PUMA **PROGRAMMABLE USER MODELLING APPLICATIONS**

Using a verification system to reason about post-completion errors

Paul Curzon *Ann Blandford*

February 14, 2000

WP31

Principal Investigator	Prof Ann Blandford	Middlesex University
Research Fellow	Dr Richard Butterworth	Middlesex University
Academic Collaborator	Dr David Duke	University of York
Research Fellow	Jason Good	Middlesex Unversity
Industrial Collaborator	Sue Milner	Praxis Critical Systems Ltd
Academic Collaborator	Prof Richard Young	University of Hertfordshire

http://www.cs.mdx.ac.uk/puma/

Contact: Prof Ann Blandford
School of Computing Science
Middlesex University
Bounds Green Road
London. N11 2NQ. UK.
tel : +44 (0)181 362 6163
fax : +44 (0)181 362 6411
email : a.blandford@mdx.ac.uk

Project funded by
EPSRC grant
number GR/L00391

Using a verification system to reason about post-completion errors

Paul Curzon *Ann Blandford*

Abstract

Faults in the way a system works are often attributed to user error. In critical systems such user errors can be catastrophic. Formal verification is one approach advocated to help avoid catastrophic errors. Previous work has concentrated on ensuring that implementations meet specifications or that safety or liveness properties hold of a specification. However, systems verified in this way are still prone to catastrophic user errors. The designs of computer systems can often provoke certain classes of user errors. An example is the post-completion error where the user omits some necessary termination action. Work from the fields of cognitive psychology and human computer interaction suggest that such user errors are avoidable if systems are designed appropriately. We demonstrate that, by adopting a user-centric approach to system verification, formal proof methods can both detect and prove the absence of such system design errors. Furthermore, this approach can be integrated with traditional system verification methods. In particular, we show how the HOL proof system can be used to verify the absence of post-completion errors within the framework of a traditional hardware verification.

1 Introduction

Formal verification tools are, in general, either used to prove that an implementation meets a formal specification, or that particular safety or liveness properties hold of the system. The aim is to analyse the computer system; users are peripheral to the verification. A consequence of this is that errors made by users are not seen as being within the scope of verification tools. Similarly it is assumed that mistakes made by users are not the concern of the system designer. Such an approach can give a false sense of security in the infallibility of the system. If the system is designed without human users in mind, system failures may occur due to human error. Many such classes of error have distinct cognitive causes and are predictable [13]. Furthermore, changes to the design of systems can eliminate such errors [1, 8, 4]. A corollary of this is that verification tools can be designed to detect the presence of such intrinsic usability problems in a system design.

In this paper, we investigate a verification methodology that catches one class of user error. We show how a generic user model can be defined which incorporates cognitive psychology theory. In particular, we build into the model a single well-studied [4] cognitive property which results in a class of user errors known as post-completion errors. We illustrate this by instantiating the model for a simple interactive device: a chocolate machine. We show how this concrete chocolate machine user model can be used in conjunction with a traditional system specification to detect, or show the absence of, post-completion errors. We first attempt to verify a machine design for which post-completion errors can occur. We show that this attempt fails. We then successfully prove that such errors are absent from a modified design. Finally, we demonstrate how such results can be integrated with traditional system verification results.

2 Formal User Modelling

Various general approaches have been suggested as the basis of formal tools to aid in the verification of the usability of systems. One approach is to work with a formal specification of the user interface. Campos and Harrison [5] review various approaches of this kind. An alternative approach is that of *formal user modelling*: generating and reasoning about a formal specification of the user in conjunction with one of the computer system, rather than concentrating on the interface. Formal user modelling has emerged from interactions between cognitive scientists, who are concerned with producing rigorous descriptions of user cognition based on results of empirical research, and formal methods specialists. Examples include the work of Duke *et al* [6], Butterworth *et al* [3], Moher and Dirda [9] and Paterno *et al* [11] [12]. Each of these approaches takes a distinctive focus. Duke *et al* concentrate on channels and resources, Moher and Dirda look at users' mental models and their changing expectations whilst Butterworth *et al* focus on user knowledge and goal-based behaviour. The latter work of Paterno corresponds closely to that which is done in state space exploration verification. The user model corresponds to the environment machine used to give a closed system. The user model in this approach describes how users are *intended* to behave. Consequently it does not support reasoning about possible sources of error in the interaction.

