Using automated reasoning in the design of an audio-visual communication system*

José C. Campos and Michael D. Harrison

Human-Computer Interaction Group
Department of Computer Science, University of York
e-mail: {Jose.Campos,Michael.Harrison}@cs.york.ac.uk

Abstract. Formal reasoning about how users and systems interact poses a difficult challenge. Interactive systems design provides a context in which the subjective area of human understanding meets the objectivity of computer systems logic. We present results of a case study in the use of automated reasoning to aid the formal analysis of interactive systems. We show how we can use human-factors issues to generate properties of interest, and how we can use model checking and theorem proving to analyse our specifications against those properties. This is part of ongoing work in the development of a tool to allow the automatic translation of interactor based specifications into SMV, and in the analysis of the role which different verification techniques might have during the development of interactive systems.

1 Introduction

In this paper we present results of an ongoing case study in the use of automated reasoning to aid the formal analysis of a proposed design of an interactive system. This case study is being undertaken as part of the development of a tool to allow the automated verifi cation (through model checking) of interactor specifi cations: a "MAL-based interactors" to SMV compiler (cf. [5,4]). We are also interested in discussing the role which different verifi cation techniques might have during the development of interactive systems.

When reasoning about models of systems, we can identify important classes of properties, including:

1. the coherence of the model (for example, type checking), used to answer questions of the type: "are we building the model right?";
2. the functional behaviour of the system (for example, safety properties), used to answer questions of the type: "are we modelling the right functionality?";
3. how will system and users interact, used to answer questions of the type: "will the system be easy to use?".

* Published In D. J. Duke and A. Puerta, editors, *Design, Specification and Verification of Interactive Systems '99*, Springer Computer Science, pages 167-188. Eurographics, Springer-Verlag/Wien, 1999. This version with minor correction made in 23/09/99.

While properties of type 1 and 2 are, obviously, important, our research deals with properties of type 3. The ability to identify and verify this type of property is fundamental, as is confirmed by the recognition that even the best functionality can be rendered useless by a badly designed user interface.

Formal (mathematically based) methods have proven useful in dealing with class 1 and 2 properties. This is specially true when we can resort to automated tools to aid reasoning. It is natural therefore to consider the applicability of such methods and tools to the development of interactive systems. This has been an active area of research in recent years (see, for example, [18,2,6]).

Formal reasoning about how users and systems interact poses a difficult challenge. Interactive systems design provides a context in which the subjective area of human understanding meets the objectivity of computer systems logic. In order to analyse, in a formal context, aspects of system design which have a degree of subjectivity, we must find some way to merge both areas. In this paper we show how we can use human-factors issues to generate properties of interest. We build an interactor based model of a system, and use the interactors to SMV compiler to enable us to check those properties against the model, using the SMV [16] model checker. The use of PVS [17], a theorem prover, is also illustrated. We will also point out how the results of such analysis can then be fed back into a human-factors context for subjective analysis.

1.1 The case study

We selected for this case-study the ECOM system [1]. ECOM is a audio-visual communication system. Audio-visual communication systems attempt to enhance collaboration and awareness between a community of users distributed over a number of distinct physical locations. This is done by offering a number of means of contact between users. These will include audio, video, and any other form of exchange of information that might be judged appropriate for a specific system.

This type of system presents a tension between the need to promote a sense of awareness between users, and the need to preserve individual privacy. In order to address the privacy issue users are offered some mechanism to control how/by whom they can be contacted (their accessibility).

One of the features that makes ECOM interesting from a design analysis point of view is the attempt at integrating two different such mechanisms. ECOM integrates features from both CAVECAT and RAVE, two previous media space systems (see [1] for a description). In CAVECAT accessibilities are represented by a door state. There are four such door states (open, ajar, closed, and locked), each representing a different accessibility level, with its associated set of allowed connections. Users can select an appropriate door state and can see the doors states of other users in the system. In this way users can select an appropriate level of accessibility. If, for example, some user is in an important meeting, and does not want to be interrupted, he can *lock the door* thus preventing connections to him. At the same time users can have a notion of how accessible other users are, by looking at each other's door state.

In RAVE accessibilities are set on a per user basis. Users can select a specific type of connection and specify which users are allowed to establish it. Awareness is promoted

by a panel showing periodically updated snapshots of all users. The idea being that these snapshot will give an idea of how busy (or not) a user is and hence how receptive to connections.

CAVECAT allows for an easy change in the accessibility level. However it does not include the possibility of exceptions to the general setting: you might want the meeting not to be interrupted *unless* it is the boss! On the other hand, RAVE allows a better tailoring of the accessibility setting, but makes it harder to make a global change since the accessibility setting of every user will have to be updated manually.

ECOM proposes the integration of both mechanisms. Users can set a general accessibility level using the door state metaphor, but a mechanism for exception setting is introduced that allows for specifi c users to have different accessibility rights.

Setting exceptions is done by selecting a user, a specifi c type of connection and the most conservative door state that still allows the user to connect (see Figure 1). Hence, if the exception level is set to *when Door Ajar*, the connection will also be allowed when the door is open. We will call this the cumulative nature of exceptions. Additionally the

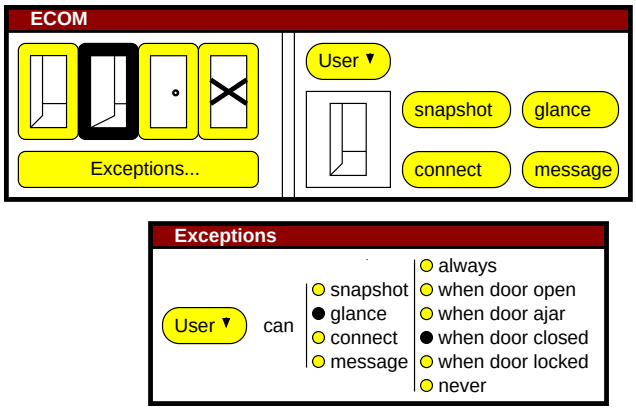

Fig. 1. The ECOM system (adapted from [1])

exception can be set to *always* or *never*.

By merging both designs, the designers hope to improve on the previous systems by incorporating the best features of each.

