Chapter 7
Formal Methods for Human-Computer Interaction

Antonio Cerone

Abstract Human-computer interaction adds the human component to the operational environment of a system. Furthermore, the unpredictability of human behaviour largely increases the overall system complexity and causes the emergence of errors and failures also in the systems that have been proved correct in isolation. Rather than trying to capture and model human errors that have been observed in the past, as it has been done traditionally in human reliability assessment, we consider cognitive aspects of human behaviour and model them in a formal framework based on the CSP process algebra. We consider two categories of human behaviour, automatic behaviour, mostly representative of a user carrying out everyday activities, and deliberate behaviour, mostly representative of an operator performing tasks driven by specific goals set up within the purpose of a working context. The human cognitive model is then composed with the physical interface/system and with a number of environmental aspects, including available resources, human knowledge and experience. Finally, the overall model is analysed using model checking within the verification framework provided by the Process Analysis Toolkit (PAT). The ATM case study from Chap. 3 and a number of other case studies illustrate the approach.

7.1 Human Errors and Cognition

You are back home from work, tired and hungry. Your partner welcomes you announcing that a nice cake is coming out of the oven soon and, this time, 'properly baked'. You sniff the air and perceive a light burning smell. You then recall that last time the cake did not properly rise, probably because the oven was kept open for too long while inserting the cake and thus the

Antonio Cerone
Nazarbayev University, Nur-Sultan, Kazakhstan

M. Roggenbach et al., *Formal Methods for Software Engineering*,
Texts in Theoretical Computer Science. An EATCS Series,
https://doi.org/10.1007/978-3-030-38800-3_7

initial baking temperature was not high enough. Your partner is announcing that this time there won't be any problems with rising because

1. during the oven pre-heating phase, the temperature was set 20 degrees higher than the temperature indicated in the cake recipe,
2. when such higher temperature was reached, the oven was opened and the cake inserted (supposedly the opening of the oven would have decreased the temperature 20 degrees down, to the one indicated in the recipe), and
3. after closing the oven the temperature setting was immediately lowered to the value indicated in the recipe.

However, the burning smell you perceive is now getting stronger, clearly showing that something went wrong in performing the three-step algorithm above, which supposedly implement our 'baking task'. Your partner swears that the increase of 20 degrees is not too much, because it is a widely tested suggestion from a cooking internet forum and it is confirmed by many positive comments. Can you explain what went wrong? Well, there was some kind of cognitive error during the task execution. But which error exactly?

Normally, cognitive errors occur when a mental process aiming at optimising the execution of a task causes instead the failure of the task itself. The existence of a cognitive cause in human errors started to be understood already at the beginning of last century, when Mach stated: "knowledge and error flow from the same mental sources, only success can tell the one from the other" [Mac05]. But it took till the 1990s to understand that "correct performance and systematic errors are two sides of the same coin" [Rea90].

In our cake baking example, the three-step algorithm that implements the task is in principle correct, but the mental processes used to carry out the task may lead to a cognitive error. In fact, it is the human cognitive processing that does not perform the algorithm correctly, thus causing the error to emerge. Here, the key design point is that we cannot expect human behaviour to adapt to a specific algorithm when performing a task. It is instead the algorithm that must realise the task by taking human performance into account.

In the rest of this section we will briefly review the research trends and milestones in Formal Methods for HCI (Sect. 7.1.1) and state what we mean for *user* (Sect. 7.1.1) and *operator* (Sect. 7.1.1). Section 7.2 introduces the structure of human memory and its main cognitive processes and, in particular, short-term memory (STM), including alternative CSP-based models (Sect. 7.2.1), and long-term memory (LTM) and its further structuring (Sect. 7.2.2). Section 7.3 illustrates how to formally model human behaviour while Sect. 7.4 shows how to combine the model of the human component and the model of the interface to produce the overall model of the interactive system. Finally, Sect. 7.5 addresses the formal verification of the overall interactive system model and delves into the formal analysis of soundness and completeness of cognitive psychology theories; in Example 69.2 of Sect. 7.5.1 we will also reveal what cognitive error caused the cake to burn and why the algorithm used by your partner caused such an error to emerge.

7.1.1 Background

The systematic analysis of human errors in interactive systems has its roots in Human Reliability Assessment (HRA) techniques [Kir90], which mostly emerged in the 1980s. However, these first attempts in the safety assessment of interactive systems were typically based on *ad hoc* techniques [Lev95], with no efforts to incorporate a representation of human cognitive processes. within the model of the interaction.

During the 1980s and 1990s, the increasing use of formal methods led to more objective analysis techniques [Dix91] that resulted, on the one hand, in the notion of *cognitively plausible user behaviour*, based on formal assumptions to bind the way users act driven by cognitive processes [BBD00] and, on the other hand, in the formal description of expected effective operator behaviour [PBP97] and the formal analysis of errors performed by the operator as reported by accident analysis [Joh97]. Thus, research in the formal analysis of interactive systems branched into two separate directions: the analysis of cognitive errors of users involved in everyday-life [Cer11, CB04, CE07, RCB08] and work-related [MRO+15, RCBB14] interactive tasks, and the analysis of skilled operator's behaviour in traditionally critical domains, such as transportation, chemical and nuclear plants, health and defence [CCL08, CLC05, De 15, MPF+16, SBBW09]. The different interaction contexts of a user, who applies attention very selectively and acts mainly under automatic control [Cer11, NS86], and an operator, who deals with high cognitive load and whose attentional mechanisms risk to be overloaded due to coping with Stimulus Rich Reactive Interfaces (SRRIs) [SBBW09], have led to the development of distinct approaches, keeping separate these two research directions. However, users have sometimes to deal with decision points or unexpected situations, which require a 'reactivation' of their attentional mechanisms, and operators must sometime resort to automatisms to reduce attentional and cognitive loads.

In this chapter we propose a modelling approach [Cer16] that unifies these two contexts of human behaviour, which were traditionally considered separately in previous literature, namely

- **user**, i.e., a human who performs everyday activities in a fairly automatic way, and
- **operator**, i.e., a human who performs deliberate tasks making large use of attention explicitly.

User

User refers to ordinary people carrying out everyday activities, such as baking a cake, driving a car, using a smartphone, interacting with an ATM, etc. During such activities, users perform tasks that are initially triggered by

specific goals, and then normally proceed in a fairly automatic way until the goal is accomplished.

As an example of everyday activity let us consider the natural language description of the user interaction with an ATM in Example 65 [Cer11, Cer16].

Example 65: ATM Withdrawal Task

The user's *goal* is 'cash withdrawal' and consists of the following basic activities (listed in no specific order).

- When the interface is ready, the user inserts the card and keeps in mind that the card has to be taken back at a later stage.
- When the interface requests a pin, the user enters the pin.
- When the cash has been delivered, the user collects the cash.
- When the card has been returned, the user collects the card and no longer needs to remember to collect it.

The goal 'cash withdrawal' is achieved when the cash is collected.

Notice that there is .no specific order among the basic activities. The user performs a specific basic activity depending on the observed state of the interface associated with that activity. Normally, some ordering is driven by the specific interface with which the user interacts. If we consider the general ATM description in Example 36 from Chap. 3, we notice that all ATMs will deliver the cash only after the user has inserted the card and entered the pin. And, obviously, the card can only be returned after being inserted. Specific ATMs impose further orderings, between card insertion and pin entering as well as between card return and cash delivery. However, if you approach the ATM to start an interaction and notice some cash already delivered, and supposedly forgotten by the previous user, ...you definitely collect it! (independently of whether you give it to the bank or you keep it.) Thus the basic activity of collecting cash may even be the first to occur while performing the task.

Although the task described in Example 65 requires some practice or training, during which the novice user performs deliberate actions, then, after repeated interactions, sufficient knowledge, skill and familiarity will be acquired, thus allowing the user to perform the task in a fairly automatic way. For example, an expert user will automatically insert the card in the right slot when the interface appears in the normal ready state, which the user is familiar with (whatever such a state looks like), and without any need to look for the appropriate slot (which is automatically reached by the hand movement). Such an acquired automatic behaviour allows the user to perform the task efficiently and quickly. However automatic behaviour is also the context in which typical cognitive errors analysed in previous research are most likely to occur as we will see in Sect. 7.5.1.

Furthermore, automatic behaviour is by no means purely reactive, but actually features an implicit, latent form of attention. In this chapter, we will see that, on various occasions during automatic behaviour, deliberate and conscious low-level actions are still required and, when this happens, attention becomes explicit and takes control. We will thoroughly explore the mechanism of attention and we will see that, in some situations, it may also be activated by the failure of those very expectations that the user has developed through experience and training, thus leading to the emergence of cognitive errors in the form of inappropriate deliberate responses.

Operator

Operator refers to a human who performs a task with a general purpose whereby specific goals are set along the way. In this case, failing to achieve the goal is not a task failure, provided the system state is still consistent with the purpose. Examples are operators of an Air Traffic Control (ATC) system, a nuclear power control room, a device to administer a therapy to a patient and a machine of an industrial plant. The operator's task is normally a monitoring one, which requires the performance of deliberate actions when the observed system behaviour is assessed as abnormal.

In Example 66 we consider the natural language description of a task of an operator interacting with a ATC simulator, which shows position, direction and speed of aircraft moving withing a specific sector of the air space [CCL08, CLC05].

Example 66: ATC Task

The operator's *purpose* is to ensure that the aircraft moving through the sector remain horizontally separated by no less than the defined minimum separation distance (5000 m): failure of this requirement is called *separation violation*. Vertical separation is ignored by the simulator. The operator can see position, direction and speed of the aircraft on the screen. The operator's task involves monitoring the movement of aircraft on the screen, searching for pairs of aircraft that are in conflict, that is, which may violate separation. This task comprises the following subtasks:

- **scan the screen** searching for a pair to monitor as possibly being in conflict,
- **classify the pair** as a conflict or a non conflict,
- **prioritise the conflict** by deciding whether there is a need for a plan to resolve an identified conflict,
- **decide an action** to resolve the conflict, possibly defer it or reclassify the conflict as a non conflict while trying to work out the plan of action,

- **perform the action** that has been decided, and
- **new phase subtask** whether to go back to scan the screen or perform an action that was previously deferred or exit the ATC operator role by abandoning the purpose (end of the simulation session, which in real-life would be the end of the operator shift).

Each subtask is driven by a goal, which is set deliberately under the influence of the purpose. For instance, the **scan the screen** task is driven by the deliberately set goal of identifying a part of the air space where there might be a conflicting pair of aircraft. Such a goal has an holistic flavour, since we cannot fully characterise all parameters that the operator considers in order to identify the critical part of air space. Furthermore, not being able to achieve the goal does not represent a task failure, since it is possible that no pair of aircraft violates separation, consistently with the ATC purpose.

Similarly,

- The purpose of the operator of a nuclear plant control room is to ensure the safe functioning of the plant. This purpose results in the monitoring of all system readout, searching for readout configurations that may be indicators of anomalies: goals are deliberately set in order to check specific readout configurations but also, in a more holistic way, by considering configurations which are not normally associated with anomalies and set new subgoals to further investigate them.
- The purpose of the operator of a machine of an industrial plant is to follow standard and specific operating procedures while using the machine. The operator must make deliberate choices depending on the perceived situation and consequently set goals that are consistent with the operating procedures. Furthermore, since operating procedures refer to generic situations and are by no means exhaustive, the operator's choices are not made among a predefined set of possibilities, but normally require a global assessment of the current situation.

We can conclude that an operator cannot automatically act in response to observations, but has to globally assess the observed situation and make informed, deliberate decisions on whether to act and what to do. Goals are thus established throughout the process according to the purpose of the task.