Our approach is to build a generic user specification based on established results from cognitive science [10], and to use that user specification with a device specification to prove properties of the system. The user is described in terms of their goals (i.e. things they wish to achieve - such as having a chocolate bar) and in terms of the actions they may perform in order to achieve those goals. Generic user behavioural properties are formalised as definitions within the logic, so that we work in a common framework to that used for traditional system verification. The two activities are thus unified.

3 The HOL System

The work described here uses the HOL system [7]. It is a general purpose, interactive proof tool that has been used for a wide variety of applications. A typical proof will proceed by the verifier proving a series of intermediate lemmas, that ultimately can be combined to give the desired theorem. The system provides a wide range of definition and proof tools, such as simplifiers, rewrite engines, and decision procedures, as well as lower level tools for performing more precise proof steps. The architecture of the system means that specific proof tools can be built on top of the core system, combining the general proof procedures with ones specifically written for a given application. All specifications, goals and theorems in HOL are written in higher-order logic. Higher order logic treats functions as first class objects. The notation used in this paper is summarised in Table 1.

4 The Chocolate Machine

To demonstrate the approach we use a very simple example: a chocolate machine (see Figure 1). It takes pound coins only, returning 50p change. To release the chocolate

$a \wedge b$	both a and b are true
$a \vee b$	either a is true or b is true
$a \supset b$	a is true implies b is true
$\forall n.\ P(n)$	for all n, property P is true of n
$\exists n.\ P(n)$	there exists an n for which property P is true of n
f n	the result of applying function f to argument n
$a = b$	a equals b
if a then b else c	if a is true then b is true, otherwise c
$x : t$	x has type t
$\vdash P$	P is a definition or proven theorem in the logic

Table 1: Higher-order Logic notation

and change, corresponding buttons must be pressed. The machine uses lights next to the coin slot and buttons to indicate the order things should be done. The order of operation is that a coin is inserted, the chocolate button is pressed and the chocolate removed, and then finally the change button is pressed and the change removed. If the user does not press the appropriate button the machine does nothing until the correct button is pressed. We assume for the sake of simplicity that the chocolate machine always contains sufficient chocolate and change. Whilst this machine is simplified, its elements appear in real machines.

