Formal Aspects of Computing (1999) 11: 107–131

Formal Aspects of Computing

A Case Study in the Specification and Analysis of Design Alternatives for a User Interface

David Duke, Bob Fields and Michael D. Harrison

Human-Computer Interaction Group, Department of Computer Science, University of York, UK

Keywords: Human computer interaction; CSCW; MAL; Design space analysis

Abstract. There is considerable interest within the Human Computer Interaction (HCI) community in the use of media spaces to enhance awareness and interaction between workers in offices or other spatially distributed environments. In addition to the technical challenges of providing reliable and efficient audio-visual communication, there are important social questions, in particular how users are able to control access to their personal environments, and how to advise other users about their level of availability. Within AMODEUS-2[1], an ESPRIT Basic Research Action concerned with the development, transfer and assessment of techniques for modelling human-computer interaction, a prototype media space has been analysed by various user and system oriented modelling techniques. This paper describes how formal specification can be used to express requirements on the interfaces needed to control access and availability in a media space. Beyond its obvious use in clarifying the subtle relationship between these concerns, the paper describes how the specification assists in assessing design options originating from other modelling disciplines.

1. Introduction

An effective user interface is usually as important to the success of a system as any of the components that manage the underlying functionality of the system. A good user interface cannot turn an error-ridden product into a success, but a poor user interface can render even the most rigorously designed and verified system

Correspondence and offprint requests to: David Duke, Department of Computer Science, University of York, Heslington, York, YO10 5DD, UK. e-mail: duke@cs.york.ac.uk

[1] AMODEUS-2 technical reports are available electronically via http://www.mrc-apu.cam.ac.uk/amodeus/index.html

unfit for use. Indeed, it is often the case that when failures in critical systems occur, the failure is due to 'human error' rather than faults in the software system [Mac94], and too often humans are assisted into error by inadequacies in the user interface [WJC94]. Of course, user interface design is a difficult problem. It encompasses concerns ranging from the capabilities of low level input and output devices, through interaction techniques that increasingly involve concurrent aspects, through to social and cultural issues over the role of the system and its interface in the workplace. Interface designers must thus consider how the interface will support the functional behaviour of the system, how it will support a user of the system in achieving their goals, and how it will fit into the working environment.

The consequence of these issues is that human factors specialists and interface designers frequently need to compare and assess design alternatives based on criteria from a number of disciplines. How design decisions are reached is beyond the scope of this paper; rather, this paper describes how formal models of interactive systems can contribute to the analysis of interaction requirements. By organising a formal model around a framework for describing interaction, it becomes possible to highlight significant issues early in the development process. As a result, we show that formal representations and analyses can address issues of relevance at the user interface of a system, not just within its "functional core" (see [Arc92]).

In user interface development, design alternatives arise from the analysis or assessment of prototype interfaces by specialists such as cognitive user modellers and task analysts. It is not always obvious whether recommendations coming from such analyses are consistent or complementary, or indeed by what criteria such alternatives can be compared and assessed. This paper demonstrates how such design alternatives can be modelled formally, and consequently how such models can be used to understand the relationship and trade-offs between alternatives. The approach of this paper is a practical one; it demonstrates how the user interface of an audio-visual communications environment (a 'media space') can be modelled formally, and how such a model can be used to explore design alternatives that address the problem of controlling accessibility within such an environment. The media space described in this paper was used as an exemplar within the AMODEUS-2 project, an ESPRIT funded research action that investigated models of human-computer interaction. AMODEUS brought together experts from both system and user modelling communities, and the design alternatives considered in this paper were the result of analyses of the exemplar carried out by two groups of user modellers. The project also included researchers on methods for integrating design contributions, and on the problems of transferring modelling and integration techniques into practice, and a wider picture of the issues and techniques involved in HCI modelling can be found in [BBD96].

Section 2 provides an informal description of media space systems, and the problems of controlling access and availability through the interface to such systems. Within the AMODEUS project, two approaches to user modelling were applied to the access control problem, and both lead to proposals for new interface designs, which are described informally. The section concludes by introducing the basic concepts and definitions that will underly subsequent formal models of the interface and the design alternatives. Our approach to the specification of interactive systems, introduced in Section 3, is based on the interactor model reported in [DuH93, DuH94b]. A high-level model is developed to specify the meaning of access control in an audio-visual environment. This model is then

extended to encompass the design recommendations from the two cognitive modelling perspectives. Section 4 considers the analysis of the resulting models to obtain an understanding of the differences and trade-offs involved. The paper concludes with comments on the role of formal methods in the design of the user interface.

2. Background

2.1. Media Spaces and Mediated Communication

The media space system described in this paper was a prototype, developed as part of a project to support awareness and communication between personnel on a construction project [BBD96]. It was based on technologies similar to those developed at the Xerox Research Centre Europe in Cambridge (formerly known as Rank Xerox Europarc), which was one of the partners in the AMODEUS-2 project. Examples of the kind of system being discussed here can be found in [DoB92b, DoB92a]. In general, media space environments involve the integration of audio and video systems with a software interface. For example, in the 'Portholes' environment described in [DoB92b] users are provided with a panel of digital 'snapshots' taken from offices, plus facilities that allow a user to make various kinds of audio-visual connections to the people shown on the panel. Apart from providing the user with awareness about general activities within an office environment, snapshots and brief one-way 'glance' connections support a form of user discipline, whereby a user may desist from making 'intrusive' two-way connections if they perceive that the person they wish to communicate with is busy or non-interruptible. This relies on the goodwill and social responsibility of the user population. In practice, discipline on the part of the users is backed up by system software that allows users to decide who can make particular types of connection to their nodes.

Two concepts have emerged to support the ability of users to control the kind of connections that can be made to their node in a media space. First, a user may signal their general *availability* to other users by a mechanism that provides other users of the system with some indication of whether it is appropriate to attempt communication. Second, the general availability of a user can be augmented or overridden by the granting or revoking of specific *accessibility* rights to selected users. Whether or not the specific accessibility settings that one person has granted to another should be perceivable by the would-be-caller is a non-trivial issue of social policy. Access permission and availability settings are enforced by the software system whenever a connection is requested.

If users' access permissions are static, i.e. are set once and then remain unchanged, a simple array of toggle-buttons is probably adequate as a user interface. However in a dynamic environment users may wish to change settings relatively frequently, to reflect their changing availability to different groups within an organisation. For example, during a meeting a manager may wish to block all interruptions (regardless of source), while allowing certain people, her supervisor for example, to make non-intrusive connections. Ideally, the user should be able to set some level of availability that partially determines the connections allowed by the system, relative to the access permissions granted in general to individuals. One problem faced by the designers of such systems has been to understand the interplay between these two forms of control. For example, are they orthogonal,

or if not, how does one form interact or interfere with the other? In the case of interactive systems such as the portholes interface, interference may not only exist at the functional level, but also at the level of what a user understands about the system. For this kind of problem a formal model may not, by itself, provide much insight. Instead, a designer may need to call on techniques such as user modelling or task analysis, and incorporate the results of such analyses into the designed artefact.

2.2. User Models of a Media Space

Within the AMODEUS project, two approaches to cognitive user modelling were applied to an example of the access control problem within ECOM, a prototype media space developed within the ESPRIT EuroCODE project to support remote collaboration on a large-scale civil engineering project. These analyses were carried out by separate groups of modellers. A comprehensive account of all the modelling approaches is presented in [BBD96]; here we are concerned specifically with the results of the user modelling approaches known as CTA (Cognitive Task Analysis) and PUM (Programmable User Models).

CTA employs a broad resource-based account of human information processing to reason about aspects of a design that place particular demands on human cognitive resources. The CTA analysis (see [BaM93, BaM95]) of the ECOM prototype [MaB94] considers the 'propositional' knowledge that users will need to interpret the interface, and the assumptions under which that knowledge will be 'proceduralised' into semi-automatic skills.