7.2 Human Memory and Memory Processes

Following the *information processing* approach normally used in cognitive psychology, we model human cognitive processes as processing activities that make use of input-output channels, in order to interact with the external

environment, and three main kinds of human memory, in order to store information:

- **sensory memory**, where information perceived through the senses persists for a very short time,
- **short-term memory (LTM)**, also called *working memory*, which has a limited capacity and where the information that is needed for processing activities is temporarily stored with rapid access and rapid decay, and
- **long-term memory (LTM)**, which has a virtually unlimited capacity and where information is organised in structured ways, with slow access but little or no decay [DFAB04].

A usual practice to keep information in memory is *rehearsal*. In particular, *maintenance rehearsal* allows us to extend the time during which information is kept in STM, whereas *elaborative rehearsal* allows us to transfer information from STM to LTM.

7.2.1 Short-Term Memory and Closure

The limited capacity of short-term memory has been measured using experiments in which the subjects had to recall items presented in sequence. By presenting sequences of digits, Miller [Mil56] found that the average person can remember 7 ± 2 digits. However, when digits are grouped in *chunks*, as it happens when we memorise phone numbers, it is actually possible to remember larger numbers of digits. Therefore, Miller's 7 ± 2 rule applies to chunks of information and the ability to form chunks can increase people's STM actual capacity.

The limited capacity of short-term memory requires the presence of a mechanism to empty it when the stored information is no longer needed. When we produce a chunk, the information concerning the chunk components is removed from STM. For example, when we chunk digits, only the representation of the chunk stays in STM, while the component digits are removed and can no longer be directly remembered as separate digits. Generally, every time a task is completed, there may be a subconscious removal of information from STM, a process called *closure*: the information used to complete the task is likely to be removed from STM, since it is no longer needed.

We can use CSP to define a general STM model as shown in Example 67.

Example 67: Short-Term Memory: CSP Model

The STM model consists of n states `STM` i*, with* $i = 1, \ldots, n$*, where*
- *n is the STM maximum capacity,*

– *action* `store` *represents the storage of a piece of information and decreases the available capacity by one unit,*
– *action* `remove` *represents the removal of a piece of information and increases the available capacity by one unit,*
– *action* `closure` *represents the occurrence of a closure, due to the successful completion of the task, and completely clears STM, and*
– *action* `delay` *occurs every time STM is emptied and represents a time delay following the successful or unsuccessful end of the task.*

The empty STM of capacity `7` is modelled by process `STMempty` by defining it as `STM7`:

```
STMempty = STM7;
STM7 = store -> STM6 []
         delay -> STMempty []
         closure -> delay -> STMempty;
STM6 = store -> STM5 [] remove -> STM7 []
         delay -> STMempty []
         closure -> delay -> STMempty;
...
STM2 = store -> STM1 [] remove -> STM3 []
         delay -> STMempty []
         closure -> delay -> STMempty;
STM1 = store -> STM0 [] remove -> STM2 []
         delay -> STMempty []
         closure -> delay -> STMempty;
STM0 = store -> STMmanagement []
         remove -> STM1 []
         delay -> STMempty []
         closure -> delay -> STMempty;
STMmanagement = overloadedSTM ->  delay -> STMempty;
```

The attempt to store information in a full STM is handled by process `STMmanagement`, which in this example is associated with action `overloadedSTM` followed by a `delay`.

Notice that this memory model does not include the representation of the actual pieces of information that can be stored in STM. Information contents need to be represented by further CSP processes, one for every possible piece of information to define the two possible information states, stored and not stored. These further processes must synchronise with the CSP process in Example 67, thus resulting in a complex model, which is not easy to understand and manage and has limited scalability.

In order to develop a more intuitive, manageable and scalable model, we consider the CSP extension implemented in the *Process Analysis Toolkit (PAT)* [PAT19]. In particular, PAT provides integer variables and arrays as

syntactic sugar to define system states, without any need to explicitly represent such states as additional synchronising processes. Processes can be then enabled by guards, which check the current values of variables, while events are annotated with performed assignments to variables and, more in general, with any statement block of a sequential program. Notice that the statement block is an atomic action, i.e. it is executed until the end without any interruption or interleaving. However, annotated events cannot synchronise with events of other processes, i.e., the parallel composition operator treats annotated events in the same way as the interleaving operator. PAT also supports the definition of constants, either singulnnand or as part of an enumeration, which associates consecutive integer numbers starting from 0 to the enumerated constants. For example

```
#define low 0;
#define medium 1;
#define high 2;
```

are three declarations of constants, which can be globally introduced in an alternative way as the enumeration

```
enum {low, medium, high};
```

The most obvious array implementation of STM would use each position of the array to store a piece of information. Thus the size of the array would represent the STM maximum capacity. However, the retrieval of information from STM would require to go through all elements of the array. Instead, we consider the implementation in Example 67.1.

Example 67.1: Short-Term Memory: PAT Model

The STM model consists of an array `stm` *whose capacity is given by the number of possible pieces of information that can be stored. Such a number is defined as a constant* `InfoNumber`*, which, in this example, equals* `10`*. The various pieces of information (e.g.* `Info`*) are introduced using an enumeration. The STM maximum capacity is defined as a constant* `M`*, which, in this example, equals* `7`*.*

```
enum { ... , Info , ... };

#define InfoNumber 10;
#define M 7;

var stmSize = M;
var stm[InfoNumber];
```

By default all positions of the array are initialised to 0*. The storage of information* `Info` *in the STM is performed by the occurrence of event* `store` *which is enabled by guard* `stmSize` $<$ `M` *, which ensure that the STM is not full, and results in setting the* `Info`*-th position of array*

`stm` *to* `1` *and incrementing variable* `stmSize`*. This is achieved with the following construct:*

```
[stmSize < M] store {stm[Info] = 1; stmSize++}
```

The retrival and removal of information `Info` *from STM is performed by the occurrence of event* `retrieve` *which is enabled by guard* `stm[Info] == 1`*, which ensure that* `Info` *is in STM, and results in setting the* `info`*-th position of array* `stm` *to* `0` *and decrementing variable* `stmSize`*. This is achieved with the following construct:*

```
[stm[Info] == 1] retrieve {stm[Info] = 0; stmSize--}
```

Closure is achieved by resetting the contents of all positions of the `stm` array to `0` and assigning `0` to variable `stmSize`.

All aspects of closure implementation using PAT are explained in details in Sect. 7.3.3.

7.2.2 Long-Term Memory

Long term memory is divided into two types.

- **Declarative** or **explicit** memory refers to our knowledge of the world ('knowing what') and consists of the *events* and *facts* that can be *consciously* recalled:
  - our experiences and specific events in time stored in a serial form (*episodic memory*), and
  - structured record of facts, meanings, concepts and knowledge about the external world, which we have acquired and organised through association and abstraction (*semantic memory*).
- **Procedural** or **implicit** memory refers to our skills ('knowing how') and consists of *rules* and *procedures* that we *unconsciously* use to do things, particularly at the motor level.

Emotions and specific contexts and environments are factors that affect the storage of experiences and events in episodic memory. Information can be transferred from episodic to semantic memory by making abstractions and building associations, whereas *elaborative rehearsal* facilitates the transfer of information from STM to semantic memory in an organised form.

Note that also declarative memory can be used to do things, but in a very inefficient way, which requires a large mental effort in using the short-term memory (*high cognitive load*) and a consequent high energy consumption. In fact, declarative memory is heavily used while learning new skills. For

example, while we are learning to drive, ride a bike, play a musical instrument or even when we are learning to do apparently trivial things, such as tying a shoelace, we consciously retrieve a large number of facts from the semantic memory and store a lot of information into STM. Skill acquisition typically occurs through repetition and practice and consists in the creation in procedural memory of rules and procedures (*proceduralisation*), which can be then unconsciously used in an automatic way with limited involvement of declarative memory and STM.

7.3 Human Behaviour and Interaction

In this section we present how to model the human components using PAT.

7.3.1 Input as Perceptions and Output as Actions

Input and output occur in humans through senses and the motor system. In this chapter we give a general representation of input channels in term of *perceptions*, with little or no details about the specific senses involved in the perception, but with a strong emphasis on the semantics of the perception in terms of its potential cognitive effects. For instance, if the user of a vending machine perceives that the requested product has been delivered, the emphasis is on the fact that the perception of the product being delivered induces the user to collect it and not on whether the user has seen or rather heard the product coming out of the machine. We represent output channels in term of *actions*. Actions are performed in response to perceptions.

In Example 65 of Sect. 7.1.1 we can identify a number of perceptions and actions, which we describe in Example 65.1

Example 65.1: Perceptions and Actions

Perceptions:

$cardR$	the interface is perceived ready,
$pinR$	the interface is perceived to request a pin,
$cashO$	the cash is perceived delivered, and
$cardO$	the card is perceived returned.

Actions:

$cardI$	the user inserts the card,
$pinE.$	the user enters the pin,
$cashC$	the user collects the cash, and
$cardC$	the user collects the card.

7.3.2 Cognitive Control: Attention and Goals

We have seen in Sect. 7.2.2 that skill acquisition results in the creation in procedural memory of the appropriate rules to automatically perform the task, thus reducing the accesses to declarative memory and the use of the STM, and, as a result, optimising the task performance. Inspired by Norman and Shallice [NS86], we consider two levels of cognitive control:

- **automatic control** is a fast processing activity that requires little or no attention and is carried out outside awareness with no conscious effort implicitly, using rules and procedures stored in the procedural memory, and
- **deliberate control** is a processing activity triggered and focussed by attention and carried out under the intentional control of the individual, who makes explicit use of facts and experiences stored in the declarative memory and is aware and conscious of the effort required in doing so.

For example, let us consider the process of learning to drive

Example 68: Learning to Drive a Car

Automatic control is essential in driving a car and, in such a context, it develops throughout a learning process based on deliberate control: during the learning process the driver has to make a conscious effort to use gear, indicators, etc. in the right way (deliberate control) and would not be able to do this while talking or listening to the radio. Once automaticity in driving is acquired, the driver is aware of the high-level tasks that are carried out, such as driving to office and stopping along the way to buy a newspaper, but is not aware of low-level details that automatically affect the action performance, such as changing gear, using the indicator and the colour of the light, amber or red, while stopping at a traffic light or even turning and whether stopping or not at a traffic light (automatic control).

Let us consider a narrative description of the baking task illustrated at the beginning of Sect. 7.1 in terms of perception, actions and information stored in and retrieved from the STM.

Example 69: Advanced Baking Task

Assuming that we have already put all ingredients in a bowl, the sequence of activities (which may be further decomposed) is as follows.

1. All ingredients are mixed in the bowl.
2. When the mix is perceived having the right consistency, it is poured in a tin.
3. The cake baking temperature is read on a recipe or retrieved from LTM and it is then stored in STM.
4. It is planned to set initially a temperature higher than the baking temperature.
5. The oven is switched on by setting the temperature higher than the cake baking temperature, keeping in mind that the temperature will have to be eventually lowered.
6. After the set temperature is reached, which is perceived through a distinctive warning sound, the oven is opened, the tin is inserted in the oven, the oven is closed and the timer is set, keeping in mind that the cake will have to be eventually taken out of the oven.
7. The temperature setting is lowered to the cake baking temperature.
8. When the cake is baked, which is perceived through a distinctive warning sound associated with the timer, the oven is switched off.
9. The cake is removed from the oven.

Perceptions are briefly stored in sensory memory and only relevant perceptions are transfered to STM using *attention*, a selective processing activity that aims to focus on one aspect of the environment while ignoring others. We can see this focussing activity as the transfer of the selected perception from sensory memory to STM.