So far we have only described how each individual user sets an accessibility level. As in CAVECAT, awareness is promoted by presenting the door state of each user to all other users. One interesting question is whether this mechanism is still valid in the new system.

2 Checking the Proposed Design

To study the applicability of automated reasoning techniques to this case study, we must start by selecting some candidate features of the system, that we wish to analyse. We can resort to a number of sources to aid us in this process:

– *off-the-shelf* catalogues of design principles — these can be found in the literature on human computer interaction (see, for example, [7, Chapter four]);
– the results of other approaches to the analysis of the system — in the present case, some of the decision that were made regarding what properties to check were informed by the results of previous cognitive and formal analysis reported in [1];
– past experience in the development of systems — the *critical eye*.

We will look at two particular aspects of the system, because they seem interesting, and also because we hope they will allow us to illustrate the application of different reasoning techniques. First we will look at the issue of predictability. Being a general principle, predictability might be looked at from a variety of angles. We will consider the following angle: "the user should understand from the interface if an operation they might like to invoke cannot be performed" [7]. This is particularly relevant since we are attempting to promote a sense of awareness of how available other users are in the system: if the mechanism to establish connections is not predictable, then users will lose faith in the state they perceive the system to be in, thus defeating awareness. Hence, we will evaluate the design against the following criterion:

Can a user predict whether attempting a connection will succeed?

The previous criterion was obtained by applying a generic design principle (predictability) to the specific system being considered. In the second case we base our analysis on simple *gut feeling*. It was the authors' first impression, on reading the systems description, that there is redundancy in the levels that can be used to set an exception. More specifically, we want to investigate whether there is any real need to have the *always* and *never* exception setting levels. We then establish another two criteria for analysis:

– The effect of setting the exception level to *always* cannot be achieved by setting it to some other level.
– The effect of setting the exception level to *never* cannot be achieved by setting it to some other level.

We will now try to assess the proposed design against the criteria enumerated above. To do that we develop models that focus on those aspects of the system that are most relevant to the principles being analysed (cf. [13,5]). Note that this is yet another instance of the general process of model building: we abstract away what is considered to be accessory, and focus on what is considered to be relevant. As with any other model, care must be taken that all which is relevant is conveniently included.

2.1 Predictability

We will now look at the first criterion:

Can a user predict whether attempting a connection will succeed?

We start by building a model of the proposed system. To do that, we have to identify those aspects of the system which relate to the property we want to investigate. In the present case we have to model:

- the mechanisms for accessibility and exceptions setting — they have an impact on which connections can be established;
- the mechanism for establishing connections — it is what we are looking at;
- the mechanism for promoting awareness — it gives users information about other users.

The model is loosely based on one presented in [1]. We will consider a user panel representing the interface to a single user, and the system core representing the remainder of the system. The user accesses the system core through the user panel.

We will use interactors [12,10] to write the model. Modal Action Logic (MAL) [19] will be used to specify interactor behaviour. First, some types are introduced:

types

User	# all users
Service	# available services
Door	# door states
Conn = User × Service × User	# connections

The type names should be self explanatory. Services represent possible connection types, and connections are defined as tuples built with two users (the caller and the callee), and a service (the type of connection).

The system core model is presented in figure 2. Its main task is to manage connections. At each instant there are a number of connections in progress (attribute current). Two actions manipulate current: action establish initiates a given connection, and close ends it. Attribute default associates, with each door state, the valid services for that state. The validity of a specific connection depends on the level of accessibility of the callee (which, in turn, is determined by the callee's door state — cf. attribute accessibility). However, this mechanism can be overridden by setting exceptions. These are registered by associating the desired connections with the most conservative door state that still allows each connection to take place (attribute exceptions).

To simplify reading the specification, a further attribute is used which represents the allowed connections at each moment (attribute allowed). The process of determining if a connection is allowed (i.e. the value of attribute allowed) is specified by axiom (5). It mimics the reasoning just described.

Two more actions are considered: action setexcep introduces an exception into the system, and action setacc sets the accessibility of a given user. Note that the behaviour of establish and close has been left deliberately under-specified (see axioms 1 and 2). The axioms state what happens if the parameters to the actions are valid, action behaviour under invalid conditions is left unspecified. It would have been possible to specify that the actions can only happen when the parameters are valid, but that would

interactor core
attributes
allowed: $\mathbb{P}$ Conn
accessibility: User $\rightarrow$ Door
current: $\mathbb{P}$ Conn
default: Door $\rightarrow \mathbb{P}$ Service
exceptions: Conn $\rightarrow$ Door
action
establish(Conn) close(Conn)
setexcep(User,Service,User,Door) setacc(User,Door)
axioms
(1) c $\in$ allowed $\rightarrow$ [establish(c)] current' = current $\cup$ {c}
$\wedge$ unchanged(accessibility, default, exceptions)
(2) c $\in$ current $\rightarrow$ [close(c)] current' = current - {c}
$\wedge$ unchanged(accessibility, default, exceptions)
(3) [setexcep(u1,s,u2,d)] exceptions'=exceptions+[(u1,s,u2) $\rightarrow$ d]
$\wedge$ unchanged(accessibility, default, current)
(4) [setacc(u,d)] accessibility'=accessibility+[u $\rightarrow$ d]
$\wedge$ unchanged(exceptions, default, current)
(5) (caller,type,callee) $\in$ allowed $\leftrightarrow$
(((caller,type,callee) $\notin$ dom(exceptions)$\rightarrow$
type $\in$ default(accessibility(callee)))
$\wedge$
((caller,type,callee) $\in$ dom(exceptions) $\rightarrow$
exceptions((caller,type,callee)) $\geq$ accessibility(callee)))

Fig. 2. Core

have been too restrictive, as it would limit further development of the model: for example, we might wish to add later that an error message should be generated when an illegal action is attempted.

The user panel is built on top of the core, its model is presented in figure 3. Since, in the present case, we are looking at a situation where one user is trying to establish a connection, the model of the user panel is developed only so far as to make the analysis possible. Hence, the model includes the buttons that are used to request connections (attribute buttons), the user that has been selected as callee (attribute chosen), and its door state (attribute door-icon). The callee is set by action select. Finally, the user panel has an owner, the caller (attribute owner). The axioms should be self explanatory. Axiom 3 defines that then a button is pressed (buttons(s).$>_{\texttt{action}}$=pressed) a request to establish the corresponding connection is originated ($>_{\texttt{action}}$=request(owner, s, chosen)). The special attribute $>_{\texttt{action}}$ represents the action that has taken place (see [4]).