PUM uses a high-level description of a user's goals and a system's resources to identify problems in the planning and execution of tasks to achieve intended goals [BlY95]. The approach derives from the "Soar" [New90] tradition of goal-directed cognitive simulation and concentrates much more directly on 'procedural' models of users' tasks. The process of building a PUM involves expressing aspects of the device, tasks and user knowledge within an 'instruction language'. So, while the CTA approach concentrates on a detailed model of the cognitive resources and processes involved in accomplishing tasks, a focus of PUM modelling is the procedural knowledge users have about their tasks. As is the case with formalisation in general, the process of representing the problem explicitly often draws out new insight, and the benefit of actually being able formally to reason about or execute the resulting description can be marginal.

An important contribution from both of the analyses was the idea that the notion of a general 'level of availability' should be separated clearly from the concept of access permission granted to (or revoked from) particular users. In both cases, these recommendations were expressed, not as abstract principles, but as explicit design suggestions in the form of revised interface designs, and the procedures by which users would achieve key tasks using these interfaces. This point is particularly important; the experience of the HCI community is that designers are not in general comfortable working with abstract recommendations [BBS95], but want to see what the analysis is suggesting in terms of a concrete interface. The danger of this approach is that details extraneous to the design argument become captured in the redesign, and it becomes difficult to appreciate what is fundamental to a design recommendation as opposed to what is superficial detail. It might seem, therefore that the practice of HCI, with its focus on the actualities of user interfaces, comes into conflict with accepted practice in the

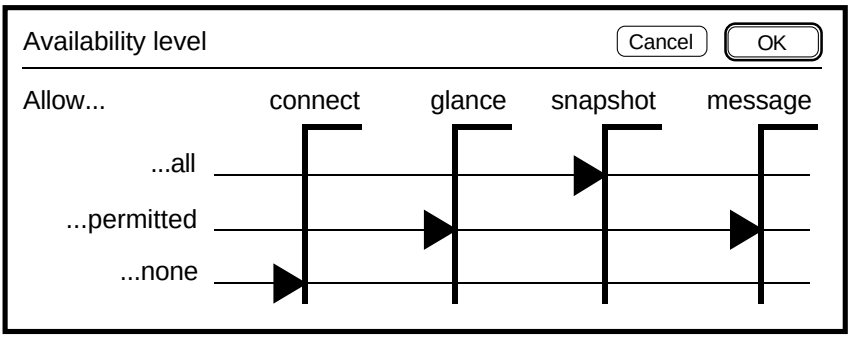

Fig. 1. Setting availability level in the CTA interface.

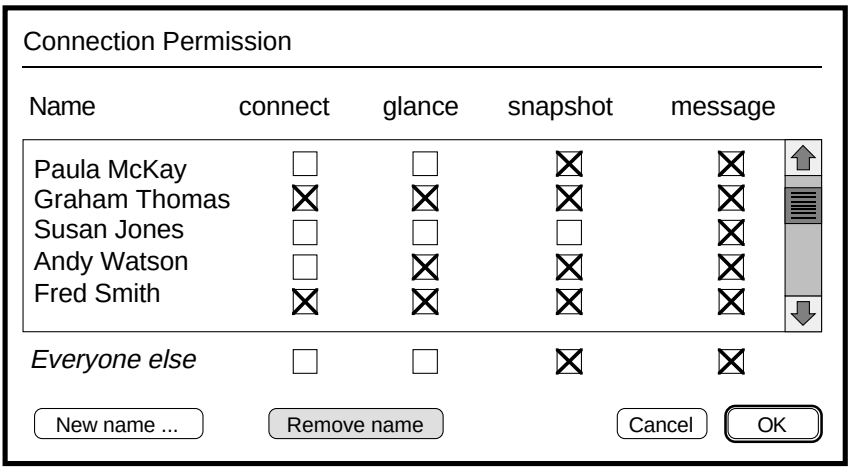

Fig. 2. Setting access permissions in CTA interface.

formal methods community, with its emphasis on abstract description. Resolving this apparent tension is an area where the specification technique introduced in the next section has been able to make valuable contributions.

The CTA Design Recommendation

In the CTA scheme, a user's general level of availability may be overridden by providing specific access control for certain combinations of users, and groups of related media services called classes. As an example, the screen structure in Fig. 1 shows a possible implementation for setting general availability suggested by the CTA analysis.

Here there are four classes of service; connect, glance, snapshot and message. A second interface, illustrated in Fig. 2, is required in order to set specific access permissions. This interface has two parts. A scrolling list contains the names of users for whom specific permissions have been defined, and the checkboxes across each row indicate those permissions. Users who have not been specifically

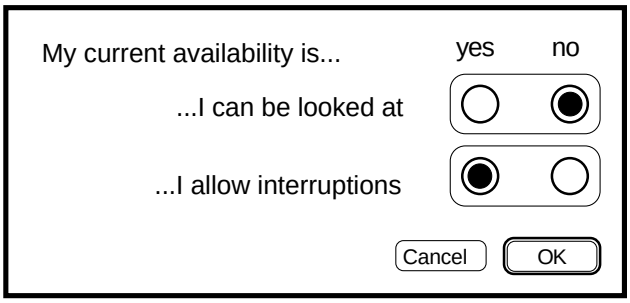

Fig. 3. Setting general level of availability in the PUM interface.

named in the list are given the access permissions specified by the row of buttons underneath the scrolling region. Informally, a would-be caller can obtain a connection using a particular kind of communications service, 'C', if

1. the callee has set their availability level to allow all 'C' connections, *or*
2. if the callee has set the availability level for 'C' to allow only permitted 'C' connections, *and either*

   (a) the callee has granted the caller specific permission to make this kind of connection, *or*
   (b) the callee has not explicitly listed the caller but has instead allowed anyone not listed to make this class of connection.

The PUM Design Recommendation

The PUM suggestion for separating availability and access control is to use the general availability only when user-specific permissions have not been set to override the general settings. Availability in this model could be set using a panel like the one shown in Fig. 3.

In their approach the PUM modellers have chosen to identify two classes of connection, those that involve the caller obtaining visual information about the callee (for example, a snapshot) and those that require the active participation of the callee, for example a "vphone" link. Each user can indicate whether, by default, they wish to allow either of these classes.

Figure 4 shows one possible realisation of the interface for setting specific access permissions. The user selects the would-be-caller for whom the access permissions are to be set from a pull-down menu on the left of the interface containing the names of all of the users in the system. This user can then, for each class connection, be given permission to always make a connection, be refused permission for a class of connection, or can have their permission determined by the callee's default availability settings.

As an example, in Fig. 4, regardless of the fact that the callee's default prohibits looking but allows interruptions, the user Susan is always allowed to look but never allowed to interrupt.

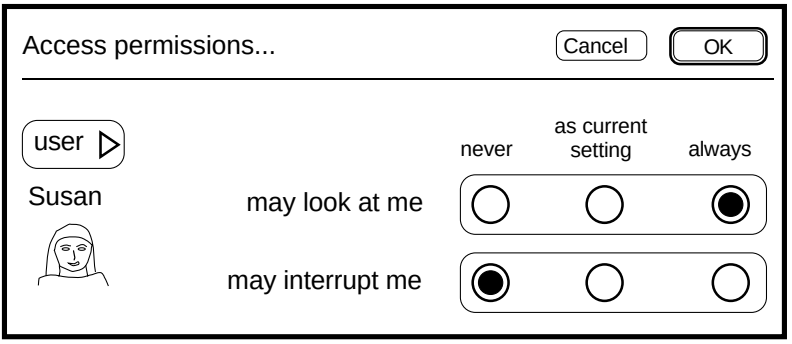

Fig. 4. Setting specific access permissions in the PUM interface.

