Combining Human Error Verification and Timing Analysis

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

[1] Department of Computer Science, Queen Mary, University of London
{rimvydas,pc}@dcs.qmul.ac.uk
[2] University College London Interaction Centre
{a.blandford,j.back}@ucl.ac.uk

Abstract. Designs can often be unacceptable on performance grounds. In this work, we integrate a GOMS-like ability to predict execution times into the generic cognitive architecture developed for the formal verification of human error related correctness properties. As a result, formal verification and GOMS-like timing analysis are combined within a unified framework. This allows one to judge whether a formally correct design is also acceptable on performance grounds, and vice versa. We illustrate our approach with an example based on a KLM style timing analysis.

Key words: human error, formal verification, execution time, GOMS, cognitive architecture, model checking, SAL.

1 Introduction

The correctness of interactive systems depends on the behaviour of both human and computer actors. Human behaviour cannot be fully captured by a formal model. However, it is a reasonable, and useful, approximation to assume that humans behave "rationally": entering interactions with goals and domain knowledge likely to help them achieve their goals. If problems are discovered resulting from rational behaviour then such problems are liable to be systematic and deserve attention in the design. Whole classes of persistent, systematic user errors may occur due to modelable cognitive causes [1, 2]. Often opportunities for making such errors can be reduced with good design [3]. A methodology for detecting designs that allow users, when behaving in a rational way, to make systematic errors will improve such systems. In the case of safety-critical interactive systems, it is crucial that some tasks are performed within the limits of specified time intervals. A design can be judged as incorrect, if it does not satisfy such requirements. Even for everyday systems and devices, the time and/or the number of steps taken to achieve a task goal can be an indication of the usability or otherwise of a particular design.

¶ Preprint to appear in: *Engineering Interactive Systems (EIS 2007)*, Lecture Notes in Computer Science.

We previously [4, 5] developed a generic formal user model from abstract cognitive principles, such as entering an interaction with knowledge of the task and its subsidiary goals, showing its utility for detecting some systematic user error. So far we have concentrated on the verification of functional correctness (user achieving a task goal) and usability properties (the absence of post-completion errors). Also, the cognitive architecture was recently used to verify some security properties – detecting confidentiality leaks due to cognitive causes [6]. However, none of this work addressed the timing aspects of user interaction. For example, a successful verification that a task goal is achieved only meant that it is *eventually* achieved at some unspecified point in the future. This is obviously insufficient, if the goal of verification is to give evidence that a system satisfies specific timing requirements.

Timing analysis is one of the core concerns in the well-established GOMS methodology [7]. A GOMS model predicts the trace of operators and task completion time. However, since GOMS models are deterministic, this prediction assumes and applies to a single, usually considered as expert or optimal, sequence of operators. Such assumptions may be invalid for everyday interactive systems whose average users do not necessarily know or are trained to follow optimal procedures, or they simply might choose a less cognitively demanding method. Moreover, under pressure, even the operators (expert users) of safety-critical systems may choose sub-optimal and less likely plans of action. This suggests that a timing analysis of interactive systems should include a broader set of cognitively plausible behaviours.

The main goal of this paper is to add into our verification methodology, based on a generic cognitive architecture, a GOMS-like ability to predict execution times. For this, we intend to use timing data provided by HCI models such GOMS. It should be noted of course that such timings are only estimates so "proofs" based on such timings are *not* formal guarantees of a particular performance level. They are not proofs of any real use, just proofs that the GOMS execution times are values within a particular range. Provided that distinction is remembered they can still be of use.

Using the SAL verification tools [8], we combine this ability to prove properties of GOMS timings with the verification of human error related correctness properties based on the traversal of all cognitively plausible behaviours as defined by our user model. This way, rather than considering a single GOMS "run", a whole series of runs are analyzed together, automatically generating a range of timings depending on the path taken. Such a setting allows one to do error (correctness) analysis first and then, once an error free design is created, do a broad timing analysis within a single integrated system. An advantage of doing so is that the GOMS timings can be used to argue that a systematically possible choice is "erroneous" on course performance grounds: the user model does achieve the goal but very inefficiently. If one potential method for achieving a goal was significantly slower, whilst the task completion would be proved, this might suggest design changes to either disable the possibility of choosing that method or change the design so that if it was taken then it would be easier to accomplish the goal. Similarly, a design chosen on performance grounds to eliminate a poor path might be rejected by our GOMS-like analysis due to its potential for systematic error discovered by the integrated human error analysis.

Many GOMS models support an explicit hierarchy of goals and subgoals. Our previous cognitive architecture was “flat” allowing only atomic user goals and actions. This meant that any hierarchy in user behaviour (task or goal structures) could be specified only implicitly. In this work, we take a step towards supporting hierarchical specifications of user goals. When needed (e.g., to capture an expert behaviour within a complex interactive system), these can be structured in an appropriate way. Note however that this extension to our cognitive architecture does not necessarily impose hierarchical goal structures on specific user models. To represent unstructured goals, one can simply choose a “flat” hierarchy, as is done in this paper.

One indication of cognitively plausible behaviour is choosing options that are relevant to the task goals when there are several alternatives available. Currently our cognitive architecture is fully non-deterministic in the sense that any user goal or action that is possible according to the principles of cognition, and/or prompted by the interface might be selected for execution. Here we introduce a facility for correlating, in such situations, user choices and task goals, thus ensuring that the user model ignores available but irrelevant alternatives.