In the present context we will use SMV [16] to perform the verification. Since SMV is a model checker, the analysis of SMV specifications is completely automated. To make the model checking of interactor specifications easier, we are developing a compiler that generates SMV code directly from interactor specifications. An initial version of the compiler was introduced in [5]. In [4] the tool is further developed and the correctness of the translation process from the interactor language to SMV is demonstrated.

interactor userpanel
importing core
attributes
owner: User
vis chosen: [User]
vis buttons: Service → button
vis door-icon: [door] # door state of selected user
action
vis select(User)
axioms
(1) chosen ≠ nil → door-icon=accessibility(chosen)
(2) chosen = nil → door-icon=nil
(3) buttons(s).$>_{action}$=pressed ↔ $>_{action}$=establish(owner,s,chosen)
(4) per(buttons(s).pressed) → chosen≠nil
(5) [select(u)] chosen=u

Fig. 3. User Panel

Having a compiler to SMV means that, given an *appropriate* interactor specification, we can check properties automatically. In the context of model checking, an *appropriate* interactor specification means one that can be expressed as a finite state machine, and ideally with a minimum of states. In order for that to be true of our specification, some adjustments must be made. The first step is to make all types finite. From Figure 1 it can be seen that there are four possible connection types (snapshot, glance, connect, and message), and also four possible door states (open, ajar, closed, and locked). Since door states are also used to set exceptions, all and none are added as possible door states (and all attributes except exceptions are restricted to the original four values). The number of users in the system is arbitrary, three users are used. The types become:

```
types
User = {user1, user2, user3}                        # all users
Service = {snapshot, glance, connect, message}      # available services
Door = {all, open, ajar, closed, locked, none}      # door states
```

Since all other types are defined on top of these three, they become finite by definition. Finally, all structured types have to be rewritten as arrays. Type $\mathbb{P}$ Conn, for example, becomes:

```
PConn = array user1 .. user3 of
            array snapshot .. message of array user1 .. user3 of boolean
```

The same process is applied to all other structured types present in the specification.

We now rewrite the specification to take into consideration these concrete definitions of the types. As an example we show the new version of axiom 5 in interactor core (compare with figure 2):

```
allowed[caller][type][callee] ↔
(exceptions[caller][type][callee]=null →
 default[accessibility[callee]][type])
∧ (exceptions[caller][type][callee]≠null →
 exceptions[caller][type][callee] ≥ accessibility[callee])
```

Inspection of the axiom above, however, reveals an error in the specification: in the presence of an exception, allowed is calculated by seeing if the door level set for the exception is greater (more conservative — look at the enumeration order in the definition of Door) than the accessibility level set by the callee. However, since in the definition of Door we have all as the smallest value, and none as the greatest, then setting the exception level to all/none prohibits/allows all connections! This behaviour is exactly the opposite of what is the reasonable interpretation of both value names. We need to exchange the position of all and none.

Since the order in which the values were enumerated was based on the order they appear in the proposed presentation, this might mean that there is some problem with that aspect of the interface. We will see how to investigate this further in Section 2.2.

In order to verify the proposed design against the criterion set forth above, that criterion needs to be defined as a CTL formula. Looking at the specification of the user panel, it can be seen that the user gets information on the callee's accessibility level through its door state. We will suppose that the user can *remember* which connections are valid for each door state (it could be argued that this information should be encoded in the interface by disabling *illegal* buttons for the given door state, as it will be shown this is not enough). If that is assumed, then the system is predictable if all valid button presses for each door state result in the corresponding connection being established, i.e. all valid button presses *to the user*, are valid button presses *to the system*. This can be expressed as a family of CTL formulae:

$$\forall_{d \in \mathsf{Door}, s \in \mathsf{default}(d)} \cdot \mathsf{AG}(\mathsf{chosen} \neq \mathsf{nil} \wedge \mathsf{door\_icon} = d \wedge \mathsf{buttons}(s).\ >_{\mathtt{action}} = \mathsf{press} \rightarrow (\mathsf{owner}, s, \mathsf{chosen}) \in \mathsf{current})$$

While looking at how the property is expressed might already give some notion of the type of problems that the system would suffer from, we will go on and show how the problem can be detected by model checking the specification.

To make sure the property holds we have to test it for all possible users, and all types of connections that are valid for each door state. If it fails for any given combination, then clearly the property does not hold. A problem now arises. The finite state machine generated by the specification is too *big* for model checking to be practical. In order to reduce the size of the finite state machine generated by our specification, we can do two things: eliminate state variables, and decrease the size of the state variables domain [11]. It should be stressed that this must be done carefully, in order not to affect the meaning of the specification. More specifically, the simplified version of the specification must preserve all the behaviour of the original specification regarding the property that we are checking. Several simplifications were introduced:

- only two types of connection were considered — this is valid since the specification/property does not depend on the number of services available;
- the number of door states was reduced to four (all, open, ajar, none) — note that all and none had to be kept since they are special cases;

– attributes default, owner and callee were *hard-coded* into the specification — this is valid since changes in those values are not being considered (i.e. they are thought of as constants).

With these alterations the specification becomes model checkable. We try the following instance of the property:

$$\mathsf{AG}(\mathsf{door\_icon} = \mathsf{open} \wedge \mathsf{button\_snapshot}.>_{\mathtt{action}}= \mathsf{press} \rightarrow (\mathsf{snapshot}, \mathsf{user1}) \in \mathsf{current})$$

and SMV's reply is (after some editing for readability):

```
-- specification AG (door_icon = 2 & do_snapshot.ac... is false
-- as demonstrated by the following execution sequence
state 1.1:
allowed[snapshot][user1] = true
current[snapshot][user1] = false
door_icon = open

state 1.2:
action = setexcep(snapshot,user1,none)
allowed[snapshot][user1] = false
current[snapshot][user1] = false
door_icon = open

state 1.3:
button_snapshot.action = press
action = establish(snapshot,user1)
current[snapshot][user1] = false
door_icon = open

resources used:
user time: 8.58 s, system time: 0.09 s
BDD nodes allocated: 206873
Bytes allocated: 4521984
BDD nodes representing transition relation: 47519 + 409
```

It can be seen that the property does not hold. What the counter example shows is that the callee might set an exception for the particular connection being tried. This is done in state 1.2, and the connection becomes not allowed. Unfortunately the caller has access only to the callee door state (see visibility annotations in the specification), so the user is unable to predict whether a connection is going to be accepted or not.