For both users and operators the top-level task can be decomposed in a hierarchy of goals and tasks until reaching *basic activities*, which do not require further decomposition and can be performed by executing a single action.

In our model of cognitive behaviour we consider a set Π of perceptions, a set Σ of actions, a set Γ of goals, a set Ξ of purposes, and a set Δ of pieces of cognitive information. The information that can be processed by the human memory is given by the set

$$\Theta = \Pi \cup \Sigma \cup \Gamma \cup \Xi \cup \Delta.$$

In our model, we assume that a piece of information in Θ may belong to one or more of the following categories.

- **Perception transferred to STM** (set Π): a perception transferred from sensory memory to STM as the result of attention.
- **Reference to the future** (set Σ): an action to perform at some point in the future.
- **Cognitive state** (set Δ): a description of the human knowledge about a state of the task or of the system.
- **Received/retrieved information** (set Δ): a piece of information that has been received (i.e., read or heard) or retrieved from LTM.
- **Goal** (set Γ): the outcome of the task, which is initially in STM.
- **Purpose** (set Ξ): the underlying reason for performing the task, which normally influences the goal.

All categories of information apart from purposes may be stored in STM. Therefore $STM \in 2^{\Theta \setminus \Xi}$.

Example 69.1: Categories of information

In the baking task we can distinguish the six possible categories of information.

Perception transfered to STM

The perceived sound that the oven has reached the right temperature is transferred to STM in Activity 6 and will be then retrieved once another task, which is carried out while waiting for the oven to heat, has been completed or can be interrupted (which will occur in Activity 6).

Reference to the future

References to the future action of lowering the temperature (to be performed in Activity 7) and to the action of taking the cake out of the oven (to be performed in Activity 9) are stored in STM in Activities 5 and 6.

Cognitive state

Activities 2, 3 and 6 must store a cognitive state pointing at the next basic activity in order to ensure the correct sequentialisation; in addition Activity 3 must remove its cognitive state, stored by the previous basic activity.

Received/retrieved information

The read/retrieved baking temperature (Activity 3) is transferred to STM.

Goal and Purpose

The goal of having the cake baked is initially in STM and is influenced by the purpose of baking the cake in a way that ensure proper raising.

Notice that all categories of information, except for the cognitive state and purpose, are explicit in the narrative description.

A task goal is formally modelled as

$$goal(info)$$

where $info \in 2^{\Theta \setminus \Gamma} \setminus \{\emptyset\}$ is a non-empty set of pieces of information except goals.

Information *info* characterises the accomplishment of the goal, which results in flashing out STM.

7.3.3 Automatic Control

In automatic control our behaviour is not affected by goals but is driven by perceptions plus pieces of information 'automatically' stored in STM during the top-level task processing. As an example of automatic control let us consider the natural language description of driving a car.

Example 70: Car Driving

Suppose that during working days we always drive to our office, whereas on Saturdays we drive to a supermarket, initially taking the same route as to the office, but then turning into a different road.

It might sometimes happen, especially in a situation of high cognitive load, that we actually drive to our office rather than to the supermarket, as instead we intended. The underlying *cognitive reason* (*genotype error*) of this *observed error* (*phenotype error*) is that our automatic control (not driven by the goal to go to the supermarket) may not switch to deliberate control (driven by the goal to go to the supermarket) when we reach the intersection where the two routes diverge.

For each $A \subseteq \Theta$ we define $\bar{A} = \{\bar{\imath} \mid i \in A,\ i \notin \Xi \cup \Gamma\}$ and $\hat{A} = A \cup \bar{A}$. Each element $\bar{\imath} \in \bar{\Theta}$ denotes the absence of the piece of information $i \in \Theta$. Obviously $\hat{\emptyset} = \bar{\emptyset} = \emptyset$.

We model a basic activity under automatic control (*automatic activity*) as a quadruple $(perc, info_1, info_2, act)$, where

- $perc \in \Pi$ is a perception,
- $info_1 \in 2^{\hat{\Theta} \setminus \Xi \setminus \Gamma}$ is the information retrieved and removed from STM,
- $info_2 \in 2^{\Theta \setminus \Xi}$ is the information stored in STM, and
- $act \in \Sigma$ is a human action.

The quadruple $(perc, info_1, info_2, act)$ is subsequently written as

$$info_1 \uparrow perc \Longrightarrow act \downarrow info_2.$$

We formally denote by *none* when a component of a basic activity is absent (perception, action) or is the empty set (information).

Actions may involve an interaction with the system interface or be purely human physical actions with no support from the system. A basic activity whose action is an interaction is called *interactive activity*. A basic activity whose action is a physical action is called *physical activity*. Information is kept promptly available, while it is needed to perform the current top-level task, by storing it in STM. A basic activity is *enabled* (and can be performed) when

- $info_1 \cap \Theta \subseteq STM$,
- there exists $info_3 \subseteq \Theta$ such that $info_1 \cap \bar{\Theta} = \overline{info_3}$ and $info_3 \cap STM = \emptyset$, and
- *perc* is available in the environment.

Thus the basic activity is triggered by the presence of $info_1 \cap \Theta$ in STM, the absence of $info_3 \subseteq \Theta$ from STM, with $\overline{info_3} = info_1 \cap \bar{\Theta}$, and the presence of *perc* in the environment.

The performance of the basic activity results in the removal of $info_1 \cap \Theta$ from STM, the execution of action *act* and the storage of $info_2$ in STM. Therefore, in the absence of closure, the performance of the basic activity changes the value of STM from STM to

$$STM' = (STM \backslash info_1) \cup info_2.$$

When $goal(info) \in STM$, the performance of the basic activity causes closure if

$$info \backslash \Xi \subseteq (STM \backslash info_1) \cup info_2 \cup \{perc, act\}$$

where STM is the content of STM before the performance of the basic activity. In the presence of closure, the performance of the basic activity changes the STM from STM to

$$STM' = (STM \backslash \{info_1, goal(info)\}) \cap \Gamma \cup info_2.$$

Therefore, the closure is determined by the perception, the performance of the action and some pieces of information in STM that make, possibly together with some purposes, the argument of the goal. The closure causes the removal from STM of all information except $info_2$ and the non achieved goals. Note that at least one component of the basic activity on the left of '$\Longrightarrow$' and one on its right have to be distinct from *none*. When the action is *none* and the perception present, the basic activity is an *automatic attentional activity*, in which implicit attention causes the transfer of a perception to STM. When both the action and the perception are *none*, the basic activity is called *cognitive activity*.

The automatic behaviour described in Example 65 is formalised in Example 65.2

Example 65.2: Automatic Behaviour

Let be

Perceptions:	$\Pi = \{cardR, pinR, cashO, cardO\}$,
Actions:	$\Sigma = \{cardI, pinE, cashC, cardC\}$,
Purposes:	$\Xi = \emptyset$,
Cognitive Information:	$\Delta = \emptyset$,
Goals:	$\Gamma = \{goal(cashC)\}$,

Set Ξ is empty since the purpose is not relevant here.

A simple ATM task, in which the user has only the goal to withdraw cash, is modelled by the following four basic activities:

1. $none \uparrow cardR \Longrightarrow cardI \downarrow cardC$
2. $none \uparrow pinR \Longrightarrow pinE \downarrow none$
3. $none \uparrow cashO \Longrightarrow cashC \downarrow none$
4. $cardC \uparrow cardO \Longrightarrow cardC \downarrow none$

The goal ('to withdraw cash') is formally modelled as

$$goal(cashC)$$

Initially the STM only contains the goal:

$$STM = \{goal(cashC)\}$$

All basic activities in this task are automatic interactive activities. A reference to action $cardC$ is stored in STM by Activity 1, which will then be essential in enabling Activity 4. The goal is accomplished when action $cashC$ is performed in Activity 3.

Modelling Automatic Control using PAT

Example 65.3 illustrates how to use PAT to define the infrastructure to model the closure phenomenon for the ATM task described in Example 65.2. The task aims at achieving the goal of withdrawing cash (`getCashGoal`).

Example 65.3: Closure in Automatic Control using PAT

```
enum { getCashGoal };
enum { None,
       CardR, PinR, CashO, StatO, CardO ,
       CardI, PinE, CashC, CardC,
       Interaction }; // 10 items
```

```
#define G 1;      // No. of goal
#define N 10;     // No. of stm array positions
#define M 7;      // STM maximum capacity

var stmGoal = [ 0 ];
var stm[N];
var stmSize;
var perc[N];
```

The storage of goal in STM is modelled by the one position array `stmGoal`. The content of this position is initialised to `0`. Arrays `stm` and `perc` implement the STM non-goal contents and the perceptions available in the environment, respectively.

The closure controls the removal of the achieved goal and the removal of non-goal information in order to free memory space for further processing towards the achievement of other goals. Example 65.4 illustrates how to use PAT to model the removal of goal `getCashGoal` for the ATM task.

Example 65.4: Closure in Automatic Control using PAT

```
Closure() = ba-> (
  [stmGoal[getCashGoal] == 1] cashC ->
    achieveGetCash {stmGoal[getCashGoal] = 0;
                    stmSize--;} -> FlashOut() []
  eact -> Closure() );
```

Event `ba` marks the beginning of the basic activity, `eact` marks the end of the action, and event `eba` marks the end of the basic activity. Process `Closure` is guarded by a condition on the presence of the goal in STM (`stmGoal[getCashGoal] == 1`). When the action associated with the goal is performed (`cashC` models that the cash is collected) the goal is achieved (`achieveGetCash`) and removed from STM by changing to `0` the position of the `stmGoal` array corresponding to the achieved goal (`getCashGoal`) and decrementing `stmSize`.

Example 65.5 illustrates how to use PAT to model the removal of the non-goal information for the ATM task.

Example 65.5: Closure in Automatic Control using PAT

```
FlashOut() = closure { var cell = 0;
                       while (cell < M) {
                         if (stm[cell] == 1) {
                             stmSize--;
                         };
                         stm[cell] = 0 ;
```

```
                    cell = cell + 1;
                  }
                } -> eact -> Closure();
```

Process `FlashOut` clears the contents of the non-goal part of the STM (array `stm`). In fact, the storage of goal in STM is separately implemented by array `stmGoal` to ensure that closure does not remove goals other that the achieved one.

Example 65.6 illustrates how to use PAT to initialise the task with the appropriate goal for the ATM task.

Example 65.6: Closure in Automatic Control using PAT

```
Goals() = [stmSize < M && stmGoal[getCashGoal] == 0]
                    getCash {stmGoal[getCashGoal] = 1;
                                stmSize++} -> Goals() []
          [stmGoal[getCashGoal] == 1] ba -> eba -> Goals();
```

Process `Goals` initialises the task by adding the goal (`getCashGoal`) to the STM by setting to `1` the corresponding position of the `stmGoal` array and incrementing `stmSize`, provided that the STM does not exceeds its maximum capacity (`stmSize` < `M`).

In general, the storage of goals in STM is modelled by array `stmGoal`, whose positions are initialised to `0`.

Example 65.7 illustrates how to use PAT to model the basic activities for the ATM task described in Example 65.4.

