Innovations in Systems and Software Engineering manuscript No.
(will be inserted by the editor)

Rimvydas Rukšėnas · Paul Curzon · Ann Blandford

Modelling and Analysing Cognitive Causes of Security Breaches

Received: date / Accepted: date

Abstract In this paper we are concerned with security issues that arise in the interaction between user and system. We focus on cognitive processes that affect security of information flow from the user to the computer system and the resilience of the whole system to intruder attacks. For this, we extend our framework developed for the verification of usability properties by introducing two kinds of intruder models – an observer and an active intruder – with the associated security properties. Finally, we consider small examples to illustrate the ideas and approach. These examples demonstrate how our framework can be used (i) to detect confidentiality leaks, caused by a combination of an inappropriate design and certain aspects of human cognition, and (ii) to identify designs more susceptible to cognitively-based intruder attacks.

Keywords human error · security · cognitive architecture · formal verification · SAL

1 Introduction

There has been much research on security (confidentiality) of information flow (see Sabelfeld and Myers' overview [25]). The starting point is the assumption that computation uses confidential inputs. The goal is to ensure a *noninterference policy* [12], which essentially means that no difference in outputs can be observed between two computations that are different only in their confidential inputs. Various approaches to this problem, such as access control [3] and static information flow control [11], have been proposed, and formalisms and mechanisms developed, e.g. security-type systems [26] and type-checkers [19].

Rimvydas Rukšėnas · Paul Curzon
Dept. of Comp. Science, Queen Mary, Mile End, London E1 4NS, UK
Tel.: +44-207-8825257, +44-207-8825212, Fax: +44-208-9806533
E-mail: {rimvydas, pc}@dcs.qmul.ac.uk

Ann Blandford
UCL Interaction Centre, 31-32 Alfred Place, London WC1E 7DP, UK
E-mail: a.blandford@ucl.ac.uk

All this research focuses on the technical aspects of software systems. It aims at ensuring that the implementation of a system does not leak confidential information. However, technology is only one aspect of security. Within interactive systems, there is another actor besides a computer system – its human user. Even perfectly designed and implemented systems cannot prevent users from unwittingly compromising confidential information they have. Users can breach security for many reasons. Nevertheless, research in human-computer interaction [1; 15] reveals systematic causes of such violations, including cognitive overload, lack of security knowledge, and mismatches between the behaviour of computer systems and the mental model that their users have. Even in the absence of software errors security can be breached when the functionally correct behaviour is inconsistent with user expectations [15].

The relationship between users and security mechanisms is addressed by *user-centred security* which provides "security models, mechanisms, systems, and software that have usability as a primary motivation or goal" [27]. Much of this work takes social dimensions, considering problems like user motivation and understanding of security mechanisms, work practices, the relationships between system users, including authorities and communities of users, and threats to security exploiting social engineering techniques.

Our work lies between the technical aspects of information flow security and the social aspects of user-centred security. More specifically, we are interested in information flow; however, the locus of this flow is now not within a computer system but within the inputs provided to it by its user. We are not considering the social aspects of human-computer interaction and security. Instead, the focus of our attention is cognitive processes that affect both the information flow from the human user into the computer system and the resilience of the whole system to intruder attacks.

We build upon the generic user model (cognitive architecture) we developed in our work on usability [9]. It was developed from abstract cognitive principles, such as a user entering an interaction with knowledge of the task and its subsidiary goals. Incorporating such models of user behaviour into models of security is advocated by user-centred

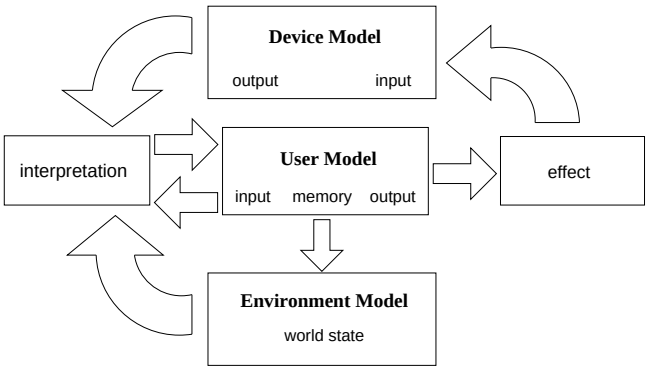

Fig. 1 The cycle of interaction

security [27]: e.g. Ka-Ping derives the guidelines (design rules) for secure interaction design from an informal user model [15].

Our cognitive architecture was later extended and restructured [22]. The new structure distinguishes *input* signals (originating from user perception), *output* signals (consequences of user actions) and user *memory* (internal state) as shown in Fig. 1. This restructuring also introduced intermediate entities that we refer to as *interpretation* and *effect*, relating the now distinct user and device state spaces. Intuitively, the effect model is an abstract view of how user actions are translated into device commands (e.g. interactive systems with voice or gesture recognition of user inputs). Similarly, the interpretation model is an abstract view of the pathways from device signals and environment objects to the user decision of what they could mean.

Our cognitive architecture has proved of use for detecting various types of systematic user errors in the context of usability and task completion [9; 22]. Here we extend our approach by introducing two kinds of intruder models: an *observer* and an active *intruder*. Our aim is to show that the behaviours emerging from the cognitive architecture also expose security problems and so facilitate the improvement of security aspects in user interaction design. To demonstrate this, we first informally discuss, from a security viewpoint, several examples of user error dealt with in our earlier work [9]. Then we consider two sets of examples using the model checking tool SAL [18] to detect potential security breaches. The first set involves confidentiality leaks emerging from our cognitive architecture and conditioned by the user interpretation of system prompts (see Section 4). More specifically, we consider security problems that may arise from the combination of user habits, shape and (relative) positioning of input fields in authentication interfaces. The second set of examples involves an adversary user, an intruder, executing a cognitive attack on a regular user (see Section 5). The examples are small and intended to illustrate an approach and ideas that we believe are more generally applicable.

Contribution. This paper is based on and extends our earlier work [23] on security verification in interaction design. Compared to that work, here we enrich our generic user model by considering the salience of user actions and the related condition for voluntary task completion. We add to our verification approach a model of an active intruder and the relevant security property. Additional examples are provided to illustrate security problems arising due to the shape-induced user confusion over interface inputs, and cognitively-based intruder attacks. Summarising, the main contribution of our work is the following:

- An investigation into the formal modelling of cognitive aspects of security breaches in interactive systems.
- An extension of our framework, developed for usability verification, to deal with the security problems in user interaction.
- Adding to our framework two intruder models: an observer and an active intruder.
- An illustrative example of confidentiality leaks, caused by cognitive interpretation and detectable by model checking using our cognitive architecture.
- An illustrative example of a security breach, caused by a cognitive attack by an intruder, and detectable by model checking using our cognitive architecture.

1.1 Related work

Whilst conducted independently and in parallel, Beckert and Beuster's work [2] takes a similar approach to ours. They also develop a formal user model, and combine it with specifications of the application and the user's assumptions about that application to verify security properties of interactive systems. Their user modelling is based on the formalisation of an established methodology, GOMS [14], which is the core of their work. The modelling of user's assumptions partially coincides with our user interpretation. However, their "assumptions" model user choice between multiple plausible options, whereas our "interpretation" deals, in addition, with the user perception of interface objects depending on their shape, position, etc. Beckert and Beuster informally define three HCI security requirements in terms of requirements on user interfaces: confidentiality (no information is leaked), integrity (user's assumptions correspond to the system's state) and availability (only desirable states can be reached). They, however, formalise only the integrity requirement, whereas correctness properties in our framework also address the remaining two. It is also unclear whether they provide tool support for automatic verification. On the other hand, their methodology supports hierarchical models: an advantage when dealing with larger systems.

In the related area of safety-critical systems, Rushby *et al* [24] 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 model user interpretation. Cerone *et al's* [7] CSP model of an air traffic control system

includes controller behaviour. A model checker was used to look for new behavioural patterns, missed by the analysis of experimental data. The classification stage in their model is similar to user interpretation.

Ka-Ping [15] gives a list of design rules, justified by an informal user model and tailored to increase security of interactive systems. As the rules are informal (many are probably too abstract to be formalised), there is no tool support for verifying whether designs obey them.

2 Cognitive architecture

Our cognitive architecture is a higher-order logic formalisation of abstract principles of cognition and specifies a form of cognitively plausible behaviour [5]. 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 [20]. Their focus is on the internal goals and knowledge of a user. It should be noted that our formal cognitive model was not developed directly from an existing, complete psychological theory. Rather we have taken an exploratory approach, starting with some simple principles of cognition, such as non-determinism of action, goal-based termination and reactive behaviour, and exploring what erroneous behaviour can emerge from them. Our paper [10] discusses the development of the cognitive model in more detail. Next we briefly present our cognitive principles.

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.

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 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.