We formally specify the chocolate machine using a traditional finite state machine description (see Figure 2) 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 to represent the machine state. The machine will have as outputs 3 lights and 2 signals to release change and chocolate. Inputs correspond to the buttons being pressed and a coin inserted. This gives a total of 8 components. Each is represented by a *history function*. This is a function from time (a natural number) to a value of the signal at that time: it thus gives a full history of the values on the signal. For the chocolate machine the values at each time are booleans. We define a new type `mstate_type` to represent this. It is essentially just an abbreviation for the type of tuples of 8 boolean history functions. 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 |  CHANGE_STATE
```

The RESET state is the initial state. Each of the other states represent the corresponding action having been done: in the COIN state a coin has been accepted; in the CHOC state the chocolate is dispensed and in the CHANGE state, change is dispensed.

For each state we define two relations. The first specifies the outputs in that state, and the second gives the state transition function. Each takes as arguments the history

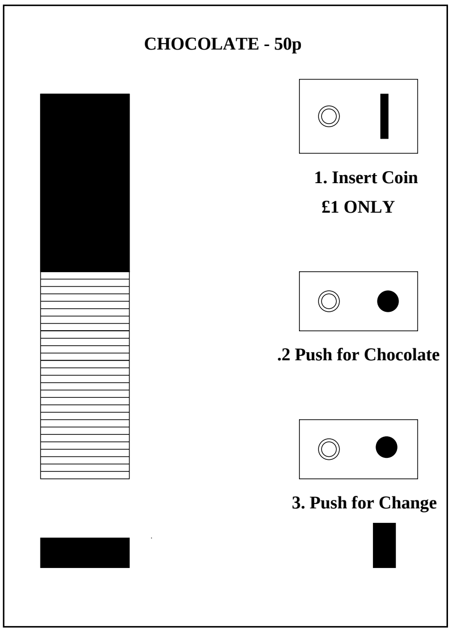

Figure 1: The Chocolate Machine

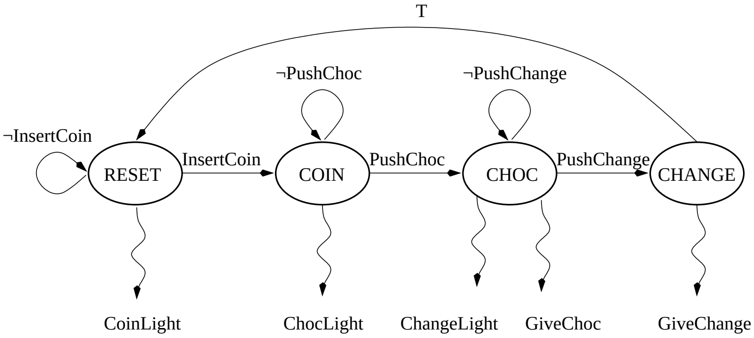

Figure 2: Finite State Machine Specification of the Chocolate Machine

function representing the machine state and the time of interest. The second takes the internal abstract state as an additional argument.

For example, initially the machine is in the RESET state with the insert coin light lit, waiting for a new user to insert a coin. The insert coin history function has value true (T) to indicate it is lit. None of the other outputs are set - their history functions have value false (F) at the time of interest.

```
⊢ RESET_OUTPUTS (mstate: mstate_type) t =
  (GiveChoc mstate t = F) ∧
  (GiveChange mstate t = F) ∧
  (ChocLight mstate t = F) ∧
  (ChangeLight mstate t = F) ∧
  (CoinLight mstate t = T)
```

If a coin is entered it moves to a COIN state in the next cycle, otherwise it remains in the RESET state.

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

These two relations are combined in a relation RESET_SPEC (omitted here). Similar definitions are given for each state. These definitions are bound together in the top level relation for the specification which decodes the state and indicates the appropriate relation that should hold in that state:[1]

```
⊢ CHOC_MACHINE_SPEC s (mstate: mstate_type) =
  ∀t.  if (s t = RESET_STATE) then RESET_SPEC s mstate t
       else if (s t = COIN_STATE) then ...
```

5 A Generic User Model

Our aim is to detect, or verify the absence of, user errors. To do so we treat the user as an explicit component of the system being verified. We therefore introduce a formal definition of the user: a formal user model. We could take the approach that a new user model be provided for each machine to be verified. However, we do not wish the model to be ad hoc but based on general cognitive psychology and HCI theory. Such theory applies to all users and is machine independent. We therefore provide a single generic user model that captures that theory once and for all, and provide machine specific information as parameters. Here we concentrate on one universal psychological property to illustrate the idea. Furthermore, a generic user model opens the possibility of building general proof tools around it.

However, for purposes of exposition, rather than immediately describing the generic user model, we first sketch a machine-centric specification. That is, we specify that the user provides inputs as expected by the designer of the machine. We then examine how such a naive user model can be extended to take into account the way real people behave based on results from cognitive science.

5.1 User Options

When confronted with a machine, at any given time, a user will have a series of options open. We describe this formally as a series of disjunctive clauses, each describing a possible action of the user at a given time instance. If we wished to model a user acting totally at random (pressing buttons, etc) then all possible actions would be given a disjunct. This would allow anything to be done at any time. However, we assume that some options are not sensible in a given context. Some actions will only be made if particular observations are made. We thus combine observations with actions, and only include those combinations that a "typical" user would make. In particular we specify a reactive strategy in which a user is guided in their actions by output from the machine. Thus, for example, if the only light lit on the machine is the one next to the coin slot, then a typical user who requires chocolate would take that as an indication that a coin should be inserted. We

[1] We omit detail from this and later definitions for the sake of clarity of exposition.

therefore formally specify inserting a coin as a user option under these circumstances. More specifically we specify that if the coin light is on, the user may after some time delay insert the coin. We do not, on the other hand, explicitly give an option for pushing the chocolate button when the insert coin light is lit. We assume there is no reason in general for a user to do this. We thus restrict the behaviour from the totally random by omitting possible behaviours.

Note that we are treating the user as a *reactive* agent. It is assumed that users who have a goal that pertains to the chocolate machine will react to the outputs of the machine. In a richer model of knowledge-based rationality, a user would only perform actions that they believe make progress towards the goal state [2]. We intend to introduce such a richer goal oriented model in the future. However, it is beyond the scope of the present paper.

To formally specify this reactive strategy we could give a series of definitions such as that below. It states that a possible behaviour is that the coin light is on at the current time `t` and after some delay the user inserts a coin.

