Reasoning about Order Errors in Interaction

Paul Curzon and Ann Blandford

School of Computing Science, Middlesex University, London, UK
{p.curzon,a.blandford}@mdx.ac.uk

Abstract. Reliability of an interactive system depends on users as well as the device implementation. User errors can result in catastrophic system failure. However, work from the field of cognitive science shows that systems can be designed so as to completely eliminate whole classes of user errors. This means that user errors should also fall within the remit of verification methods. In this paper we demonstrate how the HOL theorem prover [7] can be used to detect and prove the absence of the family of errors known as order errors. This is done by taking account of the goals and knowledge of users. We provide an explicit generic user model which embodies theory from the cognitive sciences about the way people are known to act. The user model describes action based on user communication goals. These are goals that a user adopts based on their knowledge of the task they must perform to achieve their goals. We use a simple example of a vending machine to demonstrate the approach. We prove that a user does achieve their goal for a particular design of machine. In doing so we demonstrate that communication goal based errors cannot occur.

1 Introduction

People commonly make mistakes when interacting with computer-based devices. Whilst some errors cannot always be prevented, such as those caused by users behaving randomly and maliciously, there are whole classes of error that have distinct cognitive causes and are predictable [13]. Furthermore, changes to the design of systems can eliminate such errors [9, 3]. Formal verification aims to either detect system errors or show their absence. If user errors can be eliminated using appropriate design then their detection ought to be within the remit of formal verification methodologies. However, formal verification is generally done in a machine-centered way. A consequence is that avoidable user errors are not detected or corrected as part of the verification process.

In this paper, we describe a verification methodology for detecting user errors. Our approach is to formally model *rational* users as part of the system being verified. We focus here on errors resulting from a mismatch between the device design and the order a user expects to supply information or objects. This extends earlier work concerning a different class of errors known as post-completion errors [5]. Our verification approach is capable of detecting both classes of error simultaneously. The verification described has been fully machine-checked using interactive proof with the HOL theorem prover [7].

We define a generic user model that can be instantiated for different machines. This user model describes *rational* user behaviour based on established results from cognitive science [11]. The verification approach therefore detects *rational* user errors. This differs from similar approaches in which the environment of the machine is specified to provide the input required (treating users as logical as opposed to rational agents). With such an approach user errors are treated as never occurring. Our approach is also different from assuming that the environment could perform any action at any time (users as "monkeys at keyboards"). That would amount to saying that whatever the user's goal and whatever actions they perform, they will eventually achieve the goal. This is not appropriate for interactive systems as the functionality of such a system would need to be trivial for it to be considered "correct". Instead, our approach recognises that users are important but do not act randomly. The user is described in terms of the things they wish to achieve; the actions they may perform in order to achieve those goals and in terms of the device-independent knowledge they have about the task. We are interested in eliminating errors from systems that occur when users act in this way as such errors are liable to be persistent.

2 Formal User Modelling

There are, broadly speaking, two main approaches to formal reasoning about the usability of systems. One approach is to focus on formal specification of the user interface; Campos and Harrison [4] review several techniques that take this approach. However, such techniques do not support reasoning about errors. The alternative, which we take in this work, is based on formal user modelling. This involves generating a formal specification of the user in conjunction with one of the computer system, in order to support reasoning about their conjoint behaviour. It should be noted that a formal specification of the user is a description of the way the user is, rather than one of the way the user should be, since users cannot be designed in the way that computer systems can [1]. Examples of formal user modelling include the work of Duke *et al* [6], Butterworth *et al* [2], Moher and Dirda [10] and Paterno' and Mezzanotte [12]. Each of these approaches takes a distinctive focus. Duke *et al* [6] use a mathematical notation to express constraints on the channels and resources within an interactive system; this makes their 'syndetic modelling' technique particularly well suited to reasoning about multi-modal interaction (such as that combining the use of speech and gesture). Butterworth *et al* [2] use Lamport's [8] TLA to reason about behaviour traces and reachability conditions within an interaction; this approach describes behaviour at an abstract level that does not support re-use of the user model from one computer system to another, so while it can support reasoning about errors, each model has to be individually hand-crafted. Moher and Dirda [10] use Petri net modelling to reason about users' mental models and their changing expectations over the course of an interaction; this approach supports reasoning about learning to use a new computer system – which, in turn may be an important source of errors, but focuses on changes in

user belief states rather than proof of desirable properties. Finally, Paterno' and Mezzanotte [12] use LOTOS and ACTL to specify intended user behaviours and hence reason about interactive behaviour; their approach corresponds closely to that which is done in state space exploration verification, but because their user model describes how the user is intended to behave, rather than how users might actually behave, it does not support reasoning about errors.