2.3. An Abstract System Model of a Media Space

The results of the two user modelling analyses of ECOM certainly appear different, but although the informal text provided with the design sketches suggests that the results diverge, it is not obvious whether this is the result of fundamentally different views over the management of accessibility and availability, or merely superficial differences resulting from the independent development of two sets of design sketches. It should be emphasised that this is not a criticism or problem with the modelling; these 'design suggestions' are intended to help illustrate the consequence of the modelling for the benefit of designers by showing one option from a potentially large space that addresses an identified problem. By using a common specification of the access control problem as a baseline, we will explain exactly how the design proposals resulting from the two modelling approaches are related, and will consequently be able to contribute arguments and criteria that could be used within a design team to choose between these alternatives.

Before defining a model of a media space system it is necessary to introduce some definitions for the basic concepts that will appear in the model. These are: system users, kinds of service, and actual connections. At this point we are deliberately loose about specification notation; the reader should be able to map the definitions into their favourite language. We say more about the role of notation later in the paper.

type *user* - a set representing the identities of users.
service - the kind of connections that are possible.
conn - connections linking two users by a service.

The type *service* is intended to include values such as *snapshot*, *glance*, and *connect*, representing the different kinds of audio-visual facilities provided by the system. The two parties to a connection are identified as the *caller* and the *callee*. Their identities are extracted from a value $c : conn$ by the expressions '$caller(c)$' and '$callee(c)$'.

$caller : conn \rightarrow user$ - user who initiates a connection.
$callee : conn \rightarrow user$ - user who is connected to
$type : conn \rightarrow service$ - each connection involves one service.

It is also convenient to have a function that, given a caller, a service, and a callee, will return the appropriate *connection* value. This function, written as

'©', is defined below. The axiom requires that any connection built from the components of some connection 'c' will be equal to 'c'; thus a connection is defined completely by its components.

$$\copyright : user \times service \times user \rightarrow conn$$
$$\forall c : conn \bullet \copyright(caller(c), type(c), callee(c)) = c$$

The most obvious requirement that a media space system should satisfy is that it must only be possible to establish a connection if that connection is permitted by the accessibility and availability settings in force at the time. Further, it should be possible for users to change their accessibility and availability settings so that specific access rights are defined. A number of desirable features can also be identified. For example, a user should be able to determine what connections they can make without having to request a connection and then have it refused. Conversely, if a user perceives that a connection is allowed, then it should be the case that such a connection will be permitted. One aim of separating out accessibility from availability is that it should be possible for a user to change their general availability easily, to reflect common changes of work, for example, starting or ending a private meeting.

3. Modelling Interactive Systems

3.1. Formal Methods and Interactive Systems

One way in which formal methods have been finding successful application is by working in combination with other modelling techniques, for example the integration of formal methods with structured design and analysis techniques.

A number of approaches to the use of formal methods for the specification of interactive systems have been developed and reported in the literature, see for example [PaP97]. The approach that has been developed at York involves organising a formal description into structures called *interactors*. These were inspired, in part, by object-based approaches to specification, such as Object-Z [CDD90] and the agent model [Abo90]. Interactors allow the specification to be decomposed into self-contained units which focus attention on structures and relationships that are of concern to user interface designers. Specifically, an interactor [DuH93] consists of:

- an internal state, representing the functionality that some part of an interface is controlling;
- a perceivable state, representing the information that is made available to users of the system via some presentation [DuH94b]; and
- the actions that bring about changes in the internal and perceivable state.

Various specification notations, including Z and VDM, have been used to express the behaviour of interactors; this paper employs a form of Modal Action Logic (MAL) similar to that described by Ryan et al [RFM91]. The basic idea employed here is to use MAL to express relations between the states before and after the execution of actions. So an expression "$pre \Rightarrow [a]\ post$" means that if an action a occurs in a state where the condition pre holds, the resulting state will be one in which $post$ holds. A summary of the logic, and brief description of the mathematical notation used in this paper, is given in Appendix A. A simple

example of an interactor, expressed using modal action logic, can be found in Appendix B.

3.2. Access Control in a Media Space

The core of the media space specification defines the information that is accessed and controlled through the user interface, specifically, the set of connections that exist within the system and the set of connections that is permitted. Within this model it is possible to state formally the requirement on access control, that the system will establish only those connections that are permitted by the callee.

The model of core functionality is represented as an *interactor* called *system*, which will subsequently be extended into other interactors that capture the interface proposals in 2.2. The internal state of the interactor comprises three attributes, *permitted*, *established*, and *busy*, which indicate, respectively, the set of calls the system will allow, the set of calls that are currently established, and the set of users who are currently busy with a call. The presentation of this state consists simply of the attribute *can-make*, for which the '**vis**' annotation indicates that information about the connections that each user is allowed to initiate is represented visually. That is not to say that the information is necessarily always *visible* to the user (for instance, it could be obscured by another window).

interactor system

attributes

	$permitted$	$: \mathbb{P}\, conn$	- the set of all calls that are permitted
	$established$	$: \mathbb{P}\, conn$	- the set of calls currently established
	$busy$	$: \mathbb{P}\, user$	- the set of users who are currently busy
vis	$can\text{-}make$	$: user \rightarrow \mathbb{P}\, conn$	- the calls that a user is allowed to make

actions

$request(conn)$		- request the system to make a connection

axioms

1 $busy = \{caller(c) \bullet c \in established\} \cup \{callee(c) \bullet c \in established\}$

A user is busy if they are the caller or callee of some connection.

2 $$\left(\begin{array}{l} caller(c) \notin busy \\ \wedge\ callee(c) \notin busy \\ \wedge\ c \in permitted \\ \wedge\ established = X \end{array}\right) \Rightarrow [request(c)]\ established = X \cup \{c\}$$

Provided that the callee and caller are not busy, and the connection is permitted by the system, the effect of requesting a connection is that it becomes established within the system.

3 $c \in permitted \Leftrightarrow c \in can\text{-}make(caller(c))$

The set of permitted connections contains exactly those connections that can be made by any of the users in the system.

4 $can\text{-}make(u) \subseteq \{c : conn \bullet u = caller(c)\}$

The set of connections that a user can perceive as possible is a subset of those connections for which that user is the caller.

It should be noted that once a connection has been requested successfully, it may be possible for the callee to change access permission and deny permission

for the connection. Exactly how the system should mediate between existing connections and access control is an issue that lies beyond the scope of this paper, but for this reason the specification does not require that every established call be in the permitted set.

The specification does not, and should not, mention the technical aspects of establishing a connection (e.g. that suitable system resources must be available), nor does it provide a complete model of the interaction involved in calling, for example call termination is not mentioned. A more extensive specification of a media space environment can be found in [DuH94a, DuH95a, DuH95b].

3.3. Incorporating Access Control

The *system* interactor describes the information used to control access within a media space, but does not describe how users are expected to interact with the system to modify that information. This kind of information was expressed informally in the commentary that accompanied the interface design sketches. Rather than attempt to extract the specifics of each proposal directly, we first give a general model that characterises the difference between accessibility and availability from the viewpoint of a user. This characteristic is common to both proposed designs, and extracting the shared components into a single interactor will provide a better basis for understanding the critical differences.