Salience. Even though user choices are non-deterministic, they are affected by the salience of possible actions. For example, taking money released by a cash-point is a more salient, and thus much more likely, action to take than to terminate the interaction by walking away from the machine without cash. In general, salience could be affected by several factors such as the sensory (visual) salience of an action, its procedural cueing as a part of a learned task, and the cognitive load imposed by the complexity of the task performed. In this paper we abstract from these aspects and simply associate higher or lower (for whichever reason) salience with pre-determined goals and reactive actions. In a related work [21], however, we model in more detail both salience and how it is affected by task complexity (memory load).

Voluntary task completion. A person may decide to terminate the interaction. As soon as the main task goal has been achieved, users intermittently, but persistently, terminate interactions [6], 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. Users also may terminate interactions when the signals from the device or environment suggest that task continuation is impossible due to some fault. For example, if the cash-point signals that the inserted card is invalid (and therefore retained), a person is likely to walk away and try to contact their bank.

Table 1 A fragment of the SAL language

Notation	Meaning
`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 `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 its field `x` updated by `e`
`g → 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 c_i with `i` in range `T`

Forced task termination. If there is no apparent action that a person can take that will help to complete the task then the person is forced to 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 will give up, assuming the goal is not achievable.

2.2 Cognitive architecture in SAL

We have formalised the cognitive principles within the SAL environment [18]. 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. 2, 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. The SAL specification is a transition system. Non-determinism is represented by the non-deterministic choice, `[]`, between the named guarded commands (i.e. transitions). For example, *CommitGoal* in Fig. 2 is the name of a family of transitions indexed by `a` within range `GoalRange` (a type parameter). Each guarded command in the specification describes an action that a user *could* plausibly take. The pairs *CommitGoal – PerformGoal* of the corresponding transitions reflect the connection between the physical and mental actions. The first of the pair models committing to a pre-determined goal, the second actually taking the corresponding action (see below).

Action execution. To see how action execution is modelled in SAL consider the guarded command *PerformGoal* for doing a user action that has been previously committed to:

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

The left-hand side of $\rightarrow$ 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 `gcommit` 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, $\mathit{GoalTransition}(a)$ 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. Each goal is specified by a record with the fields `grd`, `slc`, `tout`, `tmem` and `tenv`. All these fields except `slc` are *higher-order* parameters (functions) that are applied to the concrete states in the instantiations of the generic user model. The `grd` and `slc` fields are discussed below. 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 discharging this goal. These relations, provided when the generic user model is instantiated, are used to specify $\mathit{GoalTransition}(a)$ as follows:

$$\begin{array}{l} \texttt{out}' \in \{\texttt{x:Out} \mid \texttt{goals[a].tout(in,out,mem)(x)}\}; \\ \texttt{mem}' \in \{\texttt{x:Memory} \mid \texttt{goals[a].tmem(in,mem,out}')\texttt{(x)}\}; \\ \texttt{env}' \in \{\texttt{x:Env} \mid \texttt{goals[a].tenv(in,mem,env)(x)} \wedge \\ \qquad\qquad \mathit{possessions}\} \end{array}$$

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 ($\texttt{out}'$) 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.

PerformGoal is enabled by executing the guarded command for selecting a pre-determined goal, *CommitGoal*, which switches the commit flag for some action `a` to `committed` thus *committing* to that action (i.e., enabling *PerformGoal*). The predicate `grd`, extracted from the `goals` parameter, is a form of a pre-condition that specifies when there are opportunities to discharge this user goal. The predicate `salient` is true when the goal `a` has salience `Higher` (as specified by the `slc` field of the corresponding element of `goals`), or all the pre-determined goals and reactive actions are of salience `Lower`. In the latter case, we assume that one of them will become salient. Because we assign `done` to

```
DEFINITION
  gcomm = ∃(a:GoalRange): gcommit[a] = committed;
  rcomm = ∃(a:ReactRange): rcommit[a] = committed

TRANSITION
  [](a:GoalRange):CommitGoal:
     goals[a].grd(in,mem,env)
     ∧ salient(a,...)
     ∧ gcommit[a] = ready      → gcommit'[a] = committed
     ∧ NOT(gcomm ∨ rcomm)
     ∧ finished = notf
[]
  [](a:ReactRange):CommitReact:
     react[a].grd(in,mem,env)
     ∧ salient(a,...)
     ∧ rcommit[i] = ready      → rcommit'[a] = committed
     ∧ NOT(gcomm ∨ rcomm)
     ∧ finished = notf
[]
  [](a:GoalRange):PerformGoal:
     gcommit[a] = committed    → gcommit'[a] = done;
                                 GoalTransition(a)
[]
  [](a:ReactRange):PerformReact:
     rcommit[a] = committed    → rcommit'[a] = ready;
                                 ReactTransition(a)
[]
  ExitTask:
     (Perceived(in,mem,env) ∨
      BrokenState(in,mem,env)
      ∧NOT(EnabledHighGoals(...))
      ∧NOT(EnabledHighReact(...)))  → finished' = ok
     ∧ NOT(gcomm ∨ rcomm) ∧
     ∧ finished = notf
[]
  Abort:
     NOT(EnabledGoals(in,mem,env))          finished' =
     ∧ NOT(EnabledReact(in,mem,env))        IF Wait(...)
     ∧ NOT(Perceived(in,mem,env))     →     THEN notf
     ∧ NOT(gcomm ∨ rcomm)                   ELSE abort
     ∧ finished = notf                      ENDIF
[]
  Idle:
     finished = notf    →
```

Fig. 2 User model in SAL (simplified)

the corresponding element of the array `gcommit` in the *PerformGoal* command, once fired the command below will not execute again. If the user model discharges a pre-determined goal, it will not do so again without an additional reason such as a device prompt.

Reactive actions are modelled by the pairing *CommitReact – PerformReact* in the same way as pre-determined goals but on different variables, e.g. parameter `react` (a record with the same fields as `goals`) of the `User` module rather than `goals`. `ReactRange` is a type parameter similar to `GoalRange`. *ReactTransition* is specified in the same way as *GoalTransition*. The array element `rcommit[a]` is reassigned `ready` rather than `done`, once action `a` has been executed, since reactive actions, if prompted, *may* be repeated.

Task completion. In the user model, we consider three ways of terminating an interaction. Voluntary task completion (`finished` is set to `ok`) can be achieved in two ways (see the `ExitTask` command): when the main task goal, as the user perceives it, has been achieved; or when continuing the interaction seems impossible due to an apparent break in the device. Forced termination (`finished` is set to `abort`) models random user behaviour (see the `Abort` command).

Thus the guarded command *ExitTask* models two ways of voluntary task completion. It states that the user may complete the interaction if either of the predicates `Perceived` or `BrokenState` becomes true. However, in the latter case (due to an apparent break), the user model makes this choice only if there are no enabled actions of salience `Higher` expressed as the negation of the predicates `EnabledHighGoals` and `EnabledHighReact`. This assumption agrees with experimental data [16; 17] which suggests that "just in time" cues can almost completely eliminate the premature termination of tasks. The value of `finished` being `notf` means that the execution of the task continues. Since the choice between enabled guarded commands is non-deterministic, the *ExitTask* action may still not be taken. Also, it is only possible when there are no earlier commitments to other actions.

In the guarded command *Abort*, the condition of forced termination (no enabled actions) is expressed as the negation of the predicates `EnabledGoals` and `EnabledReact`. Note that, in such a case, a possible action that a person could take is to wait. However, the user model 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.

3 Verification of Security Aspects in User Interaction

In this section, we discuss examples of user error, focussing on the security aspects of interaction. We first introduce the properties to verify.

3.1 Correctness properties: usability and security

Previously, our approach dealt with two kinds of usability properties. First, we want to be sure 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 as the SAL assertion

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

where `F` means 'eventually'. Second, in achieving a goal, subsidiary tasks are often generated that the user must complete to complete the task associated with their main goal. If the completion of the subsidiary tasks is represented as a predicate, `Secondary`, the required condition is (where `G` means 'always'):

$$\mathtt{G}(\mathtt{finished} \neq \mathtt{notf} \Rightarrow \mathtt{F}(\mathtt{Secondary}(\mathtt{in},\mathtt{mem},\mathtt{env}))) \quad (2)$$

This guarantees that the secondary goal is always eventually achieved once the perceived goal has been. In addition, it also requires that the condition `Secondary` is eventually restored when the user terminates an interaction either voluntarily or simply because there is no other option to choose. Often secondary goals can be expressed as interaction invariants [9] which state that some property of the system state, that was perturbed to achieve the main goal, is restored. Previously, we viewed property (2) in terms of pure usability, applying it to, e.g. user possessions. The verification of (2) can, however, also be used to detect security problems.