3 Classes of User Error

A common form of error made by humans in a wide variety of situations is the *Post-completion Error* [3]. Examples include taking the cash but leaving a bank card in an Automatic Teller Machine and leaving the original on the platen and walking away with the copies when using a photocopier. Most ATM machines have been redesigned to force users to remove their cards before cash is delivered to avoid this problem, but the phenomenon persists in many other environments. There are of course other situations where a user does not complete all the sub-tasks associated with a main goal. For example, if a fire alarm went off whilst a person was using a photocopier, they might not take their original. However, such an error would not be a post-completion error in our sense as it would have a different underlying cause. A design that eliminated post-completion errors would not necessarily guarantee the user would not make the same surface level "mistake" for other reasons.

Post-completion errors are interesting because they are not predictable (i.e. they do not occur in every interaction) but they are persistent. They are not related to missing knowledge so cannot be eliminated by increased user training. They can, however, be eliminated with careful system design. Curzon and Blandford [5] illustrate the use of HOL to reason about such errors by considering alternative device designs. Here we develop that approach by extending the generic user model to identify a new class of errors with a distinctive cognitive cause. In particular, we look at errors that occur when there is a mismatch between the design of a device and the knowledge that a user has about the task (independent of the particular device used to complete that task). A user will often know of specific information that must be communicated to any such device for the task to be completed. They may not know precisely how or when the information must be imparted to a particular machine. They thus maintain a mental list of *communication goals*: information that they must communicate at some point. If the order that the information must be imparted to the machine is not known, or the user's mental model of the task suggests a different order then order errors can result. The user attempts to fulfill their goals in an order different to that required by the machine.

Order errors can also arise due not to information that must be communicated, but to objects that must be supplied: an ATM card, coins, etc. For example, with a vending machine, the user will know they must make a selection of chocolate and that they must supply money, but for a given machine they will not necessarily know the order. If they know exactly what they want but not

the price, they may be inclined to press the selection first (some machines would display the price at this point). Alternatively, they may have the coin in their hand and so insert it first before working out exactly which buttons to press to make their selection.

Each of the above classes of errors have distinct cognitive causes. We provide a verification approach that detects such errors in a structured way. Whilst we cannot eliminate all user errors, we *can* eliminate whole classes of error that have such specific cognitive causes.

4 Proving Usability

A proof of usability, in the sense that particular classes of errors cannot occur, involves proving a theorem of the form

```
⊢ ∀(ustate: ustate_type) (mstate: mstate_type).
  MACHINE_USER ustate mstate ∧ MACHINE_SPEC s mstate ⊃
      MACHINE_USABLE ustate mstate
```

`MACHINE_SPEC` is a traditional machine specification: a relation over an internal state `s` and inputs and outputs `mstate`. The latter represents the interface between the device and its users. States and signals are represented by *history functions*: functions from time to the value at that time. `MACHINE_USER` is also a specification of a component of the system: the user of the device. It describes the actions a *rational* user might take based on their knowledge and goals. It is a relation on an internal user state `ustate` and the inputs and outputs of the device. We will look at in more detail in the next section. The conjunction of these two relations provides a specification of the system as a whole: both device and user. The conclusion we prove about this combined device is not phrased in terms of what the device can do, or explicit properties of it. Instead it is a specification of whether the user achieves their goal in interacting with the device.

Note that the above usability theorem is of the basic form

```
⊢ implementation ⊃ specification
```

It can thus easily be combined with a traditional correctness theorem that an implementation of the machine meets the given specification [5].

In one sense the user model fills a similar role to an environment machine in traditional model-checking based verification. It provides inputs to the device being verified. The difference is not in the fact that such an environment is provided but in the *kind* of environment provided. Rather than providing values based on what the machine specification requires, or on other devices connected to the device, it is modelling the way people behave based on results from cognitive science. The user of course may not be providing all the inputs to the device. Thus unlike with an environment machine, the combined user-device system is not necessarily closed. We are treating the user as part of the system under verification, rather than just a test rig to verify the system. The kind of errors we

are looking for are those that result from the user component of the system, but which can be eliminated by modifying the device component of that system.