Rather than describing access control in terms of specific connection types, a useful generalisation is to group connections into *classes*. At one extreme there may be a single class of connection, so a user could grant or revoke all access 'in one step'. At the other extreme each connection type may define a distinct class, so this scheme also describes systems in which the user can permit or deny each kind of connection selectively. It is assumed that each connection belongs to exactly one class, as defined by:

type *class*	- user's view of classes of connection
class-of : *service* → *class*	- the class of a given service type

The interactor *access* represents the information that the system provides to a specific user (identified by the attribute *owner*) about accessibility and availability settings. The 'availability' of the owner is represented by the set of classes that will (generally) be accepted, while 'accessibility' is represented by a mapping that associates a set of service classes with users of the system. In the interactor, these concepts are expressed by the attributes *general* and *specific*. Ultimately, the value of these attributes determine the permissions granted to other users of the system by a specific user.

interactor access

	system	- inherit the model of the system
attributes		
	owner : *user*	- the user who "owns" this node
vis	*general* : ℙ *class*	- general availability of owner
vis	*specific* : *user* → ℙ *class*	- accessibility for specific users
	granted : *user* → ℙ *class*	- connection classes permitted by owner

Three actions allow the user to toggle the setting for either the default or specific connections:

actions

$set(\mathbb{P}\, class)$ - set general availability
$permit(user, class)$ - grant access to a user for a service class
$revoke(user, class)$ - revoke access from a user for a service class

The axioms, given below, link the user's access permission settings to the global system view, and describe how access settings can be changed. However, they do not describe how default and specific settings determine the calls for which permission has been granted. The CTA and PUM analyses diverge on this issue. The approach of each will be described by extending this general model.

axioms

1 $class\text{-}of(s) \in granted(u) \Leftrightarrow ©(u, s, owner) \in permitted$

A user can only grant access permission for calls that have that user as callee, and a call to the owner of a node is permitted (by the system) if and only if permission for the call has been granted

2 $[set(lev)]\ general = lev$

The 'set' action updates the owner's availability setting.

3 $specific(u) = A \Rightarrow [permit(u, c)]\ specific(u) = A \cup \{c\}$

Granting permission for a user 'u' to make connections of class 'c' results in the addition of 'c' to the class of connections granted specifically to 'u'.

4 $specific(u) = A \Rightarrow [revoke(u, c)]\ specific(u) = A - \{c\}$

Revoking permission for a user 'u' to make connections of class 'c' results in the removal of 'c' from the class of connections granted specifically to 'u'.

5 $u \neq v \wedge specific(v) = A \Rightarrow$
$[permit(u, c) + revoke(u, c)]\ specific(v) = A$

Granting or revoking specific access rights for a user 'u' does not affect the rights granted by the owner to any other user.

3.4. The CTA Recommendation

The CTA user modellers recommended a design in which general availability determined access permissions except for specific classes of connection. In these cases permissions are defined on an individual basis, i.e. specific accessibility permissions. This approach is represented in the interactor *CTA-model*. This extends the generic access control framework with an attribute, *exceptions*, that defines the connection classes for which access permission is to defer to specific permission settings. It is then possible to give an axiom that links accessibility and availability to the permissions granted by a specific user.

interactor CTA-model

access - inherit the access control model

attributes

$exceptions : \mathbb{P}\, class$ - service classes that defer to specific control

axioms

1 $granted(u) =$

1.a $(general - exceptions) \cup$

1.b $(specific(u) \cap exceptions)$

Permission for a connection 'c' is granted by the owner of a node if either (1.a) the class of that connection has not been deferred to access control and that class of service is permitted at the current availability level, or (1.b) access permission for that service class is deferred, and specific permission has been granted for the caller to make that class of connection.

The *CTA-model* interactor describes a strategy for reconciling availability and accessibility, but not describe how the state might be presented to users of the access control system. Although both groups of user modellers employed user interface sketches to illustrate their recommendations, the design of the actual interface may best be undertaken by a graphic designer. A user interface development toolkit may be needed, but the actual design of the contents should be informed by psychological principles of display structure, for example [MSB95], knowledge of representational structures [Tuf90], and an understanding of what the display is to accomplish. In other words, the display must meet both human and functional requirements, and the latter fall within the remit of formal modelling. Whatever design is ultimately realised, that design should implement the functionality of the underlying model.

In practice, the implementation of user interface components, at least for GUIs composed of 'standard' widgets such as buttons, is a well understood problem and it is unlikely that there would be much value in developing the specification further. However, there are two cases where the application of formal methods may be helpful.

- New interface technologies that involve novel use of modalities such as speech and gesture, either sequentially or in combination, present both technical and human factors challenges and designers may benefit from the insight into any problem that can be gained by building a suitable model.
- Designers may need to assess or argue about the effectiveness of alternative interface proposals in terms, for example, of how easily it allows users to complete a task, or how much opportunity it presents for user error, and what the consequences of error might be.

Although we do not claim that the media space system necessarily falls into the first of these categories, it does provide a reasonable example with which to illustrate the extension of the basic modelling approach to address lower-level issues of interaction. We thus consider a possible realisation of the CTA design alternative based on the design sketches for controlling availability (Fig. 1) and accessibility (Fig. 2).

To describe the availability interface as an interactor we first define a type to represent the 'types of setting' that appear in the column on the left hand side of the interface component:

which-allowed ::= AllowAll | AllowSpecific | AllowNone

The interface is then expressed as an interactor:

interactor CTA-availability

attributes

vis $settings : class \rightarrow which\text{-}allowed$ - settings for each service class

actions

$set\text{-}to(class, which\text{-}allowed)$ - set the allowed level for a class

axioms

1 $settings = S \Rightarrow [set\text{-}to(c, a)]\ settings = S \oplus \{c \mapsto a\}$

Setting the type of access for connection class 'c' to 'a' overrides the old setting for 'c' and leaves all other settings unchanged.

The CTA-accessibility interactor represents the proposal to control specific access settings. It has two attributes:

listed represents the users named within the 'scrolling list' in Fig. 2;

selected models the setting (i.e. whether selected or not) of all buttons available via the interface at any given time. By 'available', we mean those buttons that are present somewhere on the interface; we are not, at this level of detail, modelling the effect of the scroll bar on the presentation, nor of buttons being obscured by other windows.

The status of specific buttons is modelled in the specification by mapping the name of button to a boolean value. However, to accommodate the 'default' accessibility information given by the special row of buttons labelled 'Everyone else' in Fig. 2, the type defined to represent button names is defined as a union type in the style of VDM [Jon90].

$$bt\text{-}name == (user \times class) \cup class$$

That is, *bt-name* is the union of *user-class* pairs representing buttons used to control access rights of specific users, and the service *class* names that identify the buttons for 'everyone else'.

interactor CTA-accessibility

attributes

vis $selected : bt\text{-}name \rightarrow \mathbb{B}$ - state of each button in the interface

$listed : \mathbb{P}\, user$ - the users listed explicitly

actions

$toggle(bt\text{-}name)$ - change the selected state of a button

$remv\text{-}name(user)$ - remove a user from the specific list

$new\text{-}name(user)$ - add a user to the list

axioms

1 $\mathrm{dom}\, selected = (listed \times class) \cup class$

The interface contains a button corresponding to each listed user and service combination, and a button for each service class.

2 $listed = L \wedge n \in L \Rightarrow [remv\text{-}name(n)]\ listed = L - \{n\}$

A user can be removed from the set of listed users.

3 $listed = L \wedge n \notin L \Rightarrow [new\text{-}name(n)]\ listed = L \cup \{n\}$

A user can be added to the set of listed users.

4 $[new\text{-}name(n)]\ \forall c : class \bullet selected(n, c) \Leftrightarrow selected(c)$

When a user is added to the set of listed users, the button settings for that user are taken from the button settings in force for everyone not explicitly listed.

5 $selected(b) = X \Rightarrow [toggle(b)]\ selected(b) = \neg X$

Toggling a button 'b' changes it from selected to not selected, or vice versa.

6 $a \neq b \wedge selected(b) = X \Rightarrow [toggle(a)]\ selected(b) = X$

Toggling a button 'a' leaves the selected state of all other buttons unchanged.