The analysis above tells us that the system is not predictable. The user can not tell whether a request for connection will be successful or not. From the user's point of view, this happens because the information that is displayed regarding the callee's receptiveness to connections is inappropriate.

Since exceptions can override the accessibility level, what should be presented to the caller is not the general accessibility level of the callee, but the result of applying

the exceptions regarding the caller to the callee's accessibility level (and, for instance, disabling inappropriate buttons). In fact, it could even be argued that the general accessibility level of the callee should not be displayed at all, so as to avoid callers detecting that they were being in some way *segregated*. Note how this shows that the initial suggestion of disabling buttons according to the door state only would be inappropriate.

The problem is that two mechanisms with different philosophies are being integrated. Determining what is the best compromise solution falls outside the scope of formal/automated verification. What these techniques offer is a way to study the different proposal against specific criteria of quality.

Going back to SMV's answer, and looking at it from the specification side, we can see that the property fails because there is a mismatch between the precondition of the core level action that establishes the connection (action establish), and the preconditions (in the user's head) of the user interface commands that trigger that action (the buttons). It is easy to see that a necessary condition for a user interface to be predictable is that the preconditions at the two levels match. Although this is an indirect test, it still allows us to detect if a system will not be predictable.

State exploration type properties, like the first property checked above, demand that the system be reduced to a finite state machine. It is clear that as our specifications grow in complexity this becomes increasingly hard. Moreover, even if we can express the system as a finite state machine, it can also happen that this machine is too big and model checking is not feasible. Checking for the satisfaction of preconditions, on the other hand, can be done by hand or using a theorem prover. Hence, even if the specification is too complex for model checking, we can still analyse it regarding predictability, verifying if all preconditions at the user interface level match the corresponding system level ones. Similarly we can think of analysing if the result of system level actions matches what the user expects.

2.2 Checking the Presentation

In the previous Section we have seen how we can use model checking to analyse predictability in ECOM. In this Section we look at two aspects of the proposed presentation. The properties that we will be interested in lend them selves more naturally to theorem proving, so we will be using PVS.

At this stage we do not yet have a tool to automatically translate our interactor specifications into the PVS notation. Hence, we had have to perform the translation manually. This is not hard, as each interactor can be expressed as a PVS theory with relative ease.

Cumulative exception setting As we have seen, there was an error in the order in which we first enumerated the door states, and this had influence on how the cumulative nature of exceptions worked. Since we were using the same order which is used at the interface, we have reason to suspect that the presentation used for the exceptions might not be in agreement with the abstract model. More specifically, we want to guarantee that the proposed interface conveys the notion of cumulative exception setting.

In order to verify this we follow the same process as already applied in [8]. We develop a model of the proposed interface presentation, and a function (ρ) which builds a presentation from an abstract specification. We then develop two operators capturing the relevant concepts we want to analyse, one for each model. Since we want to study how the cumulative nature of exceptions is conveyed by the presentation, we will consider how a user determines if a connection is allowed by its current exception setting. We then have to show that using the operator at the abstract level yields the same result as mapping the abstract state into a presentation and using the operator defined in the presentation level (see Figure 4). This type of problem is better solved using theorem

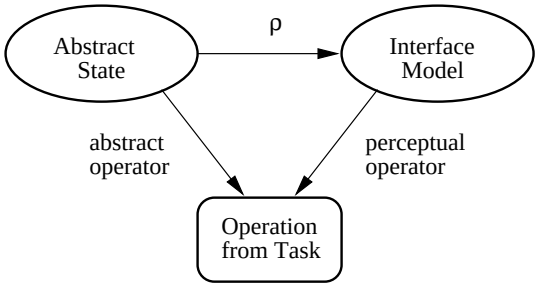

Fig. 4. Verifying presentation issues (adapted from [9])

proving (see [3]), so we choose to use PVS [17], a theorem prover.

As stated above, we will consider the case where a user is setting the exception level for some other given user and specific type of connection service (since those are fixed, we can omit them from now on). The task is to predict whether the selected exception level will allow the connection or not.

Since we are only looking at a single user, we can use a simplified version of the model developed previously. Also, because we will be looking only at how attributes are mapped in the presentation, we don't include actions in this restricted version of the model. At the abstract level we need the accessibility, and exceptions attributes. We can begin to model this in the PVS notation as the theory:

```
restrictedpanel : THEORY
  BEGIN
  Attributes :
    TYPE = [# accessibility : {d : Door | d ≠ none ∧ d ≠ alld}, exceptions : Door#]
  ...
  END restrictedpanel
```

In the presence of an exception, whether or not a request for establishing a connection will succeed can be modeled by the following operator, which we add to the theory above:

```
willsucceed : [Attributes → bool] =
  (λ (a : Attributes) : exceptions(a) ≥ accessibility(a))
```

This definition is easily deduced from axiom 5 of core (remember that we are considering that the user has set an exception).

We now build a model of the actual interface that is being proposed for the system. For the current purpose we are only interested in the level of accessibility and exception setting:

```
rho_restrictedpanel : THEORY
  BEGIN
  Attributes : TYPE = [# accessibility : AccPanel, exceptions : ExcepPanel #]
  ...
  END rho_restrictedpanel
```

where the accessibility panel is represented by an array of four buttons (one for each door state):

$$\text{AccPanel} : \text{TYPE} = \text{ARRAY}[\text{DoorIcon} \rightarrow \text{bool}]$$

and the exceptions panel is represented by six buttons (one for each possible level):

$$\text{ExcepPanel} : \text{TYPE} = \text{ARRAY}[\text{ExcepItem} \rightarrow \text{bool}]$$

In order to model the order in which the buttons are present in the interface, we introduce the operator:

$$\text{isabove} : [\text{ExcepItem}, \text{ExcepItem} \rightarrow \text{bool}]$$

We can now write a function that builds a concrete interface from an abstract state:

$$\begin{array}{l}\rho : [\text{restrictedpanel.Attributes} \rightarrow \text{rho\_restrictedpanel.Attributes}] = \\ \quad (\lambda\,(\text{rp} : \text{restrictedpanel.Attributes}) : \\ \qquad (\#\text{accessibility} := \text{rho\_accessibility}(\text{accessibility}(\text{rp})), \\ \qquad\quad \text{exceptions} := \text{rho\_exceptions}(\text{exceptions}(\text{rp}))\#)\end{array}$$

where rho_accessibility and rho_exceptions perform the translation of each of the door states (accessibility and exceptions setting) to the corresponding array of buttons.