5 A Generic User Model

We could adopt the approach of providing a separate user model for each distinct device that we wish to verify. However, this approach could lead us back into a machine-centered specification approach, specifying users that do exactly what the specific device requires of them. Moreover, we wish to detect classes of user error that are widespread and not just confined to specific devices. It therefore makes sense to provide once-and-for-all a generic user model that incorporates cognitive science theory about the way people behave in general. Such a generic model can then be targeted to specific machines, simply by providing details about the machine state, the user's knowledge of the task and their goals. Higher-order logic provides an elegant framework in which to specify such a generic model. It allows functions and relations providing details of a specific interaction to be an argument to the generic user model. For example to support reasoning about post-completion errors the user model contains general machinery regarding termination conditions. This is defined in terms of a variable representing an *interaction invariant*: a relation indicating the part of the state that should be restored for the task to be considered completed. The user model takes a specific instance of such an invariant as an argument.

The generic user model is given as a relation `USER` over the user and machine states as described above. In addition however, it takes a series of other arguments representing the details of the specific machine. To instantiate the user model for a given machine, we must provide:

- concrete types for the state of the machine and of the user,
- a list of actions a user might take (inserting coins, pushing buttons, etc),
- a history function to record the communication goals of users of the device at each instant in time,
- a list giving the user's initial communication goals,
- a list pairing device outputs with user inputs, indicating relationships where the output is intended to make the user react by taking the action resulting in the input (for example, a light might be located next to a button, with the light being on indicating the button should be pressed),
- history functions recording the possessions of the user and how they change over time as the interaction progresses,
- a history function recording when the user terminates the interaction (by leaving the device) together with that signal's position in the list of possible actions,
- the goal users of the device are trying to achieve, and
- a history function describing an interaction invariant that should hold both at the start and end of the interaction.

We will discuss each of these in more detail below as we describe the definition of the user model.

The core of the user model is a series of temporally guarded statements about possible actions a rational user might take. For example, one disjunct is associated with each of the paired lights and actions, reflecting the fact that a rational user could react to a light coming on by pressing the associated button. This is specified by:

```
⊢ LIGHT user_actions light action ppos (mstate:'m) t =
      (light mstate t = T) ∧
      NEXT user_actions (action mstate) ppos t
```

This states that if the light is on at a time `t` then the *next* action performed by the user from the list of possible actions `user_actions` will be the one paired with the light (`action`). Since this is just one clause of a list of disjuncts, it is not guaranteed that the action will be taken. A recursive definition `LIGHTS` forms a disjunct of all the pairs in the given list of lights and actions. Note that mstate (similarly ustate) has a polymorphic type in this and the other definitions of this section representing the fact that we are defining a generic user model that can apply to machines and users with different states.

The relation `NEXT` specifies the next action to occur. To define it we first define relations `LSTABLE` and `LF`. The former is used to specify that the signals do not change in some interval. The latter then states that at the end of that interval all but one of the signals remains false.

More formally, `LSTABLE` is a temporal operator that states that all the history functions in the given list have a value `v` between the start and end time.

```
⊢ (LSTABLE [] t1 t2 v = T) ∧
  (LSTABLE (CONS a l) t1 t2 v =
          (∀t. t1 <= t ∧ t < t2 ⊃ (a t = v)) ∧
          (LSTABLE l t1 t2 v))
```

`LF` states that all but one of the actions in the list (that indicated by position `ppos`) are false at a given time. This is defined recursively on the action list.

```
⊢ (LF n [] P ppos t = T) ∧
  (LF n (CONS a l) P ppos t =
      (((n = ppos) ∨ ~(a t)) ∧ (* miss the numbered signal *)
      LF (n+1) l P ppos t))
```

Note that we can not simply use a list MEMBER function here as it would check whether the *values* in the list were equal to one being checked. We wish to identify a specific action, not the value of an action. In the absence of a syntax for user actions, we use the position in the list to identify the action.

`NEXT` uses the above definitions to specify that there is a time later than that given when the action identified by the position occurs (its history function is true), the other actions do not occur (their history functions are false), and for which all the actions do not occur in all the intervening time instances.

```
⊢ NEXT al P ppos t1 =
    ∃t2. t1 <= t2 ∧ (LSTABLE al t1 t2 F) ∧ (LF 0 al P ppos t2) ∧ (P t2)
```

If the temporally guarded statements that make up the user model were based only on the pairs of lights and actions as defined above, we would be specifying a *reactive* user who did exactly what was required. However, other clauses are included to reflect *rational* behaviour based on user goals and knowledge. The first such disjunct describes the fact that a rational user may terminate the interaction on achieving their goal. If this action is taken, before the user's interaction invariant is restored, a post-completion error is made.

```
⊢ COMPLETION user_actions finished finishedpos goalachieved ustate t =
    (goalachieved ustate t = T) ∧
    NEXT user_actions (finished ustate) finishedpos t
```

In this paper we are primarily concerned with errors that result from devices not taking communication goals of users into account. For more detail of verification of designs with respect to post-completion errors see [5].

