Preprint to appear in Formal Aspects of Computing

An Approach to Formal Verification of Human-Computer Interaction

Paul Curzon,[1] Rimvydas Rukšėnas[1] and Ann Blandford[2]

[1]Department of Computer Science, Queen Mary, University of London, Mile End Road, London E1 4NS, UK
[2]University College London Interaction Centre, Remax House, 31-32 Alfred Place, London WC1E 7DP, UK

Abstract. The correct functioning of interactive computer systems depends on both the faultless operation of the device and correct human actions. In this paper, we focus on system malfunctions due to human actions. We present abstract principles that generate cognitively plausible human behaviour. These principles are then formalised in a higher-order logic as a *generic*, and so retargetable, cognitive architecture, based on results from cognitive psychology. We instantiate the generic cognitive architecture to obtain specific user models. These are then used in a series of case studies on the formal verification of simple interactive systems. By doing this, we demonstrate that our verification methodology can detect a variety of realistic, potentially erroneous actions, which emerge from the combination of a poorly designed device and cognitively plausible human behaviour.

Keywords: Formal verification; Human error; Formal cognitive architecture; Interactive systems, Theorem proving

1. Introduction

Contemporary society is increasingly reliant on interactive computer systems. They are used not only in avionics, traffic control systems, robotics, the automotive industry and e-business, but also in the devices people encounter every day. In these circumstances, their smooth operation becomes increasingly important. Since interactive computer systems combine a human operator and a computer based device, their correct functioning depends on the behaviour of both parts — the faultless operation of the device and correct human actions. In this paper, we focus on the malfunctioning of interactive systems caused by human actions that are essentially cognitive slips, not rich knowledge-based errors as studied within the PUM work [BBC04].

Much traditional system verification work has focused on verification of the device components and their interactions, trusting that the human elements in the system will perform as intended. However, many "human errors" have causes that lie in the design of people's interactions with other system components. Thus, in addition to a device specification, a user model (at least an implicit one) is necessary to detect such

Correspondence and offprint requests to: Rimvydas Rukšėnas, Department of Computer Science, Queen Mary, University of London, Mile End Road, London E1 4NS, UK. E-mail: rimvydas@dcs.qmul.ac.uk, Fax: +44 (0)208 980 6533

errors during formal system verification. Developing a formal user model is a conceptually clean method of extending traditional verification methods to the domain of interactive systems. Having a formal user model allows rigorous reasoning about the properties of an interactive system as a whole, including its users. Furthermore, a formal specification of users inevitably makes *explicit* the assumptions made about their behaviour. This facilitates their inspection, validation and modification (if necessary) later on.

The complexity of human behaviour, its variance within the population and its dependence on context and situation [Suc87] makes it difficult to capture by any formal model. However, it is a reasonable, and useful, approximation to say that humans behave rationally in the following sense. They enter an interaction with goals they try to achieve and have some knowledge that seems likely to help them achieve those goals. Within certain designs of interactive systems, whole classes of persistent and systematic user errors may occur due to cognitive causes [Rea90]. This means that the opportunities for making such errors can be eliminated with appropriate designs [ByB97].

We describe an approach to the verification of interactive computer systems that allows the detection of systematic user errors. Examples of these are termination related errors, order errors due to the preconceived knowledge of the task, random user behaviour due to the delay in the machine's response, and errors induced by the gap between the physical action and the moment that action is triggered by the human brain. In our approach, both components of an interactive system, the user and the device, are modelled explicitly. This contrasts with machine-centered verification methods in which the environment of the device is assumed to provide the input required. The errors detected by our approach emerge from the interaction of both components of an interactive system. This differs from approaches based on formulating device properties corresponding to user errors and checking that those properties do not hold of the device.

Our user model is based on simple principles of cognition [CuB04]. They generalise the way humans act in terms of the mental attributes of tasks and goals. The principles are each backed up by evidence from HCI and/or psychology studies. They do not describe a particular individual, but generalise across people as a class. We have earlier formalised [CuB01b, CuB04] these principles as a *cognitive architecture* within higher-order logic. The cognitive architecture is generic in the sense that it can be instantiated for a given task and device.

Formal verification aims to either detect design errors or show their absence. A better understanding of the design is obtained during verification efforts. Thus when errors are detected the verifier also gains a detailed understanding of why they occur. Basing the verification on principles of cognition can reveal the cognitive causes of errors. Therefore, even though our method does not explicitly identify them, design improvements can be suggested as a result of the verification, as has been demonstrated by hardware verification efforts [CuL96].

1.1. Aims and objectives

In this paper, we describe a series of case studies. In each we consider a simple interactive system. The intention is to illustrate how our verification method can detect a range of errors emerging from cognitively plausible user behaviour. Some of these case studies are similar to the ones described in our earlier papers [CuB00a, CuB00b, CuB01a, CuB01b]. However, those earlier case studies were conducted while the current version of the formal cognitive architecture was being developed (thus facilitating and validating its development). Consequently, different, and simpler, cognitive architectures have been used in each of the previous verification efforts. Here we consolidate our earlier case studies by using a more advanced version of the cognitive architecture. This not only presents our approach to the verification of interactive systems in a coherent way; it also ensures that our earlier verification results have not been invalidated by the later developments of the cognitive architecture.

Some of the presented case studies are new and demonstrate how new classes of user errors can be detected using our method. One example is errors induced by the delay between the physical action and the moment when that action was triggered in the human brain. In addition to studying various classes of errors, we also investigate more complex styles of interface. In particular, a new way of representing an interface allows us to model proper soft-key interfaces in which the functionality associated with device inputs can change over time.

Finally, we investigate a new way of using our cognitive architecture in the verification of interactive systems. Originally, our cognitive architecture was developed to formalise principles of cognition, to validate them once, and then rely on them in a verification by using concrete user models obtained by suitable instantiations of the generic architecture. The idea was to develop a single fixed user model for each combination of a task and a device. Here we argue that it is feasible to use several user models for each such combination. Different versions of the user model would represent different types of users, for example, novices and expert users. What represents a particular type of user is to be determined by HCI experts. Once this has been decided, however, the most suitable version of the user model can be used to verify interactions with the corresponding device in various environments and/or different modes of operation.

1.2. Related work

There are several approaches to formal reasoning about the usability of interactive systems. One approach is to focus on a formal specification of the user interface [CaH97, MJR98] while assuming a completely *unpredictable* user. Most commonly such specifications are used with model-checking-based verification; investigations include whether a given event can occur or whether desired properties hold of all device states. Another formal approach is exemplified by the work of Bumbulis *et al* [BAC$^+$96], who verified properties of interfaces based on a guarded command language embedded in the HOL theorem proving system. Back *et al* [BMW99] use a contracts based framework which distinguishes two kinds of nondeterminism; *angelic* nondeterminism is associated with user choices, whereas nondeterminism in device behaviour is considered as *demonic*. Using this distinction, they illustrate how properties can be proved and data refinement performed on a specification of an interactive device. However, techniques that focus on the interface do not directly support reasoning about design problems that lead to users making systematic errors; also, the usability properties checked are necessarily specific to an individual device, and have to be reformulated for each system verified.

An alternative approach is formal modelling of specific user behaviours within the underlying system. In a simpler form, this involves writing both a formal specification of the device and task models for that device, to support reasoning about the behaviour of the interactive system [PaM95, MoD95, Fie01, Cam03]. Moher and Dirda [MoD95] 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 device but focuses on changes in user belief states rather than proof of desirable properties. Paternò and Mezzanotte [PaM95] obtain task specifications by first structuring the tasks in a hierarchical way and then describing temporal relationships among them. The task specifications provide information for deriving the architectural description of a user interface in LOTOS. Various properties of the interactive system are expressed in ACTL and verified by model checking. Campos [Cam03] integrates task knowledge into the analysis of interactive systems based on the assumption of unpredictable users.

Task models describe how users are intended to behave; as such they are most useful for reasoning about expert behaviour. However, they do not address human fallibility. Proving a property of a system holds if the user does the correct steps is useful but different to proving properties when the user may not do so given cognitively plausible opportunity. If verification is to detect such user errors, a formal specification of the user behaviour, unlike one of a device, is not a specification of the way users should be; rather, it is a description of the way they are [BBD00]. This idea underlies approaches based on formal user modelling [DBD$^+$98, BoF99, DuD99]. By representing the cognitive structures of the user in a generic way, formal user models specify how user behaviour is generated. To reason about the behaviour of an interactive system, a formal user model is combined with a formal specification of the device. Doing so provides a conceptually clean method of bringing usability concerns into the domain of traditional verification in a consistent way. Both device and user are considered as central components of the system and modelled as part of the analysis known as *syndetic modelling* [DBM$^+$95].

Within the paradigm of syndetic modelling, Duke *et al* [DBD$^+$98] use Interactive Cognitive Subsystems (ICS) [BaM95] as the underlying model of human information processing. This cognitive architecture is formalised using a form of modal action logic which is also applied to capture constraints on the channels and resources within a specific interactive system. Their framework, however, lacks tool support, which would make it more difficult to apply in practice. In a related approach, Bowman and Faconti [BoF99] formalise ICS using the process calculus LOTOS, and then apply a temporal interval logic to analyse its properties. Both these approaches are particularly well suited to reasoning about multi-modal interaction that, for example, combines the use of speech and gesture [DuD99].

Our work also falls within the scope of syndetic modelling approaches. Its major difference from the ICS-based work is that it takes an even more abstract view of cognitive processes. The ICS model deals with information flow between the different cognitive subsystems and constraints on the associated transformation processes. As a result, the above work focusses on reasoning about multi-modal interfaces and analyses whether interfaces based on several simultaneous modes of interaction are compatible with the capabilities of human cognition. In its current form, our cognitive architecture is too abstract to deal with such issues. Instead, we focus on goals and task knowledge of users. This allows us to detect potential interaction problems in the situations when an interface design is, in some way, in conflict with user knowledge and goals. Furthermore, our more abstract model should make detecting such issues easier compared to the approaches based on more detailed modelling. A similar approach is taken by Butterworth *et al* [BBD99] who use TLA to reason about reachability conditions within an interaction.

Rushby *et al* [RCP99, CJR00, Rus01a, Rus01b, Rus02], like us, focus on user errors, though in their case just on one important aspect of human limitation. In particular they formalise plausible mental models of devices, looking for discrepancies between these and actual behaviour using the Murϕ state exploration tool. They are specifically concerned with the problem of mode errors and the ability of pilots to track mode changes. Essentially, the mental models are abstracted device models; they do not rely upon some structure provided by cognitive principles. Various ways of developing them are suggested including an empirical approach based on questioning pilots about their beliefs about the control system [CJR00]. Like interface-oriented approaches, each model is individually hand-crafted for each new device in this work. It is concerned with user knowledge which may vary depending on experience, training and other circumstances. Our work complements this approach by concentrating on more stable aspects of cognition, common to most humans and thus formalised as a cognitive architecture.

Bredereke and Lankenau [BrL02] develop a rigorous view of mode related errors using a refinement relation. To avoid such errors, the user's perception of reality must be a refinement of the mental model. The perception of reality is expressed as a relation from environment events to mental events that could in principle be lossy, corresponding to physical or psychological reasons for an operator not observing all interface events of a device. However, Bredereke and Lankenau note that in practice in their work the relation does no more than renaming of events and so is not lossy. Their approach is illustrated by a case study on service robotics [Lan01] using the CSP-based FDR model checking tool.

Buth [But04] extends Rushby's *et al* basic approach [Rus02], but using CSP and the FDR tool rather than Murϕ. Whereas in Rushby's original work the device model and mental model are intertwined in a single set of definitions, in CSP they can be described as two separate processes. This separation eliminates any potential for the introduction of a dependency between the device and mental models that does not exist in reality. Such a dependency may obscure flaws in a device design, as illustrated by an error highlighted by Buth that was detected using Murϕ. The FDR tool also provides a more direct way for comparing the two models; they are compared as regards their external behaviour without any need to define an invariant of the combined model.

Rushby's work was inspired by that of Leveson and her colleagues [LDS$^+$97, LeP97]. They [LDS$^+$97] identified six categories of indicators (design features) for potential mode related errors. These can be used as criteria against which a formal device specification can be checked. Leveson and Palmer [LeP97] apply their method to the kill-the-capture bug in the MD-88 autopilot analyzed later by Rushby. Butler *et al* [BMP$^+$98] develop formal models for two of the categories identified by Leveson *et al* in PVS. They show how these, potentially problematic, situations can be detected in a generic autopilot architecture by theorem proving. Lüttgen and Carreño [LüC99] note that the proof style employed by Butler *et al* resembles a form of state exploration. They compare the strengths and weaknesses of three state exploration (model checking) tools, Murϕ, SMV and Spin, in detecting three categories of potential mode confusion problems.

An approach to interactive system verification that focuses directly on a range of errors is exemplified by Fields [Fie01]. He explicitly models external, observable manifestations of erroneous behaviour, error patterns, analysing the consequences of each possible pattern. He thus models the effect of errors rather than their underlying causes. A problem of this approach is the lack of discrimination about which errors are the most important to consider. It does not discriminate random errors from systematic errors which are likely to re-occur and so be most problematic. It also implicitly assumes there is a "correct" plan, from which deviations are errors.

Table 1. Higher-order logic notation

Notation	Meaning
a ∧ b	both a and b are true
a ∨ b	either a is true or b is true
a ⊃ b	a is true implies b is true
∀n. P(n)	for all n, property P is true of n
∃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
λa.f	the function specified by f with the argument a
a = b	a equals b
if a *then* b *else* c	if a is true then b is true, otherwise c is true
:bool	the type of booleans
:num	the type of natural numbers
:time	the type representing time by natural numbers
:a→ b	the type of functions from type a to type b
:a#b	the type pairing elements of type a with elements of type b
(a, b)	the pair with first element a and second element b
FST p	the first element of pair p
SND p	the second element of pair p
[ ]	the empty list
x :: l	cons: the list consisting of first element x and remaining list l
l1 APPEND l2	the list consisting of list l1 appended onto list l2
EL n l	the nth element of list l
MAP f l	apply f to each element of list l
⊢*thm* P	P is a theorem proved in HOL
⊢*def* P	P is a definition

1.3. Contribution

The main contribution of this paper is a *generic* formalisation, within a theorem prover, of principles of cognition based on results from the cognitive sciences. The distinction to previous work is in our use of generic models in a form useful for formal reasoning. Most previous formal verification (e.g. [BBD99, BoF99, MoD95, PaM95]) work creates specific models from scratch each time. Most cognitive science work (e.g. [JoK96, New90]) is aimed at simulating human cognition at a level of detail much greater than ours, with different aims. We show how our generic cognitive architecture is instantiated to obtain various specific user models whose behaviours all emerge from the same embedded principles of cognition. We then use these formal user models, in combination with device specifications, to explore within the theorem prover environment the behaviours of the corresponding interactive systems. Our exploration demonstrates that a variety of realistic potentially erroneous actions, which emerge from these behaviours, can be discovered by our formal verification methodology. Finally, we show how the generic cognitive architecture can be specialised to represent different types of users for the same device.

2. Formal cognitive architecture

2.1. Generic user model

As already mentioned, our cognitive architecture is a higher-order logic formalisation of simple principles of cognition. These principles are discussed below as we specify the cognitive architecture. The formalisation and the subsequent verification of interactive systems presented in this paper has been done within the HOL interactive proof system [GoM93]. As the HOL system is widely used in a variety of applications, this provides a framework which can combine the verification of interactive systems and complex hardware [CuB04]. The HOL notation used in this paper is summarized in Table 1.