In order for the presentation to convey the notion that exceptions accumulate (i.e. when an exception level is set, all door states up to that one are allowed) the exception setting level must vary progressively along the column of buttons. If that happens, the user will be able to know if a request for a service will succeed based on the relative position of the current accessibility level and current exception setting level. The request is allowed if the exception setting is equal or *bigger* than the current accessibility (for example, the situation illustrated in Figure 1 allows for glance to happen). This is conveyed by the following operator, which compares the exception level with the accessibility setting:

```
rho_willsucceed : [nonempty_Attributes → bool] =
  (λ (a : nonempty_Attributes) :
    (set_access = set_excep ∨ isabove(set_access, set_excep))
      WHERE
      set_access : DoorIcon = identify_access(accessibility(a)),
      set_excep : ExcepItem = identify_excep(exceptions(a)))
```

where identify_access and identify_excep model the cognitive tasks of identifying which button is selected in each of the panels.

If we try to prove that this is an adequate operator to check for permissions:

```
equivalence : THEOREM
  ∀ (rp : restrictedpanel.Attributes) : willsucceed(rp) = rho_willsucceed(ρ(rp))
```

we end up in a situation where the theorem prover asks us to show that:

```
|
|-------------------------------------------
| [1] isabove(maptoexceplist(OpenDoor), Always)
```

That is, the prover tells us that, in order for the theorem to be true, the button for *when Door Open* must be above the button for *Always*. This is clearly not true (look at Figure 1). The problem is that, as with the definition of `Door` in the previous Section, `Never` and `Always` are placed the wrong way around. The order of the buttons in the interface should be:

1. Never
2. (only if) Door Open
3. (up to) Door Ajar
4. (up to) Door Closed
5. (up to) Door Locked
6. Always

This problem was initially reported in [1]. There, PUM analysis was used to model the system from an human-factors perspective. Here we show how in the context of a more software engineering oriented approach, the selection of appropriate abstractions allows us to reach the same conclusion.

Analysing Redundancy We will now look at the second and third criteria set forth initially. More specifically, we want to investigate whether there is any difference between:

- setting the exception level to always or to when door locked;
- setting the exception level to never or to when door open.

This analysis has been prompted by two factors: as a general principle, redundancy should be avoided at the interface; it is the authors impression that there is redundancy in the number of buttons used to set the exception level. In order to corroborate/discharge this suspicion, we will try to prove that there is no difference between the pairs of buttons mentioned above. For the first case we write:

alwaysvslocked : THEOREM
$\forall (a_1, a_2 : \text{Attributes}) :$
$((\text{exceptions}(a_1) = \text{alld} \wedge$
$\text{exceptions}(a_2) = \text{locked} \wedge$
$\text{accessibility}(a_1) = \text{accessibility}(a_2)) \Rightarrow$
$\text{willsucceed}(a_1) = \text{willsucceed}(a_2))$

We are trying to show that given two situations where the accessibility is the same, and the exception is set to always in one case, and locked in the other, then if a connection succeeds in one case it will succeed in the other.

Similarly for the second case we write:

nonevsopen : THEOREM
$\forall (a_1, a_2 : \text{Attributes}) :$
$((\text{exceptions}(a_1) = \text{none} \wedge$
$\text{exceptions}(a_2) = \text{open} \wedge$
$\text{accessibility}(a_1) = \text{accessibility}(a_2)) \Rightarrow$
$\text{willsucceed}(a_1) = \text{willsucceed}(a_2))$

In the first case we are able to prove the theorem. What this points out/proves is that there is no difference between setting the exception level to Always or to Locked. Whence, we do not need both levels in the system.

In the second case the proof fails (i.e. we cannot perform it). This happens because setting the exception level to open still allows some connections while setting it to never allows no connections at all.

In the light of the results above, we can propose another change to the design of exception setting:

1. never
2. only if door open
3. up to door ajar
4. up to door closed
5. up to door locked (always)

3 Discussion

3.1 On the tools

We have seen how we can use automated reasoning techniques to help the formal analysis of interactive systems designs. Two techniques have been used: model checking, and theorem proving. Traditionally theorem proving is considered more difficult to use, however the case study shows that this is not necessarily always the case.

In using these tools to analyse interactive systems we are putting them to uses that were not initially envisaged. This is specially true of model checking. Interactor specifications have proven brittle in terms of the time taken to perform the model checking

step: small changes in the interactor specification can produce huge differences in the time taken to get an answer. This is aggravated by the fact that it is not easy to predict how long the model checking process of a particular specification is going to take. While these problems are inherent to model checking in general, the fact that we are using such concepts as parametrised actions and sets in our specifications (remember that each parameter in an action means that the action will originate a number of actions at the SMV level) seems to make our specifications more susceptible to them. Additionally, bringing the specification down to a model checkable size is a step that must be done with care and in a stepwise manner..

On the other side, the proofs turned out to be easy to perform, as PVS solved most of the situations. In general, the sub-cases that were left to prove were easily solved. The PVS learning curve is not easy though.

In the end, the decision on which is the best tool to apply will always depend on the style of property we are looking at.

3.2 On the analysis

It is not realistic to assume we can analyse all aspects of an interactive system with one single model. Hence, the first step in an analysis process must be to decide which aspects are worth looking into. We have illustrated different possibilities for this step.