Summarising, the main goal and contribution of the work presented in this paper is the integration of user-centred timing analysis with formal verification approach originally developed for reasoning about human error. Our aim here is to demonstrate how this can be done and to indicate the potential of combining the approaches in this complementary way to analyse the behaviour of the interactive system in terms of timing and timing-related errors. More specifically:

– It provides a way of creating GOMS-like cognitively plausible variations of methods of performing a task that emerge from a formal model of behaviour.
– It provides a way of detecting methods that have potential for systematic human error occurring using the same initial GOMS-like specification.
– The GOMS-like predictions of timings open the possibility of detecting some (though not all) classes of specific errors that could occur due to those timings, whilst still doing in parallel time-free error analysis based on the verification of various correctness properties.
– It allows our concept of systematic error to be extended in an analysis to include “erroneous” choices in the sense of choosing an alternative that, whilst eventually achieving the result, is predicted to be slower than acceptable.
– It introduces into our cognitive architecture a correlation between task goals and user choices thus refining the notion of cognitive plausibility captured by the formal user model.

1.1 Related work

There is a large body of work on the formal verification of interactive systems. Specific aims and focus vary. Here we concentrate on the work most directly linked to our work in this paper.

Whilst GOMS assume error-free performance, this does not preclude them from being used in a limited way to analyse erroneous performance. As noted by John and Kieras [9], GOMS can be used for example to give performance predictions for error recovery times. To do this one simply specifies GOMS models for the task of recovering from error rather than the original task, perhaps comparing predictions for different recovery mechanisms or determining whether recovery can be achieved with minimal effort. With these approaches the analysis does not identify the potential for human error: the specific errors considered must be decided in advance by the analyst.

Beckert and Beuster [10] present a verification environment with a similar architecture to our user model – connecting a device specification, a user assumption module and a user action module. They use CMN-GOMS as the user action module. The selection rules of the GOMS model are driven by the assumption model and the actions drive the device model. This gives a way of exploring the effect of errors made by the user (incorrect selection decisions as specified in the user assumption module). However, the assumption module has no specific structure, so the decision of what kind of errors could be made is not systematic or formalised but left to the designers of the system. This differs from our approach where we use a cognitive model combined with aspects of a GOMS model. This allows us to reason about systematic error in a way that is based on formalised principles of cognition. They also have not specifically focused on predicting performance times using GOMS, but rather are using it as a formal hierarchical task model.

Bowman and Faconti [11] formally specify a cognitive architecture using the process calculus LOTOS, and then apply a temporal interval logic to analyse constraints, including timing ones, on the information flow and transformation between the different cognitive subsystems. Their approach is more detailed than ours, which abstracts from those cognitive processes.

In the area of safety-critical systems, Rushby *et al* [12] focus on mode errors and the ability of pilots to track mode changes. They formalise plausible mental models of systems and analyse them using the Murϕ verification tool. The mental models though are essentially abstracted system models; they do not rely upon structure provided by cognitive principles. Neither do they attempt timing analysis. Also using Murϕ, Fields [13] explicitly models observable manifestations of erroneous behaviour, analysing error patterns. A problem of this approach is the lack of discrimination between random and systematic errors. It also implicitly assumes there is a correct plan, from which deviations are errors.

Temporal aspects of usability have also been investigated in work based on the task models of user behaviour [14, 15]. Fields *et al* [14] focus on the analysis of situations where there are deadlines for completing some actions and where the user may have to perform several simultaneous actions. Their approach is based on Hierarchical Task Analysis and uses the CSP formalism to specify both tasks and system constraints. Lazace *et al* [15] add quantitative temporal elements to the ICO formalism and use this extension for performance analysis. Both these approaches consider specific interaction scenarios which contrasts to our verification technique supporting the analysis of all cognitively plausible behaviours. The efficiency of interaction, albeit not in terms of timing, is also explored by Thimbleby [16]. Using Mathematica and probabilistic distributions of usage of menu functions, he analyses interface complexity. The latter is measured as the number of actions needed to reach desired menu options.

2 HUM-GOMS architecture

Our cognitive architecture is a higher-order logic formalisation of abstract principles of cognition and specifies a form of cognitively plausible behaviour [17]. The architecture specifies possible user behaviour (traces of actions) that can be justified in terms of specific results from the cognitive sciences. Real users can act outside this behaviour of course, about which the architecture says nothing. However, behaviour defined by the architecture can be regarded as potentially systematic, and so erroneous behaviour is similarly systematic in the design. The predictive power of the architecture is bounded by the situations where people act according to the principles specified. The architecture allows one to investigate what happens if a person acts in such plausible ways. The behaviour defined is neither "correct" nor "incorrect". It could be either depending on the environment and task in question. We do not attempt to model the underlying neural architecture nor the higher-level cognitive architecture such as information processing. Instead our model is an abstract specification, intended for ease of reasoning.

2.1 Cognitive principles

In the formal user model, we rely upon abstract cognitive principles that give a *knowledge level* description in the terms of Newell [18]. Their focus is on the internal goals and knowledge of a user. These principles are briefly discussed below.

Non-determinism. In any situation, any one of several cognitively plausible behaviours might be taken. It cannot be assumed that any specific plausible behaviour will be the one that a person will follow where there are alternatives.

Relevance. Presented with several options, a person chooses one that seems relevant to the task goals. For example, if the user goal is to get cash from an ATM, it would be cognitively implausible to choose the option allowing one to change a PIN. A person could of course press the wrong button by accident. Such classes of error are beyond the scope of our approach, focussing as it does on systematic slips.