Example 65.7: Automatic Control Task using PAT

```
Task() = ba -> (
 [stmSize < M && perc[CardR] == 1]
   cardI -> eact -> store {stm[CardC] = 1; stmSize++}
   -> eba -> Task() []
 [perc[PinR] == 1]
   pinE -> eact -> eba -> Task() []
 [perc[CashO] == 1]
   cashC -> eact -> eba -> Task() []
 [stmSize > 0 &&
   perc[CardO] == 1 && stm[CardC] == 1]
   retrieve {stm[CardC] = 0; stmSize--}
   -> cardC -> eact -> eba -> Task()
               );

User() = Closure() || Goals() || Task();
```

Each basic activity $info_1 \uparrow perc \Longrightarrow act \downarrow info_2$ is defined by one choice of the `Task` process. The choice is a process guarded by

- condition `perc[P] == 1`, if $perc \neq none$, where `P` is the position of array `perc` that implements perception $perc$,
- condition `stmSize > n`, if `n+1` is the number of pieces of information in $info_1$, and one condition `stm[I] == 1`, for each piece of information $i \in info_1$, if $info_1 \neq none$, where `I` is the position of array `stm` that implements i, and
- condition `stmSize` $<$ `M-n`, where `M` is the maximum capacity of STM, if `n` is the cardinality of $info_2 \backslash info_1$.

The first event of the process implements action act. Annotated actions `store` and `retrieve` contain the assignments described in Example 67.1

Process `User` is the parallel composition of the three processes defined in Examples 65.4 and 65.7

Notice that the use of events `eact` and `eba` forces the closure to occur between the action performance and the storage of information in STM. In this way, the same basic activity that causes closure and removal of a goal or subgoal may also store a new goal or information in STM.

In Sect. 7.4 we will see how to combine this user model with an interface model. Then, in Sect. 7.5.1 we will show how to use model checking to formally verify properties of such an overall system.

7.3.4 Deliberate Control

In deliberate control, the role of the goal is not only to determine when closure should occur but also to drive the task: we act deliberately to achieve goals. Thus basic activities are not only driven by perceptions and non-goal information stored in STM, but also by one specific goal stored in STM. A typical case of deliberate behaviour is *problem solving*, in which the task goal is normally reached through a series of steps involving the establishing of subgoals. Achieving the subgoal takes the operator somehow closer to the task goal until this can be achieved directly. This process is illustrated in Example 71.

Example 71: Moving a Box

We need to move a box from point A to point B. The box is full of items. If the box is light enough then we just move it. Otherwise we have first to empty it, then move it and finally fill in it again. Emptying the box is a subgoal that allows us to move a heavy box.

In Example 71 we can identify a number of perceptions and actions as described in Example 71.1

Example 71.1: Perceptions and Actions

Perceptions:
light the box is perceived light;
heavy the box is perceived heavy.

Actions:

moveBox the human moves the box;
emptyBox the human empties the box;
fillBox the human fills in the box.

Cognitive information:

boxMoved the fact that the box has been moved with its contents;
boxEmptied the fact that the box is empty;
boxMovedEmpty the fact that the box has been moved without its contents.

We model a basic activity under deliberate control (*deliberate activity*) as a quintuple $(goal(info), perc, info_1, info_2, act)$, where

- $goal(info) \in \Gamma$ is the driving goal,
- $perc \in \Pi$ is a perception,
- $info_1 \in 2^{\hat{\Delta}\setminus\Xi\setminus\Gamma}$ is the information retrieved and removed from STM,
- $info_2 \in 2^{\Delta\setminus\Xi}$ is the information stored in STM, and
- $act \in \Sigma$ is a human action.

As above, the tuple is denoted as a rule:

$$goal(info) : info_1 \uparrow perc \Longrightarrow act \downarrow info_2.$$

If $info_1 = none$, the model of the basic activity can be shortened as

$$goal(act) \uparrow perc \Longrightarrow act \downarrow info_2$$

As for automatic activities, also a deliberate activity is

– *interactive* when its action is an interaction,
– *physical* when its action is a purely physical action,
– *attentional* when the action is *none* and the perception is present, and
– *cognitive* when both the action and the perception are *none*.

The basic activity is *enabled* (and can be performed) when

- $\{goal(info')\} \cup (info_1 \cap \Delta) \in STM$, with $info \subseteq info'$ and $info\setminus\Xi = info'\setminus\Xi$,
- $info_1 \cap \bar{\Delta} \notin STM$, and
- *perc* is available in the environment.

The first condition means that the goal in STM is the same as the one in the basic activity apart from some additional purposes. In fact, on the one hand, a specific purpose $\xi \in \Xi$ does not prevent goals for less specific purposes to be used, since they will still get closer to the goal for purpose ξ, on the other hand, goals for more specific purposes should not be used, since they might take far away from the goal for purpose ξ. The performance of the basic activity and the closure are the same as in the case of automatic control.

The automatic behaviour described in Example 71 is formalised in Example 71.2

Example 71.2: Deliberate Behaviour

Let be

- $\Pi = \{light, heavy\}$,
- $\Sigma = \{moveBox, emptyBox, fillbox\}$,
- $\Xi = \emptyset$,
- $\Gamma = \{goal(boxMoved), goal(boxEmptied)\}$,
- $\Delta = \{boxMoved, boxEmptied, boxMovedEmpty\} \cup \Gamma \cup \Pi \cup \Sigma$.

Set Ξ is empty since the purpose is not relevant here.
The task is modelled by the following seven basic activities:

1. $goal(boxMoved) \uparrow light \Longrightarrow none \downarrow light$
2. $goal(boxMoved) \uparrow heavy \Longrightarrow none \downarrow heavy$
3. $goal(boxMoved) : light \uparrow none \Longrightarrow moveBox \downarrow boxMoved$
4. $goal(boxMoved) : heavy \uparrow none \Longrightarrow none \downarrow goal(boxEmptied)$
5. $goal(boxEmptied) \uparrow none \Longrightarrow emptyBox \downarrow boxEmptied$
6. $goal(boxMoved) :$
   $boxEmptied \uparrow none \Longrightarrow moveBox \downarrow boxMovedEmpty$
7. $goal(boxMoved) :$
   $boxMovedEmpty \uparrow none \Longrightarrow fillBox \downarrow boxMoved$

The task goal is formally modelled as

$$goal(boxMoved)$$

and requires the use of subgoal

$$goal(boxEmptied).$$

Initially the STM only contains the task goal:

$$STM = \{goal(boxMoved)\}$$

Example 71.3 shows the usage of STM while performing the task modelled in Example 71.2 with a heavy box.

Example 71.3: STM usage in Deliberate Behaviour

Initially STM contains the goal of moving the box ($goal(boxMoved)$). The evolution of the content of STM is driven by the deliberate control activities in LTM as described in Example 71.2.

- $STM = \{goal(boxMoved)\}$
  2. $goal(boxMoved) \uparrow heavy \Longrightarrow none \downarrow heavy$
- $STM = \{goal(boxMoved), heavy\}$
  4. $goal(boxMoved) : heavy \uparrow none \Longrightarrow none \downarrow goal(boxEmptied)$
- $STM = \{goal(boxMoved), goal(boxEmptied)\}$
  5. $goal(boxEmptied) \uparrow none \Longrightarrow emptyBox \downarrow boxEmptied$
  (Goal $goal(boxEmptied)$ achieved and removed due to closure)
- $STM = \{goal(boxMoved), boxEmptied\}$
  6. $goal(boxMoved) :$
  $boxEmptied \uparrow none \Longrightarrow moveBox \downarrow boxMovedEmpty$
- $STM = \{goal(boxMoved), boxMovedEmpty\}$
  7. $goal(boxMoved) :$
  $boxMovedEmpty \uparrow fillBox \Longrightarrow moveBox \downarrow boxMoved$
  (Goal $goal(boxMoved)$ achieved and removed due to closure)
- $STM = \{boxMoved\}$

After the fact that the box is heavy ($heavy$) is internalized through Activity 2, the performance of Activity 4 determines the addition of the new goal $goal(boxEmptied)$ to STM and Activity 4 determines the achievement of such a goal. Then Activity 6 determines the moving of the box and, finally, Activity 7 its refilling. The final mental state is the awareness that the box has been moved, which is modelled by the presence of cognitive state $boxMoved$ in STM. All goals have been removed from STM once achieved.

Modelling Deliberate Control using PAT

Example 71.4 illustrates how to use PAT to model the closure phenomenon for the task described in Example 71.3.

Example 71.4: Closure in Deliberate Control using PAT

```
enum { boxMovedGoal, boxEmptiedGoal};
enum { None,
       Heavy, Light,
       moveBox, emptyBox, fillBox,
       BoxMoved, BoxEmptied, BoxMovedEmpty,
       Interaction }; // 10 items
```

```
#define G 2; // No. of goal
#define N 10; // No. of stm array positions
#define M 7; // STM maximum capacity

var stmGoal = [ 0 , 0 ];
var stm[N];
var stmSize;
var perc[N];
var info[N];

Closure() = ba-> (
  [stmGoal[boxMovedGoal] == 1 &&
       (info[boxMoved] == 1 || stm[BoxMoved])]
    achieveBoxMoved {info[BoxMoved] = 0;
                     stmGoal[BoxMoved] = 0;
                     stmSize--;} -> FlashOut() []
  [stmGoal[boxEmptiedGoal] == 1 &&
       (info[boxEmptied] == 1 || stm[BoxEmptied])]
    achieveBoxMoved {info[BoxEmptied] = 0;
                     stmGoal[BoxEmptied] = 0;
                     stmSize--;} -> FlashOut() []
  eact -> Closure() );

 FlashOut() = closure { var cell = 0;
                        while (cell < M) {
                          if (stm[cell] == 1) {
                              stmSize--;
                          };
                          stm[cell] = 0 ;
                          cell = cell + 1;
                        }
                       } -> eact -> Closure();

Goals() =
  [stmSize < M && stmGoal[BoxMovedGoal] == 0]
     move {stmGoal[BoxMovedGoal] = 1;
                                  stmSize++} -> Goals() []
  [stmGoal[BoxMovedGoal] == 1] ba -> eba -> Goals() []
  [stmSize < M && stmGoal[BoxEmptiedGoal] == 0]
     move {stmGoal[BoxEmptiedGoal] = 1;
                                  stmSize++} -> Goals() []
  [stmGoal[BoxEmptiedGoal] == 1] ba -> eba -> Goals();
```

Array `info` implements the possibility that the information associated with the goal achievement is a new piece of information stored in STM by the basic

activity. Thus not only the position of the `stmGoal` array corresponding to the achieved goal is changed to `0` but also the same position of array `info`. This is followed by the execution of process `FlashOut`, which clears the contents of the non-goal part of STM (array `stm`).

Example 71.5 illustrates how to model Example 71.2 in PAT:

Example 71.5: Deliberate Control Task using PAT

```
Task() = ba -> (
 [stmGoal[BoxMovedGoal] == 1 &&
  stmSize < M && perc[Light] == 1]
newInfo {info[Light] = 1}
   -> eact -> store {stm[Light] = 1; stmSize++}
   -> eba -> Task() []
 [stmGoal[BoxMovedGoal] == 1 &&
  stmSize < M && perc[Heavy] == 1]
 newInfo {info[Heavy] = 1}
   -> eact -> store {stm[Heavy] = 1; stmSize++}
   -> eba -> Task() []
 [stmGoal[BoxMovedGoal] == 1 &&
  stm[Light] == 1 && stmSize < M]
moveBox -> newInfo {info[boxMoved] = 1}
   -> eact -> store {stm[boxMoved] = 1; stmSize++}
   -> eba -> Task() []
 [stmGoal[BoxMovedGoal] == 1 &&
  stm[Heavy] == 1 && stmSize < M]
 eact -> store {stmGoal[BoxEmptiedGoal] = 1; stmSize++}
   -> eba -> Task() []
  [stmGoal[BoxEmptiedGoal] == 1 && stmSize < M]
  emptyBox -> newInfo {info[boxEmptied] = 1}
   -> eact -> store {stm[boxEmptied] = 1; stmSize++}
   -> eba -> Task() []
  [stmGoal[BoxMovedGoal] == 1 &&
   stm[boxEmptied] == 1 && stmSize < M]
  moveBox -> newInfo {info[boxMovedEmpty] = 1}
   -> eact -> store {stm[boxMovedEmpty] = 1; stmSize++}
   -> eba -> Task() []
  [stmGoal[BoxMovedGoal] == 1 &&
   stm[boxMovedEmpty] == 1 && stmSize < M]
  fillBox -> newInfo {info[boxMoved] = 1}
   -> eact -> store {stm[boxMoved] = 1; stmSize++}
   -> eba -> Task() []
               );

User() = Closure() || Goals() || Task();
```

For each choice of the process corresponding to basic activity

$$goal(info) : info_1 \uparrow perc \Longrightarrow act \downarrow info_2$$

for each piece of information $i \in info_2$, the possibility that the information i is associated with the goal achievement is implemented by the assignment `info[I] = 1`, where `I` is the position of array `info` that implements i.