The cognitive architecture is specified as a higher-order logic relation `USER` which is defined in full in Appendix A; its top levels are given in Figure 1. It takes as arguments information such as the user's goal, `goalachieved`, a list of actions that the user may take, `actions`, *etc.* The arguments of the architecture will be examined in more detail as the corresponding parts of the architecture are described below.

```
⊢def USER flag actions commitments commgoals init_commgoals reactive_actions
              possessions finished goalachieved invariant (ustate:'u) (mstate:'m) =
      (USER_UNIVERSAL flag actions commgoals init_commgoals possessions finished ustate mstate) ∧
      (USER_CHOICE flag actions commitments commgoals reactive_actions
              finished goalachieved invariant ustate mstate)

⊢def USER_CHOICE flag actions commitments commgoals reactive_actions
              finished goalachieved invariant ustate mstate =
      (∀t.
      ¬(flag t) ∨
      (if (finished ustate (t-1))
       then (NEXT flag actions FINISHED t)
       else if (CommitmentMade (CommitmentGuards commitments) t)
       then (COMMITS flag actions commitments t)
       else if TASK_DONE (goalachieved ustate) (invariant ustate) t
       then (NEXT flag actions FINISHED t)
       else USER_RULES flag actions commgoals reactive_actions goalachieved mstate ustate t))

⊢def USER_RULES flag actions commgoals reactive_actions goalachieved mstate ustate t =
      COMPLETION flag actions goalachieved ustate t ∨
      REACTS flag actions reactive_actions t ∨
      COMMGOALER flag actions commgoals goalachieved ustate mstate t ∨
      ABORTION flag actions goalachieved commgoals reactive_actions ustate mstate t
```

Fig. 1. The USER relation

The final two arguments of the architecture, `ustate` and `mstate`, each of polymorphic type as specified by the type variables `'u` and `'m`, represent the user state and the machine state over time. Their concrete type is supplied when the architecture is instantiated for a particular user model in a given interaction. The user and machine states record over time the series of mental and physical actions made by the user, together with a record of the user's possessions and knowledge. They are instantiated to a tuple of *history functions*. In general, a history function is of type `time` $\rightarrow$ `'s`, from time instances to values of type `'s`. We mostly use boolean values. In this case, a history function indicates whether the corresponding signal is true at that time (i.e. the action is taken, the goal is achieved, *etc.*). The other arguments to `USER` specify accessor functions to one of these states. For example, `finished` is of type `'u` $\rightarrow$ `(time` $\rightarrow$ `bool)`. Given the user state it returns a history function that for each time instance indicates whether the cognitive architecture has terminated the interaction.

The `USER` relation is split into two parts. The first, `USER_CHOICE`, models the user making a choice of actions. It formalises the action of the user at a given time as a series of rules, one of which is followed at each time instance. `USER_UNIVERSAL` specifies properties that are true at all time instances, whatever the user does. For example, it specifies properties of possessions such that if an item is not given up then the user still has it. In outline, `USER_CHOICE` states that the next user action taken is determined as follows:

```
if the interaction is finished
then it should remain finished
else if a physical action was previously initiated
then the physical action should be taken
else if the whole task is completed
then the interaction should finish
else an appropriate action should be chosen non-deterministically
```

The details of each part are described in the subsequent sections.

2.2. Timing of Actions

The time taken to perform an action after it becomes plausible is not predictable. For example, once an instruction to make a choice of destination becomes available on a ticket machine, it is not predictable exactly how long a person will take to press the button corresponding to their destination choice.

The architecture is based on a temporal primitive, `NEXT`, that specifies the next user action taken *after* a given time. For example,

NEXT flag actions action t

states that the `NEXT` action performed *after* time `t` from a list of all possible user actions, `actions`, is `action`. It asserts that the given action's history function is true at some point in the future, and that the history function of all other actions is false up to that point. The action argument is of type integer and specifies the position of the action history function in the list `actions`. The `flag` argument to `NEXT` and `USER` is a specification artifact used to ensure that the time periods that each firing rule specifies do not overlap. It is true at times when a new decision must be made by the architecture. The first line of `USER_CHOICE` in Figure 1, $\neg$`(flag t)`, thus ensures, based on the truth of the flag, that we do not re-specify contradictory behaviour in future time instances to that already specified.

A detail about the formalisation that is needed to understand the definitions is that the action argument of `NEXT` is actually represented as a position into the action list. These positions are defined when the cognitive architecture is instantiated.

2.3. Non-determinism

In any situation, any one of several behaviours that are cognitively plausible at that point might be taken. It cannot be assumed in general that any specific behaviour that is plausible at a given time will be the one that a person will follow. Thus the cognitive architecture is ultimately, in the final else case above, based on a series of non-deterministic temporally guarded action rules, formalised in relation `USER_RULES`. Note the use of disjunction in definition `USER_RULES` in Figure 1. Each rule describes an action that a user *could* plausibly make. The rules are grouped corresponding to a user performing actions for specific cognitively related reasons. For example, `REACTS` in Figure 1 groups all reactive behaviour. Each such group then has a single generic description. Each rule combines a guard, such as a particular message being displayed, with an action, such as a decision made to press a given button at some later time. Apart from those included in the if-then-else staircase of `USER_CHOICE`, no further priority ordering between rules is modelled.

2.4. Mental versus physical actions

A person commits, making a mental action, to take a physical action before it is actually taken. Once a signal has been sent from the brain to the motor system to take an action, after a certain point known as a "point of no return" [Bar58], the signal cannot be stopped even if the person becomes aware that it is wrong before the action is taken. For example, on deciding to press a button labelled with the destination "Liverpool", at some point when the decision is made the mental trigger action takes place and after a very short delay, the actual action takes place. In cognitive psychology, the mental processes that immediately precede this temporal boundary are known as *ballistic* processes. Once launched, they "must proceed to completion, and, upon completion, necessitate the start of overt movement" [OKM86]. Note that some experiments suggest that human sense of volitional control over actions can be illusory [Lib89]. However, this is not a concern at the level of abstraction at which our model has been developed. What is important for us is the fact that there is a time gap between the moment an action is triggered, for whatever reason, and the moment it is actually performed.

We model both physical and mental actions. A similar distinction between a preparation phase and an execution phase is made in the EPIC architecture [KWM97]. In our case, each physical action modelled is associated with an internal mental action that commits to taking it. The argument `commitments` to the relation `USER` is a list of pairs that links the mental and physical actions. `CommitmentGuards` extracts a list of all the mental actions (the first elements of the pairs). The recursively defined `CommitmentMade` checks, for a given time instance, `t`, whether any mental action, `maction`, from the list supplied, was taken in the previous time instance (`t-1`):

$\vdash_{def}$ (CommitmentMade [ ] t = FALSE) $\wedge$
  (CommitmentMade (maction :: rest) t =
    (maction(t-1) = TRUE) $\vee$ (CommitmentMade rest t))

If a mental action, `maction`, made a commitment to a physical action, `paction`, on the previous cycle (time, `t-1`) then that will be the next action taken as given by `COMMIT` below. Definition `COMMITS` then asserts this disjunctively for the whole list of commitments:

$\vdash_{def}$ COMMIT flag actions maction paction t =
    (maction (t-1) = TRUE) $\wedge$ NEXT flag actions paction t

$\vdash_{def}$ (COMMITS flag actions [ ] t = FALSE) $\wedge$
  (COMMITS flag actions (ca :: commits_actions) t =
    ((COMMITS flag actions commits_actions t) $\vee$
     (COMMIT flag actions (CommitmentGuard ca) (CommitmentAction ca) t)))

In `COMMITS`, `CommitmentGuard` extracts the guard of a commitment pair, `ca`, that is the mental action. `CommitmentAction` returns the physical action that corresponds to that mental action.

Based on these definitions, the second if statement of `USER_CHOICE` in Figure 1 states that if a mental action is taken on a cycle then the next action will be the externally visible action it committed to:

else if (CommitmentMade (CommitmentGuards commitments) t)
  then (COMMITS flag actions commitments t)

Adding a commitment stage into the cognitive architecture is a first step to developing a more detailed model where a person changes their mind on seeing new information not initially available. Note, however, that whether this stage is modelled or not in no way affects our ability to detect the design problems discussed later in the paper. All instances of erroneous behaviour discussed would emerge from the cognitive architecture simply because of the arbitrary delay we allow before a person takes the action triggered by interface prompts.

The current version of the cognitive architecture is based on the implicit assumption that each mental action is associated with some physical action. Though this does not prevent us from specifying a purely mental action as, for example, one of the reactive actions, a better way of modelling purely mental actions would be to represent them as a separate argument of the `USER` definition.

2.5. Task-based termination behaviour

When the task the user set out to complete is finished we take the interaction to be terminated. This is how we define successful completion of the interaction. We assume the person enters an interaction with a goal to achieve. Task completion could be more than just goal completion, however. In achieving a goal, subsidiary tasks are often generated. For the user to complete the task associated with their goal they must also complete all subsidiary tasks. Examples of such tasks with respect to a ticket machine might include taking back a credit card or taking change.

The third if statement of definition `USER_CHOICE` specifies that a user will terminate an interaction when their whole task is achieved:

else if TASK_DONE (goalachieved ustate) (invariant ustate) t
  then (NEXT flag actions finishedpos t)

`TASK_DONE` asserts whether the task is completed at a given time or not. It requires arguments about the goal and an invariant. The relation argument `goalachieved` indicates over time whether the user's goal has been achieved or not. With a ticket machine, for example, this history function may correspond to the person's possessions, including the ticket. Often, though not always, subsidiary tasks could be expressed as interaction invariants [CuB01a]. The interaction invariant is an invariant at the level of abstraction of whole interactions. For example, the invariant for a simple ticket machine might be true when the total value of the user's possessions (coins and ticket) have been restored to their original value, the user having exchanged coins for tickets of the same value. Task completion involves not only completing the user's goal, but also restoring the invariant.

We can thus define task completion formally in terms of goal completion and invariant restoration. The task is completed at a time when both the goal and invariant history functions are true:

$\vdash_{def}$ TASK_DONE goal inv t = (goal t $\wedge$ inv t)

We assume that on completing the task in this sense, the interaction will be considered terminated by the user and only physical actions already committed to can then be performed. It is therefore modelled in the if-then-else staircase of `USER_CHOICE` to give it priority over other rules apart from committed actions. It is not treated as being a non-deterministically chosen action.

2.6. Goal-based Completion

Cognitive psychology studies have shown that users intermittently, but persistently, terminate interactions as soon as their goal has been achieved [ByB97]. This could occur even if the task, in the sense above, is not fully achieved. With a ticket machine this may correspond to the person walking away, starting a new interaction (perhaps by hitting a reset button), *etc.* Such behaviour is formalised as a `NEXT` rule. If the goal is achieved at a time, then the next action of the cognitive architecture can be to terminate the interaction:

$\vdash_{def}$ COMPLETION flag actions finished goalachieved ustate t =
    (goalachieved ustate t) $\wedge$ NEXT flag actions finished t

As this rule is combined disjunctively with other rules, its presence does not assert that it will be the action taken, just that according to the cognitive architecture there are cognitively plausible traces where it occurs. If it does fire, that may or may not correspond to an erroneous action as explored in Section 4.

2.7. Reactive Behaviour

A user may react to an external stimulus or message, doing the action suggested by the stimulus. For example, if a flashing light comes on next to the coin slot of a ticket vending machine, a user might, if the light is noticed, react by inserting coins if it appears to help the user achieve their goal. In a given interaction there may be many different stimuli to react to. Relation `REACT` gives the general rule defining what it means to react to a given stimulus. If at time `t`, the stimulus `stimulus` is active, the next action taken by the user out of possible actions, `actions`, at an unspecified later time, may be the associated `action`:

$\vdash_{def}$ REACT flag actions stimulus action t =
    stimulus t $\wedge$ NEXT flag actions action t

As there may be many reactive signals, the cognitive architecture is supplied with a list of stimulus-action pairs: [(s1, a1); . . . (sn, an)]. `REACTS`, given a list of such pairs, recursively extracts the components and asserts the above rule about them. The clauses are combined using disjunction, so they are non-deterministic choices. `Stimulus` and `Action` extract a pair's components.

$\vdash_{def}$ (REACTS flag actions [ ] t = FALSE) $\wedge$
  (REACTS flag actions (s :: st) t =
    ((REACTS flag actions st t) $\vee$ (REACT flag actions (Stimulus s) (Action s) t)))

2.8. Communication Goals

A user typically enters an interaction with knowledge of the task and, in particular, task dependent sub-goals that must be discharged — information that must be communicated to the device or items (such as coins) that must be inserted into the device. Given the opportunity, they may attempt to discharge any such *communication goals* [BlY98]. A communication goal specification is a pre-determined plan that has arisen from knowledge of the task in hand independent of the environment in which that task will be accomplished. In the sense of [BlY98] a communication goal is purely about information communication. Here we use the idea more generally to include other actions that are known to be necessary to complete a task. For example, when purchasing a ticket, in some way the destination and ticket type must be specified as well as payment made. The way that these must be done and their order may not be known in advance. However, a person entering an interaction with the aim of purchasing a ticket may be primed for these communication goals. Then, if the person sees an apparent opportunity to discharge a communication goal they may do so. Once they have done so they will not expect to need to do so again. No fixed order is assumed over how communication goals will be discharged if their discharge is apparently possible. For example, if a "return ticket" button is visible then the person may press that first if that is what they see first. If a button with their destination is visible then they may press it first. Communication goals are a reason why people do not just follow instructions.

We model communication goals as a list of (guard, action) pairs, one for each communication goal, in a similar way to reactive signals. The guard describes the situation under which the discharge of the communication goal appears possible, such as when a virtual button is actually on the screen. As for reactive behaviour, the communication goal rule, `COMMGOALER`, is supplied with a list of (guard, action) pairs: one for each communication goal.

Unlike the reactive signal list that does not change through an interaction, communication goals are discharged. This corresponds to them disappearing from the user's mental list of intentions. We model this by removing them from the communication goal list when done.

2.9. No-option-based termination behaviour

If there is no apparent action that a person can take that will help complete the task then the person may terminate the interaction. For example, if on a touch screen ticket machine, the user wishes to buy a weekly season ticket, but the options presented include nothing about season tickets, then the person might give up, assuming their goal is not achievable. We do not concern ourselves with precisely what behaviour constitutes terminating an interaction. Note that a possible action that a person could take is to wait. However, they will only do so given some explicit reason according to one of the other principles. For example, a ticket machine might display a message "please wait". If they see it, the person would have a cognitively plausible option: to wait. If there were no such message and no other reason to suggest that waiting indefinitely would lead to the person achieving the task, they could terminate the interaction. Thus the last rule in the cognitive architecture, `ABORTION`, models the case where a person can not see any plausible action to help them towards their goal. It just forms the negation of the guards of all the other rules, based on the same arguments.

3. Modelling specific devices

To illustrate how our approach can be used to establish correctness of designs or identify certain problems, we employ a generic example of user interaction with a machine that provides a number of services while requiring a chip-and-pin card as the means of payment. To receive a required service, the user must insert a card, enter its PIN, and choose a suitable option from the list of available services. The order in which these actions are to be performed may vary. The machine facilitates user actions by providing appropriate information in the form of light signals or merely by the presence/absence of appropriate slots and buttons; we will consider various design possibilities and their impact on user errors in the following sections. In response to the user actions, the machine provides the chosen service and returns the card. Since the precise nature of services does not make any impact on the classes of user errors considered, we do not model services in any detail in this paper.

Whilst this machine is simplified, its elements appear in real machines, interactions with which are potentially financially critical. It contains sufficient features of real interactive machines to be representative of a class of walk-up-and-use machines including vending machines and cash machines. The user must give up possessions to the machine and make selections and requests of the machine. The machine gives information about the task in hand to the user and feedback over the success of actions. All of these actions have been reduced to the simplest form, whilst still preserving their essence. Most importantly, the machine is complex enough that user errors could occur, and do occur, in real devices of similar complexity.

Using this generic example, we describe in Sections 4-7 a series of verification attempts. These help us to

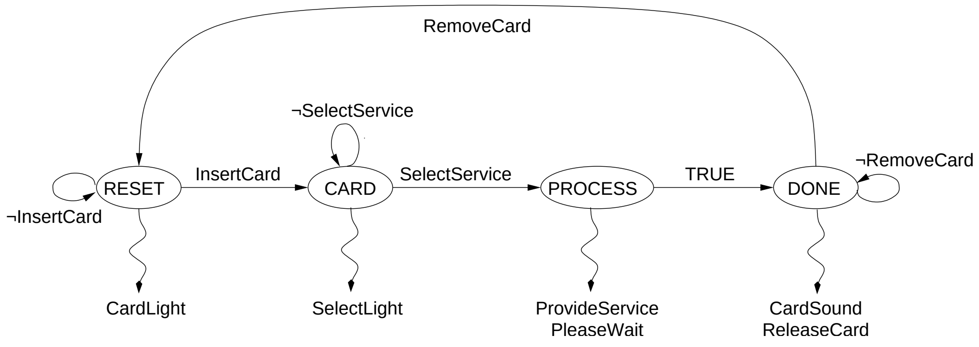

Fig. 2. Specification of a service machine

detect several potential problems, or their absence, such as termination related errors, reversing the order of actions due to communication goals, and machine delay errors. We also consider several kinds of interface. Naturally, some aspects of the machine and/or user model will vary in each of these cases. However, the structure and most of the features of both models remain the same in all specifications. As an introduction, in this section, we discuss in detail how a specification for the interactive system considered in Section 4 is developed within our approach. Later on specifications for the other interactive systems are derived as stepwise modifications of the original one. We start with the specification of the device. Then the instantiation of the cognitive architecture for this device is discussed. Finally, we finish this section with the formalisation of system properties we intend to verify.