Mental versus physical actions. There is a delay between the moment a person mentally commits to taking an action (either due to the internal goals or as a response to the interface prompts) and the moment when the corresponding

Table 1. A fragment of the SAL language

`x:T`	`x` has type `T`
λ`(x:T):e`	a function of `x` with the value `e`
`x′ = e`	an update: the new value of `x` is that of the expression `e`
`{x:T	p(x)}`	a subset of `T` such that the predicate `p(x)` holds
`a[i]`	the `i`-th element of the array `a`
`r.x`	the field `x` of the record `r`
`r WITH .x := e`	the record `r` with the field `x` replaced by the value of `e`
`g` $\rightarrow$ upd	if `g` is true then update according to `upd`
`c [] d`	non-deterministic choice between `c` and `d`
`[](i:T):` c_i	non-deterministic choice between the c_i with `i` in range `T`

physical action is taken. To capture the consequences of this delay, each *physical* action modelled is associated with an internal *mental* action that commits to taking it. Once a signal has been sent from the brain to the motor system to take an action, it cannot be revoked after a certain point even if the person becomes aware that it is wrong before the action is taken. To reflect this, we assume that a physical action immediately follows the committing action.

Pre-determined goals. A user enters an interaction with knowledge of the task and, in particular, task dependent sub-goals that must be discharged. These sub-goals might concern information that must be communicated to the device or items (such as bank cards) that must be inserted into the device. Given the opportunity, people may attempt to discharge such goals, even when the device is prompting for a different action. Such *pre-determined* goals represent a partial plan that has arisen from knowledge of the task in hand, independent of the environment in which that task is performed. No fixed order other than a goal hierarchy is assumed over how pre-determined goals will be discharged.

Reactive behaviour. Users may react to an external stimulus, doing the action suggested by the stimulus. For example, if a flashing light comes on a user might, if the light is noticed, react by inserting coins in an adjacent slot.

Goal based task completion. Users intermittently, but persistently, terminate interactions as soon as their main goal has been achieved [3], even if subsidiary tasks generated in achieving the main goal have not been completed. A cash-point example is a person walking away with the cash but leaving the card.

No-option based task termination. If there is no apparent action that a person can take that will help to complete the task then the person may terminate the interaction. For example, if, on a 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 the goal is not achievable.

```
TRANSITION
  [](g:GoalRange,p:AimRange): CommitAction:
    NOT(comm) ∧                              commit′[act(Goals[g].subgoals)] = committed;
    finished = notf ∧                        t′ = t + CogOverhead;
    atom?(Goals[g].subgoals) ∧               finished′ =
    Goals[g].grd(in, mem, env) ∧      →        IF g = ExitGoal ∧ Achieved(TopGoal)(in, mem)
    Goals[g].choice(status, g) ∧               THEN ok
    (g ≠ ExitGoal ∧ Relevant(g, p)             ELSE notf ENDIF;
     ∨                                       status′ = status WITH .trace[g] := TRUE
     g = ExitGoal ∧ MayExit)                   WITH .length := status.length + 1
[]
  [](a:ActionRange): PerformAction:
                                       commit′[a] = ready;
    commit[a] = committed     →        Transition(a)
[]
  ExitTask:
    Achieved(TopGoal)(in, mem) ∧
    NOT(comm) ∧                       →   finished′ = ok
    finished = notf
[]
  Abort:
    NOT(EnabledRelevant(in, mem, env)) ∧
                                            finished′ = IF Wait(in, mem)
    NOT(Achieved(TopGoal)(in, mem)) ∧    →                THEN notf
    NOT(comm) ∧                                         ELSE abort ENDIF
    finished = notf
[]
  Idle:
    finished = notf   →
```

Fig. 1. User model in SAL (simplified)

2.2 Cognitive architecture in SAL

We have formalised the cognitive principles within the SAL environment [8]. It provides a higher-order specification language and tools for analysing state machines specified as parametrised modules and composed either synchronously or asynchronously. The SAL notation we use here is given in Table 1. We also use the usual notation for the conjunction, disjunction and set membership operators. A slightly simplified version of the SAL specification of a transition relation that defines our user model is given in Fig. 1, where predicates in italic are shorthands explained later on. Below, whilst explaining this specification (SAL module `User`), we also discuss how it reflects our cognitive principles.

Guarded commands. SAL specifications are transition systems. Non-determinism is represented by the non-deterministic choice, `[]`, between the named guarded commands (i.e. transitions). For example, *CommitAction* in Fig. 1 is the name of a family of transitions indexed by `g`. Each guarded command in the specification describes an action that a user *could* plausibly take. The pairs *CommitAction* – *PerformAction* of the corresponding transitions reflect the connection between the physical and mental actions. The first of the pair models committing to a goal, the second actually taking the corresponding action (see below).

Goals structure. The main concepts in our cognitive architecture are those of user *goals* and *aims*. A user aim is a predicate that partially specifies model states that the user intends to achieve by executing some goal. User goals are organised as a hierarchical (tree like) goal–subgoals structure. The nodes of this tree are either compound or atomic:

atomic Goals at the bottom of the structure (tree leaves) are atomic: they consist of (map to) an action, for example, a device action.

compound All other goals are compound: they are modelled as a set of task subgoals.

In this paper, we consider an essentially flat goal structure with the top goal consisting of atomic subgoals only. We will explore the potential for using hierarchical goal structures in subsequent work.