As well as these two usability properties, we introduce two correctness properties, relevant to security of interaction. Both properties assume an intruder. The first one concerns confidentiality leaks in user input and is expressed in terms of a passive intruder, an *observer*, who simply monitors user input on low security channels. Intuitively, one would like to prevent confidentiality leaks in all system states, so we are aiming at a safety property. In terms of information flow security [25], let us have, for simplicity, two confidentiality levels of user inputs, "high" and "low". A safety property that addresses some security aspects is that high inputs cannot be observed on low channels. A boolean, `ObserverSuccess`, represents system states that breach this. The property, stating that it is always true there is no such security breach, is then:

$$\texttt{G}(\texttt{NOT}(\texttt{ObserverSuccess})) \tag{3}$$

We explain when `ObserverSuccess` is set to true, thus indicating breaches, in Section 4.

Note that neither of the first two correctness properties capture confidentiality leaks modelled as `ObserverSuccess`. Property (1) is first of all a usability property; the essential condition is a user achieving the main goal. The fact that this goal might be achieved by first making a mistake then undoing the erroneous action is irrelevant. The example from Section 4 illustrates this. With respect to security, however, undo is not good enough [15]: an erroneous action could already have leaked confidential information. Though checking property (2) can reveal some security problems related to, e.g. post-completion errors (see below), it is still a liveness property. As such, it does not require a system to satisfy the condition `Secondary` in all states, only at some point after the main goal has been achieved.

The other security property we consider involves an *active intruder* who at some point may take actions to achieve their goal. The property states that there is no system state where the intruder goal is achieved:

$$\texttt{G}(\texttt{NOT}(\texttt{IntruderGoal}(\texttt{in},\texttt{memint},\texttt{env}))) \tag{4}$$

We illustrate this property with a relevant example in Section 5.

3.2 User error and security

Erroneous actions are the proximate cause of failure, since it was a particular action that caused the problem: e.g. a user entering data in the wrong field. To eliminate the problem, however, one must consider the ultimate causes of an error. In our framework, we consider situations where the ultimate causes are aspects (limitations) of human cognition that have not been addressed in the interface. An example is that a person enters data in a particular field because the interface design suggests it as appropriate for that data. In Hollnagel's terms [13] which distinguish between human error *phenotypes* (classes of erroneous actions) and *genotypes* (the underlying, e.g. psychological, cause), our cognitive architecture deals with genotypes. Since there is no evidence that security errors are conditioned by different cognitive causes to usability errors, our cognitive architecture can exhibit behaviours leading to security problems, even though it was developed without security concerns in mind. Some of these errors have the same cognitive causes as the usability errors we dealt with in our earlier work [9]. Next we discuss several types of user error, related to security but still detectable within the usability based approach represented by properties (1) and (2).

A persistent user error that emerges from the cognitive architecture is the post-completion error [6], where a user terminates an interaction with completion of subsidiary tasks outstanding. People have been found to make such errors even in lab conditions [6]. An example of this error, which is also a security breach, is when, with old cash machines, users persistently took cash but left their bank card. Within our cognitive architecture, such behaviour emerges because of an action (*ExitTask*) that allows a user to stop once the goal has been achieved. Using our verification framework, this is detected by checking property (2). For this, the predicate `Secondary` would state that the total value of user possessions (bank cards included) in a state is the same as it was before the interaction. It can also be verified that design changes eliminate this error genotype. The formal verification of a similar example is described in [9].

Blandford and Rugg [4] give an example of an extant security breach caused by users forgetting to log out when moving away from an industrial printer, leaving it vulnerable to sabotage – e.g. by unauthorised users changing the printed message, etc. Being a case of the post-completion error, it can be detected by verifying property (2) with the appropriately chosen predicate `Secondary`.

Previously we [22] also considered user error due to the shape-induced confusion over the meaning of interface prompts. The example was that of a user attempting to top-up a phone card using an ATM. We showed how model checking, based on our cognitive architecture, can identify user confusion as to which of two numbers, phone number or top-up card number, is requested. The property checked was of type (1), i.e. whether the user achieves the main goal. User confusion in a similar situation can also result in confidentiality leaks.

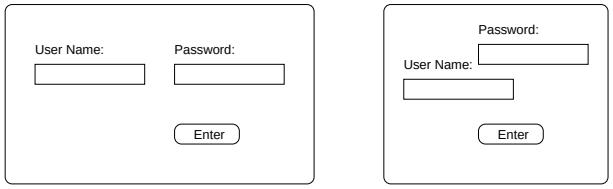

Fig. 3 (a, b) Two layouts of an authentication interface

In this paper, we extend our previous work and investigate how other security problems, not considered in that work, can be detected using our cognitive architecture formalised in SAL. In Section 4, we consider how user interpretation, due to cognitive reasons, can lead to confidentiality leaks potentially observed by an intruder. In Section 5, we take a look at a cognitively-based attack of an intruder which can result in financial losses.

4 Authentication Interface: an Example

In this section, we consider how user habits in combination with some designs, can lead to the incorrect interpretation of interface prompts, resulting in the leakage of confidential information. To determine when such leaks are possible, we introduce into our framework a new entity, generic module `observer`. This module is instantiated by providing a collection of channels and security sensitive (high security) data. The observer model thus derived monitors whether the security sensitive data appears on any of the low security channels.

4.1 An Authentication Interface

Our example concerns an authentication step present in various everyday interactive systems, e.g. internet banking. Before any transaction, users must establish their identities by providing a user name and a login password. The system checks whether the provided password is the same as the one associated with the provided user name and stored in the system's database. On the surface, one could expect the design of an authentication interface to be simple, e.g. like the one in Fig. 3 (a). In reality, the situation is more complicated. For example, the size of such an interface window on a computer screen is not fixed; users might adjust it at any time. This means that the layout of input fields is determined by some algorithm. Depending on how good/bad this algorithm is, the layout shown in Fig. 3 (b) is possible when the window size is reduced. We will first demonstrate that the two interfaces are not equally secure and will show how confidentiality leaks in the second one can be detected using our verification framework.

We assume that a high security channel is associated with login passwords and a low security channel with user names. This could mean, e.g., that the text entered into the name box is echoed on the screen whereas an entry into the password box is hidden. The data is sent whenever the users press the `Enter` button. The operation of the authentication mechanism is illustrated by a finite state machine in Fig. 4 (interface outputs are indicated in the states when they appear on the screen).

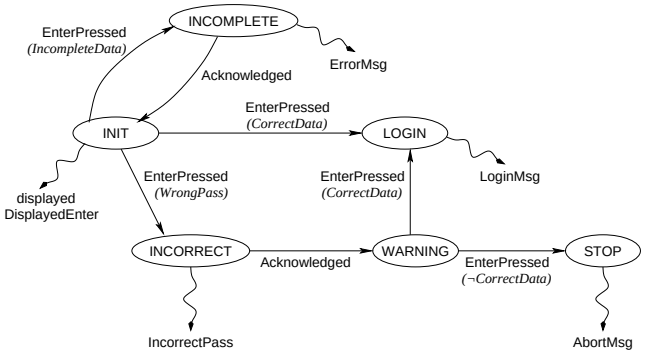

Fig. 4 Authentication procedure

We distinguish two cases of incorrect input represented by the transitions *IncompleteData* and *WrongPass*. The authentication procedure moves into the state `INCOMPLETE` when `Enter` is pressed and either (i) the text entered into the name field is not recognized as a user name, or (ii) it is but the password is missing from the password field. An appropriate error message is displayed by the interface in each case: `IncorrectName` or `MissingPass`, respectively (we assume that these messages are read by the user). The authentication procedure then returns to the `INIT` state. The transition *WrongPass* represents the case when both a user name and a password are provided but the password is incorrect. An error message, `IncorrectPass`, indicates this, also warning the user that the next attempt to login with invalid data will be the last one, and the procedure moves into the `WARNING` state. The idea is that, for security reasons, a single authentication attempt with an incorrect password is allowed before the authentication procedure aborts the interaction (`STOP` state). Finally, authentication succeeds if the user provides correct data, represented by the `LOGIN` state reachable from either the `INIT` or `WARNING` state.

The corresponding SAL specification of the authentication procedure, module `authentication`, is a straightforward translation of the diagram in Fig. 4. The input boxes are modelled as the type `Inbox = {A, B}`. Each box has a number of attributes: position, security level, "visibility", label, shape, text entered and text displayed, all modelled as arrays with the range `Inbox`. Thus, `position[j]` is a record with the coordinate fields `x` and `y`, denoting the top-left corner of box `j`. Its width and height are represented by the constants `dx` and `dy`. The attribute `level[j]` is the security level of `j` which is either `Low` or `High`; `displayed[j]` indicates whether `j` is visible or not. The attribute `label[j]` is a value of type `{NameLabel, PasswordLabel}`, whereas `shape[j]` is of type `{OnePart, TwoPart}`. In this example, we assume that the shape of both input boxes is `OnePart` (a single continuous box as in Fig. 3) and corre-

sponds to the user name and password. Finally, `value[j]` and `display[j]` represent, respectively, text entered and text displayed, which are different when the entered text is hidden. The array `value` and boolean `EnterPressed` are the inputs of the authentication procedure, whereas the arrays `label`, `shape`, `position`, `displayed`, and `display` with the booleans `DisplayedEnter`, `IncorrectName`, `MissingPass`, `IncorrectPass`, `LoginMsg`, and `AbortMsg` are its outputs.