As discussed earlier, a user of a device generally enters into an interaction with some knowledge about the task. Specifically they are likely to know of some of the information that must be communicated to the device, because they know the task cannot be completed, whatever the device design, unless it receives this information. They will not necessarily know the order the information must be communicated, however.

We model this using a list of actions, corresponding to the communication goals. We first extract the communication goal list from the user state for the time of interest. This allows COMMGOALS to be defined recursively on that argument.

```
⊢ COMMGOALER user_actions actions goal ustate mstate t =
    COMMGOALS user_actions (actions ustate t) goal ustate mstate t
```

This gives a list of communication goals with their position in the list of all possible actions the user could perform. We recurse on this list to produce a list of action disjuncts based on the communication goals.

```
⊢ (COMMGOALS user_actions [] goal ustate mstate t = F) ∧
  (COMMGOALS user_actions (CONS a actions) goal ustate mstate t =
      ((COMMGOALS user_actions actions goal ustate mstate t) ∨
       (COMMGOAL user_actions (FST a) (SND a) goal ustate mstate t)))
```

COMMGOAL describes a temporally guarded action similar to LIGHT and COMPLETION given earlier. A separate relation is defined for this for consistency throughout the user model: each guarded action is given by a similar definition. Provided the user's main goal has not yet been achieved, the next action they will take if this disjunct is activated (i.e. true) is the given communication goal.

```
⊢ COMMGOAL user_actions action n goal ustate mstate t =
    ~(goal ustate t) ∧
    NEXT user_actions (action mstate) n t
```

Since all the communication goals are disjuncts and all have the same guard, no ordering of them is prescribed by these definitions. The user may attempt to complete them in any order. Once a communication goal related action has been completed, it will cease to be a communication goal. We examine how this is specified below.

Each of the actions that a rational user might make when confronted with the machine are combined in a single definition GENERAL_USER_CHOICE. It contains a final default disjunct, ABORTION. It asserts that if none of the guards of the other disjuncts hold (and so no rational action is available) then the user will terminate the interaction without having achieved their goal.

```
⊢ GENERAL_USER_CHOICE user_actions commgoals lights_actions
            finished finishedpos goalachieved mstate ustate t =
    COMMGOALER user_actions commgoals goalachieved ustate mstate t ∨
    LIGHTS user_actions lights_actions 0 mstate t ∨
    COMPLETION user_actions finished finishedpos goalachieved ustate t ∨
    ABORTION user_actions finished finishedpos goalachieved commgoals
            lights_actions ustate mstate t
```

This relation describes the series of options that a user has open to them on any given cycle. There are other conditions that must apply at every instance in time, however. For example, we assume it is always the case that if the user terminates the interaction then they cannot then continue with it.

```
∀t. finished ustate t ⊃ finished ustate (t+1)
```

We similarly assume various rules about the possessions of a user. For example, we assume it is always the case that if a user gives up a possession then they have one less of that possession. These rules are encapsulated into a relation POSSESSIONS. We omit the details of this relation here.

We also assert universal properties of the communication goal list. It is not a constant over time. As the user performs the actions associated with a communication goal, that goal is discharged and so is removed from the user's internal list of things to do: it ceases to be a communication goal. This behaviour is modelled by asserting that if an action that appears on the communication goal list occurs at a time t, then that action will be removed from the communication goal list on the subsequent cycle.

```
⊢ (FILTER [] mstate t = []) ∧
  (FILTER (CONS a actions) mstate t =
    if (FST a) mstate t then (FILTER actions mstate t)
                         else (CONS a (FILTER actions mstate t)))

⊢ FILTER_HLIST mstate hlist = ∀t. hlist (t+1) = FILTER (hlist t) mstate t

⊢ FILTER_USER_HLIST ustate mstate hlist = FILTER_HLIST mstate (hlist ustate)
```