```
⊢ USER_COIN (mstate:  mstate_type) t =
  (CoinLight mstate t = T) ∧ EVENTUALLY (InsertCoin mstate) t
```

`EVENTUALLY` is a temporal operator, stating that there exists a time in the future (i.e. after the given time `t`), that its first argument (a relation) is true.

We could give a similar specific definition for each light on the machine. However, we are assuming that the user is following a general strategy guided by the machine interface: if lights are placed in close proximity to the inputs of a machine, and they have a visual design that pairs them, then a user will use the lights to guide them over the next action to take. We can formalise this generically using a recursive definition which takes as a parameter the list of pairs of lights and inputs.

We first give a non-recursive definition `LIGHT` that corresponds to the definitions such as `USER_CHOC` (which it replaces):

```
⊢ LIGHT light action (mstate:  'm) t =
  (light mstate t = T) ∧ EVENTUALLY (action mstate) t
```

It takes a single light and action pair as parameters and asserts that a possible event is that if the light is on, the user may consequently eventually take the action. We then give a recursive definition `LIGHTS` (not included here) that applies this definition to each of the pairs in a list of light-action pairs, specifying that each is a possible behaviour.

5.2 Considering Real Users

So far we have developed the simple user model in a machine-centric way. By taking into account results from cognitive science we can give a more accurate user model. We initially consider a simple model that is prone to just one class of general user error, the post-completion error, to illustrate the point.

If our chocolate machine were put into use, a significant number of user errors would actually be made. In particular, HCI theory and usability studies predict that a type of error known as a *Post-completion Error* would frequently be made by users of the machine.

This is a common form of error made by humans in a wide variety of situations. Examples of the phenomenon include taking the cash but leaving a bank card in an Automatic Teller Machine, entering information in the wrong window on a windows system, 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.

Byrne and Bovair [4] have developed the most convincing and sophisticated explanatory model of post-completion errors so far, based on working memory load. Other researchers (e.g. Young [16] and Rieman *et al* [14]) give alternative accounts of the underlying cognitive mechanism. However, several things are clear. Post-completion errors 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. For our purposes, the details of the mechanism are unimportant; what matters is that we should be able to specify user behaviour in sufficient detail to support reasoning about interactive behaviour when using a particular device.

When entering an interaction, a user does so with a main goal such as to obtain a chocolate bar from the machine. Achieving this will involve some precondition which is not currently satisfied. For example, for the chocolate machine inserting a coin is a precondition to obtaining chocolate. The user therefore identifies a way to achieve that precondition. However, achieving it will perturb the state of the machine (too much money has been inserted so change is due). The outcome will depend on the design of the machine. If the user achieves their main goal before the perturbation has been corrected, the user is liable to forget about the perturbation as they have achieved what they entered the interaction to achieve. In the case of the chocolate machine, if the user has taken their chocolate, they may forget that change is due and consequently leave with just the chocolate bar. If the change were automatically and noisily dispensed, then the user would probably notice and take both. If the change were delivered before the chocolate bar, then the user would almost certainly take both.

Post-completion errors occur due to the goals of the user; we thus need to introduce the concept of a goal into our user model. We can formalise post-completion error behaviour in terms of whether the user has achieved their goal and whether they believe they have finished. This gives a further choice (i.e. disjunct) to add to those specified by our machine-centric user model: the goal is achieved, and the user leaves. This suggests users might still leave without change even from a machine that gives change automatically and noisily, if they had achieved their goal before this occurred.

The naive (reactive) user model given earlier was only concerned with the machine state and machine signals. Post-completion error behaviour could be specified in terms of the `GiveChoc` button in a similar way. However, it is more natural to consider the user as having their own state, `ustate`, which keeps track of, amongst other things, their possessions. We will describe a concrete user state in more detail in a subsequent section. For the purposes of our generic user model we just assume we have accessor functions to the state `goal` and `finished`. The former indicates when the user has achieved their goal and the latter indicates when the user has terminated the interaction.