The analysis in Section 2.1 was based on a generic design principle: predictability. We analysed whether the commands to establish connection were predictable. To perform this analysis we used SMV, a model checker. We concluded they were not predictable since there was a mismatch between the actual accessibility settings of a user and the information made available to the community. Once a problem, and its causes, are identified, it usually falls out of the scope of the formal approach to decide the best solution. That solution will have to be reached in collaboration with other user centered approaches. What formal methods have to offer is the possibility of, given a set of quality measures, comparing the different proposals.

During the modelling process for the analysis above, our attention was drawn to the importance of the ordering of the different door states. Here we see how the modelling process can, in itself, be useful at raising issues. In Section 2.2 we investigate whether the proposed presentation is coherent with the underlying semantics of the door states. This analysis is done with PVS, a theorem prover.

Finally, the properties analysed in Section 2.2 are the result of a critical look at the proposed system. As a generic principal, redundancy at the interface should be avoided. It was the authors' impression that redundancy existed in the number of buttons available for exception setting. A formal analysis, performed with the theorem prover, enabled us to determine a situation where in fact there was redundancy, and another where that didn't happen.

Several approaches to the analysis of the ECOM system were used in [1]. Comparing our analysis with those, we see that our approach can be used to complement the formal system analysis performed there. We are using automated tools exactly to perform that kind of analysis. Regarding the cognitive user modelling approaches (PUM

and CTA), we can see that we were able to reach similar results. However, our results were obtained while still in the context of a software engineering approach. PUM and CTA demand human-factors expertize. We see our approach as being complementary to those in the sense that they can be used to define a set of quality measures which can then be rigorously analysed using formal methods. Furthermore, they can help in interpreting the results of such analytic process.

A fourth analysis (PAC-AMODEUS) of the case study is reported. The PAC-AMODEUS analysis is architecture oriented, hence not so much interested in usability issues, but more in implementation issues.

4 Conclusion

We looked at the use of automated reasoning tools in the formal analysis of an interactive system design. While mastering the use of the tools will inevitably take some effort, they enable us to be more confident of the results of the analysis.

Unlike other approaches to the use of automated reasoning in HCI, we do not focus on the tool, instead we focus our approach on integrating verification with the development process (cf. [5]). Once an interesting aspect of the system has been identified, we investigate which type of tool will fit best to its analysis. In this way, we are not tied to a particular type of model, and we have greater freedom in terms of what we can model and reason about.

Our long term objective is to develop a framework enabling us to integrate formal software engineering (automated reasoning in particular) with the other disciplines involved in HCI.

Acknowledgements

José Campos is supported by Fundação para a Ciência e a Tecnologia (FCT, Portugal) under grant PRAXIS XXI/BD/9562/96.

References

1. Victoria Bellotti, Ann Blandford, David Duke, Allan MacLean, Jon May, and Laurence Nigay. Interpersonal access control in computer-mediated communications: A systematic analysis of the design space. *Human-Computer Interaction*, 11:357–432, 1996.
2. Peter Bumbulis. *Combining Formal Techniques and Prototyping in User Interface Construction and Verification*. PhD thesis, University of Waterloo, 1996.
3. Jos´e C. Campos. *Automated Reasoning and Interactive Systems Development*. DPhil thesis, Department of Computer Science, University of York, 1999. in preparation.
4. Jos´e C. Campos and Michael D. Harrison. Detecting interface mode complexity with interactor specifications. submitted, 1998.
5. Jos´e C. Campos and Michael D. Harrison. The role of verification in interactive systems design. In Markopoulos and Johnson [15], pages 155–170.
6. Bruno d'Ausbourg. Using model checking for the automatic validation of user interfaces systems. In Markopoulos and Johnson [15], pages 242–260.

7. Alan Dix, Janet Finlay, Gregory Abowd, and Russell Beale. *Human-Computer Interaction*. Prentice-Hall, 1993.
8. G. Doherty, J. C. Campos, and M. D. Harrison. Representational reasoning and verification. In J. I. Siddiqi, editor, *Proceedings of the BCS-FACS Workshop: Formal Aspects of the Human Computer Interaction*, pages 193–212. SHU Press, 1998. ISBN 0 86339 7948.
9. Gavin Doherty and Michael D. Harrison. A representational approach to the specification of presentations. In Harrison and Torres [14], pages 273–290.
10. David J. Duke and Michael D. Harrison. Abstract interaction objects. *Computer Graphics Forum*, 12(3):25–36, 1993.
11. Matthew B. Dwyer, Vicki Carr, and Laura Hines. Model checking graphical user interfaces using abstractions. In Mehdi Jazayeri and Helmut Schauer, editors, *Software Engineering — ESEC/FSE '97*, number 1301 in Lecture Notes in Computer Science, pages 244–261. Springer, 1997.
12. G. Faconti and F. Paternò. An approach to the formal specification of the components of an interaction. In C. Vandoni and D. Duce, editors, *Eurographics '90*, pages 481–494. North-Holland, 1990.
13. Bob Fields, Nick Merriam, and Andy Dearden. DMVIS: Design, modelling and validation of interactive systems. In Harrison and Torres [14], pages 29–44.
14. M. D. Harrison and J. C. Torres, editors. *Design, Specification and Verification of Interactive Systems '97*, Springer Computer Science. Springer-Verlag/Vien, June 1997.
15. P. Markopoulos and P. Johnson, editors. *Design, Specification and Verification of Interactive Systems '98*, Springer Computer Science. Springer-Verlag/Vien, 1998.
16. K. L. McMillan. *The SMV system*. Carnegie-Mellon University, draft edition, February 1992.
17. S. Owre, N. Shankar, and J. M. Rushby. *User Guide for the PVS Specification and Verification System*. Computer Science Laboratory, SRI Internatinal, Menlo Park CA 94025, USA, (beta release) edition, March 1993.
18. Fabio D. Paternò. *A Method for Formal Specification and Verification of Interactive Systems*. PhD thesis, Department of Computer Science, University of York, 1995.
19. Mark Ryan, Jos´e Fiadeiro, and Tom Maibaum. 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.