7.3.5 Operator's Deliberate Behaviour

Operator's behaviour is mainly deliberate. Although there is normally a prefixed sequence of basic activities through which the operator needs to go, each of these activities is driven by a specific goal to be accomplished. However, the operator task does not have a top-level goal. Instead it has a purpose, which influences all goals established (and accomplished) during the task performance.

The 'scan the screen' operator's subtask informally described in Example 66 may be formalised as in Example 66.1.

Example 66.1: 'Scan the Screen' Operator's Subtasks

Let be

- $\Pi = \{globalView, needsFurtherInvestigation, nothingAbnormal\}$,
- $\Sigma = \{moveBox, emptyBox, fillbox\}$,
- $\Xi = \{atcPurpose\}$,
- $\Gamma = \{goal(atcPurpose, identifiedPart),$
  $goal(atcPurpose, assessedPart)\}$
- $\Delta = \{identifiedPart, assessedPart, investigatedPart\} \cup \Gamma \cup \Pi \cup \Sigma$.

The task is modelled by the following four basic tasks:

1. $goal(atcPurpose, identifiedPart) \uparrow globalView$
   $\Longrightarrow identifiedPart \downarrow goal(actPurpose, assessPart)$
2. $goal(atcPurpose, assessedPart) \uparrow needsFurtherInvestigation$
   $\Longrightarrow none \downarrow goal(atcPurpose, investigatedPart)$
3. $goal(atcPurpose, assessedPart) \uparrow nothingAbnormal$
   $\Longrightarrow none \downarrow goal(atcPurpose, identifiedPart)$

Initially

$$STM = \{goal(atcPurpose, identifiedPart)\}$$

Through a global perception of the screen the operator identifies a part of the screen in which there might be a conflict (Activity 1) and sets the subgoal to assess that part ($goal(actPurpose, assessPart)$), while the closure due to the storage of information *identifiedPart* causes the removal of goal $goal(atcPurpose, identifiedPart)$. If the part of the

screen is perceived as in need of further investigation, then subgoal $goal(actPurpose, assessPart)$ is established (Activity 2). If, instead, nothing abnormal is noticed, then subgoal $goal(actPurpose, assessPart)$ is established. In both cases, the closure due to the storage of information $assessPart$ causes the removal of goal $goal(atcPurpose, assessPart)$ (Activity 3).

The purpose is present in STM as argument of some goals, as long as the operator is engaged in the task.

The 'new phase' operator's subtask informally described in Example 66 may be formalised as in Example 66.2.

Example 66.2: 'New Phase' Operator's Substask

Let be

- $\Pi = \{endTask\}$,
- $\Sigma = \emptyset$,
- $\Xi = \{atcPurpose\}$,
- $\Gamma = \{goal(atcPurpose, newPhase),$
  $goal(actPurpose, identifyPart),$
  $goal(atcPurpose, actedOnPair)$
- $\Delta = \{newPhase\} \cup \Gamma \cup \Pi \cup \Sigma$.
  The task is modelled by the following three basic tasks:

1. $goal(atcPurpose, newPhase) \uparrow none$
   $\Longrightarrow none \downarrow goal(actPurpose, identifyPart)$
2. $goal(atcPurpose, newPhase) \uparrow none$
   $\Longrightarrow none \downarrow goal(atcPurpose, actedOnPair)$
3. $goal(atcPurpose, newPhase) \uparrow endTask$
   $\Longrightarrow none \downarrow newPhase$

Initially

$$STM = \{goal(atcPurpose, newPhase)\}$$

In Activity 3 the closure due to the storage of information $newPhase$ causes the removal of goal $goal(atcPurpose, newPhase)$, which is the only goal in STM influenced by purpose $atcPurpose$. Therefore, any trace of the purpose disappears from STM.

7.3.6 Switching Process Control

Familiar perceptions provide a mechanism to switch from deliberate control to automatic control. In an environment, with familiar perception, such as

the ones provided by an ATM, the user behaviour proceeds independently of the goal that has triggered it. However, during automatic behaviour there are situations in which the cognitive control must switch back to deliberate control.

This situation is illustrated by Example 65.8, which extends the Example 65.2 by considering two possible goals 'cash withdrawal' and 'statement printing' for the ATM task.

Example 65.8: Automatic and Deliberate Behaviour

Let be

- $\Pi = \{cardR, pinR, selR, cashO, statO, cardO\}$,
- $\Sigma = \{cardI, pinE, cashS, statS, cashC, statC, cardC\}$,
- $\Xi = \emptyset$,
- $\Gamma = \{goal(cashC), goal(statC)\}$.
- $\Delta = \Gamma \cup \Pi \cup \Sigma \cup \{interaction\}$.

Set Ξ is empty since the purpose is not relevant here.

A simple ATM task, in which the user has only the goal to withdraw cash, is modelled by the following four basic tasks:

1. $goal(cashC) : \overline{interaction} \uparrow none \Longrightarrow none \downarrow interaction$
2. $goal(statC) : \overline{interaction} \uparrow none \Longrightarrow none \downarrow interaction$
3. $interaction \uparrow cardR \Longrightarrow cardI \downarrow cardC, interaction$
4. $interaction \uparrow pinR \Longrightarrow pinE \downarrow interaction$
5. $goal(cashC) \uparrow selR \Longrightarrow cashS \downarrow none$
6. $goal(statC) \uparrow selR \Longrightarrow statS \downarrow none$
7. $interaction \uparrow cashO \Longrightarrow cashC \downarrow interaction$
8. $interaction \uparrow statO \Longrightarrow statC \downarrow interaction$
9. $cardC, interaction \uparrow cardO \Longrightarrow cardC \downarrow interaction$

Perception $selR$ denotes that the ATM requests the user to select the transaction between 'cash withdrawal' and 'statement printing'. Perception $statO$ denotes that the statement has been delivered. Actions $cashS$ and $statS$ are the user's selections of 'cash withdrawal' and 'statement printing', respectively. Information $interaction$ models the cognitive state of the user interacting with the ATM; it is initially absent from STM.

The behaviour starts under deliberate control with one of the two possible goals, $goal(cashC)$ ('cash withdrawal') or $goal(cashC)$ ('statement printing') determining the beginning of the interaction (Activities 1 and 2, respectively) by storing $interaction$ in STM. The storage of $interaction$ in STM activates the automatic control driven by perceptions until perception $selR$ requires a decision about which transaction to select. Activities 5 and 6 determine the decision based on the goal in STM, thus under deliberate control. After the decision has been made, automatic control is restored for the rest of the task.

Example 65.9 illustrates how to use PAT to define the infrastructure and model the closure phenomenon for the ATM task described in Example 65.8.

Example 65.9: Closure in Automatic Control using PAT

```
enum { getCashGoal , getStatGoal};
enum { None,
       CardR, PinR, SelR, CashO, StatO, CardO ,
       CardI, PinE, CashS, StatS, CashC, StatC, CardC,
       Interaction }; // 15 items

#define G 1; // No. of goal
#define N 15; // No. of stm array positions
#define M 7;    // STM maximum capacity

var stmGoal = [ 0 , 0];
var stm[N];
var stmSize;
var perc[N];

Closure() = ba-> (
  [stmGoal[getCashGoal] == 1] cashC ->
    achieveGetCash {stmGoal[getCashGoal] = 0;
                    stmSize--;} -> FlashOut() []
  [stmGoal[getStatGoal] == 1] cashC ->
    achieveGetCash {stmGoal[getStatGoal] = 0;
                    stmSize--;} -> FlashOut() []
  eact -> Closure() );

 FlashOut() = closure { var cell = 0;
                        while (cell < M) {
                          if (stm[cell] == 1) {
                              stmSize--;
                          };
                          stm[cell] = 0 ;
                          cell = cell + 1;
                        }
                       } -> eact -> Closure();]

Goals() =
[stmSize < M && stmGoal[getCashGoal] == 0 &&
 stm[Interaction] == 0] getCash {stmGoal[getCashGoal] = 1;
                                 stmSize++} -> Goals() []
[stmGoal[getCashGoal] == 1] ba -> eba -> Goals() []
[stmSize < M && stmGoal[getStatGoal] == 0 &&
```

```
    stm[Interaction] == 0] getCash {stmGoal[getStatGoal] = 1;
                                    stmSize++} -> Goals() []
  [stmGoal[getStatGoal] == 1] ba -> eba -> Goals();
```

With respect to Example 65.4, processes `Closure` and `Goals` have the additional choice for the new goal. Moreover, guards in process `Goals` also include a condition on the absence of `interaction` from STM: initially there is no goal in STM and the role of process `Goals` is to establish one goal non deterministically, when the user in not interacting with the ATM.

Example 65.10 illustrates how to use PAT to model the basic activities for the ATM task described in Example 65.8.

Example 65.10: Automatic Control Task using PAT

```
Task() = ba -> (
 [stmGoal[getCashGoal] == 1 && stm[Interaction] == 0] eact
   -> store {stm[Interaction] = 1;
             stmSize++} -> eba-> Task() []
 [stmGoal[getStatGoal] == 1 && stm[Interaction] == 0] eact
   -> store {stm[Interaction] = 1;
             stmSize++} -> eba-> Task() []
 [stm[Interaction] == 1 && stmSize < M &&
  perc[CardR] == 1] cardI
   -> eact -> store {stm[CardC] = 1; stmSize++}
   -> eba -> Task() []
 [stm[Interaction] == 1 &&
  perc[PinR] == 1] pinE
   -> eact -> eba -> Task() []
[stm[Interaction] == 1 &&
  perc[CashO] == 1] cashC
   -> eact -> eba -> Task() []
 [stm[Interaction] == 1 &&
   perc[StatO] == 1] statC
   -> eact -> eba -> Task() []
 [stm[Interaction] == 1 && stmSize > 0 &&
   perc[CardO] == 1 && stm[CardC] == 1]
   retrieve {stm[CardC] = 0; stmSize--} -> cardC
   -> eact -> eba -> Task()
              );

User() = Closure() || Goals() || Task();
```

7.4 Interface/System Model

In Sect. 7.3.1 we have defined *perceptions* to characterise the human input channel and *actions* to characterise the human output channels, with actions performed in response to perceptions. This describes the human perspective of input/output channels. From a machine perspective, we can also say that a user perception refers to an interface output, which acts as a stimulus for the human. During the interaction, such an output is normally the reaction of the interface to the action performed by the human.

Hence we identify a visible state created by an interface or system with the perception it produces in humans. For example, the interface state created by the action of giving change, performed by the interface of a vending machine, is identified with the perception (sound of falling coins or sight of the coins) produced.

Example 65.11 models one possible ATM interface to support the ATM task described in Example 65.8.