4.2 A User Model

Now we instantiate the generic module `User` for the authentication task. We start by specifying the state space of the concrete user model. For each input box `j`, we assume that a person either sees it or not, and perceives its label, shape and the text displayed, represented by `seen[j]`, `label[j]`,`shape[j]` and `value[j]`, respectively. The perception of whether the `Enter` button is active is denoted by `EnterActive`. The person also perceives whether an error, warning or successful authentication message is given, denoted by `ErrorMsg`, `WarningMsg` and `LoginMsg`, respectively. Variables `InputName` and `InputPass` denote the perception of which of the two boxes prompts for the user name and which for the password. Finally, `name` and `password` denote the values the person perceives as a user name and password. All these components form a record type, `In`, which is used to instantiate the corresponding type variable in `User`.

Next, we specify variables related to the actions users might take. The text typed into box `j` is represented by `value[j]`. The boolean `EnterPressed` denotes whether the `Enter` button is pressed. These components form a record type, `Out`. We assume users remember their user name, `name`, and login password, `password`. They also remember whether they already typed information into box `j`, denoted `entered[j]` (reset to false when an error message is read), and keep track of whether there was a failure to authenticate, denoted `failed`. These form a record type, `Mem`, which also records, in a component of the type `Out`, the actions taken in the previous step. The reality surrounding our system is given by a record type, `Env`. It includes the user name, `name`, and the correct password, `password`.

Pre-determined goals. We assume that user knowledge of authentication includes the need to communicate (1) user name and (2) login password. This knowledge is specified as pre-determined goals (elements of array `goals`) instantiated by giving the action guard and the updates to the output component. For the goal of communicating the user name, the guard is that an input box, regarded as the name box, is seen. Also, we assume that it is impossible to take this action (as all others) at the same time as the `Enter` button is being pressed. The output action is to enter the name as the user perceives it:

```
grd := λ(in,mem,env): in.seen[in.InputName] ∧
                      NOT(out.EnterPressed)
tout := λ(in,out0,mem):λ(out):
         out = Default(out0.value)
         WITH .value[in.InputName] := in.name
tmem := λ(in,mem0,out):λ(mem): mem = mem0
         WITH .entered[in.InputName] := TRUE
         WITH .out := out
```

Here `Default(x)` is a record with the field `value` set to `x` and all other fields set to false thus asserting that nothing else is done. The memory update records the fact of entering the password and the action taken. Since the environment updates change nothing (in all the actions), they are omitted here. The action of communicating the login password is specified similarly. We are not concerned with salience in this example, so the same inconsequential value `Lower` is assumed for all user goals and reactive actions.

Reactive actions. We assume that the user might *respond* to the displayed `Enter` button by pressing it. For this to happen, the user must not have the recollection of a failure to authenticate. Alternatively, if there was such a failure, we expect the user to be more careful and only press `Enter` when both input boxes are filled in:

```
grd := λ(in,mem,env):
        in.EnterActive ∧ NOT(out.EnterPressed)
        ∧ (NOT(mem.failed) ∨ careful)
tout := λ(in,out0,mem):λ(out):
         out = Default(out0.value)
               WITH .EnterPressed := TRUE
```

Here `careful` stands for the following condition:

```
∀(j): in.seen[j] ⇒
      mem.entered[j] ∨
      in.value[j] ≠ EmptyVal ∧ NOT(in.ErrorMsg)
```

Intuitively, it states that, for any input box perceived by the user, (i) the requested data has been entered anew, or (ii) there is no error message perceived by the user, and the corresponding box contains data entered previously.

We also assume that the user acknowledges interface messages by reading them. This action is only possible if it was not taken in the previous step, as indicated by the output component `out.EnterPressed` being true (this means that the previous action was pressing the `Enter` button). By acknowledging an error message, the user model records in the memory the fact of a failed authentication attempt, and "forgets" that data has been entered into the input boxes (since the data was rejected), formally specified as:

```
grd := λ(in,mem,env): out.EnterPressed
tout := λ(in,out0,mem):λ(out):
         out = Default(out0.value)
tmem := λ(in,mem0,out):λ(mem): mem =
         IF in.ErrorMsg THEN mem0
           WITH .failed := TRUE
           WITH .entered := [[j:Inbox] FALSE]
           WITH .out := out
         ELSE mem0 WITH .out := out ENDIF
```

As discussed earlier, the need to communicate the name and password is modelled as user goals. However, it is plausible that the user makes an error when trying to achieve those goals, e.g., enters a wrong password or presses `Enter` when an input box is empty. Errors can also occur due to user habits; relying on previous experience, the user might expect the input box for the name to precede that for the password. In such cases, the system prompts for a new authentication attempt. We assume that the user will respond to this prompt. The response is modelled as two reactive actions. In the case of the password, the action guard is that an input box is seen (as for the corresponding pre-determined goal), the user remembers failing earlier and the password was not already entered, as indicated by the memory. The output (and memory) update is the same as for the corresponding pre-determined goal:

```
grd := λ(in,mem,env):
        in.seen[in.InputPass] ∧ mem.failed
        NOT(mem.entered[in.InputPass]) ∧
        NOT(out.EnterPressed)
tout := λ(in,out0,mem):λ(out):
        out = Default(out0.value)
        WITH.value[in.InputPass]:=in.password
```

The reactive action for entering the name is analogous to that of entering the password.

Instantiation. The goal, wait and break predicates are the last parameters required to instantiate the generic module `User` from Fig. 2. Thus, the display of `LoginMsg` confirms authentication which is the main goal. We also assume that there are no signals that a user would perceive as a suggestion to wait nor as an indication that further interaction is impossible. The corresponding predicates are formally specified as follows:

```
Perceived = λ(in,mem,env): in.LoginMsg
Wait = λ(in,mem,env): FALSE
BrokenState = λ(in,mem,env): FALSE
```

Finally, the concrete user model for the authentication task, `AuthUser`, is derived by instantiating the generic user model with the parameters (goals, reactive actions, perceived goal, wait and break conditions) just defined:

```
AuthUser =
 User[goals,react,Perceived,Wait,BrokenState]
```

4.3 User Interpretation

So far we have specified an authentication interface and have developed a formal model of its user. As in reality, the state spaces of the two specifications are distinct. The changing interface state is first attended to then interpreted by the user. Our approach requires two additional models: those of user interpretation of interface signals and action effect on the machine [22], connecting the state spaces of the user model and the machine specification. In this example, the effect specification is trivial – it simply renames appropriate variables. Next we specify user interpretation as a SAL module, `interpretation`. The module, being a connector, has input variables that are the output variables of the interface, and an output variable that is the input (perception) component of the `AuthUser` module (record `in`).

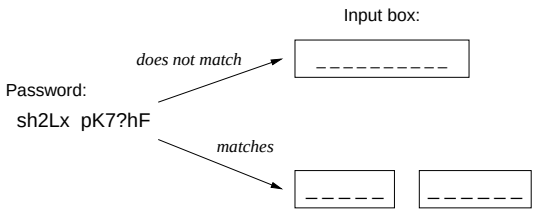

Fig. 5 Interpretation of input boxes based on their shape

In the authentication task, the crucial aspect of user interpretation is the perception of the meaning (function) of the two input boxes. Though their function is indicated by labels, in reality, however, people frequently do not pay sufficient attention to the labels. Their interpretation might be based on the shape of input boxes or the habit-induced expectation that the name box comes first. The perception of precedence depends on the layout (coordinates) of boxes in the interface window. Formally, we define the condition indicating that the input box `i` precedes `j` as follows (`p` is an array of coordinates):

```
precedes(i,j,pos) =
  (p[i].x + dx < p[j].x ∧ p[i].y ≤ p[j].y) ∨
  (p[i].x ≤ p[j].x ∧ p[i].y + dy < p[j].y)
```

Intuitively, this means that `j` is placed to the right and to the bottom of `i`. Thus, the name box in Fig. 3 (a) precedes the password box, whereas neither of the boxes in Fig. 3 (b) precedes the other.