```
⊢ COMPLETION finished goal (ustate: 'u) t =
  (goal ustate t = T) ∧ EVENTUALLY (finished ustate) t
```

We give a type variable `'u` as the type of the user state in our generic user model. This means it can apply to a wide range of different possible user states for different machines. We only specify those details of it that are specifically required for the general cognitive properties we are describing (for example, one field indicates whether the user has finished).

We add the completion clause as a new disjunct to our user model: it is another possible behaviour of a user. We do not assert here that all users in all circumstances will make the post-completion error. We give a single generic definition, `USER_CHOICES` describing the user's options. It takes as parameters the list of light-action pairs, the signals indicating when the user has finished and achieved their goal and both the user and machine state.

```
USER_CHOICES lights_actions finished goal (mstate:'m) (ustate:'u) t =
  (LIGHTS ...  t) ∨ (COMPLETION finished goal ustate t)
```

For the chocolate machine, `goal` would be set to a signal indicating the points in time when the user has chocolate. Leaving will be an option once the user has taken the chocolate. This now means that there are two alternatives once the user has taken the chocolate: leaving, or pushing the change button.

Superficially this appears to be a small change to the user model. However, the point is that it has only been introduced because we have shifted to a human-centric modelling approach. In doing so we have opened the possibility of verifying that a whole class of "user errors" are absent or, conversely, to discover them, before a device is manufactured.

5.3 The Full Generic User Model

We must add some other infrastructure to our user model. First we consider how to give a general description of the behaviour of a user terminating the interaction normally. The user leaving is tied up with them having achieved their goals. However, simply specifying that a user leaves on achieving their goal would suggest that a post-completion error would always be made. In practice this is not so.

One way to describe the termination condition in user terms is as an invariant that they wish to maintain, but that the system has perturbed. When using a vending machine the user ultimately wishes to leave with at least the same value of money and chocolate as they came with. Similarly, when using a photocopier, the user wishes to leave with the original they came with. In interacting with the machine this invariant is temporarily disrupted, and they will generally only wish to leave when it is restored. This can be specified generically by treating the invariant as a history function. It returns a boolean indicating whether the invariant property is true or false at each instance in time. Rather than make this an option we specify that the interaction always finishes once both the goal is achieved and the invariant holds. Otherwise the user's behaviour is governed by the options previously specified.

```
∀t.  if ((invariant ustate t = T) ∧ (goal ustate t = T))
     then (finished ustate t = T)
     else (USER_CHOICES ...  t)
```

In addition to specifying options open to the user, we also need to specify some universal laws. In particular we give laws about the giving and taking of possessions; we also need to specify that once an interaction has been terminated it cannot be restarted. We combine these universal truths about a user into a single definition USER_UNIVERSAL. It is generic, and must be provided with the finished history function as above. Details must also be supplied of the possessions of the user, including signals indicating when they are obtained and given up, their value and a count of the number of each possession held.

The complete behaviour of a user is the combination of the things we assume always to be the case together with the normal termination condition and the set of options that a user may take given the input signals. It is specified by a relation USER.