Example 65.11: Old ATM Interface using PAT

```
ATMold() =
  atomic{ [perc[CardR] == 1] cardI ->
    readCard {perc[CardR] = 0 ;
              perc[PinR] = 1} -> ATMold() } []
  atomic{ [perc[PinR] == 1] pinE ->
    readPin {perc[PinR] = 0 ;
             perc[SelR] = 1} -> ATMold() } []
  atomic{ [perc[SelR] == 1] cashS ->
    setCashS {perc[SelR] = 0 ;
              perc[CashO] = 1} -> ATMold() } []
  atomic{ [perc[SelR] == 1] statS ->
    setStatS {perc[SelR] = 0 ;
              perc[StatO] = 1} -> ATMold() } []
  atomic{ [perc[CashO] == 1] ( cashC ->
    detectCashC {perc[CashO] = 0;
                 perc[CardO] = 1} -> ATMold() ) } []
  atomic{ [perc[StatO] == 1] statC ->
    detectStatC {perc[StatO] = 0;
                 perc[CardO] = 1} -> ATMold() } []
  atomic{ [perc[CardO] == 1] cardC ->
    detectCardC {perc[CardO] = 0} ->
    reset {perc[CardR] = 1} -> ATMold() ) };
```

A simple interface may be modelled by a choice between all possible transitions. Each choice is guarded by the source state of the transition, normally

represented by the perception provided to the user. The first event of the choice to be performed is the synchronisation event that models the interaction of the human (events `CardI`, `PinE`, `CashS`, `StatS`, `CashC`, `StatC` and `CardC` in Example 65.11). Each synchronisation event is followed by the 'local' interface event that modifies the interface state by assigning `0` to the source state and `1` to the target state. These local events my be split to increase the readability of the model, as it happens in the last choice of the `ATMold` process in Example 65.11.

In order to keep the synchronisation event and the associated interface events as an one atomic action, we use the *atomic process* construct available in PAT. The `atomic` keyword associates higher priority with a process: if the process has an enabled event, the event will execute before any events from non atomic processes. Moreover, the sequence of statements of the atomic process is executed as one single step, with no interleaving with other processes.

The ATM interface modelled in Example 65.11 was very common in the past. However, it was observed that delivering cash or statement before returning the card sometimes caused the user error of forgetting the card. This error is due to the fact that once the goal of collecting the cash or the statement is achieved, STM may be flashed out by the closure phenomenon thus losing some information, possibly including the reference to the action to collect the card. The discovery of this error led to the development of a new ATM interface that returns the card before delivering cash or statement. In terms of interface model this means that the user's selection of a transaction, although it results in the same user perception of seeing the card returned, should determine two distinct state transitions depending on whether the user selects 'cash withdrawal' or 'statement printing'. The new state will then produce the appropriate perception at a later stage.

In general, when dealing with a fairly complex behaviour, possibly resulting from the parallel composition of several subsystems, it is necessary to consider internal system states, which do not present themselves as human perceptions. Therefore, in addition to the `perc` array to implement perception, we also use an array `state` to implement internal states.

Example 65.12 models the new ATM interface.

Example 65.12: New ATM Interface using PAT

```
var state[N];

ATMnew() =
  atomic{ [perc[CardR] == 1] cardI ->
    readCard {perc[CardR] = 0;
              perc[PinR] = 1} -> ATMnew() } []
  atomic{ [perc[PinR] == 1] pinE ->
    readPin {perc[PinR] = 0;
```

```
                perc[SelR] = 1} -> ATMnew() } []
    atomic{ [perc[SelR] == 1] cashS ->
      setCashS {perc[SelR] = 0; perc[CardO] = 1;
                state[CashO] = 1} -> ATMnew() } []
    atomic{ [perc[SelR] == 1] statS ->
      setStatS {perc[SelR] = 0; perc[CardO] = 1;
                state[StatO] = 1} -> ATMnew() } []
    atomic{ [state[CashO] == 1 && perc[CardO] == 1] cardC ->
      detectCardC {perc[CardO] = 0; perc[CashO] = 1;
                   state[CashO] = 0} -> ATMnew() } []
    atomic{ [state[StatO] == 1 && perc[CardO] == 1] cardC ->
      detectedCardC {perc[CardO] = 0; perc[StatO] = 1
                     state[StatO] = 0;} -> ATMnew() } []
    atomic{ [perc[CashO] == 1] cashC ->
      detectCashC {perc[CashO] = 0} ->
      reset {perc[CardR] = 1} -> ATMnew() } []
    atomic{ [perc[StatO] == 1] statC ->
      detectCashC {perc[StatO] = 0} ->
      reset {perc[CardR] = 1} -> ATMnew() };  };
```

7.4.1 Experiential Knowledge and Expectations

Section 7.2 illustrated various kinds of memory, which play different roles in processing information. Then in Sects. 7.3.3 and 7.3.4 we described automatic and deliberate behaviour, respectively, and provided a formal notation (and its implementation in PAT) to model basic activities under these two forms of cognitive control. If we wish to associate the location of the rules that model basic activities with distinct parts of the human memory, we can imagine that they are stored in LTM and, more specifically, that automatic basic activities are stored in procedural memory and deliberate basic activities are stored in semantic memory.

We have also mentioned that information may be transferred from sensory memory to STM through attention while facts and knowledge may be transferred from semantic memory to STM. We must add that information may flow from STM to LTM, first to episodic memory, and then produce changes to semantic and procedural memory. In fact, automatic and deliberated basic activities are created through a long-term learning process. In general, users make large use of deliberate activities while learning a task and, during the learning process, they create automatic rules in procedural memory to replace the less efficient rules in semantic memory. However, although automatic control is efficient and requires less STM usage than deliberate control, it may result inappropriate in some situation. In such a case, experiential knowledge

already stored in the LTM may be used to solve the situation. Norman and Shallice [NS86] propose the existence of a *Supervisory Attentional System* (SAS), sometimes also called *Supervisory Activating System*, which becomes active whenever none of the automatic tasks are appropriate. The activation of the SAS is triggered by perceptions that are assessed as danger, novelty, requiring decision or are the source of strong feelings such as temptation and anger. The SAS is an additional mechanism to switch from automatic to deliberate control.

In Sect. 7.3.6 we described how to model the switching from automatic to deliberate control due to a required decision. Now we consider how such switching may occur due to the user's assessment of perceptions as the result of acquired experiential knowledge. Example 65.13 extends Example 65.9 with the infrastructure for representing factual and experiential knowledge and the mechanisms to assess perception and produce an appropriate response based on experiential knowledge.

Example 65.13: Closure with Experiential Knowledge

```
enum { safe , danger };      // assessment
enum { normal , abort };     // response
var assessment = safe;
var response = normal;

enum { getCashGoal , getStatGoal};
...

Closure() = ba-> (
  [response == normal && stmGoal[getCashGoal] == 1]
    cashC ->
    achieveGetCash {stmGoal[getCashGoal] = 0;
                    stmSize--;} -> FlashOut() []
  [response == normal && stmGoal[getStatGoal] == 1]
    cashC ->
    achieveGetCash {stmGoal[getStatGoal] = 0;
                    stmSize--;} -> FlashOut() []
  eact -> Closure() );

FlashOut() = ...

Goals() = ...
```

Variable `assessment` records the assessment of the user's perception following the user's action. We enumerate only two possible values: `safe` means that the perception will not affect the user's automatic control, wheras `danger` means that the perception requires a switch to deliberate control and an appropriate

response, which will be assigned to variable `response`, whose possible values are `normal` and `abort`. Initially variable `assessment` has value `safe` and variable `response` has value `normal`.

Variable `assessment` is set depending on the user's expectation. We use additional processes to constrain the user's behavior depending on expectations. Such processes are specific to the considered interface/system. Example 65.14 defines the constraints for the ATM.

Example 65.14: Constraints Modelling Expectations

```
ExpectOld() = ba ->
 ( eba -> ExpectOld() []
   cashS -> eba ->
     ( [perc[CashO] == 1]
          cashExpectMet -> ExpectOld() []
       [perc[CardO] == 1]
           cashExpectFailed {assessment = danger}
              -> ExpectOld() ) []
 statS -> eba ->
     ( [perc[StatO] == 1]
           statExpectMet -> ExpectOld() []
       [perc[CardO] == 1]
           statExpectFailed {assessment = danger}
                -> ExpectOld() )

ExpectNew() = ba ->
 ( eba -> ExpectNew() []
   cashS -> eba ->
     ( [perc[CardO] == 1]
          cardExpectMet -> ExpectNew() []
        [perc[CashO] == 1]
          cardExpectFailed -> ExpectNew() ) []
    statS -> eba -> ( [perc[CardO] == 1]
          cardExpectMet -> ExpectNew() []
        [perc[StatO] == 1]
          cardExpectFailed -> ExpectNew() )
                    );
 );
```

The above model caters for two different user expectations. The first one is `ExpectOld`, where a user used to interact with the old ATM expects to see the cash or statement delivered after selecting 'cash withdrawal' or 'statement printing'; such expectations are not met (`cashExpectMet` or `statExpectMet`, respectively) if the card is returned instead. The second one is `ExpectNew`, where a user used to interact with the new ATM expects to see the card returned after selecting

'cash withdrawal' or 'statement printing'; such expectations is not met (`cardExpectMet`) if the cash or statement is delivered instead.

Example 65.15 extends Example 65.10 by including the appropriate guards on the assessment of the perception and an `abortSession` event which assigns `abort` to the `response` variable when the `assessment` of the perception is `danger`.

Example 65.15: Task with Response to Perception Assessment

```
Task() = ba -> (
 [assessment == safe &&
  stmGoal[getCashGoal] == 1 && stm[Interaction] == 0] eact
   -> store {stm[Interaction] = 1;
             stmSize++} -> eba-> Task() []
 [assessment == safe &&
  stmGoal[getStatGoal] == 1 && stm[Interaction] == 0] eact
   -> store {stm[Interaction] = 1;
             stmSize++} -> eba-> Task() []
 [assessment == safe &&
  stm[Interaction] == 1 && stmSize < M &&
  perc[CardR] == 1] cardI
   -> eact -> store {stm[CardC] = 1; stmSize++}
   -> eba -> Task() []
 [assessment == safe &&
  stm[Interaction] == 1 &&
  perc[PinR] == 1] pinE
   -> eact -> eba -> Task() []
 [assessment == safe &&
  stmGoal[getCashGoal] == 1 && perc[SelR] == 1] cashS
   -> eact -> eba -> Task() []
 [assessment == safe &&
  stmGoal[getStatGoal] == 1 && perc[SelR] == 1] statS
   -> eact -> eba -> Task() []
[assessment == safe &&
 stm[Interaction] == 1 &&
  perc[CashO] == 1] cashC
   -> eact -> eba -> Task() []
 [assessment == safe &&
  stm[Interaction] == 1 &&
   perc[StatO] == 1] statC
   -> eact -> eba -> Task() []
 [assessment == safe &&
  stm[Interaction] == 1 && stmSize > 0 &&
   perc[CardO] == 1 && stm[CardC] == 1]
```

```
      retrieve {stm[CardC] = 0; stmSize--} -> cardC
      -> eact -> eba -> Task()
    [assessment == danger &&
     stm[Interaction] == 1 && perc[CardO] == 1 ]
      abortSession {assessment = safe;
                    response = abort} -> cardC
        -> eact -> eba -> Task() []
    [assessment == danger &&
     stm[Interaction] == 1 && perc[CardO] == 0 ]
      abortSession {assessment = safe;
                    response = abort}
        -> eact -> eba -> Task()
                  );

  User() = Closure() || Goals() || Task();
```

The response to a danger (guard `assessment == danger`) is to collect the card (event `cardC`), if this is perceived (guard `perc[CardO] == 1`), and abort the interaction session (event `abortSession`, which set `response` to `abort`) or just abort the interaction section if the cards is not perceived (guard `perc[CardO] == 0`), while variable `assessment` is reset to `safe`. Although normally the danger assessment is due to the perception of the card, not considering the assessment of danger possibly due to other reasons would be an overspecification.