A ECOM model (model ckeckable)

```
define
  none   = 1
  open   = 2
  locked = 5
  all    = 6
  snapshot = 1
  glance   = 2
  user1 = 1
  user2 = 2
  null = 0

types
  Door = {none, open, locked, all}
  OptDoor = {none, open, locked, all, null}
  Service = {snapshot, glance}
```

```
  PConn = array snapshot..glance of \
                array user1..user2 of boolean
  ConnDoor = array snapshot..glance of array user1..user2 of \
                                {none, open, locked, all, null}
  User = {user1, user2}
  UserDoor = array user1..user2 of {open, locked}

interactor core
# core modelled from the view point of a caller
attributes
  allowed: PConn
  accessibility: UserDoor
  current: PConn
  exceptions: ConnDoor
actions
  establish(Service,User) close(Service,User) \
  setexcep(Service,User,OptDoor) setacc(User,Door)
axioms
# (1)
  allowed[s][u] -> [establish(s,u)] \
          current[s][u]' = 1 & current<s><u>' = current<s><u> \
        & unchanged(accessibility, exceptions)\
  !allowed[s][u] -> [establish(s,u)] \
          unchanged(current, accessibility, exceptions)
# (2)
  per(close(s,u)) -> current[s][u]
  [close(s,u)] \
           current[s][u]'=0 & current<s><u>'=current<s><u> \
         & unchanged(exceptions, accessibility)
# (3)
  [setexcep(s,u,d)] \
     exceptions[s][u]'=d & exceptions<s><u>'=exceptions<s><u> \
   & unchanged(current,accessibility)
# (4)
  [setacc(u,d)] \
     accessibility[u]'=d & accessibility<u>'=accessibility<u> \
   & unchanged(current,exceptions)
# (5)
  allowed[snapshot][user1] <-> \
      (exceptions[snapshot][user1]=null -> \
                 accessibility[user1] in {open, locked}) \
    & (exceptions[snapshot][user1]!=null -> \
         exceptions[snapshot][user1] >= accessibility[user1])
  allowed[snapshot][user2] <-> \
      (exceptions[snapshot][user2]=null -> \
                 accessibility[user2] in {open, locked}) \
    & (exceptions[snapshot][user2]!=null -> \
         exceptions[snapshot][user2] >= accessibility[user2])
  allowed[glance][user1] <-> \
      (exceptions[glance][user1]=null -> \
```

```
                accessibility[user1] in {open}) \
    & (exceptions[glance][user1]!=null -> \
              exceptions[glance][user1] >= accessibility[user1])
  allowed[glance][user2] <-> \
      (exceptions[glance][user2]=null -> \
                accessibility[user2] in {open}) \
    & (exceptions[glance][user2]!=null -> \
              exceptions[glance][user2] >= accessibility[user2])
# initial state
  []   exceptions[snapshot][user1] = open \
     & exceptions[glance][user1] = open \
     & exceptions[snapshot][user2] = open \
     & exceptions[glance][user2] = open \
     & !current[snapshot][user1] & !current[glance][user1] \
     & !current[snapshot][user2] & !current[glance][user2] \
     & accessibility[user1] = open & accessibility[user2] = open

interactor button
attributes
  enabled: boolean
actions
  press
axioms
  per(press) -> enabled
  [press] enabled’ = enabled

interactor main
importing
  core
includes
  button via do_snapshot
  button via do_glance
attributes
  door_icon: Door
axioms
# (1) hard-code user1 as callee
  door_icon=accessibility[user1]
# (3) we have to
  do_snapshot.action=press <-> action=establish_1_1
  do_glance.action=press <-> action=establish_2_1
# initial state
  [] do_snapshot.enabled
test
  AG(door_icon=open & do_snapshot.action=press -> \
                              current[snapshot][user1])
```

B PVS model

ecom : THEORY

```
    BEGIN

    IMPORTING restrictedpanel, rho_restrictedpanel

    rho_accessibility : [{d : Door | d ≠ none ∧ d ≠ alld} → AccPanel] =
      (λ (d : {d : Door | d ≠ none ∧ d ≠ alld}) :
        COND
        d = open → (λ (di : DoorIcon) : di = OpenDoor),
        d = ajar → (λ (di : DoorIcon) : di = AjarDoor),
        d = closed → (λ (di : DoorIcon) : di = ClosedDoor),
        d = locked → (λ (di : DoorIcon) : di = LockedDoor)
        ENDCOND)

    rho_exceptions : [d : Door → ExcepPanel] =
      (λ (d : Door) :
        COND
        d = none → (λ (ei : ExcepItem) : ei = Never),
        d = open → (λ (ei : ExcepItem) : ei = whenOpen),
        d = ajar → (λ (ei : ExcepItem) : ei = whenAjar),
        d = closed → (λ (ei : ExcepItem) : ei = whenClosed),
        d = locked → (λ (ei : ExcepItem) : ei = whenLocked),
        d = alld → (λ (ei : ExcepItem) : ei = Always)
        ENDCOND)

    ρ : [restrictedpanel.Attributes → rho_restrictedpanel.Attributes] =
      (λ (rp : restrictedpanel.Attributes) :
        (#accessibility := rho_accessibility(accessibility(rp)),
          exceptions := rho_exceptions(exceptions(rp))#))

    equivalence : THEOREM
      ∀ (rp : restrictedpanel.Attributes) : willsucceed(rp) = rho_willsucceed(ρ(rp))

    END ecom


restrictedpanel : THEORY
    BEGIN

    Door : TYPE = {none, open, ajar, closed, locked, alld}

    door_order : [Door → int]
    door_order_none : AXIOM (∀ (d : Door) : d = none ⇒ door_order(d) = 1)
    door_order_open : AXIOM (∀ (d : Door) : d = open ⇒ door_order(d) = 2)
    door_order_ajar : AXIOM (∀ (d : Door) : d = ajar ⇒ door_order(d) = 3)
    door_order_closed : AXIOM (∀ (d : Door) : d = closed ⇒ door_order(d) = 4)
    door_order_locked : AXIOM (∀ (d : Door) : d = locked ⇒ door_order(d) = 5)
    door_order_all : AXIOM (∀ (d : Door) : d = alld ⇒ door_order(d) = 6)
    CONVERSION door_order

    Attributes :
```