The separate relations describing universal properties are cojoined together into a single relation GENERAL_USER_UNIVERSAL.

```
⊢ GENERAL_USER_UNIVERSAL commgoals possessions finished ustate mstate =
    (∀t. finished ustate t ⊃ finished ustate (t+1)) ∧
    (POSSESSIONS possessions ustate mstate) ∧
    (FILTER_USER_HLIST ustate mstate commgoals)
```

We need two further elements to our generic user model, however. We must assert that at the start of the interaction, the user's communication goals are in fact those supplied as the initial list.

```
⊢ USER_INIT cgoals init_cgoals ustate = (cgoals ustate 0 = init_cgoals)
```

Finally we must describe the situation where the user terminates the interaction normally. We have considered the situation where a user completes their goal and leaves. However, we argued that this may lead to post-completion errors. Normal, non-erroneous termination involves leaving not just when the goal is completed, but also when any necessary house-keeping tasks have been completed. A non-device specific way of describing this is by using the notion of an interaction invariant that the user wishes to maintain. The invariant may be perturbed in the course of the interaction, but must be reinstated by the time the interaction is terminated.

If the goal is achieved and the interaction invariant satisfied, then we assume that the rational user will always terminate the interaction as the next action. If either condition is not fulfilled, the user will take some action from the set of options. This is combined with the initialisation and universal relations to give the complete generic user model.

```
⊢ USER user_actions commgoals init_commgoals lights_actions possessions
          finished finishedpos goalachieved invariant ustate mstate =
  (USER_INIT commgoals init_commgoals ustate) ∧
  (GENERAL_USER_UNIVERSAL commgoals possessions finished ustate mstate) ∧
  (∀t.
    if  ((invariant ustate t = T) ∧ (goalachieved ustate t = T))
    then  NEXT user_actions (finished ustate) finishedpos t
    else  GENERAL_USER_CHOICE user_actions commgoals lights_actions
                 finished finishedpos goalachieved mstate ustate t)
```

This user model, instantiated with the details of a specific machine, specifies aspects of a general rational user of that machine. Because all the options are modelled as guarded disjuncts, the model does not specify that users always make mistakes, just that they are capable of making mistakes of specific kinds. To verify that the modelled user always achieves their goal, the device specification must be such that the opportunities for such errors are not present. For example, if a chocolate machine design always gives out change before chocolate, the guard on the `COMPLETION` disjunct will only be activated when the interaction invariant has already been restored. In this way we have provided a facility which can be used to verify that whole classes of errors cannot occur with a given design.

6 Case Study: A Chocolate Machine

To demonstrate how our user model can be used to verify the absence of classes of errors we will look at a simple case study. In [5] we used an earlier, less sophisticated version of the user model to investigate the verification of simple vending machines with the potential for post-completion errors. Here we consider a similar example, but instead concentrate on communication goal related errors. The design consists of features that appear in real machines. However, it has been reduced to the simplest form with which to demonstrate our approach.

Our chocolate machine takes exact money only and it is assumed it will only take a single coin of that value. To release the chocolate a button must be pressed (this is intended as a simplified version of the selection that most machines would offer). The design of the machine could require a specific ordering: coin inserted, then button pressed, or button pressed then coin inserted. In either case order errors could result. The problem can be eliminated if either order is allowed. We verify here a machine that does allow either ordering. We will also discuss the effect of trying to verify faulty designs. We assume for the sake of simplicity that the chocolate machine always contains chocolate.

We formally specify the chocolate machine using a traditional finite state machine description (see Figure 1) within higher order logic. The specification is represented by a relation on the machine's inputs and outputs. We group these inputs and outputs into a tuple of history functions to represent the machine state. We define a new type `mstate_type` to represent this. The machine has two inputs indicating that the button has been pressed and that the coin has been inserted. It has a single output that releases chocolate. Each of the history functions is a function from time (a natural number) to booleans indicating the value of the signal at that time. We define a series of accessor functions to obtain the values of particular components of the state. For example the function `InsertCoin` extracts from a machine state the history function representing the coin slot.

We define a new enumerated type `ChocState` to represent the 4 finite state machine states (as opposed to the state representing the values input and output discussed above).

```
ChocState = RESET_STATE | COIN_STATE | CHOC_STATE | DONE_STATE
```

The RESET state is the initial state. In the DONE state the chocolate is released. The COIN state is the state in which a coin has been inserted but the button not pressed and vice versa for the CHOC state.

For each state we define a relation indicating the value on the single output in that state, together with a relation indicating the next state. These are then combined in a relation giving the full specification for that state. For a small example such as that considered here, it might be simpler to just have one definition giving the whole automaton. However such an approach would not scale: in particular the resulting specification would be much less readable.