7.4.2 Environment and Overall System

Until now we have considered the following components of the overall system:

– the *user's behaviour*

```
User() = Closure() || Goals() || Task();
```

consisting of the infrastructure for STM (process `Closure`) and goals (process `Goals`) and the human task process `Task` (see Examples 65.7, 71.5, 65.10 and 65.15),
– the *interface or system* (see Examples 65.11 and 65.11), and
– the *user's experiential constraints* (see Example 65.14).

However, as illustrated in Example 65.16 there are further aspects of the environment that influence the interaction and thus ought to be part of the modelled overall system:

- the *initial interface/system state*,
- the *availability of resources*, and
- the *user's knowledge*.

Example 65.16: Closure in Automatic Control using PAT

```
InitState() = initialization {perc[CardR] = 1} -> Skip();

HasCard() = cardI -> NoCard();
NoCard() = cardC -> HasCard();

Resources() = HasCard();

KnowsPin() = pinE -> KnowsPin();

Knowledge() = KnowsPin();

User() = Closure() || Goals() || Task();

Environment() = User() || Resources() || Knowledge() ;

SysOld() = InitState() ; ( Environment() || ATMold() );
SysNew() = InitState() ; ( Environment() || ATMnew() );

UserOld() = Environment() || ExpectOld();
UserNew() = Environment() || ExpectNew();

SysOldUserOld() = InitState() ; ( UserOld() || ATMold() );
SysOldUserNew() = InitState() ; ( UserNew() || ATMold() );
SysNewUserNew() = InitState() ; ( UserNew() || ATMnew() );
SysNewUserOld() = InitState() ; ( UserOld() || ATMnew() );
```

Aspects of the environment are the following.

- *Initial interface state.* We assume that the interface is initially requesting a card. This is expressed by process `InitState`, which performs event `initialization` to set variable `perc[CardR]` to `1` and terminate successfully. This process is sequentialised with the main overall system process.
- *Availability of resources.* An essential resource for the task is the bank card, which has to be available for the user: process `Resources` consists of two states describing the availability (`HasCard`) and non availability (`NoCard`) of the card.
- *User's knowledge.* The user must know the pin in order to perform the task. In our example we implicitly assumed that the user knows the pin, thus process `KnowsPin` models only the correct pin in terms of event `pinE`. However, we might want to consider also the case of using a wrong pin, in order to explore its impact on the interaction and the emergent errors This would require a more sophisticated version of process `KnowsPin`.

Finally, the overall system is modelled by processes `SysOld` and `SysNew`, corresponding to the two possible interfaces, but with no assumptions on the user's experiential knowledge, and by processes `SysOldUserOld` and `SysOldUserNew`, `SysNewUserNew` and `SysNewUserOld`, which include constraints on user expectations.

7.5 Model Checking Analyses

From an analytical point of view we focus on two aspect: overall system verification and task failure analysis. First, in Sect. 7.5.1 we illustrate how to verify whether the design of the interface and the other environment components addresses cognitive aspects of human behaviour such as closure phenomena and user expectations that trigger the SAS to activate attention (overal system verification). Then, in Sect. 7.5.2, we consider patterns of behaviour featuring persistent operator errors may lead to a task failure.

7.5.1 Overall System Verification

Model checking techniques provide an effective analytical tool to exhaustively explore the system state space and capture the behaviour that emerges from the combination of several system components. Closure, automatic behaviour, expectancy and attention are phenomena that represent distinct components of human cognition and action, and their combination results in an apparently holistic ways of performing tasks. In this context model checking can be used to capture errors that emerge when environment design, which includes physical devices, interfaces and their operational environment, cannot deal with the closure phenomena, or when the outcome of the interaction between automatic behaviour and environment does not meet human expectations. We use Linear Temporal Logic (LTL), as described in Sect. 2.5.3, to specify system properties and then we use PAT model checking capabilities to verify such properties on the CSP model. PAT support the definition of *assertions* of the form

`#assert` *system* `|=` *property*

where *system* is the model we aim to verify and *property* is a property expressed in LTL. The PAT model checker verifies whether the property is valid on the model and, if not, provides a counterexample. The counterexample provided by the model checking analysis can then be exploited to improve the environment design.

We consider three kinds of properties:

- *Functional correctness* the user or operator can always complete the task by successfully accomplishing the goal,
- *Non-functional correctness* in spite of successfully accomplishing the goal, the system may violate some non-functional properties (e.g. the user of an ATM forgets the card after collecting the cash), and
- *Cognitive Overload* the STM is overloaded above a considered upper limit, which may lead to a failure in accomplishing the goal when the STM is loaded by additional uncompleted tasks.

We illustrate the functional correctness in Example 65.17

Example 65.17: Functional Property Verification using PAT

Since there are two possible goals, to get cash and to get a statement, **functional correctness** aims to verify for each interface design, whether there are cognitive errors that may prevent the user from collecting the cash and from collecting the statement. The property that the user is always able to collect the cash can be expressed by formalising that "a user who selects 'cash' will collect the cash before the end of the interaction section". Since the end of the interaction section may be characterised as the beginning of a new interaction section, which occurs when a card is inserted again, we can say that "a user who selects 'cash' will collect the cash before a card is inserted". Furthermore "the user collects the cash before a card is inserted" can also be expressed as "no card is inserted until the user collects the cash". Finally, our original property can be expressed as "if a user selects 'cash' then no card is inserted until the user collects the cash", which can be immediately translated into LTL. Similarly, the properties that the user is always able to collect the statement can be expressed by formalising that "a user who selects 'statement' will collect the statement before the end of the interaction section" or equivalently as "if a user selects 'statement' then no card is inserted until the user collects the statement", which again can be immediately translated into LTL.

```
#assert SystemNewUserNew() |=
     [] ( cashS -> ( ! cardI U cashC ) );
#assert SystemNewUserNew() |=
     [] ( statS -> ( ! cardI U statC ) );

#assert SystemOldUserNew() |=
     [] ( cashS -> ( ! cardI U cashC ) );
#assert SystemOldUserNew() |=
     [] ( statS -> ( ! cardI U statC ) );
```

```
#assert SystemNewUserOld() |=
     [] ( cashS -> ( ! cardI U cashC ) );
#assert SystemNewUserOld() |=
     [] ( statS -> ( ! cardI U statC ) );

#assert SystemOldUserOld() |=
     [] ( cashS -> ( ! cardI U cashC ) );
#assert SystemOldUserOld() |=
     [] ( statS -> ( ! cardI U statC ) );
```

The PAT model checker shows all systems except `SystemNewUserOld` to be functionally correct. In fact, although the new design of the ATM works in an ideal world where all ATMs are designed according to the new criterion, there are still some ATMs, especially in the developing world, that are designed according to the old criterion. Thus we can imagine that a user from one of such countries, while visiting a country where all ATMs are designed according to the new criterion, is likely to assess the early return of the card as a danger and is prone to abandon the interaction forgetting to collect the cash. This situation is formalised by the counterexample returned in the verification of `SystemNewUserOld`.

In general, when the system behaviour consists of a loop of user sessions each characterised by a *begin* event and there are two events *choose* and *accomplish* which characterise the choice and accomplishment of the goal, then functional correctness can be expressed as

`#assert` *system* `|= [] ` *choose* `-> ( !` *begin* `U` *accomplish* `)).`

In some cases, also non-functional properties may be characterised in this way. For example, in the case of *safety* properties, there might be a system internal event *internal*, rather than a user's choice, as a precondition for the user not to lose some owned resource currently used by the system. If *return* is the event characterising the return of the resource to the user, then the safety property can be expressed as

`#assert` *system* `|= []` *internal* `-> ( !` *begin* `U` *return* `));`

We illustrate the verification of safety in Example 65.18

Example 65.18: Safety Property Verification using PAT

As an example of **nonfunctional correctness** we consider the **safety** property that aims to verify, for each interface design, whether there are cognitive errors that may prevent the user from collecting the returned card.

```
#assert SystemNewUserNew() |=
     [] ( readCard -> ( ! cardI U cardC ) );
```

```
#assert SystemOldUserNew() |=
     [] ( readCard -> ( ! cardI U cardC ) );
#assert SystemNewUserOld() |=
     [] ( readCard -> ( ! cardI U cardC ) );
#assert SystemOldUserOld() |=
     [] ( readCard -> ( ! cardI U cardC ) );
```

The PAT model checker shows that the safety property is valid with the new ATM (`SystemNewUserNew` and `SystemNewUserOld`) and not with the old ATM (`SystemOldUserOld` and `SystemOldUserOld`), independently of the user experience. The counterexample captures possible post-completion errors in using the old design of the ATM and shows that such errors cannot occur in the new design of the ATM.

We can now understand what cognitive error caused the cake of Example 65 to burn and why the algorithm used by your partner caused such an error to emerge. This is illustrated in Example 69.2.

Example 69.2: Why Cakes and Engines Burn

The baking tasks is divided in two separate parts, with a long period in between that is likely to be devoted to many other tasks. Each part is actually a task in itself with a specific goal. The goal of the first task is achieved when the cake is inserted in the oven and the oven is closed (Activity 6), thus causing STM closure. Therefore, the subsidiary task of lowering the temperature setting may be forgotten (Activity 7), with the result that the cake burns.

The obvious solution to this problem is to swap Activity 6 and Activity 7, thus preventing the occurrence of a post-completion error. The problem here is not in the interface, but in the algorithm, that is, the protocol that is used to carry out the task. This subtle form of post-completion error is difficult to eliminate in practice, since the solution count on the human to strictly adhere to a given protocol.

For example, on 24 May 2013, the fan cowl doors of an aircraft were left unlatched on both engines after completing scheduled maintenance (forgetting this subsidiary task after the achievement of the maintenance goal). As the aircraft departed London Heathrow Airport, the fan cowl doors from both engines detached, puncturing a fuel pipe on the right engine and damaging the airframe and some aircraft systems. While the flight returned to Heathrow an external fire developed on the right engine, which was then shut down. The aircraft managed to safely land using the left engine. All the passengers and crew evacuated the aircraft via the escape slides.

Cognitive load expresses the amount of information stored in STM at a given time. Thus cognitive overload occurs when the amount of information stored in STM is above a considered upper limit. We illustrate the analysis of cognitive overload in Example 65.19

Example 65.19: Cognitive Overload Analysis using PAT

As an example of **cognitive overload** we consider an upper limit of 5 piece of information stored in STM.,

```
#define cognitiveOverload (stmSize > 5);

#assert SystemNewUserNew() |= [] ! cognitiveOverload;
#assert SystemOldUserNew() |= [] ! cognitiveOverload;
#assert SystemNewUserOld() |= [] ! cognitiveOverload;
#assert SystemOldUserOld() |= [] ! cognitiveOverload;
```

PAT provides a `define` construct to define constants. This can be used to define boolean constants to be used as proposition within assertions, as in Example 65.19. This way, the model checker can determine the mental capabilities an operator has to possess to avoid cognitive overload.

7.5.2 Task Failures Analysis

The purpose of the operator's behaviour is to prevent the system from reaching a failure state. In this case the unwanted result of the interaction is the task failure. Although it is acceptable that the operator makes errors, since recovery from errors is always possible, if this recovery does not occur and the operator persists in making errors, then the system will eventually reach a failure state.