TYPE = [# accessibility : {d : Door | $d \neq$ none $\wedge$ $d \neq$ alld}, exceptions : Door#]

willsucceed : [Attributes $\rightarrow$ bool] =
(λ (a : Attributes) : exceptions(a) $\geq$ accessibility(a))

alwaysvslocked : THEOREM
∀ (a_1, a_2 : Attributes) :
((exceptions(a_1) = alld∧
exceptions(a_2) = locked∧
accessibility(a_1) = accessibility(a_2)) ⇒
willsucceed(a_1) = willsucceed(a_2))

nonevsopen : THEOREM
∀ (a_1, a_2 : Attributes) :
((exceptions(a_1) = none∧
exceptions(a_2) = open∧
accessibility(a_1) = accessibility(a_2)) ⇒
willsucceed(a_1) = willsucceed(a_2))

END restrictedpanel

rho_restrictedpanel : THEORY
BEGIN

DoorIcon : TYPE = {OpenDoor, AjarDoor, ClosedDoor, LockedDoor}
AccPanel : TYPE = ARRAY[DoorIcon $\rightarrow$ bool]

AccPanel_is_RadioBox : AXIOM
∀ (ap : AccPanel) : ¬∃ (di1, di2 : DoorIcon) : di1 ≠ di1 ∧ ap(di1) ∧ ap(di2)

ExcepItem : TYPE = {Always, whenOpen, whenAjar, whenClosed, whenLocked, Never}
ExcepPanel : TYPE = ARRAY[ExcepItem $\rightarrow$ bool]

ExcepPanel_is_RadioBox : AXIOM
∀ (ep : ExcepPanel) : ¬∃ (ei1, ei2 : ExcepItem) : ei1 ≠ ei1 ∧ ep(ei1) ∧ ep(ei2)

isabove : [ExcepItem, ExcepItem $\rightarrow$ bool]
isabove_transitive : AXIOM
∀ (ei1, ei2, ei3 : ExcepItem) :
(isabove(ei1, ei2) ∧ isabove(ei2, ei3)) ⇒ isabove(ei1, ei3)
isabove_antisymmetric : AXIOM
∀ (ei1, ei2 : ExcepItem) : (isabove(ei1, ei2) ⇒ ¬isabove(ei2, ei1))
isabove_Never_whenOpen : AXIOM
(∀ (e_1, e_2 : ExcepItem) :
(e_1 = Never ∧ e_2 = whenOpen) ⇒ isabove(e_1, e_2))
isabove_whenOpen_whenAjar : AXIOM
(∀ (e_1, e_2 : ExcepItem) :
(e_1 = whenOpen ∧ e_2 = whenAjar) ⇒ isabove(e_1, e_2))
isabove_whenAjar_whenClosed : AXIOM

$(\forall\,(e_1, e_2 :\ \text{ExcepItem}) :$
$(e_1 = \text{whenAjar} \wedge e_2 = \text{whenClosed}) \Rightarrow \text{isabove}(e_1, e_2))$
isabove_whenClosed_whenLocked : AXIOM
$(\forall\,(e_1, e_2 :\ \text{ExcepItem}) :$
$(e_1 = \text{whenClosed} \wedge e_2 = \text{whenLocked}) \Rightarrow \text{isabove}(e_1, e_2))$
isabove_whenLocked_Always : AXIOM
$(\forall\,(e_1, e_2 :\ \text{ExcepItem}) :$
$(e_1 = \text{whenLocked} \wedge e_2 = \text{Always}) \Rightarrow \text{isabove}(e_1, e_2))$

Attributes : TYPE $= [\#\ \text{accessibility} :\ \text{AccPanel}, \text{exceptions} :\ \text{ExcepPanel}\#]$

nonempty_AccPanel : TYPE $= \{\text{ap} :\ \text{AccPanel} \mid \exists\,(\text{di} :\ \text{DoorIcon}) :\ \text{ap}(\text{di})\}$

nonempty_ExcepPanel : TYPE $= \{\text{ep} :\ \text{ExcepPanel} \mid \exists\,(\text{ei} :\ \text{ExcepItem}) :\ \text{ep}(\text{ei})\}$

nonempty_Attributes :
TYPE $= \{a :\ \text{Attributes} \mid$
$(\exists\,(\text{di} :\ \text{DoorIcon}) :\ \text{accessibility}(a)(\text{di})) \wedge$
$(\exists\,(\text{ei} :\ \text{ExcepItem}) :\ \text{exceptions}(a)(\text{ei}))\}$

maptoexceplist : [DoorIcon $\rightarrow$ ExcepItem]
maptoexceplist_OpenDoor : AXIOM maptoexceplist(OpenDoor) = whenOpen
maptoexceplist_AjarDoor : AXIOM maptoexceplist(AjarDoor) = whenAjar
maptoexceplist_ClosedDoor : AXIOM maptoexceplist(ClosedDoor) = whenClosed
maptoexceplist_LockedDoor : AXIOM maptoexceplist(LockedDoor) = whenLocked
CONVERSION maptoexceplist

identify_access : [nonempty_AccPanel $\rightarrow$ DoorIcon] =
$(\lambda\,(\text{ap} :\ \text{nonempty\_AccPanel}) :$
COND
ap(OpenDoor) $\rightarrow$ OpenDoor,
ap(AjarDoor) $\rightarrow$ AjarDoor,
ap(ClosedDoor) $\rightarrow$ ClosedDoor,
ap(LockedDoor) $\rightarrow$ LockedDoor
ENDCOND)

identify_excep : [nonempty_ExcepPanel $\rightarrow$ ExcepItem] =
$(\lambda\,(\text{ep} :\ \text{nonempty\_ExcepPanel}) :$
COND
ep(Always) $\rightarrow$ Always,
ep(whenOpen) $\rightarrow$ whenOpen,
ep(whenAjar) $\rightarrow$ whenAjar,
ep(whenClosed) $\rightarrow$ whenClosed,
ep(whenLocked) $\rightarrow$ whenLocked,
ep(Never) $\rightarrow$ Never
ENDCOND)

rho_willsucceed : [nonempty_Attributes $\rightarrow$ bool] =
$(\lambda\,(a :\ \text{nonempty\_Attributes}) :$
$(\text{set\_access} = \text{set\_excep} \vee \text{isabove}(\text{set\_access}, \text{set\_excep}))$

WHERE
set_access : DoorIcon = identify_access(accessibility(a)),
set_excep : ExcepItem = identify_excep(exceptions(a)))

END rho_restrictedpanel