In SAL, user goals and aims are modelled as arrays, respectively, `Goals` and `Aims`, which are parameters of the `User` module. Each element in `Goals` is a record with the following fields:

guard A predicate, denoted `grd`, that specifies when the goal is enabled, for example, due to the relevant device prompts.

choice A predicate (choice strategy), denoted `choice`, that models a high-level ordering of goals by specifying when a goal can be chosen. An example of the choice strategy is: "choose only if this goal has not been chosen before."

aims A set of records consisting of two fields, denoted `aims`, that essentially models the principle of relevance. The first one, `state`, is a reference to an aim (predicate) in the array `Aims`. The conjunction of all the predicates referred to in the set `aims`, defined by the predicate `Achieved(g)` for a goal `g`, fully specifies the model states the user intends to achieve by executing this goal. For the top goal, denoted `TopGoal`, this conjunction coincides with the main task goal. The second field, `ignore`, specifies a set of goals that are irrelevant to the aim specified by the corresponding field `state`. Note that the same effect could be achieved by providing a set of "promising" actions. However, since in our approach the relevance of a goal is generally interpreted in a very wide sense, we expect that the "ignore" set will be a more concise way of specifying the same thing.

subgoals A data structure, denoted `subgoals`, that specifies the subgoals of the goal. It takes the form `comp(gls)` when the goal consists of a set of subgoals `gls`. If the goal is atomic, its subgoals are represented by a reference, denoted `atom(act)` to an action in the array `Actions` (see below).

Goal execution. To see how the execution of an atomic goal is modelled in SAL consider the guarded command *PerformAction* for doing a user action that has been previously committed to:

$$\texttt{commit}[\mathsf{a}] = \texttt{committed} \quad \rightarrow \quad \begin{array}{l} \texttt{commit}'[\mathsf{a}] = \texttt{ready}; \\ \mathit{Transition}(\texttt{a}) \end{array}$$

The left-hand side of → is the guard of this command. It says that the rule will only activate if the associated action has already been committed to, as indicated by the element `a` of the local variable array `commit` holding value `committed`. If the rule is then non-deterministically chosen to fire, this value is changed to `ready` to indicate there are now no commitments to physical actions outstanding and the user model can select another goal. Finally, $\mathit{Transition}(\texttt{a})$ represents the state updates associated with this particular action `a`.

The state space of the user model consists of three parts: input variable `in`, output variable `out`, and global variable (memory) `mem`; the environment is modelled by a global variable, `env`. All of these are specified using type variables and are instantiated for each concrete interactive system. The state updates associated with an atomic goal are specified as an action. The latter is modelled as a record with the fields `tout`, `tmem`, `tenv` and `time`; the array `Actions` is a collection of all user actions. The `time` field gives the time value associated with this action (see Section 2.3). The remaining fields are relations from old to new states that describe how two components of the user model state (outputs `out` and memory `mem`) and environment `env` are updated by executing this action. These relations, provided when the generic user model is instantiated, are used to specify $\mathit{Transition}(\texttt{a})$ as follows:

```
t′= t + Actions[a].time;
out′ ∈ {x:Out | Actions[a].tout(in,out,mem)(x)};
mem′ ∈ {x:Memory | Actions[a].tmem(in,mem,out′)(x)};
env′ ∈ {x:Env | Actions[a].tenv(in,mem,env)(x) ∧ possessions}
```

Since we are modelling the cognitive aspects of user actions, all three state updates depend on the initial values of inputs (perceptions) and memory. In addition, each update depends on the old value of the component updated. The memory update also depends on the new value (`out′`) of the outputs, since we usually assume the user remembers the actions just taken. The update of `env` must also satisfy a generic relation, *possessions*. It specifies universal physical constraints on possessions and their value, linking the events of taking and giving up a possession item with the corresponding increase or decrease in the number (counter) of items possessed. For example, it specifies that if an item is not given up then the user still has it. The counters of possession items are modelled as environment components.

PerformAction is enabled by executing the guarded command for selecting an atomic goal, *CommitAction*, which switches the commit flag for some action `a` to `committed` thus *committing* to this action (enabling *PerformAction*). The fact that a goal `g` is atomic is denoted `atom?(Goals[g].subgoals)`. An atomic goal `g` *may* be selected only when its guard is enabled and the choice strategy for `g` is true. For the reactive actions (goals), their choice strategy is a predicate that is always true. In the case of pre-determined goals, we will frequently use the strategy "choose only if this goal has not been chosen before". When the user model discharges such a goal, it will not do the related action again without an additional reason such as a device prompt.

The last conjunct in the guard of *CommitAction* distinguishes the cases when the selected goal is `ExitGoal` or not. `ExitGoal` (given as a parameter of the `User` module) represents such options as "cancel" or "exit", available in some form in most of interactive systems. Thus, a goal `g` that is not `ExitGoal` may be selected only if there exists a relevant aim `p` in the set `Goals[g].aims`, denoted $\mathit{Relevant}(\mathtt{g}, \mathtt{p})$. We omit here the formal definition of the relevance condition. On the other hand, if `g` is `ExitGoal` then it can be selected only when either the task goal has been achieved (user does not intend to finish interaction before achieving main goal), or there are no enabled relevant goals (the user will try relevant options if such are available). Again, we omit the formal definition of these conditions here just denoting them $MayExit$.

When an atomic goal `g` is selected, the user model commits to the corresponding action `act(Goals[g].subgoals)`. The time variable `t` is increased by the value associated with "cognitive overhead" (see Section 2.3). The record `status` keeps track of a history of selected goals. Thus, the element `g` of the array `status.trace` is set to true to indicate that the goal `g` has been selected, and the counter of selected goals, `status.length`, is increased. In addition to time-based analysis, this counter provides another way of analysing the behaviour of the user model.

Task completion. There are essentially two cases when the user model terminates an interaction: (i) goal based completion when the user terminates upon achieving the task goal, and (ii) no-option based termination when the user terminates since there are no enabled relevant goals to continue. Goal based completion (`finished` is set to `ok`) is achieved by simply "going away" from the interactive device (see the $ExitTask$ command). No-option based termination (`finished` is set to `abort`) models random user behaviour (see the *Abort* command).

The guarded command $ExitTask$ states that the user may complete the interaction once the predicate `Achieved(TopGoal)` becomes true and there are no commitments to actions. This action may still not be taken because the choice between enabled guarded commands is non-deterministic. The value of `finished` being `notf` means that the execution of the task continues.