Note that, by preventing an already listed name from being added to the list again, the behaviour specified by axiom 3 prevents accidental changes to access permissions, as axiom 4 would require that the permissions of such a user be set to those of the unlisted users. Such interplay between constraints in user interface behaviour are not necessarily obvious from screen sketches and informal scenario descriptions.

To close the model of the CTA recommendations, the three aspects of the design – the basic model, and the interfaces for availability and accessibility control – are brought together in the *CTA-implementation* interactor. The axioms define the correspondence between interface features in the access and availability specifications and state attributes and actions in the underlying CTA model.

interactor CTA-implementation

CTA-model

CTA-availability

CTA-accessibility

axioms

1 $general = \{c : class \bullet settings(c) = \mathsf{AllowAll}\}$

A user's level of general availability is represented by the connection classes which any other user can request.

2 $exceptions = \{c : class \bullet settings(c) = \mathsf{AllowSpecific}\}$

General availability defers to specific access permissions for those connections which are marked as 'as AllowSpecific'.

3 $\forall u \in listed \bullet specific(u) = \{c : class \mid selected(u, c)\}$

When access permission defers from general availability to specific access rights, the permissions granted to explicitly listed users are for those connection classes where the button for that user - class combination has been selected.

4 $\forall u \in (user - listed) \bullet specific(u) = \{c : class \mid selected(c)\}$

When access permission defers from general availability to specific access rights, the permissions granted to any user not explicitly listed are determined by the status of the buttons representing connection classes.

The connection between the operation of specific buttons on the interfaces for availability and accessibility control, and changes to the set of connections that are permitted within the system, can be found through a chain of interactors:

- A connection request will be satisfied by the system only if it is a member of the permitted set (axiom 2 of *system*).
- For a given caller, a connection for that caller is in the permitted set if and only if permission for that caller to make that class of connection has been granted by the callee (axiom 1 of *access*).
- For the CTA model, the link between the set of connections for which permission is granted, and general (availability) and specific (accessibility) settings is given by the axiom of *CTA-model*.
- The connection between the setting of buttons on the user interface panels modelled by *CTA-availability* and *CTA-accessibility* is given by axioms 1 to 4 of *CTA-implementation*.

It is also possible to make the connection between interface actions and permission settings explicit, by defining equivalences between *toggle* actions in given situations and the generic *permit* and *revoke* actions within the *access* interactor. Carrying out this mapping can provide interesting insight into the design of the interface. The set of axioms given here as an extension to *CTA-implementation* is not complete, but illustrates the point. First, as we would hope, operating a button for a setting of an explicitly listed user affects only that user:

5 $\neg\, selected(u,c) \Rightarrow toggle(u,c) \equiv permit(u,c)$

If a button for a user-class combination is not selected, then toggling the button is equivalent to granting that user specific permission to make connections of a given class.

6 $selected(u,c) \Rightarrow toggle(u,c) \equiv revoke(u,c)$

If a button for a user-class combination is selected, then pressing the button is equivalent to revoking the specific permission of that user to make connections of the given class.

What is less obvious but also important is that adding a user to, or removing a user from the list of explicit users also changes access permissions for that user.

7 $\neg\, selected(c) \wedge u \in listed \Rightarrow remv\text{-}name(u) \equiv revoke(u,c)$

If the access settings for users explicitly listed do not allow connections of class 'c', then removing a user from those given specific permissions revokes permission for that user to make connections of class 'c'.

8 $selected(c) \wedge u \in listed \Rightarrow remv\text{-}name(u) \equiv permit(u,c)$

If the access settings for users explicitly listed do allow connections of class 'c', then removing a user from those given specific permissions grants permission for that user to make connections of class 'c'.

Further statements could be made about the relationship between other interface actions in terms of granting and/or revoking access permissions. Ideally, such statements should be taken as conjectures which can be shown to follow from the invariants. However, we have not developed the specification with proof in mind; the activity reported here has, rather, been in the spirit of 'Formal Methods Light' [Jon96].

At this point we have completed the specification of the design proposal from the CTA modellers. It will be revisited in Section 4 in which this design is compared with the proposal from the PUM modellers.

3.5. The PUM Recommendation

The specification of the design recommendations from the PUM modellers follows the approach adopted for the CTA analysis. First, the interactor *PUM-model* is defined as an extension to the generic access control interactor. As was the case in *CTA-model*, this interactor introduces an 'exceptions' variable, though now the interpretation is different: the variable indicates, for each user, which connection classes are subject to the user's general availability, as opposed to specific accessibility.

interactor PUM-model

access - inherit the access control model

attributes

$exceptions : user \rightarrow \mathbb{P}\, class$ - classes that don't defer to specific control

axioms

1 $granted(u) =$

1.a $(general \cap exceptions(u)) \cup$

1.b $(specific(u) - exceptions(u))$

Permission for a connection '*c*' is granted by the owner of a node if either (1.a) the class of that connection *has* been deferred to, and is permitted by, the current availability level, or (1.b) access permission for that service class has not been deferred and specific permission has been granted for the caller to make that class of connection.

To support the comparison between the PUM and CTA options, we again extend the specification to incorporate details of the interface proposals for controlling availability and accessibility. The interface sketched by the PUM modellers to illustrate their approach to availability control was given in Fig. 3, and is represented by the following interactor:

interactor PUM-availability

access - inherit the access control model

actions

$toggle(class)$ - set and unset a service class

axioms

1a $(general = G \wedge c \in G) \Rightarrow [toggle(c)]\, general = (G - \{c\})$

1b $(general = G \wedge c \notin G) \Rightarrow [toggle(c)]\, general = (G \cup \{c\})$

Toggling the status of a connection class adds it to the connections that are generally available, it was not in the set initially, or removes it from the set, if it was.

In the PUM interface for controlling accessibility (see Fig. 4), the access permission for a given user can be set for each connection class to be: always allow (AllowAll), never allow (AllowNone), or deferred to the general availability setting (AllowSpecific). In the interactor, this is expressed as an attribute that takes each user-class pair to one of the values in the type *which-allowed* that was defined as part of the CTA specification.

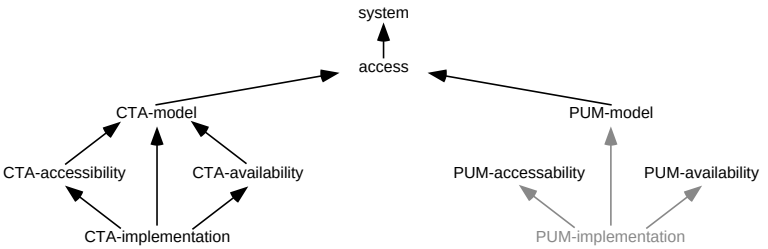

Fig. 5. Specification structure.

interactor PUM-accessibility

attributes

vis $settings : (user \times class) \rightarrow which\text{-}allowed$

actions

$set\text{-}access(user, class, which\text{-}allowed)$

axioms

1 $settings = S \Rightarrow [set\text{-}access(u, c, a)]\ settings = S \oplus \{(u, c) \mapsto a\}$

Setting a specific access permission of 'a' for user 'u' and class 'c' overrides the previous permission set for that user and class, but leaves all other settings unchanged.

That completes our discussion of the model of PUM design options for access control. Although a *PUM-implementation* interactor could be defined along the same lines as *CTA-implementation*, we do not develop the PUM specification further. For reference, Fig. 5 shows the structure of the model that has been developed.

4. Analysis

The previous section has developed specifications of the CTA and PUM design proposals from a common model of access control (the *access* interactor). In this section we consider how these specifications can be used to contribute to the analysis and comparison of these design options. Both sets of 'recommendations' separated a notion of general access permission from specific exceptions, and it is interesting to note that the two models define different interpretations for the meaning of 'general' permission defined in the *access* interactor. In the *CTA-model* interactor, general access permissions are derived from a user's availability, but can then be overridden by the user's specific accessibility settings. In contrast, the PUM design represented by *PUM-model* uses the concept of accessibility to define permissions, with the level of availability recruited only for determining the access permissions of specific individuals. These different views of the relationship between accessibility and availability have implications for how users will need to interact with the interfaces to achieve specific goals.