3.1. Formal machine model

We formally specify our machine using a traditional finite state machine description 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. Initially our machines will have as outputs two lights to suggest the card insertion and the selection of a service, and sound to draw user attention to the card release. Furthermore, two additional outputs are signals to release the card and provide the service. Finally, the machine may display a message asking users to wait. Inputs correspond to the selection button being pressed and a card being inserted and removed. This gives a total of 9 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 our machines the values at each time are booleans in most cases. Informally, our finite state machine specifications can be represented by diagrams as shown in Figure 2. In this (and other diagrams), machine outputs are indicated to be false by omitting them.

We define a new type `mstate_type` to represent the machine state type. It is essentially just an abbreviation for the type of tuples of 9 boolean history functions. We define a series of accessor functions to obtain the values of particular components of the state. For example the function `InsertCard` extracts from the machine state the history function representing the card slot.

An enumerated type `MachineState` represents the states of our finite state machine (as opposed to the state representing the input and output values discussed above). We start with a machine of four states:

$$\vdash_{def} \text{MachineState} = \text{RESET\_STATE} \mid \text{CARD\_STATE} \mid \text{PROCESS\_STATE} \mid \text{DONE\_STATE}$$

The `RESET` state is the initial state. The `CARD` state is the state in which a card has been inserted but the selection button not pressed. In the `PROCESS` state a service request is processed, while the `DONE` state is the final state where the card is returned once the requested service has been provided. Later on when considering more complex machines, we will add more states to the type `MachineState`.

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

In the initial state of our first case study, the machine, with the insert card light lit, waits for a new user to insert a card. The `CardLight` history function has value true (`T`) to indicate it is lit in this state. None of the other outputs are set - their history functions have value false (`F`) at the time of interest:

$\vdash_{def}$ RESET_OUTPUTS (mstate: mstate_type) t =
  (SelectLight mstate t = F) ∧ (CardLight mstate t = T) ∧ (ReleaseCard mstate t = F) ∧
  (CardSound mstate t = F) ∧ (ProvideService mstate t = F) ∧ (PleaseWait mstate t = F)

If a card is inserted the machine moves to the `CARD` state in the next cycle, otherwise it remains in the `RESET` state:

$\vdash_{def}$ RESET_TRANSITION s (mstate: mstate_type) t =
  if InsertCard mstate t *then* s(t+1) = CARD_STATE *else* s(t+1) = RESET_STATE

These two relations are then combined as a conjunction into a relation `RESET_SPEC`. 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:

$\vdash_{def}$ MACHINE_SPEC s (mstate: mstate_type) =
  ∀t. *if* s t = RESET_STATE *then* RESET_SPEC s mstate t
    else if s t = CARD_STATE *then* ...

As an example, a full HOL specification of one of our machines is given in Appendix B.

3.2. Instantiating the cognitive architecture

To target our cognitive architecture 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. The user state includes history functions that record for each time instance whether the user has terminated the interaction and the communication goal list. Furthermore, it includes a history function for each mental commitment that the user makes to perform a physical action. In our examples, we consider three physical actions: insert card, select service and remove card. Finally, the user state includes, for each one of the user's possessions, three attributes: the value of that possession and two history functions that record whether the user has that possession and its quantity. Note that a possession does not have to be a physical object; we also model services as users' possessions. In our examples, the user will manage three possession items: a card before its use, a card after its use and a required service. We associate the same card with two distinct possession items for simplicity reasons. This allows us to model the card value as a constant. [1] Also, an accessor function for each part of the state is defined. For example, `UserCommgoals` extracts the communication goal list from the user state.

One of the arguments to be provided to the definition `USER` of Figure 1 is a list of all the possible user actions indicated by their history functions: the state extractor applied to the appropriate state tuple. We start from the following list of possible actions:

[UserFinished ustate; CommitInsert ustate; CommitSelect ustate; CommitRemove ustate;
 InsertCard mstate; SelectService mstate; RemoveCard mstate; Pause mstate]

The next argument is a list of commitments, each one being a pair that associates a mental user action with the corresponding physical action. In our examples, this will be the following list:

[(COMMITINSERT, INSERTCARD);
 (COMMITSELECT, SELECTSERVICE);
 (COMMITREMOVE, REMOVECARD)]

where `COMMITINSERT`, `INSERTCARD`, *etc.* are the positions of the corresponding actions in the full action list.

Further, one must link the attributes, such as the quantity of an object possessed and its value, to the events of taking and giving up that possession. In our examples such events are the user actions `InsertCard mstate`, `RemoveCard mstate`, and the machine output `ProvideService mstate` which, for simplicity, is identified with the user action of receiving that service. A relation `MAKE_POSSESSIONS` gathers this information into an appropriate form, given the corresponding history functions:

[1] This is a restriction of the current version of the cognitive architecture.

```
MAKE_POSSESSIONS HasCardBefore InsertCard CountCardBefore (ValueCardBefore ustate)
                 ReceivedService ProvideService CountService (ValueService ustate)
                 HasCardAfter RemoveCard CountCardAfter (ValueCardAfter ustate)
```

Note that we identify the card value with the balance on the associated bank account at a particular time instance.

The cognitive architecture is also provided with three accessor functions to the user state. Since the list of communication goals may change over the course of time, it is represented as a part of the user state accessed by the state extractor `UserCommgoals`. One must also specify accessor functions, `UserFinished` and `ReceivedService`, that indicate, respectively, when the user has terminated the interaction and the user's main goal in taking part in that interaction.

Finally we must provide the invariant that the user wishes to restore by the end of the interaction. In our examples this is based on the value of the user's possessions which is calculated from possession quantity and value attributes using a relation `VALUE`. After interacting with our machine, users will not wish the value of their total possessions to be less than they were at the start (time 1). This is described by a history relation `VALUE_INVARIANT`:

$$
\begin{aligned}
&\vdash_{def} \text{VALUE\_INVARIANT possessions ustate t} = \\
&\qquad (\text{VALUE possessions ustate t} \geq \text{VALUE possessions ustate 1})
\end{aligned}
$$

This relation will not hold throughout the interaction. When the card is inserted the value will drop and will only return to its initial value once the service has been provided and the card has been taken back. It is only an invariant in the sense that it must be restored by the end of the interaction.

Instantiating the cognitive architecture with all these arguments yields only a partially specified user model (λ`commgoals reactive. MACHINE_USER commgoals reactive ustate mstate`), as in the above discussion we have not specified reactive actions with which the user may respond to the machine outputs, and the initial list of communication goals with which the user enters the interaction. Since these vary in each case study, they are defined in the corresponding sections. Providing them as arguments to the above (partial) user model will yield specific user models, targeted for the interaction with the corresponding devices specified as `MACHINE_SPEC s mstate`. Appendix C gives in full one possible instantiation of the cognitive architecture for the machine defined in Appendix B.

3.3. Correctness properties

Generally, we are interested in two kinds of correctness properties. Firstly, we want to be sure that, in any possible interactive system behaviour, the user is eventually able to achieve the main goal associated with the task performed. Secondarily, in achieving that goal, subsidiary tasks are often generated. For the user to complete the task associated with their goal they must also complete all subsidiary tasks. In the case of the interaction between our user and machine models, these correctness properties are represented by an assertion of the following form:

$$
\begin{aligned}
&\forall \text{ustate mstate s.} \\
&\quad \text{MACHINE\_USER ustate mstate} \wedge \text{MACHINE\_SPEC s mstate} \supset \\
&\qquad (\text{ValueCardBefore ustate} = \text{ValueService ustate} + \text{ValueCardAfter ustate}) \wedge \\
&\qquad (\text{s 1} = \text{RESET\_STATE}) \wedge \neg(\text{UserFinished ustate 0}) \wedge (\text{HasCardBefore ustate 1}) \wedge \\
&\qquad \neg(\text{CommitInsert ustate 0}) \wedge \neg(\text{CommitService ustate 0}) \wedge \neg(\text{CommitRemove ustate 0}) \supset \\
&\qquad\qquad (\exists \text{t. ReceivedService ustate t} \wedge \text{VALUE\_INVARIANT possessions ustate t})
\end{aligned}
$$

The conclusion of this assertion states that there will exist some time at which the user has received the service (i.e., has achieved their main goal) and the interaction invariant `VALUE_INVARIANT` has been restored (i.e., the subsidiary task has been completed). However, this is only guaranteed for users whose behaviour does not violate the specification `MACHINE_USER`. We also assume that the interaction starts when the service machine is in its reset state, and that the user does not consider the interaction to be finished, has a card to pay for the requested service, and has not yet made any mental commitments to physical actions.

We prove such assertions by essentially 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 user's actions, and the machine specification to deduce facts about the machine's actions. 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. 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.

Proving such a correctness theorem does not mean that users interacting with the machine will never make an error. The proof, however, guarantees that no user will make the classes of errors for known cognitive causes as specified in the cognitive architecture. On the other hand, a failed attempt to prove such a theorem is likely to indicate potential systematic problems in a machine design.

In the following four sections, we describe specific case studies based on the partial user model and variations of the machine model we have just introduced.

4. Naïve user

In the previous section we described a partially instantiated user model for our device. However, we must still specify the communication goals and reactive behaviour for each specific user and device combination to allow the analysis of the interactive system to proceed.

We start from the simplest user model, a *naïve* user, whose behaviour can be viewed as a reaction to machine signals, meaning the user has no preconceived knowledge of the task. Admittedly, a user willing to blindly follow machine instructions is not very realistic. Nevertheless, it is the model of user behaviour often assumed by designers. Subsequent sections consider more realistic user models.

Since a naïve user would enter an interaction without any information to communicate to the machine, the argument for communication goals in the cognitive architecture is instantiated with the empty list. On the other hand, we must provide a nonempty list of reactive actions. This is a list pairing observations with actions that they prompt the user to make. In the designs of this section, there are four such actions: the `CardLight` prompts users to insert a card if they have got one; the `SelectService` light prompts users to select one of the available services; the `CardSound` prompts users to remove the card and so trigger the sensor on the card slot; finally, the `PleaseWait` message suggests to users to wait while their request is processed:

```
[(CardLight mstate AND HasCardBefore ustate, COMMITINSERT);
 (SelectService mstate, COMMITSELECT);
 (CardSound mstate, COMMITREMOVE);
 (PleaseWait mstate, PAUSE)]
```

Next we investigate what impact the design of our service machine makes on the success of interactions performed by the naïve user. We consider two device designs. In both, the order of actions in an interaction is fixed: first the machine prompts the user to insert the card. After this it suggests to the user that they select a service. We show that one of the designs is prone to a widespread termination related error, the post-completion error, which occurs when a person terminates an interaction upon achieving the main goal of the task (getting cash from an ATM, for example) before completing the associated subsidiary tasks (removing the card).

4.1. Post-completion error

We start with the design where the machine provides the requested service once the user has inserted a card and made a selection (see Figure 2). The state in the machine operation when the service is provided is modelled by `PROCESS_STATE` whose outputs are defined as follows:

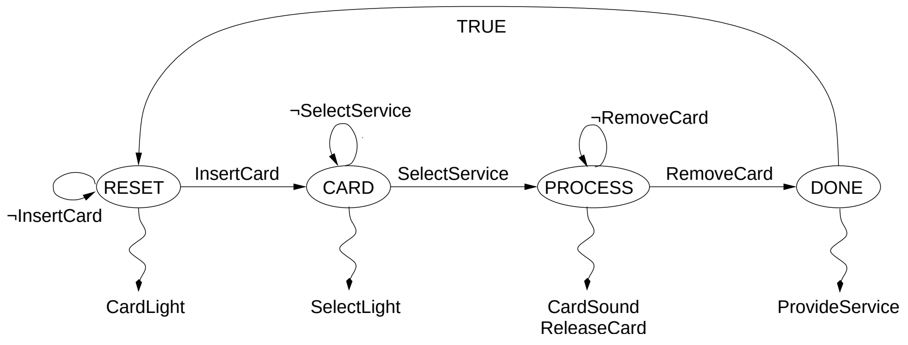

Fig. 3. Specification of an improved design

$\vdash_{def}$ PROCESS_OUTPUTS (mstate: mstate_type) t =
(SelectLight mstate t = F) $\wedge$ (CardLight mstate t = F) $\wedge$ (ReleaseCard mstate t = F) $\wedge$
(CardSound mstate t = F) $\wedge$ (ProvideService mstate t = T) $\wedge$ (PleaseWait mstate t = T)

`ProvideService` is true (by definition) in this state. At the same time the `PleaseWait` message tells the user to wait for the card to be released. At the next time instance, the machine makes a transition into the `DONE_STATE` in which the card is released while the `CardSound` prompts the user to take the card back.

However, an attempt to prove the correctness theorem from Section 3.3 fails with this design. We are left with the goal of establishing `HasCardAfter ustate` for some future time instance `t'`, while the user has already finished the interaction (i.e. `UserFinished ustate` holds at the current time instance `t`). This may happen since, upon receiving the requested service, the user has already achieved their goal and therefore may leave rather than wait for the card to be released. This is an example of a post-completion error. The design is flawed because it does not prevent users from making such errors.

Note that there is no *explicit* design decision as regards the requirement for the card to be returned. It is only implicitly captured by the interaction invariant in terms of the abstract requirement concerning the value of user possessions. Furthermore, the interaction invariant conceptually does not belong to the cognitive model. It appears in the HOL version mainly to make it a "visible" part of the instantiation process. In a parallel development of our work [RCB$^+$07a] using SAL, the cognitive architecture dispenses with it: it is formulated, if needed, only within the correctness properties during the actual verification. In any case and notwithstanding any design decisions, only the potential to make post-completion errors, not the errors per se, is inevitable. Whether the potential comes true depends on the environment (i.e. the device design) the model is in. Post-completion errors can be eliminated in an appropriate design. Our approach allows one to distinguish good, in this respect, designs from bad ones, based on the verification of appropriate correctness properties.

In the next subsection we consider a modified design of the service machine in an attempt to eliminate systematic post-completion errors from this interactive system.

4.2. Improved design

One way in which we can correct the usability problem of the machine is to change its design so that it releases the card and then, before providing the service, waits until the user removes the card from the card slot. 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. This is because a post-completion error can only occur if the device allows the goal to be achieved before the invariant is restored. The new design does not allow this to happen. It involves only minor changes to the machine specification as shown in Figure 3. In particular, the machine releases the card in the `PROCESS_STATE`:

$\vdash_{def}$ PROCESS_OUTPUTS (mstate: mstate_type) t =
(SelectLight mstate t = F) $\wedge$ (CardLight mstate t = F) $\wedge$ (ReleaseCard mstate t = T) $\wedge$
(CardSound mstate t = T) $\wedge$ (ProvideService mstate t = F) $\wedge$ (PleaseWait mstate t = F)

Upon the user removing the card, the machine makes a transition into the `DONE_STATE` in which the requested service is provided. The new machine has the same buttons and lights and the user reacts in the same way to those signals. With the new design, proof of the correctness theorem can be completed. Now, when the goal-based termination clause in the user model is enabled, the invariant has already been restored so the normal termination condition is triggered.

We have thus proved that a single class of common errors with a specific cognitive cause (post-completion errors) have been eliminated from the modified design. However, our proof relies upon a very simple, it may be argued too simplistic, instantiation of the cognitive architecture. In the following section, we consider more realistic users with some knowledge relevant to their goals. Note, however, that this additional knowledge would not preclude them from making the post-completion error identified in the previous section. The original design is still flawed, even when combined with the changed user model, as was shown by a verification attempt in the same way.

5. A user with communication goals

People enter an interaction with some knowledge of the task that they wish to perform. For example, on approaching our service machine, a person would know that they must communicate their choice of service to the machine. Similarly, they know that payment must be provided by some means (a chip-and-pin card in our example). Because of this knowledge, as anecdotal experience and research in cognitive psychology indicate [Rea90], a user may take an action as a result of seeing an apparent opportunity for it, irrespective of any guidance the machine is giving. In this section, we consider a user model that captures this kind of behaviour, by treating card insertion and service selection as communication goals.