For example when in the RESET state the machine does not release chocolate so the value of the output is false.

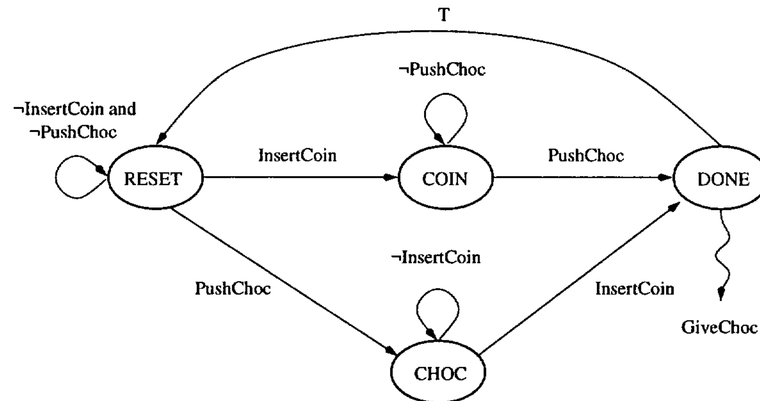

Fig. 1. Finite State Machine Specification of the Chocolate Machine

```
⊢ RESET_OUTPUTS (mstate: mstate_type) t = (GiveChoc mstate t = F)
```

We also give a relation representing the next state for each state. If a coin is entered it moves to a COIN state in the next cycle, if the button is pressed it moves to the CHOC state and otherwise it remains in the RESET state.

```
⊢ RESET_TRANSITION s mstate t =
    if       InsertCoin mstate t     then (s(t+1) = COIN_STATE)
    else if  PushChoc mstate t       then (s(t+1) = CHOC_STATE)
                                     else (s(t+1) = RESET_STATE)
```

For each state these two relations are combined in a relation that gives the whole behaviour (for example RESET_SPEC for the RESET state). A single definition then gives the full specification of the machine in terms of these definitions.

```
⊢ CHOC_MACHINE_SPEC s mstate =
    ∀t. if          (s t = RESET_STATE)     then  RESET_SPEC s mstate t
        else if     (s t = COIN_STATE)      then  COIN_SPEC s mstate t
        else if     (s t = CHOC_STATE)      then  CHOC_SPEC s mstate t
                                            else  DONE_SPEC s mstate t
```

7 Instantiating the User Model

To target the generic user model to a given machine we must provide values for all the arguments to USER except for the user state and machine state. For these we provide concrete types to instantiate the type variables given as their type.

The type of the machine state is just that used in the machine specification defined above: a tuple of history functions. For the user state we must provide a state consisting of a tuple of 6 elements. These elements are history functions that record for each time instance whether the user has chocolate, whether they

have a coin, whether they have terminated the interaction, a count of the amount of chocolate they possess, a count of the number of coins they possess, and a list of their communication goals paired with numbers giving the position of the corresponding action in the full list of actions. An accessor function for each part of the state is defined. For example, `UserCommgoals` extracts the communication goal list from the state.

The first argument we provide to `USER` is a list of all the possible user actions indicated by their history functions: the state extractor applied to the appropriate state tuple.

```
[InsertCoin mstate; PushChoc mstate; UserFinished ustate]
```

The second argument is the state extractor for the communication goals, `UserCommgoals`. We must also provide the initial communication goal list with which the user enters the interaction. In this case we assume that the user knows they must insert a coin at some point and that they must make a selection (push the chocolate button). This would be determined using a device-independent task analysis of the task of getting chocolate. We use the state extractor function to represent each communication goal. These are paired with a number giving their position in the full action list.

```
[(InsertCoin, 0); (PushChoc, 1)]
```

Note that, strictly speaking, inserting a coin is not a communication goal as it is concerned with property rather than information about a selection to be made. We intend in a later version of the user model to deal with these two kinds of knowledge separately. The main ramification for the theorem proved here is that as a communication goal no check is made in the user model as to whether the user has a coin as one of its possessions. This means the correctness theorem though not explicitly stating it says nothing about what happens if the user tries to insert a coin that they do not have.

Our particular machine provides no output to the user to indicate what must be done so an empty list is provided as the next argument for the pairings between outputs and the corresponding reactive input. A case study concerning post-completion errors where reactive pairings are provided can be found in [5].