4.1. Reasoning About Scenarios

The interface specification extracted from the user model analyses will be considered in the context of typical user level tasks that they are intended to support.

In practice the identification of such tasks is an important part of requirements gathering and human factors assessment, about which there are considerable bodies of literature, see for example [Joh92]. In the case of access control, we have chosen to define two scenarios which highlight the strengths and weaknesses of each approach. Although these lack the authority of a real task model, they are typical of scenario fragments given to the AMODEUS modellers by the media space designers who were involved in the case studies.

Scenario S1 The 'owner' wishes to change permissions to allow all calls of a certain class 'c'. This goal can be achieved by a task T satisfying the property:

$$permitted = A \Rightarrow [T]\, permitted = A \cup \{©(u, s, owner) \bullet u : user \wedge class\text{-}of(s) = c\}$$

Scenario S2 The 'owner' wishes to remove permission for some particular user 'x' to make a certain class 'c' of connection. This goal is represented the following constraint on task T:

$$permitted = A \Rightarrow [T]\, permitted = A - \{©(x, s, owner) \bullet class\text{-}of(s) = c\}$$

We will now illustrate how the specification of the interfaces can support arguments over the interaction required within these scenarios.

CTA – S1 Assuming that the connection class corresponds to the general classes, the CTA model supports S1 well; all the user need do is use *set-to* to allow all calls in the class 'c', i.e. the user initiates the action

$$set\text{-}to(c, \mathsf{AllowAll})$$

PUM – S1 For the PUM model S1 is more difficult. From the specification, it can be seen that the user has two options, described below, involving two sets of users:

$P = \{u : user \bullet settings(u, c) = \mathsf{AllowSpecific}\}$ - users permitted class c
$N = \{u : user \bullet settings(u, c) = \mathsf{AllowNone}\}$ - users denied class c

Assuming that the owner is initially unavailable for communications of class 'c', the first option is to change the default setting to allow connections of type 'c', and then change any exception that denies permission for 'c' so that 'c' is allowed. This can either be done by permitting 'c' outright, or by telling the system to take the permission from the default setting. The actions required are:

$toggle(c)$ - toggle the default permission for service class c.
$set\text{-}access(u, c, \mathsf{AllowSpecific})$ for $u \in N$ - override existing exceptions for that class.

The second option assumes once more that the current default does not allow connections of type 'c', and the user explicitly changes all exceptions where necessary to allow the call. Here the required actions are, for each $u \in P \cup N$:

$set\text{-}access(u, c, \mathsf{AllowAll})$ - override all existing exceptions for the class.

By taking the first option and changing the default permission, the user may be able to cancel permission easily for the call class at some later time – the

'best' option to take will clearly depend on the broader background of the user's pattern of access control.

CTA – S2 Scenario 2 appears to bring out the advantages of the PUMs approach. In order to remove permission for a particular connection using the CTA approach, it may first be necessary to change general availability for the connection class. However, since general availability overrides specific settings, this may also deny access to users other than the one being excluded. A worst case scenario is where the general availability for the connection class is set to 'AllowAll', but for most users the specific setting is to refuse permission for that class. If R is the set of users $\{u \bullet c \notin specific(u)\}$ then the actions required are:

$set\text{-}to(c, \mathsf{AllowSpecific})$ - allow the use of class 'c' only if given specific permission
$revoke(x, c)$ - remove permission from the designated user
$permit(u', c)$ for each $u' \in R$ - grant permission to all other users.

PUM – S2 In contrast to this, denying access to one user for a given class of connection is straightforward in the PUM model; it just requires:

$set\text{-}access(x, c, \mathsf{AllowNone})$ - refuse permission for 'x' to make connections of class 'c'.

It should be noted that the same problems exist regardless of whether permission is being granted or denied. In the CTA design it is easy to make wholesale changes in general availability, but it may become unwieldy when making localised changes. The PUM recommendation is the inverse, allowing easy control over individual users' access permissions but potentially complicating the task of setting up a general availability level.

To conclude, we note that no axioms have been given to describe the *initial* state of a system. For example, what should the initial general accessibility level be set to for CTA, or what specific exceptions should be allowed by the PUM approach? Two policies seem obvious, one to allow the user to make any call, the other to refuse permissions for all calls. It depends on whether a user is going to take a 'negative' view and only grant access to certain others, or a 'positive' view and only refuse access to particular users.

4.2. Analysing Erroneous Actions

The previous sub-section shows how alternative interface designs may be compared on the basis of the ease with which they support tasks a user may wish to perform. Another, related criterion for selecting between alternatives is to consider unintended user behaviour, or erroneous actions (see [FWH95, FHW97]). By erroneous actions, we mean those that are possible, but unintended. Typically, errors are counterproductive in the sense that they have a negative impact on the achievement of an intended outcome (cf. [Rea90]). In doing this, the aim is not to apportion blame for system failures to the user, but rather to address the design issue of how what is known about human error can be better used in design. When considering failures in human-system interaction, three factors are significant: what errors can occur (and are *likely* to occur)? how easily can the user detect and recover from these errors? and what is the effect of these errors?

Table 1. Analysis of operator errors

	Task Error	Consequences
Omission of actions		
1	Omit *toggle*(c)	No effect, leaving *permitted* unchanged, so that no new connections are allowed.
2	Omit *set-access*(u, c, AllowSpecific)	User u incorrectly denied access.
Substitution of c' for class c ($c' \neq c$)		
3	Substitute c' for c throughout	The intended users are permitted the wrong kind of access, but not given the intended access permissions
4	*set-access*(u, c', AllowSpecific)	User u is incorrectly permitted calls of class $c' \neq c$, but not given the intended access permissions
Substitution of u' for some specific user u		
5	*set-access*(u', c, AllowSpecific)	No change for user u', as user $u' \notin N$ was already permitted to make calls of class c; permissions for the intended user u are unchanged
Substitution of AllowSpecific for one user, u		
6	*set-access*(u, c, AllowAll)	Intended effect – user u is permitted class c connections.

The first two of these issues, concerning the likelihood of particular errors occurring and being detected and remedied, are empirical or psychological questions and will not be treated here. A systematic way of addressing the remaining question, of analysing the effects of errors, relies upon a formalised model of system and interface behaviour, as presented above. The starting point for an analysis of the impact of human error is an understanding of the tasks the user must perform, as well as a model allowing predictions to be made about the effects of user actions on the system state. Samples of the former are given for particular scenarios in the previous sub-section and the latter is provided by the interactor specifications in preceding sections.

The approach to using task descriptions together with models of the system's behaviour to analyse user error will involve the systematic examination of some of the ways that tasks can be carried out incorrectly. Two types of error will be considered here: *omission* of an entire task action, and *substitution* of an incorrect parameter to an action (which will typically correspond, in interaction with a real implementation, to the mis-selection of a button or other control). These two patterns of error are only a small (though important) selection from the large body of literature on classifications of erroneous actions.

As an example of how an error analysis may be conducted, consider the first PUM task given in Section 4 for scenario S1. The possible omission and substitution errors that could occur in this are tabulated in Table 1. This table shows the effects on the *permitted* set, calculated from the interactor specification, that result if the remainder of the task is completed correctly. Although this process of going from behaviours to their effects is described informally here, it is underpinned by an analysis of the formal models described previously.