```
⊢ USER lights_actions possessions invariant finished goal
      (ustate:'u) (mstate:'m) =
  (USER_UNIVERSAL possessions finished ustate) ∧
  (∀t.  if ((invariant ustate t = T) ∧ (goal ustate t = T))
        then (finished ustate t = T)
        else (USER_CHOICES ...  t))
```

Each of the generic parameters to the previous definitions, such as goal, become parameters to this one. By providing concrete values for those parameters and for the state types we obtain user models for a range of machines, all incorporating the same general cognitive theory.

6 Specific User Models

To target the generic user model to a given machine we must provide:

- concrete types for the machine and user state, to instantiate the type variables 'm and 'u;
- a list of pairs of lights and the actions associated with them;
- history functions that represent the possessions of the user;
- functions that extract the part of the user state that indicates when the user has finished and has achieved their main goal, and
- an invariant that indicates the part of the state that the user intends to be preserved after the interaction.

The machine state is just that used in the machine specification, given by the type mstate_type, defined earlier, for our machine. For the user state we must provide a state consisting of a tuple of 7 elements. Each element is a history function that indicates a given property of the user at each time point: having chocolate, having change, having a coin, finishing and counts of the amount of chocolate, change and pound coins. Actions and observations directly related to the machine, such as seeing lights, are equated with the machine state so are not needed in the user state. Accessor functions are defined to access each element of this tuple. For example, the first element of the tuple holds the signal recording details of when the user possesses chocolate. The function `UserHasChoc` is therefore defined to extract the first element of the tuple. These functions provide the abstract interface to the state.

On our chocolate machine, the coin light is positioned with the coin slot, and the chocolate and change lights with the chocolate and change buttons. We therefore give a list that pairs the corresponding state accessor functions:

```
[(CoinLight,InsertCoin); (ChocLight,PushChoc); (ChangeLight,PushChange)]
```

We also give details of the possessions. A user of the chocolate machine can have three kinds of possession: chocolate, change (i.e. 50p pieces) and pound coins. A tuple is associated with each of these. Consider chocolate, for example. The first element of the tuple is a signal indicating when the user has possessed chocolate: here given by the accessor function from the user state `UserHasChoc`, mentioned above. Similar accessor functions to either the user or machine state are given as the other elements of the tuple. In addition the value associated with the possession is given – for chocolate 50p.

We specify which accessor functions to the user state indicate when the user has terminated the interaction, `UserFinished` and the user's main goal in taking part in the interaction, `UserHasChoc`.

Finally we must provide the invariant. For vending machine applications, this can be based on the value of the user's possessions. After interacting with a vending machine a user does not wish to have lost money, in the sense that the value of their total possessions should be no less than they were at the start. We give a relation, `POSS_VAL`, that formally defines the user's total worth based on the possessions specified. It takes as arguments, the details about the possessions, the user state and the time of interest. It simply multiplies the possession count by its value, summing the results for each possession. The invariant for the chocolate machine is specified in terms of this definition. The invariant is true at a point in time if the value of all the possessions at that time is greater than or equal to the value of the possessions at time 0.

```
VALUE_INVARIANT possessions (ustate:'u) t =
  (POSS_VAL possessions ustate t >= POSS_VAL possessions ustate 0)
```

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
   [(CoinLight,InsertCoin);(ChocLight,PushChoc);(ChangeLight,PushChange)]
   ...
```

7 Verifying Usability

Ultimately, we require that if the user arrives at the machine desiring chocolate when the machine is in its initial state (i.e the coin light is on), then eventually they will get both chocolate and change. We assume that they do not already have chocolate (if they do have chocolate then they have already achieved their goal, so will not need to interact with the machine), do not already have change, do have a coin to insert, and the invariant is as it was at the base time 0.

```
⊢ CHOC_MACHINE_USABLE (ustate: ustate_type) (mstate: mstate_type) =
  ∀t. (UserHasChoc ustate t = F) ∧ (UserHasChange ustate t = F) ∧
      (UserHasCoin ustate t = T) ∧ (VALUE_INVARIANT ...)  ∧
      (CoinLight mstate t = T) ⊃
                ∃t1. UserHasChoc ustate t1 ∧ UserHasChange ustate t1
```

This states that if at any time, t, a user approaches the machine when its coin light is on, then they will at some time, t1, have both chocolate and change. This definition is specific to the chocolate machine. It would be relatively simple to give a generic version; however this has not been done at the time of writing.