Now we formally define the user interpretation of the function of input boxes, depending on their layout, shape and labelling. We distinguish four cases. First, the user might judge the function of input boxes from their labels (`l` stands for the array of labels):

```
ByLabel(l,x) =
  l[x.InputName] = NameLabel ∧
  l[x.InputPass] = PasswordLabel
```

Second, if the input box `i` precedes `j` then `i` is perceived as a name and `j` as password box:

```
ByPrecedence(p,x) =
  precedes(x.InputName,x.InputPass,p)
```

Third, if the shapes of the input box are different, their function might be inferred depending on how their shapes match the actual user name or password (see Figure 5):

```
ByShape1(s,e,x) =
  s[x.InputName] ≠ s[x.InputPass] ∧
  s[x.InputName] = e.NameShape ∧
  s[x.InputPass] = e.PassShape
```

```
DEFINITION in ∈ { x:In |
  IF NOT(gcomm ∨ rcomm) THEN
    x.EnterActive = DisplayedEnter ∧ x.seen = displayed ∧
    x.label = label ∧ x.shape = shape ∧ x.value = display ∧
    x.LoginMsg = LoginMsg ∧ x.WarningMsg = IncorrectPass ∧
    x.ErrorMsg = (IncorrectName ∨ MissingPass ∨ IncorrectPass) ∧
    x.name = IF x.WarningMsg THEN env.name ELSIF mem.name ENDIF ∧
    x.password = IF x.WarningMsg THEN env.password ELSIF mem.password ENDIF ∧
    x.InputName ≠ x.InputPass ∧
    IF MajorChanges(p,position,l,label,s,shape,d,displayed) ∨ x.WarningMsg THEN
      ByLabel(label,x) ∨
      IF displayed[A] ≠ displayed[B] THEN ByShape2(shape,displayed,mem,env,x)
      ELSIF x.WarningMsg THEN ByLabel(label,x)
      ELSE ByPrecedence(position,x) ∨ ByShape1(shape,env,x) ∨ Random(label,position,x) ENDIF
    ELSE x.InputName = st.InputName ∧ x.InputPass = st.InputPass ENDIF
  ELSE x = st ENDIF }

TRANSITION
    st′ = in; l′ = label; s′ = shape; p′ = position; d′ = displayed
```

Fig. 6 User interpretation in SAL (module `interpretation`)

Here `s` stands for the array of shapes, and `e` for the environment.

Finally, the user might get confused. This is possible when (i) the labels of input boxes are the same, or (ii) neither of the boxes precedes the other. In addition, the shapes of the corresponding boxes must match the shapes of the user name and the password. The judgment about the function of the boxes is random in this case:

```
Random(p,l,s,e,x) =
  s[x.InputName] = e.NameShape ∧
  s[x.InputPass] = e.PassShape ∧
  (l[x.InputName] = l[x.InputPass] ∨
   ∀(i,j):NOT(precedes(i,j,p)))
```

User interpretation is modelled as a SAL definition which allows one to describe system invariants. Intuitively, this means that the left-hand side of an equation is updated whenever the value of the right-hand side changes. We assume that, once the user makes a mental commitment to a goal or reactive action, the interpretation of the interface outputs does not change until the associated physical action is performed. If there is no commitment, the user directly perceives the `Enter` button, the input boxes with their labels, shapes and displayed text, and the interface messages. Hence the first seven conjuncts in Fig. 6 simply rename the interface variables to the corresponding fields of the record `in`.

Next we assume that any of the messages `MissingPass`, `IncorrectPass` and `IncorrectName` is interpreted as an error message. For the user name and password, the user relies on the memory unless a warning message is displayed. If so, we expect the user to be careful enough to provide the correct values. For simplicity, here we do not consider how this is actually achieved (perhaps they are taken from a notebook), assuming that the values from the environment specification are used.

As explained earlier, the perception of which of the two boxes is for the names and which for the passwords is more complicated; the results of this perception are assigned to `InputName` and `InputLabel`, respectively. We assume that the user model interprets the name and password boxes as being distinct, thus $\texttt{x.InputName} \neq \texttt{x.InputPass}$. If there are no "major" interface changes (defined below) and no warning message is given, the interpretation of the boxes remains the same as in the previous step. The auxiliary variables `st`, `l`, `s`, `p` and `d` are used to store the previous interpretation (see the `TRANSITION` section); this allows specifying that user interpretation does not change.

Otherwise, if there are major interface changes or the warning message is displayed (and noticed), the interpretation based on labels is always possible. In addition, however, other interpretations might be non-deterministically chosen as follows. If only one of the input boxes is displayed (the case $\texttt{displayed[A]} \neq \texttt{displayed[B]}$), then the interpretation based on its shape and user habits, defined by the predicate `ByShape2`, is possible. We will consider this case in Section 4.6; here we assume that both input boxes are always displayed. Otherwise, if both boxes are displayed and the warning message is perceived, we assume that the user becomes more careful and interprets the input boxes by their labels. However, if there is no perception of the warning message, an interpretation based on shapes (`ByShape1`) or precedence, or random interpretation might be non-deterministically chosen.

As major interface changes, we consider a change in the precedence of input boxes, or any label, shape and position change as well as the appearance/disappearance of an input box:

```
MajorChanges(l0,l,s0,s,p0,p,d0,d) =
  ∃(i,j): precedes(i,j,p0) ≠ precedes(i,j,p) ∨
          l0[i] ≠ l[i] ∨ s0[i] ≠ s[i] ∨
          p0[i] ≠ p[i] ∨ d0[i] ≠ d[i]
```

Admittedly, our attempt to formally specify how the user perceives input boxes already hints at potential problems, even before the actual verification. Note, however, that we

aim at developing a generic model of interpretation which would turn the specification process into a simple instantiation of the generic model.

4.4 The Interactive System

We have specified an authentication procedure, its user model and user interpretation. The authentication task is modelled as an interactive system that consists of the five components shown in Fig. 1. This system, denoted `System`, is defined to be a parallel composition of the corresponding SAL specifications:

```
(AuthUser [] authentication [] environment)
||
(interpretation || effect)
```

Here, `[]` denotes asynchronous (interleaving) composition, whereas `||` denotes synchronous composition.

The `effect` module specifies how the user actions from `AuthUser` are translated into the machine commands; in other words, how the output component `out` of `AuthUser` is connected to the `authentication` inputs. In this example, the translation is simple renaming:

```
value = out.value
EnterPressed = out.EnterPressed
```

The `environment` module contains no transitions; it simply defines constants such as the user name and password, and the shape of the input boxes.

4.5 Verification

Now the correctness properties of our interactive system can be analysed. We start from the interface with no constraints on the layout of the input boxes (other than that they do not intersect). The usability property (1), the user eventually achieving the perceived goal, is satisfied by the interactive system. Next we proceed with the analysis of security aspects of the system.

In this we also take a generic approach. Thus we specify a generic module, `observer`, which is composed with an interactive system and models a passive intruder who simply monitors the communication between the device and the user. When this intruder model succeeds the variable `ObserverSuccess` is set to true. What communication aspects are monitored is determined by the instantiation of the `observer` module. It has three parameters. The type variable `Chan` represents the communication channels. The predicate `filter` specifies which of the channels are monitored. Finally, `test` denotes security sensitive data. When this data appears on a monitored channel, `ObserverSuccess` is set to true. The transitions of the module are the following family of commands:

```
[](j:Chan): filter(j) ∧ value[j]=test →
            ObserverSuccess′ = TRUE
```

Thus channels `j` that can lead to observer success are those where `j` is monitored (`filter(j)` is true) and data sent through `j` is security sensitive (`value(j)=test`).

For our authentication task, `Chan` is instantiated to the input boxes (type `Inbox`). The security sensitive data is the actual user password `env.password`. Finally, the channels monitored are low security channels:

```
filter(j) = (level[j] = Low)
```

This yields a concrete observer module, `AuthObserver`.

The interactive system that also includes an observer is then specified in SAL as the following module:

```
ObsSystem = System [] AuthObserver
```

Now we check property (3) for `ObsSystem`; the verification fails. The counterexample produced by SAL indicates that the user enters the password (security sensitive data) into the name box (monitored channel). The analysis of the specifications reveals that this counterexample occurs because neither of the boxes precedes the other which confuses the user.

Why was this confusion not detected when verifying property (1)? The answer is that it does not prevent the user from achieving the main goal, authenticating their identity. Our user model is "smart" enough to recover from the mistake made due to this confusion and, after receiving a warning message, to provide the required information according to the labels of the input boxes. Such a recovery, however, is not good enough from the security point of view, since the mistake could have already breached security and undoing or redoing a wrong action cannot undo the consequences of this breach in most cases, which is detected by the failure to establish property (3).

How can we avoid this security breach? Since the confusion leading to it is caused by the layout of the input boxes, one solution is to display them as in Fig. 3(a). However, one must be careful even with such a layout. If the password box preceded the name box, people might enter their password into the name box, due to their habits (`ByPrecedence` condition) rather than confusion. Again, this security breach was detected by analysing (3). The latter holds only when the layout of the input boxes is such that `precedes(InputName,InputPass,position)` is true. Next we consider an alternative solution to the identified security problem.