The consideration of user errors and their overall impact on the system can form an important part of an analysis of a system's usability. An analysis of this

kind can help to distinguish between types of error (for example, between those that have a relatively benign effect, and those that give users unintended access permissions). Additionally, the analysis can help a designer to draw a further distinction between the design proposals. For example, although the PUM S1 task is more complex than CTA S1, and therefore contains more opportunities for error, many of the errors in the PUM task can be shown to be harmless, in the sense that they don't permit unintended connections. Subsequent design activity might seek to address the error deemed sufficiently consequential in a number of ways (for example, making the error impossible, or by allowing the error to occur but supporting the user in detecting and recovering from it). Although such error analysis remains largely an informal process, relying on the skill of the analyst, formal models of the kind described in this paper can support the analysis in two main ways.

Firstly, and perhaps most obviously, formal models can support the analyst in determining the effects of user actions (and thereby in filling in the "consequences" column of Table 1). This can be done either by proving conjectures about the state of the system following an action or action sequence, or, where possible, by executing or animating the specification. If it is found that an undesirable state is reached as the result of some error in a task, then the designer may wish to introduce interface features to discourage the error or mitigate its effects. Everyday examples of such features are the "greying out" of non-permitted menu items or the use of "modal" dialogue boxes which prevent progress being made until an important action is carried out. The above analysis has suggested that some of the erroneous performances of the task (3 and 4 in Table 1) may be more consequential than others (they grant additional, unwanted permissions). Re-design efforts can be targeted at these specific error forms, and can result in a number of design possibilities (such as providing the user with an explicit representation of the *permitted* set).

The second kind of support afforded by the use of formal models is that the perceivable effects of erroneous action can be studied with the help of the "$\boxed{\text{vis}}$" annotations of interactors' state variables. This allows the analyst to investigate whether the system's user interface provides enough information for a user to distinguish between states reached by correct and erroneous inputs.[2] Once again, formal proof or animation of the specification could be employed here to reason not about the whole state space of an interactor, but just about the portion of it marked as perceivable.

In short, formal modelling can assist designers in clarifying what erroneous actions are being considered. Furthermore, such models are important in analysing what the effects of actions will be in the context of an otherwise correctly performed task. In this way, formal modelling and an examination of error can together contribute to the design of user interfaces that are robust to the rigours of ordinary use.

5. Conclusions

This paper has described how a formal specification of two design alternatives could support arguments over design options, both by clarifying the differences

[2] Of course, the question of whether a user actually *will* notice when an error has been made, is a psychological question that is outside the scope of the modelling activity described here.

between options, and in terms of criteria such as support for specific tasks or the likelihood and consequences of user error. In presenting this analysis, we are not suggesting that user interface designers need or should become experts in formal methods. A designer is interested in the consequences of the analysis, not in its derivation, and typically there are factors affecting the choice between design alternatives for which formal techniques are either impractical or simply inappropriate.

Case studies such as this suggest that formal specification techniques can assist in developing usable interfaces. A specification can make clear what information can or should be presented to users, and what effect user actions should have both on internal and perceivable state. In particular, this paper has shown that a formal model of a system can be extended to capture recommendations from other disciplines as part of the broader process of understanding, documenting, and comparing design alternatives. We have used the resulting models for three purposes:

- A specification can capture precisely design recommendations at a high level of abstraction. CTA and PUMs recommendations for separating access permissions from general availability can be modelled independently of design 'suggestions' used to encapsulate the output of modelling. Although choice of recommendations may be made on grounds other than their impact on the functional behaviour of the system, at least that impact is known precisely.

- Competing design alternatives may also be contrasted on the basis of user level criteria such as the nature of the tasks required to achieve particular user goals or the effects that user errors will have on the system state.

- Recommendations from different modelling techniques can be compared. Doing this for the CTA and PUMs recommendations for the access control problem shows that they take opposite views of general versus specific access permission. This is useful both for the designer, in comparing modelling recommendations, and for the modellers themselves who may then be able to look at the theory or rationale that led to a particular suggestion.

The AMODEUS project utilised and developed a number of techniques for integrating multidisciplinary analyses [BlD97], and the use of formal specifications to contribute to the development to a space of design options and criteria has already been documented, see for example [BFH95, BBD96]. However, the complex interactions between accessibility and availability described in this paper have been a problem for designers of media space systems, and what we have demonstrated is that it is possible, by using mathematical models, to better understand the relationships and trade-offs between interfaces that manage these concepts. Such an analysis could be carried out formally, but in practice the techniques described in this section are more in the spirit of the 'formal methods light' approach advocated for example by Jones [Jon96]. As is being discovered in other areas such as information systems modelling and structured design, the mathematics underlying formal methods can be recruited to enhance and clarify existing techniques for describing and analysing problems. In this way, we believe that formal methods do have a role in practical HCI.

References

[Abo90] Abowd, G.: Agents: Communicating interactive processes. In D. Diaper, D. Gilmore, G. Cockton, and B. Shackel, editors, *Proceedings of INTERACT'90*, pages 143–146. North-Holland, 1990.

[Arc92] Arch, a metamodel for the runtime architecture of an interactive system. In ACM SIGCHI Bulletin, Volume 24 Number 1, 1992. UIMS Developers Workshop.

[BBD96] Bellotti, V., Blandford, A. E., Duke, D. J., Maclean, A., May, J. and Nigay, L.: Controlling accessibility in computer mediated communications: A systematic analysis of the design space. *Human Computer Interaction*, 1996.

[BBS95] Bellotti, V., Buckingham Shum, S. J., MacLean, A. and Hammond, N.: Multidisciplinary modelling in HCI design ... in theory and in practice. In *Proc. CHI'95*. ACM Press, 1995.

[BlD97] Blandford, A. and Duke, D. J.: Integrating user and computer system concerns in the design of interactive systems. *International Journal of Human-Computer Studies*, 46:653–679, 1997.

[BFH95] Bramwell, C., Fields, B. and Harrison, M. D.: Exploring design options rationally. In P. Palanque and R. Bastide, editors, *DSV-IS'95: Eurographics Workshop on Design, Specification and Verification of Interactive Systems*, pages 134–148. Springer-Verlag, 1995.

[BaM93] Barnard, P. J. and May, J.: Cognitive modelling for user requirements. In P.F. Byerley, P.J. Barnard, and J. May, editors, *Computers, Communication and Usability: Design Issues, Research and Methods for Integrated Services*, North Holland Series in Telecommunication. Elsevier, 1993.

[BaM95] Barnard, P. J. and May, J.: Interactions with advanced graphical interfaces and the deployment of latent human knowledge. In *Eurographics Workshop on Design, Specification and Verification of Interactive Systems*, pages 15–49. Springer, June 1995.

[BlY95] Blandford, A. E. and Young, R. M.: Separating user and device descriptions for modelling interactive problem solving. In K. Nordby, P. Helmersen and D. J. Gilmore, and S. Arnsen, editors, *Human-Computer Interaction: INTERACT'95*, pages 91–96. Chapman and Hall, 1995.

[CDD90] Carrington, D. A., Duke, D. J., Duke, R. W., King, P., Rose, G. A. and Smith, G.: Object-Z: An object-oriented extension to Z. In S. Vuong, editor, *Formal Description Techniques (FORTE'89)*. North Holland, 1990.

[DoB92a] Dourish, P. and Bellotti, V.: Awareness and coordination in shared workspaces. In *Proc. ACM Conference on Computer-Supported Cooperative Work CSCW'92*. ACM Press, 1992.

[DoB92b] Dourish, P. and Bly, S.: Portholes: Supporting awareness in a distributed work group. In *Proc. ACM Conference on Human Factors in Computing Systems CHI'92*. ACM Press, 1992.

[DuH93] Duke, D. J. and Harrison, M. D.: Abstract Interaction Objects. *Computer Graphics Forum*, 12(3):C–25 – C–36, 1993.

[DuH94a] Duke, D. J. and Harrison, M. D.: Connections: From A(V) to Z. Technical Report SM/WP29, ESPRIT BRA 7040 Amodeus-2, January 1994.