The usability theorem that we wish to prove states that if a user acts reactively (as specified) and the machine behaves according to its specification, then our usability property above will hold. We are effectively taking the system being verified as the combination of the user and the machine, and proving that this combined system satisfies the usability specification.

```
∀(ustate: ustate_type) (mstate: mstate_type) s.
  CHOC_MACHINE_USER ustate mstate ∧ CHOC_MACHINE_SPEC s mstate ⊃
   CHOC_MACHINE_USABLE ustate mstate
```

Note that we are proceeding as if the usability property were a requirements specification (property). The difference is that we incorporate the user model into the description of the system about which we prove the requirement. Furthermore the requirement is stated as a user-centric property rather than as a property of the machine.

By giving the generic definitions to the HOL system and defining some specific proof tools, we obtain a tool with which we can verify the usability of the specification essentially by performing a symbolic simulation by proof. We start from the assumptions about the machine and user state at the initial time. We then deduce facts about the subsequent state at the next time instance. We can use these new facts to step on a further time instance, and so on. We use the user model to deduce facts about the users actions, and the machine specification to deduce facts about the machine's actions. Our correctness statement is a liveness property so we need to step forwards until we come to a state for which the desired property holds.

This is done using semi-interactive proof in HOL. As the structure of the specification and user model is very uniform the same proof steps are used to step between each pair of states. Given the simplicity of the example presented here, automated state exploration tools could be used for such a verification. However, we believe that for more substantial systems the additional power of an interactive proof tool such as HOL will be needed.

For the machine we described, the usability property cannot be proved, however. This is because of the completion disjunct of the universal user model. On receiving the chocolate, the user has achieved their goal. They may therefore leave rather than pushing the change button. The system is flawed. This manifests itself as an unprovable HOL subgoal. We must prove that the change button will be pressed, given a list of known facts about the machine and user. However, the assumptions we have give us no information to deduce this from. We know only facts such as the state of each of the lights (only the change light is on) and that the machine is not giving change. As this goal arises from the COMPLETION disjunct, its unprovability indicates that the user can leave the machine without their change due to a post-completion error.

8 A Design Without Post Completion Errors

One way in which we can correct the usability problem of the machine, is to change its design so that it gives out the change before the chocolate. This ensures that the main user goal is not achieved until the last step of the interaction. Post-completion errors are then no longer possible. The new design involves only minor changes to the specification and implementation. The new machine has the same buttons and lights and the user reacts in the same way to those signals. Thus the user model for the new machine does not need to be changed.

With the new design, the proof of usability can be completed. Now, when the post-completion error termination clause in the user model is activated, the invariant has already been restored so the normal termination condition is triggered. We are therefore able to prove a usability theorem of the desired form:

```
⊢ ∀(ustate: ustate_type) (mstate: mstate_type).
  CHOC_MACHINE_USER ustate mstate ∧ CHOC_MACHINE_SPEC s mstate ⊃
    CHOC_MACHINE_USABLE ustate mstate
```

This states that provided the machine behaves within the bounds of its specification and the user behaves according to the user model then the user will obtain both chocolate and change. Note that we have not proved that under all circumstances this will happen. If the user is distracted by some external signal, (such as for example a fire alarm going off), they may leave the machine with nothing. This would not be a post-completion error but a different form of termination error [15]. Our claim is that we have proved that a single class of common errors with a specific cognitive cause (post-completion errors) have been eliminated from the design. By extending the user model other classes of user errors could similarly be eliminated.

9 Combining Usability and System Verification

Our usability specification, user model and system specification have been given in a single formalism and verification system: higher-order logic within the HOL proof system. However, we can go a step further. The same framework can be used to specify the implementation and formally verify that it meets the specification. Both the HOL system and our specification approach have been used for system verification [7]. For hardware, we formally specify the behaviour of each component of the system as a relation on its inputs, and give a structural description of the implementation, not unlike a hardware description language description. We then verify a correctness theorem of the form below that the implementation implements the specification.