4.6 A Modified Authentication Interface

In the previous section we saw that displaying both input boxes simultaneously increases security risk due to user errors. An obvious solution to the identified problem seems to be sequencing the display of input boxes so that at any time instance the user can enter only one data item (user name or password). The diagram of a modified design that implements this idea is depicted in Fig. 7. Since, in most respects, this design is the same as the original one, we discuss only

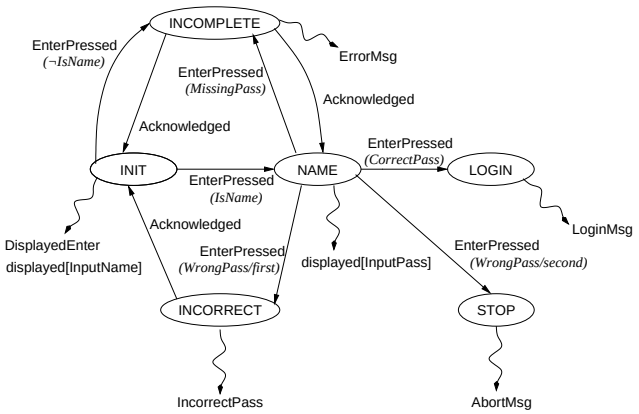

Fig. 7 Modified authentication procedure

their differences. First, the new design displays a single input box initially. To eliminate the habit-based user error due to the precedence of boxes, that is an input box for the user name. If the entered symbols are recognised as a user name, the authentication procedure displays an input box for the password next. The major modification, however, is the response to entering an incorrect password for the first time. In such a case, the interface requests to start authentication procedure from the beginning by displaying an input box for the user name.

User model. The users of the modified interface can take the same actions as with the original one. Hence we use the same user and interpretation models (`AuthUser` and `interpretation`) as before. As mentioned earlier, the user interpretation of an input box based on habits and its shape is defined by the predicate `ByShape2` in the case when only one box is displayed (the case `displayed[A]` $\neq$ `displayed[B]` in Fig. 6). This predicate is now defined as follows. If the user (i) remembers that the user name has been entered, or (ii) perceives the warning message (which informs that the entered password is incorrect), then the displayed box is interpreted as a password box when its shape matches the actual password. Otherwise, it is interpreted as a name box, if its shape matches the user name. Formally:

```
ByShape2(s,d,m,e,x) =
 IF m.entered[x.InputName] ∨ x.WarningMsg THEN
  d[x.InputPass] ∧ s[x.InputPass] = e.PassShape
 ELSE
  d[x.InputName] ∧ s[x.InputName] = e.NameShape
 ENDIF
```

Verification. We proceed with the verification of security property (3) for the modified design using the same observer model `AuthObserver` as before. Surprisingly, the verification fails. The counterexample involves an execution trace that indicates that the user model first makes a mistake by typing a wrong password (into the password box). Next the user model recovers from this errors and enters the correct password. However, it turns out that the password is typed into the low security name box, which is registered by the `AuthObserver` model.

The cause of this security breach is a combination of a user habit (the assumption that the interface asks to re-enter only the password, if the first attempt was unsuccessful) and a choice of the interface design (the shape of a name box matches the user password). In this situation, one of the possible user interpretations is defined by `ByShape2` which leads to the identified problem. It is unrealistic to change user habits; on the other hand, the design can be easily improved by ensuring that the shape of the name box does not match user passwords. We investigate this solution next.

We change the specification of design from Fig. 7 so that `shape[InputPass] = TwoPart` which models the input box at the bottom of Fig. 5 (in the original specification, the shapes of both boxes were modelled as `OnePart`, corresponding to the top box in the same figure). Verification shows that the security property 3 holds for the design with this modification. Furthermore, property (1) is also verified to be true. This shows that, with the final design, the user model always securely achieves the goal of authentication.

5 Cash-point: an Example

In the previous section, we showed how some security breaches (confidentiality leaks) can be detected within our framework using a model of a passive intruder. In this section, we consider how our approach can be used for investigating a design's resilience to the attacks from an active intruder. To illustrate this, we use a new example of an intruder who targets the cognition of cash-point users. Note that this example involves a different authentication procedure based on bank cards and PINs. The example also illustrates different outcomes of our analysis, depending on the salience of interface components.

5.1 A Cash-point Specification

We consider a simple cash-point that supports only the basic service – providing balance information and cash. Once a card is inserted, the machine checks its validity. If the card is invalid, it is retained, and an appropriate message is displayed. Then the machine returns to the initial state. If the card is valid, the machine asks the user to enter a PIN. Once this is done, the user can select one of the two options: withdraw cash or check balance (see Fig. 8). 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 cash 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. Users may cancel an interaction with this machine before entering a PIN, and selecting service and withdrawal amount, i.e., while the machine is in the `CARD`, `PIN`, or `WITHDRAW` state, respectively. If

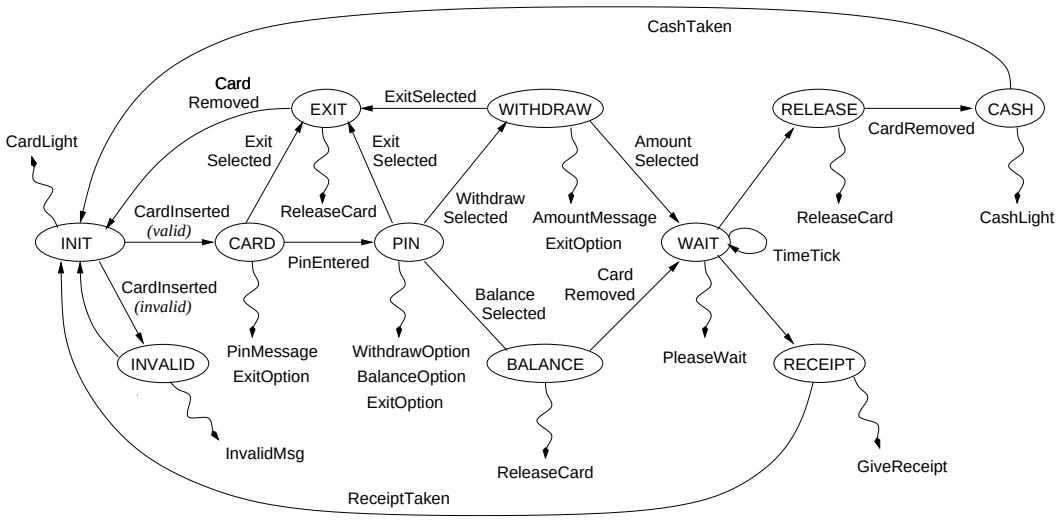

Fig. 8 Cash-point design

they choose to do so, their card is released. The corresponding SAL specification of the cash-point, module `ATM`, is a straightforward translation of the diagram in Fig. 8.

5.2 A User Model

In this example, we consider a user whose goal is to obtain cash. Therefore, we assume that the balance and exit options are not used. Our instantiation of the generic user model reflects this.

User actions. We assume that the machine users, based on their previous experience, have the following predetermined goals: `InsertCard`, `SelectWithdraw`, and `SelectAmount`. As an example, `SelectWithdraw` is the following record (the others are specified in similar ways):

```
grd := λ(in,mem,env): in.OptionWithdraw
slc := Lower
tout := λ(in,out0,mem):λ(out): out =
        Default WITH .WithdrawSelected:=TRUE
```

where `Default` is a record with all its fields set to false thus asserting that nothing is done. The memory and environment updates (omitted) are likewise default relations. Thus, `SelectWithdraw` may be selected only when the withdraw cash option is provided by the interface (and observed by the user). We assume that there is no reason to consider this action as highly salient. The output action is simply to choose the withdraw cash option, for example, by pressing the appropriate button.

In response to machine signals, the user model may take one of the following reactive actions: `EnterPin`, `RemoveCard`, and `TakeCash`. The definition of `TakeCash` is given below, the others are similar.

```
grd := λ(in,mem,env): in.TakeCash
slc := Higher
tout := λ(in,out0,mem):λ(out):
        out = Default WITH .CashTaken:=TRUE
```

This action is enabled only when the user perceives the signal to retrieve money. We assume that its salience is `Higher`. There are several justifications for this assumption. First of all, since obtaining cash is the task goal, an action achieving this is cognitively salient. Moreover, banknotes sticking out of the cash slot are visually salient. We also assume that the salience of `RemoveCard` is `Lower`. This models, for example, an interface where the card slot is further away from the display (the focus of user attention) and there is no audio signal indicating the released card. The salience of all the other actions is specified as `Lower`.