We must also indicate the possessions of the user and how they are affected by particular actions. A relation `CHOC_POSSESSIONS` gathers this information into an appropriate form, given the history functions for the user having chocolate and coins, the machine giving chocolate, the user inserting a coin and counts of the number of coins and chocolate bars possessed.

```
CHOC_POSSESSIONS UserHasChoc GiveChoc CountChoc
                 UserHasCoin InsertCoin CountCoin
```

We specify which accessor functions to the user state indicate when the user has terminated the interaction, `UserFinished`, together with the number of its position in the list of actions (as with the communication goals). We also specify

the state accessor specifying the user's main goal in taking part in the interaction, UserHasChoc.

Finally we must provide the invariant that the user wishes to restore by the end of the interaction. For vending machines this can be based on the value of the user's possessions. After interacting with a vending machine a user does not wish the value of their total possessions to be less than they were at the start. This is described by a history predicate VALUE_INVARIANT. We omit the definition here.

The general model for the chocolate machine is specified by providing each of the arguments discussed above to the generic user model and restricting the types of the states to be the concrete types for the chocolate machine.

```
⊢ CHOC_MACHINE_USER (ustate:ustate_type) (mstate:mstate_type) =
    USER [InsertCoin mstate; PushChoc mstate; UserFinished ustate]
         UserCommgoals [(InsertCoin, 0); (PushChoc, 1)]
         ... ustate mstate
```

8 Verifying Usability

The usability correctness theorem we have proved in HOL has the following form:

```
⊢ ∀ustate mstate s.
  CHOC_MACHINE_USER ustate mstate ∧ CHOC_MACHINE_SPEC s mstate ⊃
      (s 0 = RESET_STATE) ∧ ~(UserHasChoc ustate 0)
                          ⊃ (∃t2. UserHasChoc ustate t2)
```

This is of the general form discussed earlier. The usability specification part of the theorem states that if we assume the vending machine starts in its reset state, and the user does not have chocolate but has communication goals of inserting a coin (paying money) and pushing the chocolate button (making a selection), then there will exist some time at which the user does have chocolate (i.e., has achieved their main goal).

This theorem is essentially proved using simulation by proof. An induction principle concerning the stability of a signal is used repeatedly. This essentially states that:

- if the value of some boolean signal P is stable over an interval,
- a second signal, Q, is true at the start of that interval, and
- if Q is true at some time, but P has the stable value at that time, then Q will be true at the subsequent time,
- then Q will be stable over an interval starting at the same point but extending one cycle later.

This is used to step the simulation over periods of inactivity.

In proving the usability theorem we have not proved that users using the machine will never make an error. We have, however, proved that no user will make the classes of errors with known cognitive causes specified in the user

model. In particular, we have proved that a user will not make order errors due to communication goal mismatches, provided they start with the stated communication goals. If these communication goals are identified using a device-independent task analysis then they will be consistent with the majority of users. Since such errors are both common and persistent as discussed in Section 3 the reliability of the system as a whole is consequently improved.

Consider an attempt to verify a design which requires the coin to be inserted before the button was pushed. This proof attempt would fail because the user model allows the user to do either of the communication goals first. If they pushed the button first, this action would be removed from their list of goals: they would believe the selection made. On then inserting a coin to complete their other goal, there would be no longer anything in the user model to compel them to press the button. We thus would be required to prove that they pushed the button, with no assumptions with which to do this. Of course a real user would in this case eventually work out the problem and go on to complete the interaction. However, the user error has already occurred.

9 Conclusions

We have described a formal verification methodology which detects classes of user error. In particular we have so far considered order errors based on communication goal mismatches and post-completion errors. These classes of errors are considered because they can be eliminated by appropriate design.

Our approach involves defining a generic user model which describes the behaviour of rational users. As with real users, erroneous behaviour is not specified to occur during every interaction. It is just specified as a potential behaviour. Given that potential behaviour exists, if it can be proved that the user does eventually achieve their goal, then it has been proved that the erroneous behaviour cannot manifest itself with the device under verification.

The use of a generic user model reduces the work required to produce a user model for each new device considered. More importantly, it reduces the chances that the user model is created in a device-centered way, specifying that the user behaves as expected by the designer of the device. It is based only on cognitive science theory that is generally applicable.

As alternative approach would be to write liveness properties corresponding to a list of known user errors for each system to be verified. However, to do so would require informal reasoning to determine the manifestation of the error from rational behaviour for every new device considered. For example, the order errors considered here are errors because the user does not have perfect knowledge of the design. Post completion errors are errors dependent on the user's goals. It is only by reasoning about the user's goals and knowledge that we determine the actions for which the ordering is important and determine what that ordering should be. In our approach, this reasoning is formalised and machine-checked. The general rational behaviour is specified once and the errors emerge.