5.1. Reversing the order of actions

Like reactive actions, communication goals are modeled as guard-action pairs. The guard describes the situation under which the discharge of the communication goal can be attempted. The action is the user action that discharges the communication goal. In our example, we assume that the user has two communication goals: to select a service and to pay for it (by inserting a card). Thus we instantiate the cognitive architecture with the following list of communication goals:

[(HasCardBefore ustate, COMMITINSERT); (λt.T, COMMITSELECT)]

Note that now neither communication goal is guarded by machine signals as was the case with the reactive actions in the previous section. Therefore, service selection is always possible (from the user's point of view), whereas paying for the service is possible only when the user has a card amongst their possessions.

Our first user model is that of a walk-up-and-use novice. Once an action from the communication goal list has been performed, such a user may believe that necessary information has been supplied to the device, and consequently will no longer react to the relevant machine prompts. This is reflected by the list of reactive actions that is provided as an argument to the cognitive architecture. It contains only a response to the `CardSound` signal prompting the removal of the released card and the `Pause` action:

[(CardSound mstate, COMMITREMOVE); (PleaseWait mstate, PAUSE)]

With this user model, we attempt to prove the correctness theorem for the originally successful design in Figure 3. Unfortunately, our attempt fails in this case. We are left with the goal of establishing that the user performs the action `SelectService ustate` at some future time instance `t'`, while the interaction has been already finished at the current time instance `t`. This happens because the user model allows users to do either of the communication goals first. If they made the selection first, this action would be removed from their list of goals: they would believe the selection has been made. On inserting a card to complete their other goal, there would no longer be anything in the user model to compel them to press the selection button, which is the action awaited by the machine. The user has made an order error. In fact, there would be no other actions enabled so that the only option available in the user model would be to terminate the interaction.

In terms of the verification, termination before the user has received the requested service is an error. There are at least two plausible angles from which to interpret this situation. The first one is to recognize it as a design flaw. One then can look for changes in the design to eliminate the problem. In Section 5.3 we describe a modified design of the service machine, which is no longer prone to error due to premature termination for this reason. The second point of view is based on the fact that most real users would eventually work out the problem and go on to complete the interaction. To capture such user behaviour, we consider a different instantiation of the cognitive architecture in the next section.

5.2. A more sophisticated user

The problem with the current user model is that it ignores machine prompts to perform an action (from the user's point of view, to repeat it) once the same action has apparently been executed as a communication goal. In our case, it would seem natural to expect that users, even though they might perform the `SelectService` action too early, will recover from this error and repeat the same action as a response to the `SelectLight` which prompts them to make a selection. Whether or not these expectations are cognitively plausible can be determined by usability experts. Provided they are, the cognitive architecture can be instantiated with the same list of reactive actions as in Section 4:

[(CardLight mstate AND HasCardBefore ustate, COMMITINSERT);
(SelectService mstate, COMMITSELECT);
(CardSound mstate, COMMITREMOVE);
(PleaseWait mstate, PAUSE)]

Appendix C shows in full how this particular user model is derived as an instantiation of the cognitive architecture.

Now, even if the user tries to make the selection first as an attempt to discharge one communication goal, the situation discussed in Section 5.1 is no longer possible. Since selection is also on the list of reactive actions, the model allows users to respond to the machine signal prompting them to select a service. As one would expect, a verification attempt for the modified user model is successful: the correctness theorem is proved. However, as explained earlier, the verification failure of Section 5.1 can also be considered as a design flaw. Instead of modifying the user model one can look for changes in the design to eliminate the problem. We consider this next.

5.3. Permissive design

The reason for the verification failure in Section 5.1 is that the machine design we used relies on a specific order for the execution of communication goals. Namely, first a card must be inserted, and only then may a service be selected. One way of solving the problem is to make the design more permissive [Thi01], at least for communication goals: the design should accept any order in which users execute them. In our case, this requires the addition of a new state, `SERVICE_STATE`, into the device specification:

$\vdash_{def}$ MachineState = RESET_STATE | CARD_STATE | SERVICE_STATE |
PROCESS_STATE | DONE_STATE

In the initial state, the machine now accepts both user actions, `InsertCard` and `SelectService` (see Figure 4). The transitions from the initial state are now specified as follows:

$\vdash_{def}$ RESET_TRANSITION s (mstate: mstate_type) t =
  if InsertCard mstate t *then* s(t+1) = CARD_STATE
  else if SelectService mstate t *then* s(t+1) = SERVICE_STATE *else* s(t+1) = RESET_STATE

In the `CARD_STATE`, the machine waits for the user selecting a service as before, whereas in the `SERVICE_STATE` it waits for the card insertion before processing the user request. Thus, in either of the two states, the machine waits for the user executing the remaining communication goal. As a result of these changes in the design, the correctness theorem now can be proved with both previous user models: that which regards card insertion and service selection as communication goals exclusively, and that which allows for a possibility of the same actions taken as a response to the reactive signals from the device.

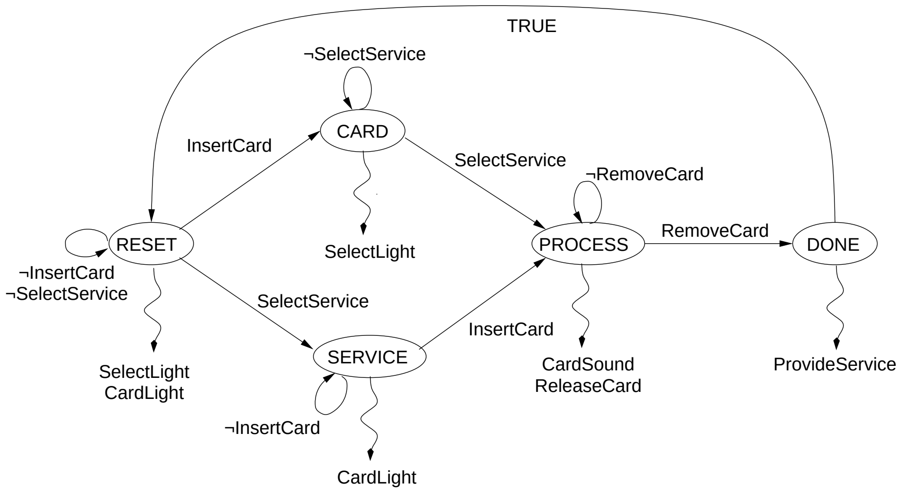

Fig. 4. Specification of a permissive design

6. Premature termination

The designs we have considered so far are idealized in one sense at least: the machine responds to the user actions instantaneously. For example, upon receiving a card and a user's selection, it fulfills the request at the next time instance. In reality, both verifying the card and providing service will take some time and may cause a delay in machine response. Next we modify our latest design so that it includes device delays.

6.1. Device delays

Our new machine specification includes an additional machine state, `WAIT_STATE`. The machine makes a transition into this state from the initial state upon receiving a card and a user's selection (see Figure 5). It makes a transition from this state into the `PROCESS_STATE` once the processing of the user request has been finished. We assume that processing user requests can take some time, thereby causing delays with device responses. Since we do not target timing properties in this case study, it makes no real difference how many time instances device delays can take as long as they do not continue forever. For simplicity, in this paper we assume that all delays are of one time instance:

$\vdash_{def}$ WAIT_TRANSITION s (mstate: mstate_type) t = (s(t+1) = PROCESS_STATE)

The HOL specification of this machine is given in Appendix B.

Assuming the simpler user model (see Section 5.1), an attempt to prove the correctness theorem reveals a problem with this design. Upon supplying a card and making a selection, the user has no outstanding communication goals. Furthermore, there are no signals from the machine to react to, even though it takes time to process users' requests. The only option available to the user is to terminate the interaction. This models the situation where a user suspects a machine has hung, so terminates prematurely (perhaps by randomly hitting buttons). Note that the same error would also be possible with the model of a more sophisticated user from Section 5.2. Similar design problems would also be detected by our verification approach in the case when a machine seemingly provides some options, but the user model cannot use them because they are not cognitively plausible with respect to the task performed (this might happen due to, for example, wrong button labels). Next we modify the flawed design to eradicate the revealed problem.

6.2. Design with feedback

It is rather obvious that the cause of the problem is the absence of feedback of any kind from the device during delays in its responses. The solution to the problem is to make a simple modification to the design in Figure 5 so that a wait message is displayed while the delay occurs:

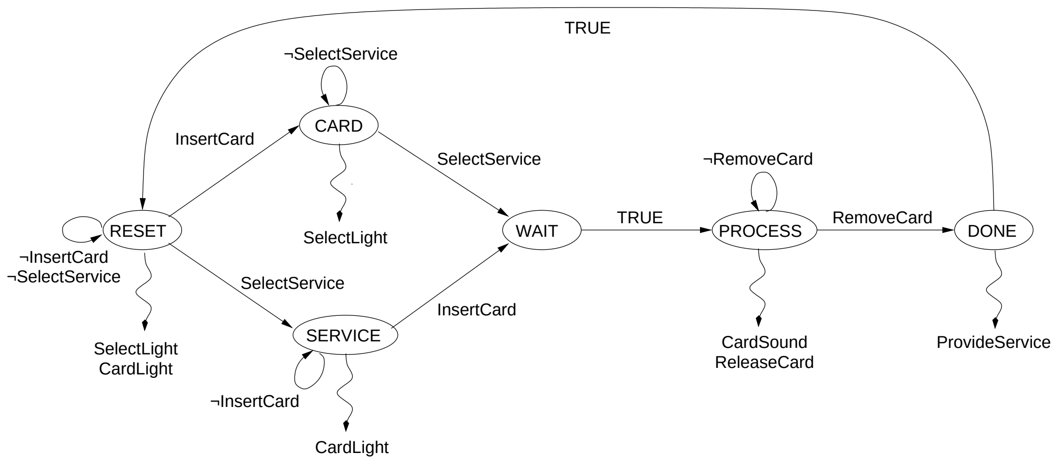

Fig. 5. Specification of a design with delays

$\vdash_{def}$ WAIT_OUTPUTS (mstate: mstate_type) t =
(SelectLight mstate t = F) ∧ (CardLight mstate t = F) ∧ (ReleaseCard mstate t = F) ∧
(CardSound mstate t = F) ∧ (ProvideService mstate t = F) ∧ (PleaseWait mstate t = T)

Now, since the user model includes a pause action, users have an option while their requests are processed; when the `PleaseWait` signal is on, they can simply react by doing nothing. The feedback from the machine prevents users from terminating the interaction too early. As a result, the correctness theorem for this design has been proved.

With the new design, a delay occurs when all the communication goals have been discharged. The presence of feedback is sufficient in this situation, since to pause is now the only option available to the user. In general, however, the feedback from a machine does not guarantee by itself the success of an interaction. Consider, for example, a similar design in which a delay occurs after the insertion of a card when a service has not necessarily been selected. In this case, users would have a choice between pausing and discharging their outstanding communication goal, service selection. Since neither of the actions has a priority over the other in the user model, it is possible that the communication goal is discharged. If the design permits this, the user choice will be registered, and the correctness of the interactive system could be proved. If not, we have the situation discussed in Section 5.1. With the simpler user model verification would fail, since there are no enabled actions in this state, which represents the user belief that all required actions have been performed. On the other hand, the correctness of the design could be proved for the more sophisticated user model, which represents users willing to repeat their actions when prompted by a machine.

7. Hard and soft-key interfaces

In previous sections, we have been looking at a range of errors and alternative designs that avoids the identified errors. Here we investigate more complex styles of interface. In particular, we focus on soft-key interfaces. We show how the hard or soft-key nature of the interface can be modelled explicitly in a way that allows us to verify either kind of interface. We also show that our methodology can detect the presence of further classes of design errors that arise as a result of soft-key interfaces.

When a user enters an interaction they will try to take actions that will help them achieve their goal. However, they will only take an action if they see an opportunity to do so. For the user to be able to rationally take an action the device needs to present appropriate information about the nature of inputs at the appropriate time. The user acts on the information presented. In previous case studies, we made no attempt to model the way that the users know what an input does. We assumed implicitly that inputs were labelled so that the user was aware of their effect. In essence we assumed the buttons were permanently and clearly labelled with their meaning. In this section, we introduce into our modelling approach some information about the interface itself.

To model both soft and hard-key signals, we associate with each (virtual) input of the device an additional

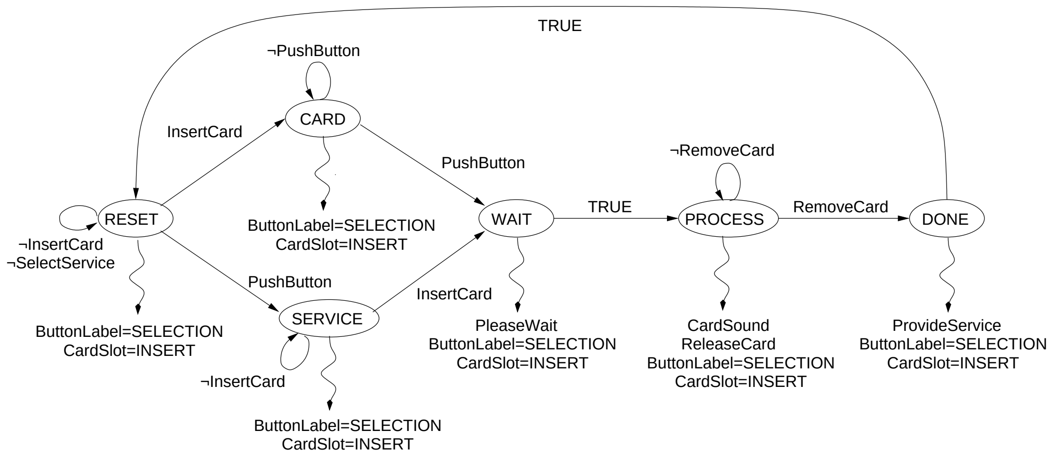

Fig. 6. Specification of a design with a hard-key interface

output label indicating its use, even if no such label exists (in which case the output would represent either a blank label or the absence of the button in question). The device specification specifies the values of these label signals in the same way as for other outputs. Such a label does not necessarily need to be in the form of text. It could be a picture or icon, or it could be the artifact itself. For example, normally it is clear what a coin slot is for without a label. Its very shape advertises its purpose. Deciding whether a given artifact, text or button does in fact advertise a given message is beyond the scope of our formal verification. We assume that a HCI expert would analyze the device to determine what messages are communicated by analyzing the specific task.

7.1. Hard-key interface

Labels (even permanent ones) are, in effect, an output from the machine. They can therefore be modelled as history functions from time to label values (text, a picture, icon, *etc.*). The output for a permanently printed label indicating the use of a given hard-key remains the same throughout an interaction, while the hard-key itself is always available.

To illustrate these ideas we use our generic service machine. Its design is essentially the same as in Section 6.2. In that design, however, we assumed implicitly that machine inputs are marked with appropriate labels. The difference here is that we explicitly model the interface. Recall that our machine has a card slot and a button for making selections. The label values associated with these inputs are represented as

$\vdash_{def}$ LabelValues = INSERT_CARD | MAKE_SELECTION

Next we specify when the coin slot and selection button communicate their use. As it is a hard-key interface we assume both are always available and have permanent labels indicating their use. There are outputs from the device permanently presenting this information as can be seen in Figure 6. In terms of machine specification, this means that the corresponding outputs are constant history functions: [2]

(CardSlotLabel mstate t = INSERT_CARD) $\wedge$ (ButtonLabel mstate t = MAKE_SELECTION)

To investigate our design, we can use either of the two user models from Section 5. They just need simple adjustments to take the explicit modelling of the machine interface into account. Namely, the guards of the communication goals (and the corresponding reactive actions, if the second model is used) ought additionally to include label signals:

[((λt. CardSlotLabel mstate t = INSERT_CARD) AND HasCardBefore ustate, COMMITINSERT);
((λt. ButtonLabel mstate t = MAKE_SELECTION), COMMITSELECT)]

[2] We also rename the variable `SelectService` to `PushButton` to reflect the changes made.