Instantiation. Informally, the main user goal is to withdraw cash. We model this by counting the number of withdrawals. Thus, the perceived goal is expressed as a predicate stating that this number is at least one (initially it is zero). The user model waits only if it perceives the machine signal `PleaseWait`. Finally, we assume users are aware that the machine can retain an invalid card. Therefore, if this happens (indicated by `InvalidMsg`), the user model perceives further interaction as apparently impossible. The corresponding predicates are formally specified as follows:

```
Perceived = λ(in,mem,env): env.Withdraws ≥ 1
Wait = λ(in,mem,env): in.PleaseWait
BrokenState = λ(in,mem,env): in.InvalidMsg
```

The updates of `env.Withdraws` are specified by the generic relation *possessions*. The number of withdrawals is modelled as an environment component, since it, as an objective fact, does not depend on user cognition.

The concrete user model for the withdrawal task, module `ATMuser`, is derived by instantiating the generic user model with the parameters (goals, reactive actions, perceived goal, wait and break conditions) just defined.

5.3 An Intruder Attack

Next we explore whether the above interactive system is resilient to an intruder attack. The scenario of the attack, based on real life cases, is as follows. The intruder installs a device that produces a voice message saying that the user's card is invalid and therefore retained by the machine. It is cognitively plausible that the user's response to this message is to walk away and try to contact their bank. If this happens at a "right" time (when the user already entered the PIN), the intruder will have an opportunity to withdraw money from the user's account.

Note that the above scenario involves an active intruder taking appropriate actions. It therefore makes sense to derive the intruder model by instantiating our generic user model. We describe such an instantiation next.

Intruder model. We assume that the intruder can perform the same actions as the user model specified earlier. The exception is the actions `InsertCard` (the intruder does not intend to use their own cards) and `EnterPin` (the intruder does not know another person's PIN). The remaining actions `SelectWithdraw`, `SelectAmount`, `RemoveCard`, and `TakeCash` are instantiated as the reactive actions of our intruder model.

The intruder's goal is similar to the user's – to make a withdrawal. To achieve this, the intruder is always prepared to wait. Finally, the intruder may always disregard the machine's messages informing them about some break in interaction. The corresponding predicates are formalised as follows:

```
IntruderGoal =
    λ(in,mem,env): env.IntrWithdraws ≥ 1
IntruderWait = λ(in,mem,env): TRUE
IntrBrokenState = λ(in,mem,env): FALSE
```

The intruder model, `ATMintruder`, is derived by instantiating our generic user model with the parameters just defined. In addition, we assume that the intruder can only operate when the user is not interacting with the machine (`finished` $\neq$ `notf`). This guard is thus added to our intruder specification.

We assume that the intruder perceives interface signals in the same way as the user; the interpretation model therefore remains the same. The effect specification takes into account that some of the actions performed can be taken both by the user and by the intruder. For the action of taking cash, this is defined as follows:

```
CashTaken = out.CashTaken ∨ outintr.CashTaken
```

For the actions of selecting the withdraw cash option and amount, and taking the card, the corresponding definitions are similar. The definitions for the remaining actions simply rename appropriate variables. Finally, the environment specification initialises variables that define both the user's and the intruder's possessions.

Interactive system. The whole interactive system is specified as the following parallel composition of the corresponding SAL modules:

```
(ATMuser [] ATMintruder [] ATM [] environment)
||
(interpretation || effect)
```

Verification. Now we are ready to check whether the intruder's goal can be achieved by performing the attack described earlier on the cash-point design from Fig. 8. For this we check security property (4). The verification produces a counterexample. It shows that the user model, after entering the PIN, terminates the interaction when a fake invalid card message is heard. At this point, the intruder model steps in and completes the interaction, withdrawing cash from the user's bank account. Is it possible to modify this cash-point design so that the same intruder attack would fail? We investigate this next.

5.4 A Safer Design

Presumably, the success of the intruder attack relied on a time gap between authentication and the actual cash withdrawal during which the user (or intruder) had to perform several actions (select the withdraw cash option and amount). Perhaps, reducing this time gap will guard against this type of intruder attacks?

The modified cash-point design is depicted in Fig. 9. Its interface retains the same functionality. The major difference from the previous design is that the authentication steps (inserting card and entering PIN) are now requested at the end of an interaction. They only have to be performed once the user provided all information relevant to the requested service.

A new interactive system includes the modified cash-point; the other models (user, intruder, etc.) remain the same. We check property (4) for the new system. However, the verification again produces a counterexample. As previously, the user model, after entering the PIN, terminates the interaction when a fake invalid card message is heard, at which point the intruder model steps in. However, this time the interaction is terminated even though the card has been released (`PIN` state). The analysis of the user model shows that this is possible since the salience of the reactive action `RemoveCard` is `Lower`, making the predicate `EnabledHighReact(...)` (see *ExitTask* in Fig. 2) false. This corresponds to the situation where a person walks away not noticing the released card because its salience is low.

Next we continue our analysis of the design in Fig. 9 by assuming that the salience associated with the user action `RemoveCard` has been increased, for example, by placing the card slot next to the display:

```
slc := Higher
```

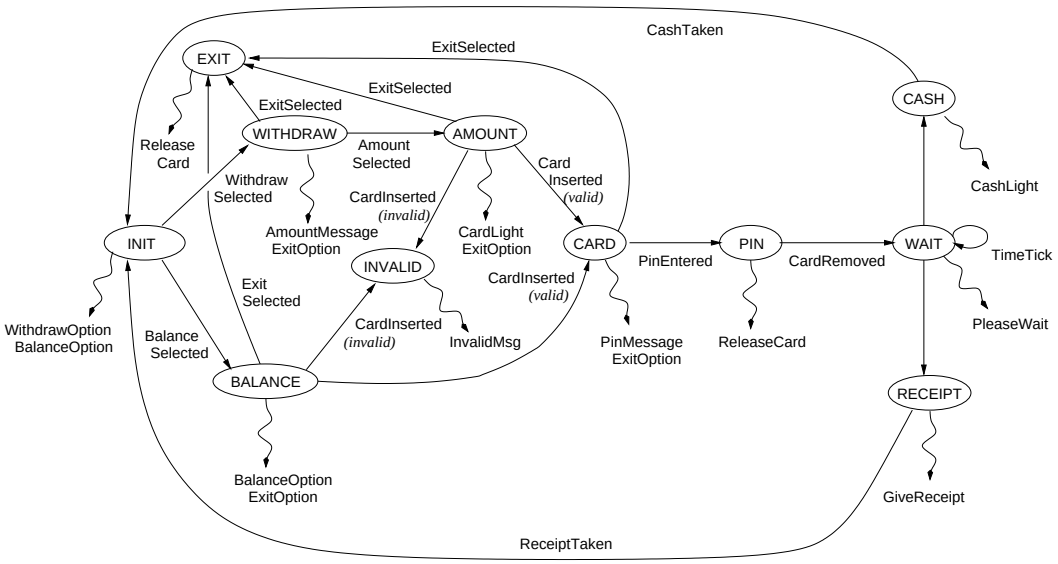

Fig. 9 Modified cash-point design

Now property (4) holds. Thus, even though our user model perceives a fake invalid card message, the card released at the same time is noticed (and taken) due to its high salience. The user model then waits (since a please wait message is displayed) until the requested cash is released, thus preventing the intruder model from taking it.

Verification, however, also shows that both usability properties (1) and (2) are still false. This is possible, since a fake message might be produced (`CARD` state) before the user model enters a PIN so that the task is terminated while the card and the cash have not been released. Be that as it may, it is unavoidable as long as we assume that users may respond to the invalid card message by terminating the interaction and walking away. Note, however, that both these properties – the user withdrawing cash, and post-completion errors like forgetting the card not occurring – are proved for the new interactive system, if no invalid card messages are produced.

Summarising, the verification results imply that users whose behaviour is consistent with our user model can always successfully perform the task of withdrawing cash, if there is no intruder attack considered here. Moreover, if such an attack is attempted, they are not liable to financial losses when interacting with the cash-point specified by Fig. 9. In fact, a sequence of user actions analogous to the one in Fig. 9 is required, for example, to top-up oyster cards in London underground. This provides higher security compared to that of regular cash-points.

Finally, our verification shows that the intruder attack can succeed even with the design from Fig. 9 when the salience associated with the actions of removing card and taking cash is low. Note, however, that the consequences of security breach in this case are less severe compared to the original design from Fig. 8. With the modified design, the intruder could only withdraw the amount of cash the user selected earlier, whereas any amount can be chosen by the intruder with the original design. Thus, our verification identified a cash-point design that, depending on the plausibility of our salience assumptions, is either completely secure against the attack discussed, or the consequences of this attack are less severe.

6 Conclusion

In previous work, we assumed that usability verification is enough to establish user-centred correctness of interactive systems. This is not true within security- or mission-critical contexts where it is possible to achieve the main goal while exposing ourselves to various security or safety risks. In this paper, we discussed how security breaches can be detected using our earlier framework. The main focus, however, was an extension of that framework to address security aspects more directly and link them to the usability properties.