In the guarded command *Abort*, the no-option condition is expressed as the negation of the predicate `EnabledRelevant`. Note that, in such a case, a possible action that a person could take is to wait. However, they will only do so given some cognitively plausible reason such as a displayed "please wait" message. The waiting conditions are represented in the specification by predicate parameter `Wait`. If `Wait` is false, `finished` is set to `abort` to model a user giving up and terminating the task.

2.3 Timing aspects

Following GOMS models, we extend our cognitive architecture with timing information concerning user actions. On an abstract level, three GOMS models, KLM, CMN-GOMS and NGOMSL, are similar in their treatment of execution

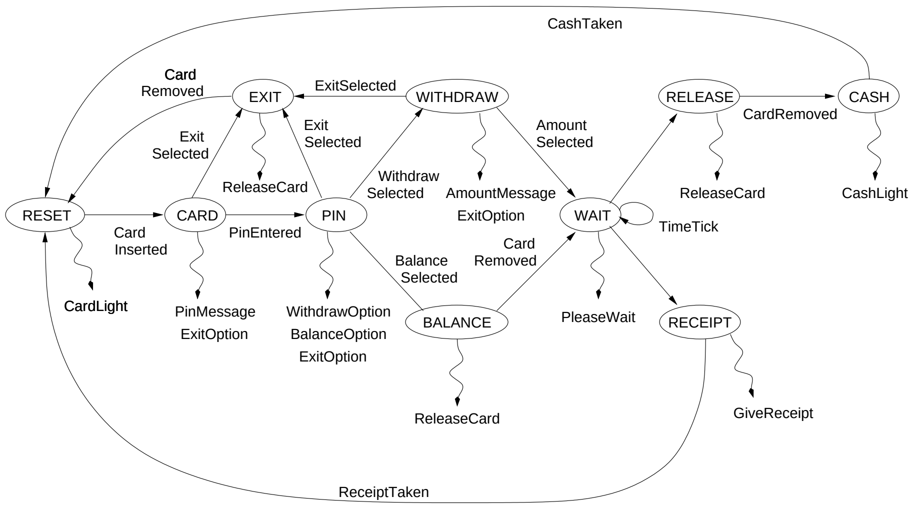

Fig. 2. A specification of the cash machine

time [7]. The main difference is that NGOMSL adds, for each user action, a fixed "cognitive overhead" associated with the production-rule cycling. In our model, this corresponds to the goal selection commands (*CommitAction*). Hence, the time variable is increased by the value `CogOverhead` which is a parameter of our user model. For KLM or CMN-GOMS-like analysis, this parameter can be set to `0`. In this case, the time variable is increased (*PerformAction* command) only by the value associated with the actual execution of action and specified as `Actions[a].time`. All three GOMS models differ in the way they distribute "mental time" among user actions, but this need only be considered when our cognitive architecture is instantiated to concrete user models. In general, any of the three approaches (or even their combination) can be chosen at this point. In this paper, we will give an example of KLM like timing analysis.

3 An example

To illustrate how the extended cognitive architecture could be used for the analysis of execution time, we consider interaction with a cash machine.

3.1 Cash machine

For simplicity of presentation, we assume a simple design of cash machine. After inserting a bank card, its user can select one of the two options: withdraw cash or check balance (see Fig. 2). If the balance option is selected, the machine releases the card and, once the card has been removed and after some delay, prints a receipt with the balance information. If the withdraw option is selected, the user can select the desired amount. Again, after some delay, the machine releases the card and, once it has been removed, provides cash. Note that users are allowed to cancel an interaction with our machine before entering the PIN, and selecting the withdraw option, balance option, or amount, i.e., while the machine is in the `CARD`, `PIN`, or `WITHDRAW` state. If they choose to do so, their card is released.

3.2 User model

Next, we instantiate our cognitive architecture to model cash machine users.

User aims. We assume there are two aims, denoted `CashAim` and `BalanceAim`, which might compel a person to use this cash machine. These predicates provide values for the array `Aims`. As an example, the predicate `BalanceAim` is as follows:

```
λ(in,mem,env): env.Receipts ≥ 1 ∨ mem.BalanceRead
```

It states that the balance is checked when either the user has at least one receipt (these are modelled as possession items), or they read the balance on the display and have recorded this fact in their memory.

User goals. Taking account of the aims specified, we assume that the machine users, based on the previous experience, have the following pre-determined goals: `InsertCardGoal`, `SelectBalanceGoal`, `SelectWithdrawGoal`, and `SelectAmountGoal`. As an example, `SelectBalanceGoal` is the following record (the others are similar):

```
grd := λ(in,mem,env): in.OptionBalance
choice := NotYetDischarged
aims := {}
subgoals := atom(SelectBalance)
```

Thus, this goal may be selected only when a balance option is provided by the interface. The choice strategy `NotYetDischarged` is a pre-defined predicate that allows one to choose a goal only when it has not been chosen before. Since this is an atomic goal, the set `aims` is empty, whereas its subgoal is the actual action (an operator in GOMS terms) of selecting the balance option (see below).

In response to machine signals, the user may form the following reactive goals: `EnterPinGoal`, `TakeReceiptGoal`, `ReadBalanceGoal`, `RemoveCardGoal`, `TakeCashGoal`, and `SelectExitGoal`. Their definitions are similar to those of the pre-determined goals, except that, in this case, the choice strategy always permits their selection.