The fact that a common user model is used means that the proofs for different devices are very uniform, increasing the possibilities for automation of the proof. For examples as simple as that presented here to illustrate the ideas it is likely that fully automated model checking/state-space exploration based verification tools could be used. However, when more realistic devices are considered it is likely that the additional power of an interactive theorem prover will be required. Furthermore, higher-order logic provides an elegant way in which a generic user model can be specified. It seems likely that this kind of proof would be a good application for a combined verification tool. The instantiated user model would be instantiated in HOL and exported to the automated system. Higher level details of the proof would be dealt with in HOL, with state exploration conducted in the automated tool. HOL could also be used to combine the usability correctness theorem with more traditional system verification theorems.

We used a very simple example of a chocolate machine to demonstrate the approach. We instantiated the generic user model with the details of a specific machine designed to avoid order errors. Despite the machine giving no indication of the steps required, because its design works with the communication goals of the task, it is usable. We also discussed how the proof would fail if other erroneous designs were considered. The design works because it has a permissive interface, allowing users to supply information in any order. It might be argued that such an approach could always be used. However, post-completion errors occur if the ordering of actions by the user is such that the user can complete their main goal before other required actions have been completed. Thus to avoid post-completion errors we must do the opposite of making the interface permissive. We must instead force a specific order. For example, if a machine dispensed change, it would be important that it was not dispensed before the chocolate. We investigated the verification of post-completion errors in an earlier paper [5]. There we investigated vending machines with and without post-completion errors. Our present user model has the ability to simultaneously detect order errors and post-completion errors. In future work we will investigate more complex machines and other classes of user errors. We will also look at machine designs with the potential for making multiple classes of errors. When considering a single class of error in isolation, it is relatively easy to ensure it is not present. When multiple kinds of errors are considered it is very easy to remove one kind of error, only to introduced another. This is where having a single generic user model is beneficial, since it ensures errors are not missed. It is in this situation that our verification approach will be of most use.

Acknowledgements This work is funded by EPSRC grants GR/M45221 and GR/L00391. The work was done in part whilst the first author was visiting Cambridge University Computer Laboratory.

References

1. R. Butterworth, A. Blandford, and D. Duke. Demonstrating the cognitive plausibility of interactive system specifications. Submitted to FACS journal. Available

from http://www.cs.mdx.ac.uk/puma/ as working paper WP25.
2. R.J. Butterworth, A.E. Blandford, and D.J. Duke. Using formal models to explore display based usability issues. *Journal of Visual Languages and Computing*, 10:455–479, 1999.
3. M. Byrne and S. Bovair. A working memory model of a common procedural error. *Cognitive Science*, 21(1):31–61, 1997.
4. J.C. Campos and M.D. Harrison. Formally verifying interactive systems: a review. In M. D. Harrison and J. C. Torres, editors, *Design, Specification and Verification of Interactive Systems '97*, pages 109–124. Wien : Springer, 1997.
5. Paul Curzon and Ann Blandford. Using a verification system to reason about post-completion errors. Presented at Design, Specification and Verification of Interactive Systems 2000. Available from http://www.cs.mdx.ac.uk/puma/ as working paper WP31.
6. D.J. Duke, P.J. Barnard, D.A. Duce, and J. May. Syndetic modelling. *Human-Computer Interaction*, 13(4):337–394, 1998.
7. M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press, 1993.
8. L. Lamport. The temproal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16:872–923, 1994.
9. W-O Lee. The effects of skills development and feedback on action slips. In Monk, Diaper, and Harrison, editors, *People and Computers VII.* Cambridge University Press, 1992.
10. T.G. Moher and V. Dirda. Revising mental models to accommodate expectation failures in human-computer dialogues. In *Design, Specification and Verification of Interactive Systems '95*, pages 76–92. Wien : Springer, 1995.
11. A. Newell. *Unified Theories of Cognition.* Harvard University Press, 1990.
12. F. Paterno' and M. Mezzanotte. Formal analysis of user and system interactions in the CERD case study. In *Proceedings of EHCI'95: IFIP Working Conference on Engineering for Human-Computer Interaction*, pages 213–226. Chapman and Hall Publisher, 1995.
13. J. Reason. *Human Error.* Cambridge University Press, 1990.