[DuH94b] Duke, D. J. and Harrison, M. D.: A theory of presentations. In M. Naftalin, T. Denvir, and M. Bertran, editors, *FME'94: Industrial Benefit of Formal Methods*, volume 873 of *Lecture Notes in Computer Science*, pages 271–290. Springer-Verlag, 1994.

[DuH95a] Duke, D. J. and Harrison, M. D.: Folding human factors into rigorous development. In F. Paternò, editor, *Interactive Systems: Design, Specification and Verification*, Focus on Computer Graphics, pages 333–347. Springer-Verlag, June 1995.

[DuH95b] Duke, D. J. and Harrison, M. D.: From formal models to formal methods. In R.N. Taylor and J. Coutaz, editors, *Proc Intl. Workshop on Software Engineering and Human-Computer Interaction*, volume 896 of *Lecture Notes in Computer Science*, pages 159–173. Springer-Verlag, 1995.

[DuH95c] Duke, D. J. and Harrison, M. D.: Interaction and task requirements. In P. Palanque and R. Bastide, editors, *DSV-IS'95: Eurographics Workshop on Design, Specification and Verification of Interactive Systems*, pages 54–75. Springer-Verlag, 1995.

[FHW97] Fields, B., Harrison, M. and Wright, P.: THEA: human error analysis for requirements definition. Technical Report YCS-294, Dept. of Computer Science, University of York, 1997. Available via http://www.cs.york.ac.uk/~bob/papers.html.

[FWH95] Fields, B., Wright, P. C. and Harrison, M. D.: A task centered approach to analysing human error tolerance requirements. In P. Zave, editor, *Second IEEE International Symposium on Requirements Engineering (RE'95)*, pages 18–26. IEEE Computer Society Press, 1995.

[Joh92] Johnson, P.: *Human Computer Interaction: Psychology, Task Analysis and Software Engineering*. McGraw-Hill, 1992.
[Jon90] Jones, C. B.: *Systematic Software Development Using VDM*. Prentice Hall International, second edition, 1990.
[Jon96] Jones, C. B.: A Rigorous Approach to Formal Methods. *IEEE Computer*, 29(4):20–21, 1996.
[KMQ93] Kent, S. J., Maibaum, T. S. and Quirk, W. J.: Formally specifying temporal constraints and error recovery. In *Proc. of the IEEE International Workshop on Requirements Engineering*, pages 208–215. IEEE Press, 1993.
[Mac94] MacKenzie, D.: Computer-related accidental death: an empirical exploration. *Science and Public Policy*, 21(4):233–248, August 1994.
[MaB94] May, J. and Barnard, P. J.: A cognitive task analysis of the EuroCODE exemplar material. Technical Report UM/WP22, ESPRIT BRA 7040 Amodeus-2, 1994.
[MSB95] May, J., Scott, S. and Barnard, P. J.: *Structuring Displays: A Psychological Guide*. Eurographics Tutorial Notes. European Association for Computer Graphics, Geneva, 1995.
[New90] Newell, A.: *Unified Theories of Cognition*. Harvard Universtity Press, 1990.
[PaP97] Paternò, F. and Palanque, P.: editors. *Formal Methods in Human Computer Interaction*. Springer-Verlag, 1997.
[Rea90] Reason, J.: *Human Error*. Cambridge University Press, 1990.
[RFM91] Ryan, M., Fiadeiro, J. and Maibaum, T.: Sharing actions and attributes in modal action logic. In T. Ito and A.R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 569–593. Springer-Verlag, 1991.
[Spi92] Spivey, J. M.: *The Z Notation: A Reference Manual*. Prentice Hall International, second edition, 1992.
[Tuf90] Tufte, E. R.: *Envisioning Information*. Graphics Press, Cheshire CT, 1990.
[WJC94] Woods, D. D., Johannesen, L. J., Cook, R. I. and Sarter, N. B.: Behind human error: Cognitive systems, computers and hindsight. State-of-the-Art Report SOAR 94-01, CSERIAC (Crew System Ergonomics Information Analysis Center), December 1994.

A. Modal Action Logic

Axioms describing the behaviour of interactors are defined in this paper using structured Modal Action Logic (MAL) [RFM91]. This has the usual operators of classical first order logic, for example $\wedge$ (and), $\neg$ (not), $\Rightarrow$ (implies) and $\forall$ (for all), but also includes modal operators of the form $[A]\_$ where A is an action expression. If P is a predicate, $[A]\,P$ means that P must hold in the state following performance of A. Modal predicates often occur within implications; $P \Rightarrow [A]\,Q$ is read as 'if P is true then performing action A will result in a state where Q is true'. The action 'A' can either be atomic, or it can be an expression representing either choice ($B + C$), parallel performance ($B\&C$) of other actions or exclusion ($\overline{B}$ – any action other than B).

Other mathematical notation is based loosely on Z [Spi92]. In particular, we use the powerset constructor ($\mathbb{P}$), cartesian product ($\times$), disjoint union ($\cup$) taken from VDM [Jon90], and function space ($\rightarrow$). Operations involving sets are the usual ones: $\cup$, $\cap$, $-$ for set difference, and $\in$ ($\notin$) for membership (non-membership). For $f : X \rightarrow Y$,

$$\mathrm{dom} f = \{x : X \mid \exists y : Y \bullet (x, y) \in f\}.$$

For clarity, the tuple (x, y) is sometimes written using maplet notation $x \mapsto y$. Function overriding is represented as $f \oplus g$; for $x \in \mathrm{dom}\, g$, $(f \oplus g)(x) = g(x)$, while for $x \in (\mathrm{dom} f - \mathrm{dom}\, g)$, $(f \oplus g)(x) = f(x)$.

B. Interactor Representation

An interactor consists of internal state, perceivable state, and actions that operate on both states. The behaviour of an interactor is described by a collection of axioms expressed within some logic. For example, the following interactor describes a simple button as might be found on a graphical user interface. The button can be selected, and can be enabled or disabled. Both of these 'properties' are visually perceivable, but exactly how the button is presented is not of concern.

interactor button

attributes

vis	$selected : \mathbb{B}$	- in this case, just one boolean-value variable
vis	$enabled : \mathbb{B}$	- a flag to indicate whether pressing is allowed

actions

press	- press the button
$setMode(\mathbb{B})$	- control whether the button is enabled

axioms

1 $enabled \wedge selected = X \Rightarrow [press]\ selected = \neg X$

If a button is enabled, and its selected state is 'X', then after the button has been pressed, its selected state is flipped to 'not X'.

2 $[setMode(b)]\ enabled = b$

Set the flag to indicate whether or not the button is enabled.

The *state* of the interactor is modelled by a set of typed attributes (variables), in this case '*selected*' and '*enabled*'. Variables that can be perceived by a user of the system form the *presentation* and are annotated with the "modality" or channel through which they can be perceived; see [DuH94b]. In the button interactor, the fact that the button is selected (or not) is perceived visually (hence the **vis** annotation). Two *actions*, *press* and *setMode*, are available; note that the latter action is parameterised. The effect of the *press* action, specified by Axiom 1, is to toggle the button between being selected or not selected. Free variables in axioms, such as X in the example above, are implicitly universally quantified. The intention is that the button will be used by other system components that might respond to the '*selected*' state, or use the *setMode* action, in application-specific ways. In principle, some form of "frame condition" is required to define which variables do not change when an action occurs. For example, axiom 2 above should also assert that the value of *selected* is unchanged by the *setMode* action. This issue is discussed in [KMQ93]; frame conditions are omitted in this paper unless they are crucial to the discussion. In addition to the modal operator employed in this paper, MAL also includes a deontic component has been utilised in related work [DuH95c, BBD96].

Received August 1996
Accepted in revised form April 1999 by D. J. Cooke