User actions. To fulfil these goals, users will perform an action referred to in the corresponding goal definition. Thus, we have to specify an action for each of the above user goals. As an example, the output update `tout` of the `SelectBalance` action is the following relation:

```
λ(in,out0,mem):λ(out): out = Default WITH .BalanceSelected := TRUE
```

where `Default` is a record with all its fields set to false thus asserting that nothing else is done. The memory and environment updates are simply default relations. Finally, the timing of this action (field `time`) is discussed below.

Task goals So far we have introduced all the basic goals and actions of a cash machine user. Now we explain how tasks that can be performed with this cash machine are specified as a suitable `TopGoal`. Here we consider essentially flat goal structures, thus a top goal directly includes all the atomic goals as its subgoals. For the task "check balance and withdraw cash," `TopGoal` is specified as the following record:

```
grd := True
choice := NotYetDischarged
aims := { (# state := CashAim, ignore := {SelectBalanceGoal, ReadBalanceGoal} #),
         (# state := BalanceAim, ignore := {SelectAmountGoal} #) }
subgoals := comp({ InsertCardGoal, EnterPinGoal, SelectBalanceGoal, ...})
```

The interesting part of this definition is the attribute `aims`. It specifies that, while performing this task, the user model will have two aims (partial goals) defined by the predicates `CashAim` and `BalanceAim`. Furthermore, when the aim is to check the balance, the user model will ignore the options for selecting the amount as irrelevant to this aim (similarly the balance option and reading balance will be ignored when the aim is to withdraw cash). Of course, this is not the only task that can be performed with this machine. A simpler task, "check balance" (or "withdraw cash") alone, is also possible. For such a task, the specification of `TopGoal` is the same as above, except that the set `aims` now only includes the first (or second) record.

Note that in this way we have developed an essentially generic user model for our cash machine. Three (or more) different tasks can be specified just by providing appropriate attributes (parameters) `aims`.

3.3 KLM timing

In this paper, we use KLM timings to illustrate our approach. For the cash machine example, we consider three types of the original KLM operators: **K** to press a key or button, **H** to home hands on the keyboard, and **M** to mentally prepare for an action or a series of closely related primitive actions. The duration associated with these types of operators is denoted, respectively, by the constants `K`, `H` and `M`. The duration values we use are taken from Hudson *et al* [19]. These can be easily altered, if research suggests more accurate times as they are just constants defined in the model.

Since our user model is more abstract, the user actions are actually sequences of the **K** and **H** operators, preceded by the **M** operator. As a consequence, the timing of actions is an appropriate accumulation of `K`, `H` and `M` operators. For example, `InsertCard` involves moving a hand (**H** operator) and inserting a card (we consider this as a **K** operator), preceded by mental preparation (**M** operator). The time attribute for this action is thus specified as `M+H+K`. We also use the same timing for the actions `RemoveCard`, `TakeReceipt` and `TakeCash`. On the other hand, `SelectBalance` involves only pressing a button, since the hand is already on the keyboard. Thus its timing is `M+K` (similarly for `SelectWithdraw`, `SelectAmount` and `SelectExit`). `EnterPin` involves pressing a key four times (four digits of PIN), thus its timing is `M+H+4*K`. Finally, `ReadBalance` is a purely mental action, giving the timing `M`.

In addition to the operators discussed, original KLM also includes an operator, **R**, to represent the system response time during which the user has to wait. Since an explicit device specification is included into our verification approach, there is no need to introduce into the user model time values corresponding to the duration of **R**. System delays are explicitly specified as a part of a device model. For example, in our ATM specification, we assumed that system delays occur after a user selects the desired amount of cash and before the device prints a receipt (the `WAIT` state in Fig. 2).

4 Verification and timing analysis

So far we have formally developed both a machine specification and a (parametric) model of its user. Our approach also requires two additional models: those of user interpretation of interface signals and effect of user actions on the machine (see [5]), connecting the state spaces of the user model and the machine specification. In this example, these connectors are trivial – they simply rename appropriate variables. Finally, the environment specification simply initialises variables that define user possessions as well as the time variable. Thus, the whole system to analyse is the parallel composition of these five SAL modules. Next we discuss what properties of this system can be verified and analysed, and show how this is done. First we consider the verification of correctness properties.

4.1 Error analysis

In our previous work [4, 5], we mainly dealt with two kinds of correctness properties. The first one (functional correctness) aimed to ensure that, in any possible system behaviour, the user's main goal of interaction (as they perceive it) is eventually achieved. Given our model's state space, this is written in SAL as the following LTL assertion:

$$\mathtt{F}(\mathtt{Perceived}(\mathtt{in}, \mathtt{mem})) \tag{1}$$

Here `F` means "eventually," and `Perceived` is the conjunction of all the predicates from the set `Goals[TopGoal].aims` as explained earlier.

The second property aimed to catch post-completion errors – a situation when subsidiary tasks are left unfinished once the main task goal has been achieved. In SAL, this condition is written as follows:

$$\mathtt{G}(\mathtt{Perceived}(\mathtt{in}, \mathtt{mem}) \Rightarrow \mathtt{F}(\mathtt{Secondary}(\mathtt{in}, \mathtt{mem}, \mathtt{env}))) \tag{2}$$

Here `G` means "always," and `Secondary` represents the subsidiary tasks. In our example, `Secondary` is a predicate stating that the total value of user possessions (account balance plus withdrawn cash) in a state is no less than that in the initial state.

Both these properties can be verified by SAL model checkers. With the cash machine design from Fig. 2, the verification of both succeeds for each of the three tasks we specified. Note, however, that both properties only guarantee that the main and subsidiary tasks are eventually finished at some unspecified point in the future. In many situations, especially in the case of various critical systems, designs can be judged as "incorrect" on the grounds of poor performance. Next we show how efficiency analysis is supported by our approach by considering execution times.