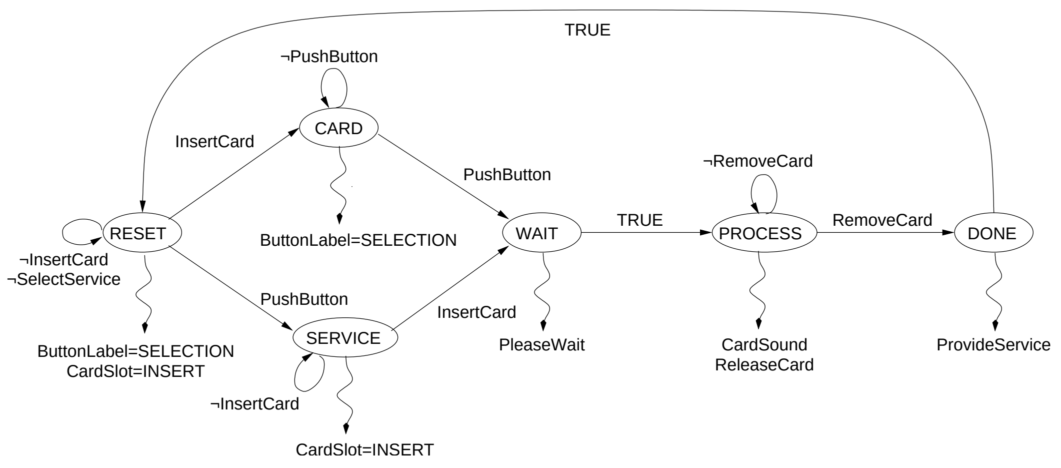

Fig. 7. Specification of a design with a soft-key interface

Unsurprisingly, the verification of this hard-key interface design composed with either of the two adjusted user models is successful. The design does not really provide new functionality. Our current setting has simply made explicit the assumptions implicit in earlier case studies concerning the machine interface. In the next section, we consider a slightly more complicated interface.

7.2. Soft-key interface

In contrast to hard-keys, a soft-key can disappear or change its functionality (the associated label) during an interaction. A simple example of an interface with changing functionality is a cash machine that has a series of buttons round a screen. Labels on the screen indicate what each button does at any particular time during the interaction. For example, at one point the buttons are used to indicate the service the user requires, whilst later in the interaction the same buttons are used to indicate the amount of money desired. This setting more precisely could be described as a soft-label interface as opposed to say a soft-button interface of a touch screen where, in addition to changing labels, a button itself may disappear. Software based graphical user interfaces provide plenty of examples of disappearing (and, possibly, later reappearing) buttons or other types of inputs. Within our modelling approach, all soft-keys are easily represented by history functions that change their value from time to time. In this section, we consider a machine identical to that of the previous section but with a soft-key interface.

Thus, our next machine has the same specification as the previous one except for its interface. Rather than having permanent buttons, it has a touch screen (soft-button) interface. Rather than having permanent labels to indicate actions, text messages appear on the screen. To model this interface, we extend the set of labels as follows:

$\vdash_{def}$ LabelValues = INSERT_CARD | MAKE_SELECTION | NONE

The value `NONE` is designated for those states where a particular machine input (button) is unavailable or unlabelled. Note that, by using the same value for both cases, we effectively identify them. If necessary, the distinction can be made by further extending the set of label values.

Now the machine interface is modelled by setting the labels to `NONE` in states where those buttons and/or messages are not presented. The card slot is covered and so not available in periods when the machine will not accept cards. The finite state machine specification is given in Figure 7. In this informal diagram, labels (machine outputs) are indicated to be `NONE` by omitting them. In the formal specification, they are explicitly set to either `NONE` or an appropriate value for each state. In particular, the card slot exists and so is labelled as `INSERT_CARD` in the `RESET` and `SERVICE` states. Similarly the selection button exists and so is labelled as `MAKE_SELECTION` in the `RESET` and `CARD` states. Modelling the labels related to inputs in this way, we can use them in the user model to restrict when a user might rationally take the corresponding action. In fact, the user models we described in the previous section can also be used to verify this soft-key based design.

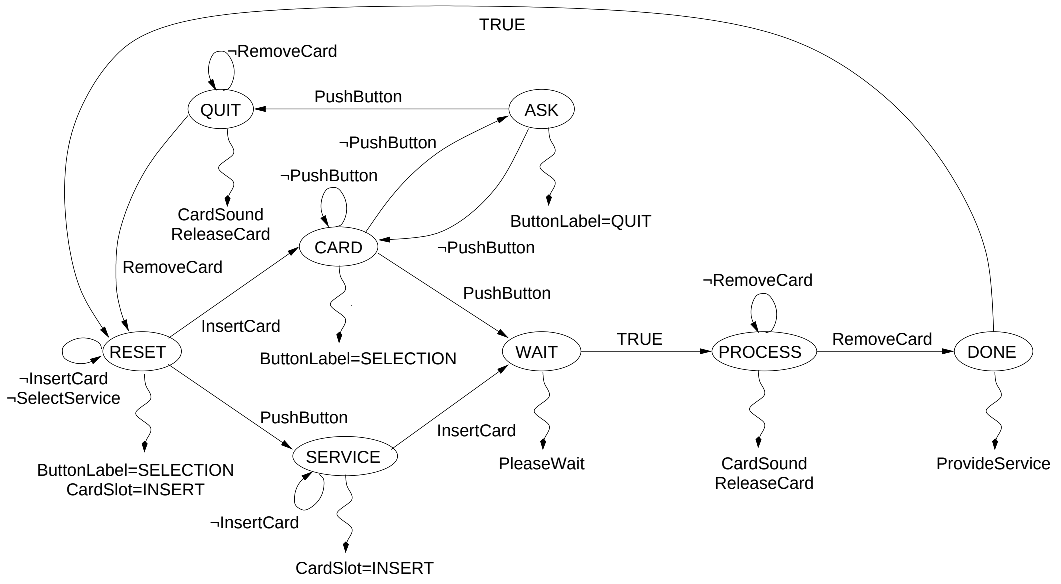

Fig. 8. Specification of a design with indirect interface changes

Furthermore, the verification succeeds using the same proof script as developed for the verification of the hard-key interface.

7.3. Indirect interface changes

In this section, we consider more advanced soft-key interfaces. Namely, we allow a button to be associated with different functionality (labels) during an interaction. We will see that this may lead to a new class of design errors.

To modify the functionality of our machine, let us first consider the `CARD` state of previous designs. Recall that it is a state where a card has been inserted and the user is about to decide what service is needed. Note that in all previous designs, the machine waits for this decision indefinitely. Clearly, this is an unrealistic scenario. For security reasons, in any reasonable design, the user, after some period of inactivity, would be asked whether the interaction should continue. If so, the machine would return to the waiting state, otherwise, it would return the card and terminate the interaction. To be able to model such functionality, we add two new states into the finite machine specification of our device:

$$\vdash_{def} \text{MachineState} = \text{RESET\_STATE} \mid \text{CARD\_STATE} \mid \text{ASK\_STATE} \mid \text{QUIT\_STATE} \mid \\ \text{SERVICE\_STATE} \mid \text{PROCESS\_STATE} \mid \text{DONE\_STATE}$$

In the `ASK` state, the user has the option of terminating the interaction by telling the machine to move into the `QUIT` state where the card is released.

It is obvious that the machine interface needs extensions to support the additional functionality. Let us assume that the designers of our machine wanted to keep the machine interface as minimalist as possible. In such a case, the same button could be used, in different states, for two purposes — making selections (as in the previous design) and telling the machine to quit. To achieve this, we only have to mark the button with appropriate labels. We thus introduce a new label `PRESS_TO_QUIT`:

$$\vdash_{def} \text{LabelValues} = \text{INSERT\_CARD} \mid \text{MAKE\_SELECTION} \mid \text{PRESS\_TO\_QUIT} \mid \text{NONE}$$

The finite state machine specification is informally shown in Figure 8. In the formal specification, the button is labelled as `PRESS_TO_QUIT` in the `ASK` state, `MAKE_SELECTION` in the `RESET` and `CARD` states, and is unavailable (its history function has the value `NONE`) in all the other states. In the `QUIT` state, the only active machine output is that of the card release. The specification of machine outputs in all the other states remains the same as in the previous design.

How does the machine with the added functionality operate? Now there are two possible outcomes when the machine is waiting for user selection (`CARD` state). First, the user may decide about the required service and push the button to make the selection known to the machine. Second, the machine may, as explained earlier, try to determine whether the user is interested in continuing the interaction by moving into the `ASK` state. These two transitions are formally specified as follows:

$\vdash_{def}$ CARD_TRANSITION s (mstate: mstate_type) t =
  if PushButton mstate t *then* s(t+1) = WAIT_STATE
    else (s(t+1) = ASK_STATE) $\vee$ (s(t+1) = CARD_STATE)

Note that the choice between moving into the `ASK` state and remaining in the `CARD` state is nondeterministic in this specification. In real machines, the transition into the `ASK` state would be conditioned by the period of user inactivity. However, as mentioned earlier, the verification of timing properties is out of the scope of this paper. Our specification thus states that such a transition is possible in principle, without providing any detail about when.

Finally, the user may choose to quit the interaction when the machine is in the `ASK` state:

$\vdash_{def}$ ASK_TRANSITION s (mstate: mstate_type) t =
  if PushButton mstate t *then* s(t+1) = QUIT_STATE *else* s(t+1) = CARD_STATE

Otherwise, the machine returns into the `CARD` state and provides the user with the option of making a service selection again. According to this definition, the user has only one time instance to quit the interaction. However, such specification suffices within our modelling approach, since the user goals formalised in our correctness theorem are incompatible with quitting an interaction early. Hence, the design can be verified to be correct only if there is no chance, even for a single time instance, for the user to terminate an interaction prematurely.

The user model for the verification of this design is essentially the same as in Section 5.1 (and earlier in this section). We only add one new action into the list of reactive actions:

[(λt. ButtonLabel mstate t = PRESS_TO_QUIT, PAUSE);
 (CardSound mstate, COMMITREMOVE);
 (PleaseWait mstate, PAUSE)]

This action states that users pause when the button is labelled as `PRESS_TO_QUIT`. This is so, since the overall user goal is to receive some service. Quitting an interaction before this goal is achieved would be cognitively implausible behaviour.

Will our verification be successful for the system composed of the machine and the user model we just specified? Intuitively, it would seem so. The user model specifies that users will pause when provided with the chance to terminate an interaction too early. In other respects, the user model and the machine are essentially the same as in the successful verifications earlier in this section. Unfortunately, the actual verification attempt fails!

Upon inspecting this failure more closely, it is clear that it is actually possible for the machine to move into the `QUIT` state, from where there is no way users can achieve their goal. How can this transition happen? The only possibility is that the user has pressed the button in an attempt to discharge one of the communication goals. Since that goal is guarded by the condition `(`λ`t. ButtonLabel mstate t = MAKE_SELECTION)`, the user must have made the decision when the button was labelled as `MAKE_SELECTION)`, which means the machine must have been in the `CARD` state. As a result of the execution of that goal, the user would commit to making a selection by pushing the button. In reality, however, the commitment to an action and the physical action itself are not simultaneous. Our user model captures this fact, and thus the following scenario is possible. The user commits to making a selection at time `t`, whereas the machine at the same time instance moves from the `CARD` state into the `ASK` state. At time `t+1`, the button is labelled as `PRESS_TO_QUIT` and it serves as the means to finish the interaction. However, the commitment already made, users cannot stop the physical action they committed to, even though they may realize it is wrong at that particular time instance. Thus the button is pressed at time `t+1`, which leads to the design error we identified.

How can this design problem be rectified? The simplest way would be to use two different buttons — one for making selections and one for terminating an interaction. Another possibility is to change the functionality of the (single) button so that, in the `ASK` state, it is no longer associated with the termination of an interaction. Instead, its effect could be made less harmful, when used with a different intention, by associating it with the return to the `CARD` state from where the user can make selections again.

Admittedly, our example is simple, and the consequences of the erroneous action are merely annoying. However, the underlying reason for this error is a widespread phenomenon and can have much more disastrous consequences. This reason is an indirect mode change taking place. By that we mean situations where changes in the state of a device (and in the functionality of its interface) are internally determined by the machine and, therefore, may occur as a surprise to users, not in response to their earlier action. Problems of this type have been identified in various interactive systems including cash machines, email systems, and sophisticated control systems in air traffic control, flight monitoring on airplanes, telecommunications and robotics [Rus02, Bil96, Bre03, Lan01].

8. Verification methodology

In this paper we have outlined an approach to verifying interactive systems. The error analysis here was mostly driven by specific examples. Clearly future work is needed before our approach could be used as an industrial methodology. That, however, does not preclude the contribution of this work in demonstrating the possibilities of such a methodology.

In principle, one can consider at least three different aspects related to the verification methodology used in this area. At a concrete level, it concerns how to specify and verify a specific task performed on the specified interactive device by a certain (fixed) class of users. Our approach supports specifying such tasks by providing a pattern in the form of a generic user model formalised within a verification system. The verifier only provides the concrete parameters of this pattern: communication goals, reactive actions, main task goal, etc. Verification is then supported by providing a pattern for the correctness (usability) property and the related proof tactics to establish it. A failed proof attempt usually indicates the design features that cause the problem. Verification can then be repeated with a modified design. Contrary to the error driven approach of Fields [Fie01], we are not modelling human errors per se. Therefore, a successful proof indicates that the property verified is correct (task goal is achieved) for any user behaviour emerging from the formalised principles of cognition. The formalisation of these principles is inspectable and the assumptions about the classes of behaviour reasoned about made clear. This aspect of the verification methodology was our main concern in this paper.

At a higher level, a verification methodology could deal with various classes of users so that a scenario based approach can be followed. In the presented work, we touched on this aspect only by considering an example of two classes: beginners and experts. The questions of what classes of users should be subject to verification and how to choose appropriate concrete parameters for the generic user model to define those classes remain topics of further research.

Finally, a meta-level verification methodology addresses the development and refinement of the generic user model itself. Exploring this aspect of a methodology in detail is beyond the scope of this paper. In these early stages of our work we have been working with an aim of seeing what classes of errors can be covered by modelling systematic behaviour which in itself is not necessarily erroneous. For example a claim of a referee from an earlier paper (that we have since shown unfounded) was “this clearly only works for post-completion errors.” Our guiding principle has therefore been to choose a range of different kinds of slips to show they can in theory be detected in this way. Our research methodology is thus one of incrementally choosing known errors from the literature and looking to how the underlying behaviour (which will not always be erroneous) can be modelled. This is supported by our empirical experiments of people using devices in controlled situations [LBC$^+$05, LCB$^+$06, PCB07] to validate the model and form the basis of further experiments. These experiments then also show us where there are major gaps: they help us to identify systematic errors that are not yet represented in our model. It also should be noted that additional (realistic) errors have been shown to emerge from aspects of the model that were not added with those errors in mind. This is the power of modelling systematic behaviour rather than errors as in Fields’ approach. In subsequent work we are now looking at larger case studies not designed specifically with the model in mind to further investigate major gaps in the behaviour covered.

9. Discussion and further work

We presented a user-centred verification methodology using HOL that allows multiple classes of user errors to be detected or verified absent from the designs of interactive systems by proving a single task completion theorem. We illustrated our methodology by a series of case studies exploring various designs of an abstract service machine. In each consecutive step, new features were introduced into the designs thereby making them prone to new classes of user errors. We applied our verification methodology either to discover these errors or to prove their absence in the designs. We demonstrated how our approach could be used to detect the possibility of several classes of error: post-completion errors, communication goal based errors, order errors, premature termination and errors due to indirect interface changes. The examples were simplified to illustrate the general approach – for example we abstract multiple selections to a single button and do not model authentication. In future work we will investigate such issues.

We also demonstrated that the verification methodology is flexible as regards users modelled. Various user classes, for example beginners or experts, can be represented by choosing different instantiations of the generic user model. These choices are to be guided by the informal analysis performed by HCI experts. The different user models could then be used to verify devices designed for specific user groups.

9.1. Formal verification

What justifies formal verification of an interactive system? A pre-requisite for formal verification in our approach is a formal specification of the user part of the interactive system.[3] In our case, such specifications are developed by instantiating the generic cognitive architecture. Our experience [RCB+07a] indicates that already an attempt to formally specify a user might reveal potential problems with a design. Experience in other applications of formal methods supports this. Initially, this seems to be the most immediately useful aspect of our verification methodology. In addition, the generic cognitive architecture can be used to formally verify the consistency (soundness) of a chosen set of design rules. These can then be applied informally to minimise potential problems in designs [CuB02, CuB04]. Thus a formal user model can also be used in ways that reduce the need for full formal verification of design as in the examples here.