We considered two aspects of interactive systems security – information-flow security and intruder attacks targeting user's cognition. To address the first aspect, we introduced into our framework a generic observer model which monitors information flow between the user and the device, and reveals potential security breaches in that communication. Using the observer model we added to our verification approach a security property that captures some confidentiality leaks in interactive systems. To address the second aspect, we introduced a model of an active intruder, derived from our generic user model, and the related security property. This allows us to model intruder attacks and verify the resilience of interactive systems to them.

Research on the technical aspects of information flow security seeks to improve the implementations of software systems so that possibilities are reduced for deducing, from the values observed on the low channels, what values appear on the high channels. The focus of our work is different

from and complementary to this research. We are concerned with detecting interface designs that, in combination with the limitations of human cognition, directly prompt humans to enter high security data into the low channels, thus potentially leaking this information to intruders. Though human limitations generally cannot be eliminated, interface designs can be modified so that the probability of such a leakage is reduced.

Admittedly, in our analysis, we relied on simple security properties. Still, even they enabled us to detect security issues that were encountered in real life designs. Similarly, a simple formulation of property (2) allows detecting a wide range of usability problems related to post-completion error. This leads us to believe that even analysis based on simple security properties will be useful in practice. In future research, we also intend to explore more complex security properties.

Our verification approach relies on using formal models of cognitively plausible user behaviour. It is *generic* in at least two ways. First, the user (also intruder) model itself is generic and serves as a pattern for the construction of concrete user models by its instantiation. Thus, by providing suitable concrete instantiations, our models can be used for analysing security of various interactive systems. Second, relying on formal models of cognitively plausible behaviour also means that our analysis is concerned with the *genotypes* [13] of security issues, i.e., their underlying cognitive causes. While the specific manifestations of these issues might vary in different domains, they are related to the same features of human cognition and can be detected by our analysis.

Our observer model is very general, detecting security sensitive data when it is transmitted through any low security (monitored) channel. The approach here is that the active intruder models are derived for each specific scenario as suitable instantiations of the generic user model. An alternative would be to develop a generic intruder model, different from the regular user model and possibly incorporating the observe behaviour. The question which of these two approaches is advantageous is a topic for further research.

To illustrate our approach, we considered two sets of examples.[1] A simple authentication interface demonstrated how the layout and shape of input fields combined with user habits can influence the user (mis)interpretation of interface prompts, potentially leading to confidentiality leaks. The cash-point example demonstrated how our approach is used to investigate cognitively-based intruder attacks.

The cash-point example also illustrated the role cognitive salience plays in our analysis. Well-designed interfaces increase the sensory salience of signals that are used to cue actions that are frequently forgotten or are performed in the wrong sequence. Chung and Byrne [8] found that a sensory signal has to be highly visually salient in order to ensure that a task-critical step, buried deep down in the task structure, is not forgotten. Non-sensory cues, known as procedural cues (internal to the cognitive system), can be used to retrieve previously formulated intentions (expert procedural knowledge) enabling the next procedural step to be performed. Remembering that, and so doing, after performing *x* always do *y* if *z* is true is an example of following a procedural cue rule. The approach here was to abstract from these aspects by simply associating with the user actions high or low salience (as determined by HCI experts). In a related work [21], we further develop our formalisation by distinguishing procedural and sensory salience and formally relating it to cognitive load.

We showed how security breaches are detected using the SAL verification tools, and how the analysis of the counterexamples produced can help in eradicating cognitively susceptible interface designs and/or identifying more secure designs. The SAL environment was primarily chosen because of its support for higher-order specifications. This is necessary for developing a generic cognitive architecture as ours.

The interpretation model was introduced into our framework from general considerations. We previously showed how modelling user interpretation allows us to detect usability problems due to the shape-induced confusion over the meaning of device prompts [22]. Here we showed that similar ideas apply in the context of security properties and their dependence on the layout and shape of input fields. Finally, we also considered user habits, which we had not dealt with before.

Human factors in the security context have been considered before [1; 15]. The novelty of our approach is dealing with the cognitive aspects of security in a completely formal way, making them amenable to automatic verification. Moreover, our cognitive architecture could be used to prove generic results on, e.g., design rules for security, using its version [10] formalised within the HOL theorem prover.

Acknowledgements This research has been funded by EPSRC grants GR/S67494/01 and GR/S67500/01.

[1] The SAL sources of these examples are available from `http://www.dcs.qmul.ac.uk/research/imc/hum/examples/isse.zip`.

References

1. Adams A, Sasse MA (1999) Users are not the enemy. CACM **42**(12):41–46
2. Beckert B, Beuster G (2006) A method for formalizing, analyzing, and verifying secure user interfaces. In: Proc. ICFEM 2006, vol. 4260 of LNCS, Springer-Verlag, 55–73
3. Bell DE, La Padula LJ (1976) Secure computer system: Unified exposition and Multics interpretation. Tech. Rep. MTR-2997, MITRE Corp., MA
4. Blandford A, Rugg G (2002) A case study on integrating contextual information with usability evaluation. Int. J. Human-Computer Studies **57**(1):75–99
5. Butterworth R, Blandford A, Duke D (2000) Demonstrating the cognitive plausibility of interactive systems. Form. Asp. Computing **12**:237–259
6. Byrne MD, Bovair S (1997) A working memory model of a common procedural error. Cognitive Science **21**(1):31–61

7. Cerone A, Lindsay PA, Connelly S (2005) Formal analysis of human-computer interaction using model-checking. In: Proc. SEFM 2005, IEEE Press, 352–362
8. Chung P, Byrne MD (2004) Visual cues to reduce errors in a routine procedural task. In: Proc. 26th Ann. Conf. of the Cognitive Science Society, Hillsdale, NJ, Lawrence Erlbaum Associates
9. Curzon P, Blandford AE (2001) Detecting multiple classes of user errors. In: Little R, Nigay L (eds) Proc. EHCI 2001, vol. 2254 of LNCS, Springer-Verlag, 57–71
10. Curzon P, Rukšėnas R, Blandford A (2007) An approach to formal verification of human-computer interaction. Form. Asp. Computing **19**(4):513–550
11. Denning DE, Denning PJ (1977) Certification of programs for secure information flow. CACM **20**(7):504–513
12. Goguen JA, Meseguer J (1982) Security policies and security models. In: Proc. IEEE Symp. on Security and Privacy, Apr. 1982, IEEE Press, 11–20
13. Hollnagel E (1998) Cognitive reliability and error analysis method. Elsevier
14. John BE, Kieras DE (1996) The GOMS family of user interface analysis techniques: Comparison and contrast. ACM Trans. CHI **3**(4):320–351
15. Ka-Ping Y (2002) User interaction design for secure systems. In: Deng R, et al (eds) Proc. ICICS 2002, vol. 2513 of LNCS, Springer-Verlag, 278–290
16. Li SYW, Blandford A, Cairns P, Young RM (2005) Post-completion errors in problem solving. In: Proc. 27th Ann. Conf. of the Cognitive Science Society, Lawrence Erlbaum Associates, 1278–1283
17. Li SYW, Cox AL, Blandford A, et al (2006) Further investigations into post-completion error: the effects of interruption position and duration. In: Proc. 28th Ann. Conf. of the Cognitive Science Society, Lawrence Erlbaum Associates, 471–476
18. de Moura L, Owre S, Ruess H, et al (2004) SAL 2. In: Alur R, Peled DA (eds) Computer Aided Verification: CAV 2004, vol. 3114 of LNCS, Springer-Verlag, 496–500
19. Myers AC (1999) JFlow: Practical mostly-static information flow control. In: Proc. of ACM Symposium on Principles of Programming Languages, 228–241
20. Newell A (1990) Unified theories of cognition. Harvard University Press
21. Rukšėnas R, Back J, Curzon P, Blandford A (in press) Formal modelling of salience and cognitive load. In: Proc. 2nd Int. Workshop on Formal Methods for Interactive Systems: FMIS 2007, Electronic Notes in Theoretical Computer Science
22. Rukšėnas R, Curzon P, Back J, Blandford A (2007) Formal modelling of cognitive interpretation. In: Proc. DSVIS 2006, vol. 4323 of LNCS, Springer-Verlag, 123–136
23. Rukšėnas R, Curzon P, Blandford A (2007) Detecting cognitive causes of confidentiality leaks. In: Proc. 1st Int. Workshop on Formal Methods for Interactive Systems: FMIS 2006, Electronic Notes in Theoretical Computer Science **183**:21–38
24. Rushby J (2001) Analyzing cockpit interfaces using formal methods. Electronic Notes in Theoretical Computer Science **43**:1–14
25. Sabelfeld A, Myers AC (2003) Language-based information-flow security. IEEE Journal on Selected Areas in Communications **21**(1):1–15
26. Volpano D, Smith G, Irvine C (1996) A sound type system for secure flow analysis. Journal of Computer Security **4**(3):167–187
27. Zurko ME (2005) User-centered security: Stepping up to the grand challenge. In: Proc. ACSAC 2005, IEEE Press, 187–202