4.2 Timing analysis

Model checkers give binary results – a property is either true or false. Because of this, they are not naturally suited for a detailed GOMS-like analysis of execution times. Still, if one is content with analysis that produces an upper (or lower) limit, model checking is a good option. For example, if it suffices to know that both the main and the subsidiary tasks are finished in time less than `T`, one can verify the condition

$$\mathtt{G}(\mathtt{Perceived}(\mathtt{in},\mathtt{mem}) \Rightarrow \mathtt{F}(\mathtt{Secondary}(\mathtt{in},\mathtt{mem},\mathtt{env}) \wedge \mathtt{time} < \mathtt{T})) \qquad (3)$$

The validity of both (1) and (3) predicts that `T` is an upper limit for the user model, and thus for any person behaving according to the cognitive principles specified, to properly finish a task. If expert knowledge is needed for such performance, SAL would produce a counter-example (a specific sequence of actions and intermediate states) for property (3). This can be used to determine design features requiring expert knowledge.

As an example, consider the task "check balance and withdraw cash". Let the threshold for slow execution times be 17 seconds (i.e. 17 000 milliseconds). The verification of property (3) with `T` equal to `17000` fails. The counter-example shows that the execution time is slow since the user model goes through the whole interaction cycle (inserting a card, entering a PIN, etc.) twice. A design allowing the task to be performed in a single cycle would improve the execution times. In the next section, we consider such a design.

By verifying property (3) for different `T` values, the estimates of the upper and lower time limits for a task execution can be determined. However, execution times given by counter-examples provide no clue as to how likely they are, in other words, whether there are many methods of task execution yielding these particular times. Neither do they give the duration of other execution methods. To gather precise timing information for possible execution methods, we use an interactive tool provided by the SAL environment, a simulator. It is possible to instruct the latter to run an interactive system so that the system states defined by some predicate (for example, `Perceived`) are reached. In general, different system states are reached by different execution methods. Thus, one can determine the precise timing of a particular method simply by checking the variable

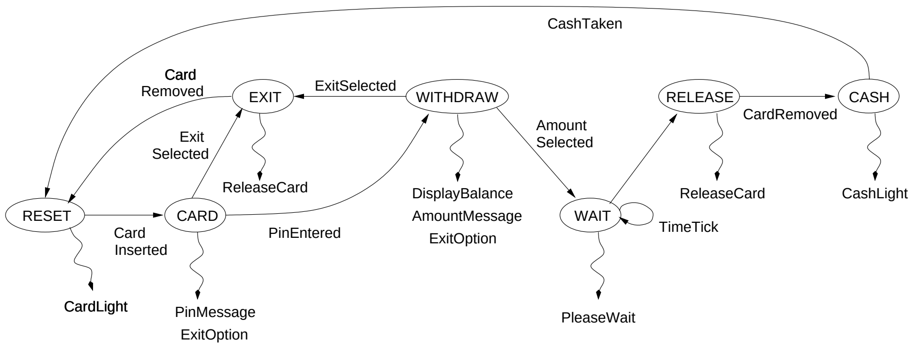

Fig. 3. A specification of the modified design

`time` in the corresponding state. A more sophisticated analysis and comparison of timing information can be automated, since the SAL simulator is a Lisp-like environment that allows programming functions for suitable filtering of required information. We will explore this in future work.

5 Modified design

An obvious "improvement" on the design is to free users from an early selection of a task. Instead, while in the `WITHDRAW` state, the machine displays the balance in addition to the amount choices (see Fig. 3). The user can read it and then choose an amount option as needed, thus achieving both task goals in one run. To check whether our expectations are valid, we run the simulator to reach system states where both predicates `Perceived` and `Secondary` are true. Checking execution time in these states indicates an improvement. To find out whether execution times improved for all possible paths reaching the above goal states, we model check property (3). However, this verification fails again. SAL produces a counter example where the user model chooses an amount option without first reading the displayed balance and, to achieve both aims, is forced to restart interaction. Furthermore, for the "improved" design, even property (2) is invalid. The SAL counter example shows that the user model, after reading the displayed balance, chooses the exit option, thus forgetting the card.

In a traditional GOMS analysis this new design is apparently fine as expert non-erroneous behaviour is assumed. However the HUM-GOMS analysis highlights two potentially systematic problems: an attention error and a post-completion error. The expert assumption is thus in a sense required here. Whilst it might be argued that an expert who has chosen that method for obtaining balance and cash would not make the mistake of failing to notice the balance when it was displayed, experimental data suggests that even experts find it hard to eliminate post-completion error in similar situations. Amongst non-expert users both errors are liable to be systematic. The HUM-GOMS analysis has thus identified two design flaws that if fixed would be significant improvements on the design.

A simple fix for both detected flaws is a cash machine similar to our second design, but which, instead of displaying the balance, prints this information and releases the receipt in the same slot and at the same time as the banknotes.

6 Conclusion

We have added support for timing analysis into our usability verification approach based on the analysis of correctness properties. This allows both timing analysis and human error analysis to be performed in a single verification environment from a single set of specifications. For this, our cognitive architecture was extended with timing information, as in GOMS models. Our approach uses the existing SAL tools, both the automatic model checkers and the interactive simulator environment, to explore the efficiency of an interactive system based on the models provided. As in our earlier work the cognitive architecture is generic: principles of cognition are specified once and instantiated for a particular design under consideration. This differs from other approaches where a tailored user model has to be created from scratch for each device to be analysed. The generic nature of our architecture is naturally represented using higher-order formalisms. This and SAL's support for higher-order specifications are the primary reasons for developing our verification approach within the SAL environment.