The work described in this paper will lead the way for more automated tool use in the future. The main difficulty in verifying complex systems is the time taken to develop the proof. This problem will be eased as we continue to develop tactics to increase the automation of the proofs. The fact that a common user model is used means that the proofs for different devices are very uniform, increasing automation possibilities. Furthermore, formal verification does not necessarily suppose a "hard" interactive proof process. It can also be done by automatic model checking [RCB+07a].

An additional strand of work we are currently exploring [RCB+07b] is how our error-detecting analysis can be combined with other methods such as cognitive task analysis. Another direction [PCB07] is investigating a range of Usability Evaluation Methods of different degrees of formality to investigate (amongst other things) such trade-offs. A detailed discussion of this is beyond the scope of this paper however.

Our approach requires traditional informal analysis of the task and of the device's interface to be performed. This is needed to gather the information upon which to instantiate the formal model. If the information so gathered is inaccurate then errors could be missed. For example, if as a result of that analysis it is decided that a particular action is a device independent communication goal, but in practice this is not so, then the associated errors would not be detected.

Our work integrates machine-centred verification (hardware verification) with user-centred verification (that user errors are eliminated). The higher-order logic framework adopted is that developed for hardware verification. Specifications, whether of implementations or behaviours, are higher-order logic relations over signals specifying input or output traces. The theorems developed therefore could be combined with hardware verification theorems about the computer component of the interactive system. Such an integration with hardware verification is described in [CuB04].

Higher-order logic and so HOL is a general purpose and extremely expressive logic. The modelling could be done in other ways (a great deal of work has been done embedding other notations in higher-order logic so, in a sense, any other existing notation could be embedded in HOL and that approach used). The history function approach was a pragmatic decision – the early work aimed to show how such verification of interactive systems could be integrated with hardware verification where history functions were a standard HOL modelling approach. It is quite a powerful and natural specification approach so in that sense was an obvious choice too. Later we implemented in SAL essentially the same cognitive architecture [RCB+07a],

[3] We assume that the device specification is provided by its developers.

albeit as a simpler state transition based system, that abstracts from history functions. On the other hand, the generic nature of our architecture *requires* a formalism, like HOL or SAL, that supports higher-order specifications.

9.2. Cognitive architecture

Our generic user model is a formal description of a very simple cognitive architecture based on results from the cognitive science and human-computer interaction literature. The cognitive architecture describes fallible behaviour. However, rather than explicitly describing erroneous behaviour, it is based on cognitively plausible behaviour. Erroneous behaviour emerges if it is placed in an environment (i.e. with a computer system or other device) that allows it. However, the model does not imply that mistakes will always be made in such cases, just that the potential is there.

The main purpose of our user model is not accurately predicting human behaviour; rather, it is taking human cognition into account when trying to detect potential design problems. Our reasoning is about what the cognitive architecture might do rather than any particular person. Still, errors that the cognitive architecture could make are cognitively plausible and so worth attention. In a sense, our nondeterministic model is more "brutal" than stochastic ones – it makes passing the test harder and so is more likely to reveal potential issues and situations that should be eliminated. Our approach (as any other) can never eradicate the possibility of human error, however. It cannot be used to avoid non-systematic errors or malicious user behaviour. Such errors can never be completely eliminated from interactive systems unless the functionality of the device is severely limited. However, by providing a methodology for detecting design features that allow users behaving in a simple rational way to make systematic errors, the usability of interactive systems can be improved.

An alternative approach would be to specify properties corresponding to known possible 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 emerge because the users do not have perfect knowledge of the design, or they are simply not focussed enough on the interaction. An advantage of our approach over specifying properties directly is that the informal and potentially error-prone reasoning implicitly required to generate appropriate properties is replaced by a formal generic user model. The cognitive basis of the errors is specified and validated once rather than for each new design or task. It also does not need to be revalidated when errors are found and the design subsequently modified.

The formal cognitive model has been developed from principles of cognition and as such has a theoretical basis. However, it should be noted that, unlike work such as that of Duke and Duce [DuD99], and Bowman and Faconti [BoF99], the formalism was not developed directly from an existing, complete psychological theory. Rather we have taken an exploratory approach (as described in Section 8), starting with some simple principles of cognition, such as non-determinism of action, goal-based termination and reactive behaviour, with particular erroneous behaviour in mind. Despite this approach, even the small number of principles is rich enough that plausible erroneous behaviour has emerged that was not directly expected when the principles were formalised. For example, the model will terminate early if not given "please wait" feedback. This was only realised when exploring the consequences of an existing version of the model [CuB01a]. This erroneous behaviour emerges from principles included for other reasons. Similarly, the possible error due to the user belief that all actions relevant to the task have been taken was discovered when using a novice user model, as discussed in Section 5.1.

There are clear advantages and disadvantages of this approach to developing the formal cognitive architecture. The main advantage is that it allows us to more easily explore the consequences of individual principles and the way they have been formalised, whilst gradually building a more complete picture. The disadvantage is that the model as it stands is, to some extent, based on arbitrarily chosen principles of cognition. It does not have the completeness and underpinning that working from a full psychologically based model would. Parallel work on the project is aiming to offset this in a way that fits our exploratory approach. We are undertaking empirical lab-based studies to validate the model [BCD$^+$06]. In experiments, the notion of a threshold is used to decide whether some phenomenon is systematic. We apply the same idea when validating our cognitive architecture against behavioural data. The aim is that the formal work feeds the empirical work suggesting experiment and similarly the experimental work feeds new developments in the user model.

9.3. Future work

Our methodology shows promise for detecting various classes of user error within more complex interactive systems. To test its utility, we will carry out case studies on more complicated interactive devices, user tasks and interfaces such as windows-based GUIs. In particular, we intend to verify some aspects of emergency dispatch systems. Currently our approach identifies interface features and their perceptions by users by using a single variable to represent both: a particular feature in the device specification and its perception in the user model. In effect, this assumes faultless user perception. Since such an assumption might hide potential problems in interface designs, we are exploring the consequences of modelling interface features and their perceptions as distinct entities and explicitly specifying appropriate relations between them [RCB+07a]. By developing our generic user model, we also intend to give a more accurate description of what is cognitively plausible. As this is done, it will be possible to model more erroneous behaviour.

For specific examples such as those presented here, it is likely that fully automated model checking or verification tools based on state-space exploration could be used. However, the additional power of an interactive theorem prover will be required to establish generic properties or to reason about design rules [CuB02]. It seems likely that this kind of verification would be a good application for an integrated environment, including both an interactive theorem prover and a model checker. In fact, we are developing a version of the cognitive architecture [RCB+07a] based on the generic formalism for transition systems within the SAL framework [MOR+04]. A close integration of both versions is one of the goals of the future development of our verification methodology. The interactive approach presented in this paper is a step towards an integrated verification environment. However, already now our approach allows detecting various classes of user error and reasoning about more general properties of interactive systems.

Acknowledgments

This work is funded by EPSRC grants GR/S67494/01 and GR/S67500/01.

References

[BAC+96] Bumbulis P, Alencar PSC, Cowan DD, de Lucena CJP (1996) Validating properties of component-based graphical user interfaces. In: Bodart F, Vanderdonckt J (eds) *Proc. Design, Specification and Verification of Interactive Systems (DSV-IS'96)*, Springer-Verlag, pp 347–365

[BaM95] Barnard PJ, May J (1995) Interactions with advanced graphical interfaces and the deployment of latent human knowledge. In: *Design, Specification and Verification of Interactive Systems: DSV-IS'95*, Springer-Verlag, pp 15–49

[Bar58] Bartlett FC (1958) *Thinking: An Experimental and Social Study.* Basic Books, New York

[BBC04] Blandford AE, Butterworth RJ, Curzon P (2004) Models of interactive systems: a case study on Programmable User Modelling. *International Journal of HumanComputer Studies* 60(2):149–200

[BBD99] Butterworth RJ, Blandford AE, Duke DJ (1999) Using formal models to explore display based usability issues. *Journal of Visual Languages and Computing* 10:455–479

[BBD00] Butterworth RJ, Blandford AE, Duke DJ (2000) Demonstrating the cognitive plausibility of interactive systems. *Formal Aspects of Computing* 12:237–259

[BCD+06] Back J, Cheng WL, Dann R, Curzon P, Blandford A (2006) Does being motivated to avoid procedural errors influence their systematicity? In: Bryan-Kinns N, Blandford A, Curzon P, Nigay L (eds) *People and computers XX–engage: proceedings of HCI 2006 (Vol. 1)*, Springer-Verlag, pp 151–158

[Bil96] Billings CE (1996) *Aviation Automation: The Search for A Human-centered Approach.* Lawrence Erlbaum Associates

[BlY98] Blandford AE, Young RM (1998) The role of communication goals in interaction. In: *Adjunct Proceedings of HCI'98*, pp 14–15

[BMP+98] Butler RW, Miller SP, Potts JN, Carreño VA (1998) A formal methods approach to the analysis of mode confusion. In: *17th AIAA/IEEE Digital Avionics Systems Conference, Bellevue, WA, October 1998*

[BMW99] Back R-J, Mikhajlova A, von Wright J (1999) Reasoning about interactive systems. In: *Formal Methods (FM'99)*, volume 1709 of *Lecture Notes in Computer Science*, Springer-Verlag, pp 1460–1476

[BoF99] Bowman H, Faconti G (1999) Analysing cognitive behaviour using LOTOS and Mexitl. *Formal Aspects of Computing* 11:132–159

[Bre03] Bredereke J (2003) On preventing telephony feature interactions which are shared-control mode confusions. In: *Feature Interactions in Telecommunications and Software Systems VII*, IOS Press, pp 159–176

[BrL02] Bredereke J, Lankenau A (2002) A rigorous view of mode confusion. In: *Computer Safety, Reliability and Security: SAFECOMP 2002*, volume 2434 of *Lecture Notes in Computer Science*, Springer-Verlag, pp 19–31

[But04] Buth B (2004) Analysing mode confusion: An approach using FDR2. In: Heisel M, Liggesmeyer P, Wittmann S (eds) *Computer Safety, Reliability and Security: SAFECOMP 2004*, volume 3219 of *Lecture Notes in Computer Science*, Springer-Verlag, pp 101–114

[ByB97] Byrne MD, Bovair S (1997) A working memory model of a common procedural error. *Cognitive Science* 21(1):31–61

[CaH97] Campos JC, Harrison MD (1997) Formally verifying interactive systems: a review. In: Harrison MD, Torres JC (eds) *Design, Specification and Verification of Interactive Systems (DSV-IS'97)*, Springer-Verlag, pp 109–124

[Cam03] Campos JC (2003) Using task knowledge to guide interactor specifications analysis. In: Jorge JA, Jardim Nunes N, Falcão e Cunha J (eds) *Interactive Systems. Design, Specification and Verification, 10th International Workshop*, volume 2844 of *Lecture Notes in Computer Science*, Springer-Verlag, pp 171–186

[CJR00] Crow J, Javaux D, Rushby J (2000) Models and mechanized methods that integrate human factors into automation design. In: *International Conference on Human-Computer Interaction in Aeronautics: HCI-Aero, September 2000*

[CuB00a] Curzon P, Blandford AE (2000) Reasoning about order errors in interaction. In: *TPHOLs 2000 Supplementary Proceedings*

[CuB00b] Curzon P, Blandford AE (2000) Using a verification system to reason about post-completion errors. In: Palanque P, Paternò F (eds) *Participants Proc. of DSV-IS 2000: 7th Int. Workshop on Design, Specification and Verification of Interactive Systems, at the 22nd Int. Conf. on Software Engineering*, pp 292–308

[CuB01a] Curzon P, Blandford AE (2001) Detecting multiple classes of user errors. In: Little R, Nigay L (eds) *Proceedings of the 8th IFIP Working Conference on Engineering for Human-Computer Interaction (EHCI'01)*, volume 2254 of *Lecture Notes in Computer Science*, Springer-Verlag, pp 57–71

[CuB01b] Curzon P, Blandford AE (2001) A user model for avoiding design induced errors in soft-key interactive systems. In: Bolton RJ, Jackson PB (eds) *TPHOLs 2001 Supplementary Proceedings*, number ED-INF-RR-0046 in Informatics Research Report, pp 33–48

[CuB02] Curzon P, Blandford AE (2002) From a formal user model to design rules. In: Forbrig P, Urban B, Vanderdonckt J, Limbourg Q (eds) *Interactive Systems. Design, Specification and Verification, 9th International Workshop*, volume 2545 of *Lecture Notes in Computer Science*, Springer-Verlag, pp 19–33

[CuB04] Curzon P, Blandford AE (2004) Formally justifying user-centred design rules: a case study on post-completion errors. In: Boiten EA, Derrick J, Smith G (eds) *Proc. of the 4th International Conference on Integrated Formal Methods*, volume 2999 of *Lecture Notes in Computer Science*, Springer-Verlag, pp 461–480

[CuL96] Curzon P, Leslie I (1996) Improving hardware designs whilst simplifying their proof. In: *Designing Correct Circuits, Workshops in Computing*, Springer-Verlag

[DBD+98] Duke DJ, Barnard PJ, Duce DA, May J (1998) Syndetic modelling. *Human-Computer Interaction* 13(4):337–394

[DBM+95] Duke DJ, Barnard PJ, May J, Duce DA (1995) Systematic development of the human interface. In: *APSEC'95: Second Asia-Pacific Software Engineering Conference*, IEEE Computer Society Press, pp 313–321

[DuD99] Duke DJ, Duce DA (1999) The formalization of a cognitive architecture and its application to reasoning about human computer interaction. *Formal Aspects of Computing* 11:665–689

[Fie01] Fields RE (2001) Analysis of erroneous actions in the design of critical systems. D. Phil Thesis, Technical Report YCST 20001/09, University of York, Department of Computer Science

[GoM93] Gordon MJC, Melham TF (eds) (1993) *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press

[JoK96] John BE, Kieras DE (1996) Using GOMS for user interface design and evaluation: which technique? *ACM Trans. on Computer-Human Interaction* 3(4):287–319

[KWM97] Kieras DE, Wood SD, Meyer DE (1997) Predictive engineering models based on the EPIC architecture for a multimodal high-performance human-computer interaction task. *ACM Trans. on Computer-Human Interaction* 4(3):230–275

[Lan01] Lankenau A (2001) Avoiding mode confusion in service-robots. In: Mokhtari M (ed) *Integration of Assistive Technology in the Information Age, Proc. of the 7th Int. Conf. on Rehabilitation Robotics*, IOS Press, pp 162–167

[LBC+05] Li SYW, Blandford A, Cairns P, Young RM (2005) Post-completion errors in problem solving. In: *Proceedings of the 27th Annual Conference of the Cognitive Science Society*, Lawrence Erlbaum Associates, pp 1278–1283

[LCB+06] Li SYW, Cox AL, Blandford A, Cairns P, Young RM, Abeles A (2006) Further investigations into post-completion error: the effects of interruption position and duration. In: *Proceedings of the 28th Annual Conference of the Cognitive Science Society*, Lawrence Erlbaum Associates, pp 471–476

[LDS+97] Leveson NG, Denise Pinnel L, Sandys SD, Koga S, Reese JD (1997) Analyzing software specifications for mode confusion potential. In: Johnson CW (ed) *Proceedings of the Workshop on Human Error and System Development*, Glasgow Accident Analysis Group Technical Report GAAG-TR-97-2, pp 132–146

[LeP97] Leveson NG, Palmer E (1997) Designing automation to reduce operator errors. In: *Proceedings of the IEEE Systems, Man and Cybernetics Conference, October 1997*

[Lib89] Libet B (1989) The timing of a subjective experience. *Behavioural and Brain Sciences* 12:183–185

[LüC99] Lüttgen G, Carreño V (1999) Analyzing mode confusion via model checking. In: Dams D, Gerth R, Leue S, Massink M (eds) *SPIN'99*, volume 1680 of *Lecture Notes in Computer Science*, Springer-Verlag, pp 120–135

[MJR98] Markopoulos P, Johnson P, Rowson J (1998) Formal architectural abstractions for interactive software. *International Journal of Human Computer Studies* 49:679–715