More explicitly this states that for any sequence of machine states `mstate` that are possible behaviours of the specified implementation, there exists some abstract state, `s`, which satisfies the specification.

```
⊢ ∀mstate.
  CHOC_MACHINE_IMPL (mstate: mstate_type) ⊃ ∃s. CHOC_MACHINE_SPEC s mstate
```

The usability theorem proved above asserts the usability of an abstract specification of a chocolate machine. Ultimately we wish to know that our concrete implementation itself is usable. Our implementation correctness theorem allows us to prove that explicitly. Because we have used the same formalism both in the correctness proof and usability proof, it is trivial to prove the correctness theorem we desire by combining the two theorems.

```
⊢ ∀(ustate: ustate_type) (mstate: mstate_type).
  CHOC_MACHINE_USER ustate mstate ∧ CHOC_MACHINE_IMPL mstate ⊃
    CHOC_MACHINE_USABLE ustate mstate
```

Note that this theorem no longer refers to the behavioural specification, nor to the states of the abstract specification, `s`. Instead it refers only to the machine implementation and its external state (mstate). It states that if the user behaves as specified, interacting with a machine with the given implementation, then the user will eventually obtain both chocolate and change.

10 Conclusions

The fact that a system has been verified to be correct in the traditional way does not mean that catastrophic user errors will not be made. Traditional formal verification is machine-centric and so cannot detect such errors. We have shown that higher-order logic provides an elegant way to provide a generic user model. A single model can be given that encapsulates general user behaviour. Specific behaviour for a given machine can then be provided as arguments. We have further demonstrated that this user model can form the basis of a verification tool with which both correctness and usability can be verified. We have used an interactive proof system, HOL, in our work. However, the general approach is applicable to other tools and in particular those based on automated state exploration.

We have presented a simple model of a reactive user augmented with one result from cognitive science: that users make post-completion errors. We do not claim to be able to prove that all users will be guaranteed to always successfully use the machine. We can prove, however, that they will not make *post-completion errors*. That is, we have proved that mistakes will not be made due to one common specific cognitive cause concerned with working memory overload.

We have used a simple example of a chocolate machine to illustrate our approach. In such an application, formal verification may appear heavyweight; however, it provides a suitable example for testing the approach. We intend to continue to develop the generic user model, adding other results from cognitive science such as that user behaviour is rational in the sense that users deliberately select actions towards achieving goals, that user knowledge is important, that users need feedback, etc. Ensuring that a single class of user error has been avoided may be straightforward. Ensuring that a whole range of such, often conflicting, requirements are met for a real, complex system is a much greater challenge without tool support. In safety critical systems such as on aircraft flight decks, usability issues are critical. Thus the integration of formal usability verification tools with system verification tools is of great importance.

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] A. Blandford. Puma footprints: linking theory and craftskill in usability evaluation. Submitted to HCI'2000.

[2] R.J. Butterworth and A.E. Blandford. The principle of rationality and models of highly interactive systems. In M. A. Sasse and C. Johnson, editors, *Human-Computer Interaction INTERACT'99.* Amsterdam: IOS Press, 1999.

[3] 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.

[4] M. Byrne and S. Bovair. A working memory model of a common procedural error. *Cognitive Science*, 21(1):31–61, 1997.

[5] 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.

[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] 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.

[9] 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.

[10] A. Newell. *Unified Theories of Cognition.* Harvard University Press, 1990.

[11] F. Paterno' and A. Leonardi. A semantics-based approach for the design and implementation of interaction objects. *Computer Graphics Forum*, 13(3):195–204, 1994.

[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.

[14] J. Rieman, M. Byrne, and P.G. Polson. Goal formation and the unselected window problem. In *Proc. CHI94 Basic Research Symposium.*, 1994.

[15] H. Thimbleby. *User Interface Design.* ACM Press, 1990.

[16] R.M. Young. The unselected window scenario: analysis based on the SOAR cognitive architecture. In *Proc. CHI94 Basic Research Symposium.*, 1994.