The example we presented aimed to illustrate how our approach can be used for a KLM style prediction of execution times (our SAL specifications are available at `http://www.dcs.qmul.ac.uk/~rimvydas/usermodel/dsvis07.zip`). A difference in our approach is that, if the goal is achieved, the user model may terminate early. Also, if several rules are enabled, the choice between them is non-deterministic. The actual execution time is then potentially a range, depending on the order – there is a maximum and a minimum prediction. These are not real max/min in the sense of saying this is the longest or shortest time it will take, however, just a range of GOMS-like predictions for the different possible paths. In effect, it corresponds to a series of KLM analyses using different procedural rules, but incorporated in HUM-GOMS into a single automated analysis.

Similarly as CCT models [20] and unlike pure GOMS, we have an explicit device specification that has its own timings for each machine response. It is likely that most are essentially instantaneous (below the millisecond timing level) and so approximated to zero time. However, where there are explicit **R** operators in KLM, the corresponding times can be assigned to the device specification.

Even though we illustrated our approach by doing a KLM style analysis, our extension of the cognitive architecture is also capable of supporting CMN-GOMS and NGOMSL approaches to timing predictions. We intend to explore this topic in future work, developing at the same time a hierarchical goal structure.

Another topic of further investigation is timing-related usability errors. We have already demonstrated the capability of our approach to detect potential user errors resulting from the device delays or indirect interface changes without any sort of feedback [4]. The presented extension opens a way to deal with real-time issues (e.g., when system time-outs are too short, or system delays are too long). We also intend to investigate "race condition" errors when two closely fired intentions to action come out in the wrong order [21]. We expect that the inherent non-determinism of our cognitive architecture can generate such erroneous behaviour in appropriate circumstances. Finally, since tool support allows experimentation be done more easily, we believe that our approach can address the scale-up issue and facilitate the analysis of trade-offs between the efficiency of multiple tasks.

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

References

1. Reason, J.: Human Error. Cambridge University Press (1990)
2. Gray, W.: The nature and processing of errors in interactive behavior. Cognitive Science **24**(2) (2000) 205–248
3. Byrne, M.D., Bovair, S.: A working memory model of a common procedural error. Cognitive Science **21**(1) (1997) 31–61
4. Curzon, P., Blandford, A.E.: Detecting multiple classes of user errors. In: Little, R., Nigay, L. (eds.): Proc. 8th IFIP Working Conf. on Engineering for Human-Computer Interaction (EHCI'01). Vol. 2254 of LNCS, Springer-Verlag (2001) 57–71
5. Rukšėnas, R., Curzon, P., Back, J., Blandford, A.: Formal modelling of cognitive interpretation. In: Doherty, G., Blandford, A. (eds.) Proc. DSV-IS 2006. Vol. 4323 of LNCS, Springer-Verlag (2007) 123–136
6. Rukšėnas, R., Curzon, P., Blandford, A.: Detecting cognitive causes of confidentiality leaks. In: Proc. 1st Int. Workshop on Formal Methods for Interactive Systems (FMIS 2006). UNU-IIST Report No. 347 (2006) 19–37
7. John, B.E., Kieras, D.E.: The GOMS family of user interface analysis techniques: Comparison and contrast. ACM Trans. CHI **3**(4) (1996) 320–351
8. de Moura, L., Owre, S., Ruess, H., Rushby, J., Shankar, N., Sorea, M., Tiwari, A.: SAL 2. In: Alur, R., Peled, D.A. (eds.): Computer Aided Verification: CAV 2004. Vol. 3114 of LNCS, Springer-Verlag (2004) 496–500
9. John, B.E., Kieras, D.E.: Using GOMS for user interface design and evaluation: which technique? ACM Trans. CHI **3**(4) (1996) 287–319
10. Beckert, B., Beuster, G.: A method for formalizing, analyzing, and verifying secure user interfaces. In: Proc. ICFEM 2006, Vol. 4260 of LNCS, Springer-Verlag (2006) 55–73
11. Bowman, H., Faconti, G.: Analysing cognitive behaviour using LOTOS and Mexitl. Formal Aspects of Computing **11** (1999) 132–159
12. Rushby, J.: Analyzing cockpit interfaces using formal methods. Electronic Notes in Theoretical Computer Science **43** (2001)
13. Fields, R.E.: Analysis of erroneous actions in the design of critical systems. Tech. Rep. YCST 20001/09, Univ. of York, Dept. of Comp. Science, D.Phil Thesis (2001)
14. Fields, B., Wright, P., Harrison, M.: Time, tasks and errors. ACM SIGCHI Bull. **28**(2) (1996) 53–56

15. Lacaze, X., Palanque, P., Navarre, D., Bastide, R.: Performance evaluation as a tool for quantitative assessment of complexity of interactive systems. In: Proc. DSV-IS 2002. Vol. 2545 of LNCS, Springer-Verlag (2002) 208–222
16. Thimbleby, H.: Analysis and simulation of user interfaces. In: Proc. BCS HCI, vol. XIV (2000) 221–237
17. Butterworth, R.J., Blandford, A.E., Duke, D.J.: Demonstrating the cognitive plausibility of interactive systems. Formal Aspects of Computing **12** (2000) 237–259
18. Newell, A.: Unified Theories of Cognition. Harvard University Press (1990)
19. Hudson, S.E., John, B.E., Knudsen, K., Byrne, M.D.: A tool for creating predictive performance models from user interface demonstrations. In: Proc. 12th Ann. ACM Symp. on User Interface Software and Technology, ACM Press (1999) 93–102
20. Kieras, D.E., Polson, P.G.: An approach to the formal analysis of user complexity. Int. J. Man-Mach. Stud. **22** (1985) 365–394
21. Dix, A., Brewster, S.: Causing trouble with buttons. In: Auxiliary Proc. HCI'94 (1994)