[MoD95] Moher TG, Dirda V (1995) Revising mental models to accommodate expectation failures in human-computer dialogues. In: *Design, Specification and Verification of Interactive Systems: DSV-IS'95*, Springer-Verlag, pp 76–92

[MOR+04] de Moura L, Owre S, Ruess H, Rushby J, Shankar N, Sorea M, Tiwari A (2004) SAL 2. In: Alur R, Peled DA (eds) *Computer Aided Verification: CAV'04*, volume 3114 of *Lecture Notes in Computer Science*, Springer-Verlag, pp 496–500

[New90] Newell A (1990) *Unified Theories of Cognition.* Harvard University Press

[OKM86] Osman A, Kornblum S, Meyer DE (1986) The point of no return in choice reaction time: controlled and ballistic stages of response preparation. *Journal of Experimental Psychology: Human Perception and Performance* 12(3):243–258

[PaM95] Paternò F, Mezzanotte M (1995) 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*, Chapman and Hall Publisher, pp 213–226

[PCB07] Papatzanis P, Curzon P, Blandford A (2007) Identifying phenotypes & genotypes: a case study on evaluating an in-car navigation system. In press: *Proceedings of Engineering Interactive Systems 2007*, Springer-Verlag

[RCB+07a] Rukšėnas R, Curzon P, Back J, Blandford A (2007) Formal modelling of cognitive interpretation. In: Doherty G, Blandford A (eds) *Interactive Systems. Design, Specification and Verification, 13th International Workshop*, volume 4323 of *Lecture Notes in Computer Science*, Springer-Verlag, pp 123–136

[RCB+07b] Rukšėnas R, Curzon P, Blandford A, Back J (2007) Combining human error verification and timing analysis. In press: *Proceedings of Engineering Interactive Systems 2007*, Springer-Verlag

[RCP99] Rushby J, Crow J, Palmer E (1999) An automated method to detect potential mode confusions. In: *18th AIAA/IEEE Digital Avionics Systems Conference (DASC), October 1999*

[Rea90] Reason J (1990) *Human Error.* Cambridge University Press

[Rus01a] Rushby J (2001) Analyzing cockpit interfaces using formal methods. *Electronic Notes in Theoretical Computer Science* 43

[Rus01b] Rushby J (2001) Modeling the human in human factors. In: Vogues U (ed) *Computer Safety, Reliability and Security: SAFECOMP 2001*, volume 2187 of *Lecture Notes in Computer Science*, Springer-Verlag, pp 86–91

[Rus02] Rushby J (2002) Using model checking to help discover mode confusions and other automation suprises. *Reliability Engineering and System Safety* 75(2):167–177

[Suc87] Suchman LA (1987) *Plans and situated actions: the problem of human-machine communication.* Cambridge University Press

[Thi01] Thimbleby H (2001) Permissive user interfaces. *International Journal of Human-Computer Studies* 54(3):333–350

Appendices

A. The formal cognitive architecture

In this appendix we give the formal HOL definitions of the cognitive architecture.

A.1. Basics

In this section the definitions underpinning the timing of actions are given. The key definition is `NEXT`. It is then used throughout the remainder of the cognitive architecture to specify when actions occur. It is based on local definitions `STABLE`, `LSTABLE` and `LF`.

`STABLE` asserts that the signal, `P`, has the same value `v` between times `t1` and `t2`.

$\vdash_{def}$ STABLE P t1 t2 v = $\forall$t. t1 $\leq$ t $\wedge$ t $<$ t2 $\supset$ (P t = v)

`LSTABLE` asserts that `STABLE` holds for a list of signals.

$\vdash_{def}$ (LSTABLE [ ] t1 t2 v = TRUE) $\wedge$
    (LSTABLE (a :: rest) t1 t2 v = (STABLE a t1 t2 v) $\wedge$ (LSTABLE rest t1 t2 v))

Given seed argument 0 for `n`, `LF` states that all actions in a list except for a specified one given by position, `ppos`, are inactive (false) at the given time. Other values for `n` as argument to `LF` are only relevant in the subsequent recursive calls of `LF`.

$\vdash_{def}$ (LF n [ ] ppos t = TRUE) $\wedge$
    (LF n (a :: rest) ppos t = (((n = ppos) $\vee$ $\neg$(a t)) $\wedge$ (LF (SUC n) rest ppos t))

`NEXT` asserts that the next action from a list of actions to become true after time `t1` is `action`.

$\vdash_{def}$ NEXT flag actions action t1 =
        $\exists$t2. t1 <= t2 $\wedge$
            LSTABLE actions t1 t2 F $\wedge$
            (LF 0 actions action t2) $\wedge$
            EL action actions t2 $\wedge$
            (flag (t2+1)) $\wedge$
            STABLE flag (t1+1) (t2+1) F

A.2. Possessions

In this subsection we define some physical restrictions on possessions and their values.

`HAS_POSSESSION` is used to define the physical restrictions on possessions of an individual, linking for a single kind of possession the events of taking and giving up a possession, having a possession, the number of an object possessed and its value. Properties include that a person has a possession if its count is greater than 0; if in a time instance a person takes a possession and does not give it up, then the count goes up by 1; *etc.*

```
⊢def HAS_POSSESSION haspossession takepossession givepossession valueofpossession countpossession
          (ustate:'u) (mstate:'m) =
      (∀t. haspossession ustate t = (countpossession ustate t > 0)) ∧
      (∀t. (givepossession mstate t ∧ ¬(takepossession mstate t)) =
             ((countpossession ustate t > 0) ∧
              (countpossession ustate (t+1) = countpossession ustate t - 1 ))) ∧
      (∀t. (takepossession mstate t ∧ ¬(givepossession mstate t)) =
             (countpossession ustate (t+1) = countpossession ustate t + 1 )) ∧
      (∀t. ((¬(givepossession mstate t) ∧ ¬(takepossession mstate t)) ∨
            ((givepossession mstate t) ∧ (takepossession mstate t))) =
             (countpossession ustate (t+1) = countpossession ustate t))
```

Possessions are recorded as a new state tuple type recording having a possession, taking one, giving one up, the value and the count of the possession. Corresponding functions that access the tuples' fields are defined.

```
pstate_type = : ('u → num → bool) # ('m → num → bool) # ('m → num → bool) #
                 num # ('u → num → num)

⊢def HasPossession (pstate: pstate_type) = FST pstate

⊢def TakePossession (pstate: pstate_type) = FST (SND pstate)

⊢def GivePossession (pstate: pstate_type) = FST (SND (SND pstate))

⊢def ValueOfPossession (pstate: pstate_type) = FST (SND (SND (SND pstate)))

⊢def CountPossession (pstate: pstate_type) = SND (SND (SND (SND pstate)))
```

A collection of possessions is recorded as a list of possession tuples. `POSSESSIONS` asserts that `HAS_POSSESSION` holds of each.

```
⊢def (POSSESSIONS [ ] (ustate:'u) (mstate:'m) = TRUE) ∧
     (POSSESSIONS ((p: pstate_type) :: possessions) ustate mstate =
         ((POSSESSIONS possessions ustate mstate) ∧
         (HAS_POSSESSION (HasPossession p) (TakePossession p)
                         (GivePossession p) (ValueOfPossession p)
                         (CountPossession p)
                         ustate mstate)))
```

The value of a list of possessions is the total value of the possessions of each type.

```
⊢def (POSSESSIONS_VAL [ ] (ustate:'u) t = 0) ∧
     (POSSESSIONS_VAL ((p: pstate_type) :: ps) (ustate:'u) t =
         ((POSSESSIONS_VAL ps (ustate:'u) t) + ((CountPossession p ustate t) * (ValueOfPossession p))))
```

A.3. Probes

In this subsection we define probes. They are signals that only record the firing of other behavioural rules at each time instance and do not alter the cognitive behaviour of the architecture [CuB04]. In this version of the cognitive architecture there is only a single probe related to a goal completion rule (given later) firing.

$\vdash_{def}$ Goalcompletion probes t = probes t

A.4. Reactive Signals

In this subsection we define rules about how a user might react to external stimulus from a device such as lights flashing.

Reactive signals are provided to the cognitive architecture as a list of pairs consisting of a stimulus and a resulting action. We first define accessor functions for the tuples.

$\vdash_{def}$ Stimulus reactive_action = FST reactive_action

$\vdash_{def}$ Action reactive_action = SND reactive_action

`REACT` is the rule for reacting to a stimulus. The relation is true at a time, `t` if the stimulus is true and the next action is the given action. As this is not the goal completion rule, the goal completion probe is false.

```
⊢def REACT flag actions stimulus action probes t =
        (stimulus t = TRUE) ∧
        (Goalcompletion probes t = FALSE) ∧
        NEXT flag actions action t
```

`REACTS` states that given a list of reactive stimulus rules, any can fire (their relation can be true) at a time instance.

```
⊢def (REACTS flag actions [ ] probes t = FALSE) ∧
     (REACTS flag actions (ra :: reactive_actions) probes t =
        ((REACTS flag actions reactive_actions probes t) ∨
         (REACT flag actions (Stimulus ra) (Action ra) probes t)))
```

A.5. Mental Commitments

In this subsection we define what it means to have made a mental commitment to take a physical action.

Mental commitments are pairs, consisting of a guard and an action that can occur if the guard is true. Given a list of such pairs, `CommitmentGuards` extracts all the guards.

$\vdash_{def}$ CommitmentGuard commitments = FST commitments

$\vdash_{def}$ CommitmentAction commitments = SND commitments

$\vdash_{def}$ CommitmentGuards commitments = MAP CommitmentGuard commitments

The rule for turning a guard mental action (irrevocable decision) into a physical one, is that if the mental action was true on the previous cycle then the next action is a corresponding physical action as given by `COMMITS`.

```
⊢def COMMIT flag actions maction paction t =
        (maction(t-1) = TRUE) ∧ NEXT flag actions paction t
```

Given a list of mental commitments the rule corresponding to any one of them can fire.

```
⊢def (COMMITS flag actions [ ] t = FALSE) ∧
     (COMMITS flag actions (ca :: commit_actions) t =
        ((COMMITS flag actions commit_actions t) ∨
         (COMMIT flag actions (CommitmentGuard ca) (CommitmentAction ca) t)))
```

Given a list of commitments, a commitment is currently made if and only if the guard of any one of them was true on the previous cycle.

$\vdash_{def}$ (CommitmentMade [ ] t = FALSE) $\wedge$
(CommitmentMade (maction :: rest) t = (maction(t-1) = TRUE) $\vee$ (CommitmentMade rest t))

A.6. Communication Goals

In this subsection we define rules related to one form of task related knowledge: communication goals.

`FILTER` takes two lists. The first is a list of all signals. The second a list of positions in that first list. For each position, it checks if that signal is true at the current time `t` and removes it from the list of positions if so. This is used to take a communication goal list and remove all those for which the action was performed.

```
⊢def (FILTER actions [ ] t = [ ] ) ∧
      (FILTER actions (a :: rest) t =
          if (EL (FST a) actions t then (FILTER actions rest t) else (a :: (FILTER actions rest t))))
```

A history list (i.e. a list of actions at time `t`) is filtered if and only if at the next time instance all those entries which were active currently are removed from the list.

```
⊢def FILTER_HLIST actions hlist =
          ∀t. hlist(t+1) = FILTER actions (hlist t) t
```

A communication goal is a pair consisting of an action and a guard. We define corresponding accessor functions.

$\vdash_{def}$ ActionOfComGoal cg = FST cg

$\vdash_{def}$ GuardOfComGoal cg = SND cg

A communication goal rule fires if the guard of the communication goal is true, the goal has not been achieved and the next action is the given action. This is not the goal completion rule so its probe is false.

```
⊢def COMMGOAL flag actions action guard goal probes (ustate:'u) (mstate:'m) t =
          ¬(goal ustate t) ∧
          (guard t) ∧
          (Goalcompletion probes t = FALSE) ∧
          NEXT flag actions action t
```

`COMMGOALER` asserts using the subsidiary definition `COMMGOALS` that given a history list of communication goals, any one of them can fire at a given time.

```
⊢def (COMMGOALS flag actions [ ] goal probes (ustate:'u) (mstate:'m) t = FALSE) ∧
      (COMMGOALS flag actions (a :: rest) goal probes ustate mstate t =
          ((COMMGOALS flag actions rest goal probes ustate mstate t) ∨
           (COMMGOAL flag actions (ActionOfComGoal a) (GuardOfComGoal a)
                goal probes ustate mstate t)))
```

```
⊢def COMMGOALER flag actions hlist goal probes (ustate:'u) (mstate:'m) t =
          COMMGOALS flag actions (hlist t) goal probes ustate mstate t
```

A.7. Goal-based Termination

In this subsection we define termination behaviour based on a person's goals.

The action that indicates when an interaction is terminated is always first in the list of all actions. Hence, its position is defined to be 0.

$\vdash_{def}$ FINISHED = 0

The goal based completion rule fires when the goal is achieved and the next action is to finish the interaction. This is the goal completion rule so the goal completion probe fires.

$\vdash_{def}$ COMPLETION flag actions goalachieved probes (ustate:'u) t =
    (Goalcompletion probes t = TRUE) $\wedge$
    (goalachieved ustate t = TRUE) $\wedge$
    NEXT flag actions FINISHED t

A.8. Termination due to no options

In this subsection we define termination behaviour that results from there being no useful steps that can be taken.

We first define some simple list operators. They are used to manipulate the lists of guards for the other rules to create a guard that fires when no other guard fires. `NOT_CONJL` takes a list of booleans and creates a new list with all its entries negated. `CONJ1L` takes a list of booleans and "ands" a boolean to each element of the list. `APPLYL` applies a function to each element of a list.

$\vdash_{def}$ (NOT_CONJL [ ] = TRUE) $\wedge$
  (NOT_CONJL (P :: rest) = $\neg$P $\wedge$ (NOT_CONJL rest))

$\vdash_{def}$ (CONJ1L P [ ] = [ ]) $\wedge$
  (CONJ1L P (Q :: rest) = ((P $\wedge$ Q) :: (CONJ1L Q rest)))

$\vdash_{def}$ (APPLYL [ ] a = [ ]) $\wedge$
  (APPLYL (f :: rest) a = ((f a) :: (APPLYL rest a)))

The basic rule for no-option based termination, `ABORT` fires if its guard is true, leading to the next action being to finish the interaction. `ABORTION` constructs the guard for `ABORT` so that it holds if no other rule's guard holds.

$\vdash_{def}$ ABORT flag actions guards probes (ustate:'u) t =
    (Goalcompletion probes t = FALSE) $\wedge$
    (guards = TRUE) $\wedge$
    NEXT flag actions FINISHED t

$\vdash_{def}$ ABORTION flag actions goalachieved commgoals react_actions probes (ustate:'u) (mstate:'m) t =
    ABORT flag actions
      (NOT_CONJL ((goalachieved ustate t) ::
              (APPEND (CONJ1L ($\neg$(goalachieved ustate t))
                      (APPLYL (MAP GuardOfComGoal (commgoals t)) t))
                  (APPLYL (MAP FST react_actions) t))))
      probes ustate t

A.9. Top Level Definitions of the Architecture

In this subsection we give the top level definitions of the architecture, pulling the separate behaviours together into a single architecture.

The initial communication goals at time 1 are set to those supplied as an argument to the cognitive architecture as a whole.

$\vdash_{def}$ USER_INIT commgoals init_commgoals = (commgoals 1 = init_commgoals)

The task is completed when the goal is achieved and the interaction invariant is restored.

$\vdash_{def}$ TASK_DONE goal inv t = (goal t $\wedge$ inv t)

The rules encoding different reasons for taking an action are combined by disjunction, so as to be non-deterministic. In the current version of the architecture they consist of goal completion, reacting to a stimulus, executing a communication goal and finishing due to no options.

$\vdash_{def}$ USER_RULES flag actions commgoals reactive_actions goalachieved probes mstate ustate t =
    COMPLETION flag actions goalachieved probes ustate t $\vee$
    REACTS flag actions reactive_actions probes t $\vee$
    COMMGOALER flag actions commgoals goalachieved probes ustate mstate t $\vee$
    ABORTION flag actions goalachieved commgoals reactive_actions probes ustate mstate t

The non-deterministic rules are wrapped in an if-then else structure. If the interaction has already been terminated it stays terminated. If a mental commitment has been made then it is carried out. If the task is completed then the interaction terminates. Only if none of the above hold do the non-deterministic options come into play. The above is only a consideration if the flag is true as it indicates the time is such that a new decision needs to be made by the architecture. It is set by each rule to cover the time period over which that rule has committed to a decision.

$\vdash_{def}$ USER_CHOICE flag actions commitments commgoals reactive_actions
        finished goalachieved invariant probes (ustate:'u) (mstate:'m) =
  ($\forall$t.
  $\neg$(flag t) $\vee$
  (*if* (finished ustate (t-1))
   then (NEXT flag actions FINISHED t $\wedge$ (Goalcompletion probes t = FALSE))
   else if (CommitmentMade (CommitmentGuards commitments) t)
   then (COMMITS flag actions commitments t $\wedge$ (Goalcompletion probes t = FALSE))
   else if TASK_DONE (goalachieved ustate) (invariant ustate) t
   then (NEXT flag actions FINISHED t $\wedge$ (Goalcompletion probes t = FALSE))
   else USER_RULES flag actions commgoals reactive_actions
        goalachieved probes mstate ustate t))

To make the above rules work, various other processes need to take place in the background at every time instance. If the interaction is terminated it stays terminated. The rules about possessions must always hold. The person's internal communication goal list is maintained from time instance to time instance with only completed actions removed. Where rules are not driving the probe value directly its signal remains false.

GENERAL_USER_UNIVERSAL actions commgoals possessions finished flag
        probes (ustate:'u) (mstate:'m) =
  ($\forall$t. finished ustate t $\supset$ finished ustate (t+1)) $\wedge$
  (POSSESSIONS possessions ustate mstate) $\wedge$
  (FILTER_HLIST actions commgoals) $\wedge$
  ($\forall$t. $\neg$(flag t) $\supset$ (Goalcompletion probes t = FALSE))

The above properties of each time instance are combined with the rule about initialising the cognitive architecture.

$\vdash_{def}$ USER_UNIVERSAL flag actions commgoals init_commgoals possessions
        finished probes (ustate:'u) (mstate:'m) =
  (USER_INIT commgoals init_commgoals) $\wedge$
  (GENERAL_USER_UNIVERSAL actions commgoals possessions finished flag probes ustate mstate)

Finally the full cognitive architecture is the combination of `USER_UNIVERSAL` about the background processes and `USER_CHOICE` about the actual rules that drive the actions.

$\vdash_{\mathit{def}}$ USER flag actions commitments commgoals init_commgoals reactive_actions
        possessions finished goalachieved invariant probes (ustate:'u) (mstate:'m) =
    (USER_UNIVERSAL flag actions commgoals init_commgoals possessions finished
        probes ustate mstate) $\wedge$
    (USER_CHOICE flag actions commitments commgoals reactive_actions
        finished goalachieved invariant probes ustate mstate)

B. A formal machine specification

In this appendix we specify in HOL the design of the service machine described in Section 6.1 (see Figure 5).

B.1. Machine state space

The machine state space is represented by a tuple type consisting of 9 history functions. The first three fields represent machine inputs, the rest represent machine outputs. Corresponding functions that access the tuples' fields are defined.

$\vdash_{\mathit{def}}$ mstate_type = :(time $\rightarrow$ bool) # (time $\rightarrow$ bool) # (time $\rightarrow$ bool) # (time $\rightarrow$ bool) #
        (time $\rightarrow$ bool) # (time $\rightarrow$ bool) # (time $\rightarrow$ bool) # (time $\rightarrow$ bool) # (time $\rightarrow$ bool)

$\vdash_{\mathit{def}}$ InsertCard (mstate: mstate_type) = FST mstate

$\vdash_{\mathit{def}}$ SelectService mstate = FST (SND mstate)

$\vdash_{\mathit{def}}$ RemoveCard mstate = FST (SND (SND mstate))

$\vdash_{\mathit{def}}$ SelectLight mstate = FST (SND (SND (SND mstate)))

$\vdash_{\mathit{def}}$ CardLight mstate = FST (SND (SND (SND (SND mstate))))

$\vdash_{\mathit{def}}$ ReleaseCard mstate = FST (SND (SND (SND (SND (SND mstate)))))

$\vdash_{\mathit{def}}$ CardSound mstate = FST (SND (SND (SND (SND (SND (SND mstate))))))

$\vdash_{\mathit{def}}$ ProvideService mstate = FST (SND (SND (SND (SND (SND (SND (SND mstate)))))))

$\vdash_{\mathit{def}}$ PleaseWait mstate = SND (SND (SND (SND (SND (SND (SND (SND mstate)))))))

B.2. Machine outputs

An enumerated type `MachineState` represents the internal states of our finite state machine. In this design, it includes six states. Each state is associated with specific machine outputs in that state.

$\vdash_{\mathit{def}}$ MachineState = RESET_STATE | CARD_STATE | SERVICE_STATE |
        WAIT_STATE | PROCESS_STATE | DONE_STATE

$\vdash_{\mathit{def}}$ RESET_OUTPUTS (mstate: mstate_type) t =
    (SelectLight mstate t = T) $\wedge$ (CardLight mstate t = T) $\wedge$ (ReleaseCard mstate t = F) $\wedge$
    (CardSound mstate t = F) $\wedge$ (ProvideService mstate t = F) $\wedge$ (PleaseWait mstate t = F)

$\vdash_{def}$ CARD_OUTPUTS (mstate: mstate_type) t =
(SelectLight mstate t = T) $\wedge$ (CardLight mstate t = F) $\wedge$ (ReleaseCard mstate t = F) $\wedge$
(CardSound mstate t = F) $\wedge$ (ProvideService mstate t = F) $\wedge$ (PleaseWait mstate t = F)

$\vdash_{def}$ SERVICE_OUTPUTS (mstate: mstate_type) t =
(SelectLight mstate t = F) $\wedge$ (CardLight mstate t = T) $\wedge$ (ReleaseCard mstate t = F) $\wedge$
(CardSound mstate t = F) $\wedge$ (ProvideService mstate t = F) $\wedge$ (PleaseWait mstate t = F)

$\vdash_{def}$ WAIT_OUTPUTS (mstate: mstate_type) t =
(SelectLight mstate t = F) $\wedge$ (CardLight mstate t = F) $\wedge$ (ReleaseCard mstate t = F) $\wedge$
(CardSound mstate t = F) $\wedge$ (ProvideService mstate t = F) $\wedge$ (PleaseWait mstate t = F)

$\vdash_{def}$ PROCESS_OUTPUTS (mstate: mstate_type) t =
(SelectLight mstate t = F) $\wedge$ (CardLight mstate t = F) $\wedge$ (ReleaseCard mstate t = T) $\wedge$
(CardSound mstate t = T) $\wedge$ (ProvideService mstate t = F) $\wedge$ (PleaseWait mstate t = F)

$\vdash_{def}$ DONE_OUTPUTS (mstate: mstate_type) t =
(SelectLight mstate t = F) $\wedge$ (CardLight mstate t = F) $\wedge$ (ReleaseCard mstate t = F) $\wedge$
(CardSound mstate t = F) $\wedge$ (ProvideService mstate t = T) $\wedge$ (PleaseWait mstate t = F)

B.3. Machine transitions

In general, the machine moves from one internal state to another prompted by user actions. However, in the `WAIT` and `DONE` states, the machine moves to a new state unprompted.

$\vdash_{def}$ RESET_TRANSITION s (mstate: mstate_type) t =
if InsertCard mstate t *then* s(t+1) = CARD_STATE
else if SelectService mstate t *then* s(t+1) = SERVICE_STATE *else* s(t+1) = RESET_STATE

$\vdash_{def}$ SERVICE_TRANSITION s (mstate: mstate_type) t =
if InsertCard mstate t *then* s(t+1) = WAIT_STATE *else* s(t+1) = SERVICE_STATE

$\vdash_{def}$ CARD_TRANSITION s (mstate: mstate_type) t =
if SelectService mstate t *then* s(t+1) = WAIT_STATE *else* s(t+1) = CARD_STATE

$\vdash_{def}$ WAIT_TRANSITION s (mstate: mstate_type) t = (s(t+1) = PROCESS_STATE)

$\vdash_{def}$ PROCESS_TRANSITION s (mstate: mstate_type) t =
if RemoveCard mstate t *then* s(t+1) = DONE_STATE *else* s(t+1) = PROCESS_STATE

$\vdash_{def}$ DONE_TRANSITION s (mstate: mstate_type) t = (s(t+1) = RESET_STATE)

B.4. Machine specification

Each internal machine state is defined by combining its output and its transition relation. These definitions are then bound together in the top level machine specification.

$\vdash_{def}$ RESET_SPEC s (mstate: mstate_type) =
RESET_OUTPUTS mstate t $\wedge$ RESET_TRANSITION s mstate t

$\vdash_{def}$ SERVICE_SPEC s (mstate: mstate_type) =
SERVICE_OUTPUTS mstate t $\wedge$ SERVICE_TRANSITION s mstate t

```
⊢def CARD_SPEC s (mstate: mstate_type) =
    CARD_OUTPUTS mstate t ∧ CARD_TRANSITION s mstate t

⊢def WAIT_SPEC s (mstate: mstate_type) =
    WAIT_OUTPUTS mstate t ∧ WAIT_TRANSITION s mstate t

⊢def PROCESS_SPEC s (mstate: mstate_type) =
    PROCESS_OUTPUTS mstate t ∧ PROCESS_TRANSITION s mstate t

⊢def DONE_SPEC s (mstate: mstate_type) =
    DONE_OUTPUTS mstate t ∧ DONE_TRANSITION s mstate t

⊢def MACHINE_SPEC s (mstate: mstate_type) =
    ∀t. if (s t = RESET_STATE) then RESET_SPEC s mstate t
        else if (s t = SERVICE_STATE) then SERVICE_SPEC s mstate t
        else if (s t = CARD_STATE) then CARD_SPEC s mstate t
        else if (s t = WAIT_STATE) then WAIT_SPEC s mstate t
        else if (s t = PROCESS_STATE) then PROCESS_SPEC s mstate t
        else DONE_SPEC s mstate t
```

C. An instantiation of the cognitive architecture

In this appendix, we show how the specific user model described in Section 5.2 is derived as an instantiation of the cognitive architecture. This user model specifies the users of the service machine defined in Appendix B.

C.1. User state space

The user state space is represented by a tuple type consisting of 16 history functions. The first one is used for the communication goal list. The following three fields represent commitment actions: one for each physical action (these correspond to the three machine inputs defined in Appendix B). The following 9 fields represent the attributes of the user's possessions: three for each possession. The remaining three history functions record whether the user has terminated the interaction, whether the next action has been selected and probes (not used in this paper). Corresponding functions that access the tuples' fields are defined.

```
⊢def ustate_type = :(time → ((time → bool) # num) list) #
                    (time → bool) # (time → bool) # (time → bool) #
                    (time → bool) # (time → num) # num #
                    (time → bool) # (time → num) # num #
                    (time → bool) # (time → num) # num #
                    (time → bool) # (time → bool) # (time → bool)

⊢def UserCommgoals (ustate: ustate_type) = FST ustate

⊢def CommitSelect ustate = FST (SND ustate)

⊢def CommitInsert ustate = FST (SND (SND ustate))

⊢def CommitRemove ustate = FST (SND (SND (SND ustate)))

⊢def HasCardBefore ustate = FST (SND (SND (SND (SND ustate))))

⊢def CountCardBefore ustate = FST (SND (SND (SND (SND (SND ustate)))))

⊢def ValueCardBefore ustate = FST (SND (SND (SND (SND (SND (SND ustate))))))
```

$\vdash_{def}$ ReceivedService ustate = FST (SND (SND (SND (SND (SND (SND (SND ustate)))))))

$\vdash_{def}$ CountService ustate = FST (SND (SND (SND (SND (SND (SND (SND (SND ustate))))))))

$\vdash_{def}$ ValueService ustate = FST (SND (SND (SND (SND (SND (SND (SND (SND (SND ustate)))))))))

$\vdash_{def}$ HasCardAfter ustate =
FST (SND (SND (SND (SND (SND (SND (SND (SND (SND (SND ustate))))))))))

$\vdash_{def}$ CountCardAfter ustate =
FST (SND (SND (SND (SND (SND (SND (SND (SND (SND (SND (SND ustate)))))))))))

$\vdash_{def}$ ValueCardAfter ustate =
FST (SND (SND (SND (SND (SND (SND (SND (SND (SND (SND (SND (SND ustate))))))))))))

$\vdash_{def}$ UserFinished ustate =
FST (SND (SND (SND (SND (SND (SND (SND (SND(SND(SND(SND(SND(SND ustate)))))))))))))

$\vdash_{def}$ Flag ustate =
FST(SND(SND(SND(SND(SND(SND(SND(SND(SND(SND(SND(SND(SND(SND ustate))))))))))))))

$\vdash_{def}$ Probes ustate =
SND(SND(SND(SND(SND(SND(SND(SND(SND(SND(SND(SND(SND(SND(SND ustate))))))))))))))

C.2. User actions

In this user model, there are 8 possible user actions: three physical actions modelled as the machine inputs, three corresponding mental actions, and two actions representing the termination of the task and a pause in the interaction. Each action is named by its position in the list of all actions. Pairs that associate a mental user action with the corresponding physical action form a list of commitments.

$\vdash_{def}$ ACTIONS = [UserFinished ustate; CommitInsert ustate; CommitSelect ustate; CommitRemove ustate;
InsertCard mstate; SelectService mstate; RemoveCard mstate; Pause mstate]

$\vdash_{def}$ COMMITINSERT = 1 $\wedge$ COMMITSELECT = 2 $\wedge$ COMMITREMOVE = 3 $\wedge$
INSERTCARD = 4 $\wedge$ SELECTSERVICE = 5 $\wedge$ REMOVECARD = 6 $\wedge$ PAUSE = 7

$\vdash_{def}$ COMMITMENTS = [(COMMITINSERT, INSERTCARD);
(COMMITSELECT, SELECTSERVICE);
(COMMITREMOVE, REMOVECARD)]

Our user model is primed to provide a card as the means of payment and to communicate the selection of a service. The model reacts when the machine prompts for inserting a card, making a selection, and removing the inserted card. Prompted by the machine, the user may also pause.

$\vdash_{def}$ INIT_COMMGOALS = [(HasCardBefore ustate, COMMITINSERT); (λt.T, COMMITSELECT)]

$\vdash_{def}$ p AND q = λt. p t $\wedge$ q t

$\vdash_{def}$ REACTIVE = [(CardLight mstate AND HasCardBefore ustate, COMMITINSERT);
(SelectService mstate, COMMITSELECT);
(CardSound mstate, COMMITREMOVE);
(PleaseWait mstate, PAUSE)]

C.3. Possessions

We record five pieces of information about each user's possession: whether the user has that possession, the events of taking and giving up that possession, the quantity of an object possessed, and its value. This information is represented as a tuple. The list of all such tuples is formed using the function `MAKE_POSSESSIONS`. Finally, the invariant that the user wishes to restore by the end of the interaction is based on the value of the user's possessions.

$\vdash_{def}$ MAKE_POSSESSIONS hasbefore insert countbefore valbefore received select
countservice valservice hasafter remove countafter valafter =
[(hasbefore, (λm t.F), insert, valbefore, countbefore);
(received, select, (λm t.F), valservice, countservice);
(hasafter, remove, (λm t.F), valafter, countafter)]

$\vdash_{def}$ POSSESSIONS ustate =
MAKE_POSSESSIONS HasCardBefore InsertCard CountCardBefore (ValueCardBefore ustate)
ReceivedService ProvideService CountService (ValueService ustate)
HasCardAfter RemoveCard CountCardAfter (ValueCardAfter ustate)

$\vdash_{def}$ VALUE_INVARIANT possessions ustate t =
(VALUE possessions ustate t $\geq$ VALUE possessions ustate 1)

C.4. User model

A specific model of our service machine users is derived by instantiating the cognitive architecture with the parameters defined above. Note that the user's goal is to receive the requested service (indicated by `ReceivedService`).

$\vdash_{def}$ MACHINE_USER (ustate: ustate_type) (mstate: mstate_type) =
USER (Flag ustate) ACTIONS COMMITMENTS (UserCommgoals ustate)
INIT_COMMGOALS REACTIVE (POSSESSIONS ustate) UserFinished ReceivedService
(VALUE_INVARIANT (POSSESSIONS ustate)) (Probes ustate) ustate mstate