Applied psychology uses experiments, natural observation and other data gathering instruments to identify and categorise the operator's patterns of behaviour that may lead to a task failure. The goal of this kind of studies is to capture all possible patterns of behaviour that may lead to a task failure in order to design system controls, support tools, environment settings and working schedules that prevent operators from entering such dangerous patterns of behaviour.

Formal methods can support applied psychology by verifying whether the decomposition of a task failure into patterns of behaviour is sound and complete. The task failure F and its empirically defined decomposition $\mathcal{D} = \{P_1, \ldots P_n\}$ into patterns of behaviour can be formalised in LTL. The decomposition $\mathcal{D}$ is

- *sound* if each of the P_i is sufficient to cause the task failure F, and
- *complete* if one of the P_i is necessary to cause the task failure F.

Then model checking can be used to verify the soundness of the decomposition

$$\bigwedge_{P \in \mathcal{D}} (P \Rightarrow F)$$

and the completeness of the decomposition

$$F \Rightarrow \bigvee_{P \in \mathcal{D}} P$$

We informally illustrate this methodology in Example 66.3.

Example 66.3: Task Failure Analysis using PAT

We can characterise a separation violation as an operator who persistently misses the intention to carry out a specific action to solve the conflict [CCL08, CLC05]. We distinguish between intention and action to be able to model an unintended action that does not match the intention [Rea90]. Although this is not part of our analysis, such a mismatch would be relevant in the analysis of errors induced by a specific interface design, which could be carried out on this case study by introducing alternative interface designs and using our formal cognitive framework as in the ATM case study.

The formalisation of the ATC task failure decomposition suggested by Lindsay and Connelly [LC02] is

1. **Failure of scanning** when the operator fails to monitor a specific part of the interface, thus missing possible conflicts,
2. **Persistent mis-classification** when the operator persistently classifies as a non conflict what is actually a conflict,
3. **Persistent mis-prioritisation** when the operator persistently gives a low priority to a conflict, thus missing to solve it, and
4. **Defer action for too long** when the operator persistently delays to implement an already developed plan to solve a conflict.

Model checking analysis using PAT shows that decomposition of the task failure is sound but not complete. The counterexample shows that the definition of persistent mis-classification by Lindsay and Connelly mixes two different kinds of behaviour, one fully characterising persistent misclassification and the other being a part of another property which was not captured through empirical analysis. This property, which we call **Contrary decision process**, occurs when a conflict is persistently reclassified as a non conflict. Once such a property is added to the decomposition and the notion of persistent misclassification is redefined in a way that does not overlap with it, the decomposition becomes complete.

7.6 Closing Remarks

In this chapter, we presented a formal approach to the specification, modelling and analysis of interactive systems in general and, more specifically, of human-computer interaction. Systems are modelled using the CSP extension implemented in the Process Analyzer Toolkit (PAT), properties are specified using temporal logic formulae, either on states or events, and analysis is carried out by exploiting the model checking capabilities of PAT.

The approach is illustrated through two classical examples. The Automated Teller Machine (ATM) example was already introduced in Chap. 3 and is used in this chapter to illustrate the automatic behaviour of a user who carries out everyday activities with just implicit attention, but who may resort to explicit attention when in need of making a decision, driven by the task goal, or when realising the occurrence of an anomalous situation, such as a danger. Both functional properties, such as being enabled to achieve the goal (withdrawing cash or printing a statement, in the case of the ATM), and safety properties (remembering to collect the card, in the case of the ATM) are analysed. The Air Traffic Control (ATC) system example is introduced in this chapter to illustrate the deliberate behaviour of an operator who performs a task with a general purpose, whereby specific goals are set along the way. Although, in general, failing to achieve the goal is not a task failure, provided the system state is still consistent with the purpose, a pattern of behaviour featuring persistent operator errors may indeed lead to a task failure. In this context, model checking is used to support applied psychology by analysing an empirically defined decomposition of the task failure into patterns of behaviour, in order to verify whether the decomposition is sound and complete.

7.7 Annotated Bibliography

There is a large number of textbooks on human-computer interaction. The most comprehensive and appropriate to provide an accessible introduction to the concepts used in this chapter are by Dix et al. [DFAB04], by Preece, Rogers and Sharp [PRS17] and by Thimbleby [Thi07]. The first has an emphasis on modelling. It provides an extensive introduction to human behaviour and interaction from a cognitive science perspective and also presents, mostly at an intuitive level, a variety of formal approaches for dealing with some aspects of HCI and tackling specific challenges. The second has an emphasis on designing for user experience. It is intended as a book for practitioners and has a broader scope of issues, topics and methods than traditional human-computer interaction textbooks, with a focus on diversity of design and evaluation process involved. However, it is less concerned with cognition

than the book by Dix et al. The third draws on sound computer science principles, with a strong formal methods flavour. It uses state machines and graph theory as a powerful and insightful way to analyse and design better interfaces and examines specific designs and creative solutions to design problems

Looking more specifically at modelling cognition, historical but still actual works are by Newell and Simon [NS72], Card et al. [CEB78]. In a later work Card, Moran and Newel [CMN83] introduced a somehow formal notation, which inspired, on the one hand, the development of a plethora of cognitive architectures over the last 40 years and, on the other hand, the use of formal methods in HCI.

Kotseruba and Tsotsos published a broad overview of these last 40 years of cognitive architectures [KT18], featuring 84 cognitive architectures and mapping them according to perception modality, implemented mechanisms of attention, memory organisation, types of learning, action selection and practical applications. A similar, but less comprehensive survey by Samsonovich [Sam10] collects the descriptions of 26 cognitive architectures submitted by the respective authors. Finally, Laird et al. [LLR17] focus on three among the most known cognitive architectures, ACT-R, Soar and Sigma, and compare them based on their structural organisation and approaches to model core cognitive abilities.

In 1991 two nice surveys on the first formal approaches in HCI were compiled by Haan, van der Veer and van Vliet [GdHvV91], based on a psychology perspective, and by Dix [Dix91], based on a computer science perspective. Although the scientific community working on the use of formal methods in HCI is quite small, there have been a number of significant results over the last 20 years. Such results mainly appear in the proceedings of the international workshops on on Formal Methods for Interactive Systems (FMIS), which run from 2006, though not every year, and in journal special issues associated with such workshop. Some of these special issues and other papers in the area appeared in the journal Formal Aspects of Computing. Two important collection of works on formal methods approaches to HCI have been recently edited by Weyers, Bowen, Dix and Palanque [WBDP17] and by Oulasvirta, Kristensson, Bi and Howes [OKBH18].

7.7.1 Current Research Directions

The way the validity of both functional and nonfunctional properties is affected by user behaviour is quite intricate. It may seem obvious for functional properties that an interactive system can deploy its functionalities only if it is highly usable. However, usability may actually be in conflict with functional correctness, especially in applications developed for learning or entertainment purpose. More in general, high usability may be in conflict

with user experience, whereby the user expects some challenges in order to test personal skills and knowledge, enjoy the interaction and avoid boredom. Usability is also strictly related to critical nonfunctional properties such as safety [CCL08] and security [CE07]. Such relationship is actually two ways. On one side improving usability increases safety and/or security [CCL08]. On the other side introducing mechanisms to increase safety and/or security may reduce usability, and as a result, may lead to an unexpected global decrease in safety [IBCB91] and/or security [CE07]. Investigating such complex relationships is an important research direction.

In the real world humans frequently have to deal with operating environments that

1. continuously produce, change and invalidate human expectations as part of an evolutionary learning process [Cer16, IBCB91],
2. deploy constraining social contexts [IBCB91] and cultural differences [Hei07], and
3. provide a large amount of stimuli, which are perceived through several modalities at the same time and interpreted and combined according to temporal and contextual constraints (multimodal interaction) [CFG07].

The formal approach proposed in this chapter as well as all approaches that aim at applying formal methods to generic HCI problems presuppose that

1. expectation are a priori constraints rather than part of a learning process,
2. cognitive behavior depends on a specific social and cultural context, and
3. human cognition and actions are directly triggered by isolated perceptions.

Therefore, it is important to

1. define cognitive mechanisms that build
   - expectations in semantic memory out of experience stored in episodic memory, and
   - procedures in procedural memory out of knowledge stored in semantic memory,

   thus mimicking the information flow from STM first to episodic memory and then to LTM (see Sect. 7.4.1),
2. enable multiple, interacting instantiations of cognitive architectures as part of a complex sociotechnical system, and
3. define, at the cognitive architecture level, mechanisms for the fusion of multiple modalities.

These objectives may not be easily accomplished using formal notations with limited data structures such as CSP, even in the extended form provided by PAT. A more sophisticated modelling language with extensive data structures, possibly featuring an object-oriented paradigm, is needed. With this aim in mind the Maude rewrite system [Ö17] has been proposed as a possible candidate [Cer18, BMO19]. Furthermore, the definition of the Behavioural

and Reasoning Description Language (BRDL) [Cer20] and its implementation using Real-Time Maude [CO20] have recently paved the way for the insilico simulation of experiments carried out in cognitive psychology with human subjects [CM21] as well as the simulation of long-term human learning processes [CP21].

Furthermore, the intrinsic unpredictability of human behaviour requires the validation of any a priori model on real data. Using text mining techniques and appropriate ontologies, abstract event logs that match the representation used in the cognitive model could be extracted from the dataset and used to constrain the model before performing formal verification. This could be done at different levels, from a correspondence one-to-one between real interaction history and constraints to the representation of a set of real interaction histories with a single constraint.

Finally, the use of formal methods for system modelling and analysis requires high expertise in mathematics and logic, which is not common among typical users, such as interaction design and usability experts as well as psychologists and other social scientists. Therefore, the development of tools that address the need and skills of such typical users is essential for the acceptance and diffusion of formal methods in the HCI area.

References

[BBD00] R. Butterworth, Ann E. Blandford, and D. Duke. Demonstrating the cognitive plausability of interactive systems. *Form. Asp. of Comput.*, 12:237–259, 2000.

[BMO19] Giovanna Broccia, Paolo Milazzo, and Peter Csaba Ölveczky. Formal modeling and analysis of safety-critical human multitasking. *Innovations in Systems and Software Engineering*, 2019.

[CB04] Paul Curzon and Ann E. Blandford. Formally justifying user-centred design rules: a case study on post-completion errors. In *Integrated Formal Methods*, LNCS 2999, pages 461–480. Springer, 2004.

[CCL08] A. Cerone, S. Connelly, and P. Lindsay. Formal analysis of human operator behavioural patterns in interactive surveillance systems. *Softw. Syst. Model.*, 7(3):273–286, 2008.

[CE07] Antonio Cerone and Norzima Elbegbayan. Model-checking driven design of interactive systems. In *Proceedings of FMIS 2006*, volume 183 of *Electronic Notes in Theoretical Computer Science*, pages 3–20. Elevier, 2007.

[CEB78] S.K. Card, W.K. English, and B.J. Burr. Evaluation of mouse, rate-controlled isometric joystick, step keys, and text keys for text selection on a CRTl. *Ergonomics*, 21:601–613, 1978.

[Cer11] Antonio Cerone. Closure and attention activation in human automatic behaviour: A framework for the formal analysis of interactive systems. In *Proc. of FMIS 2011*, volume 45 of *Electronic Communications of the EASST*, 2011.

[Cer16] Antonio Cerone. A cognitive framework based on rewriting logic for the analysis of interactive systems. In *Software Engineering and Formal Methods*, LNCS 9763, pages 287–303. Springer, 2016.

[Cer18] Antonio Cerone. Towards a cognitive architecture for the formal analysis of human behaviour and learning. In *STAF collocated workshops*, LNCS 11176, pages 216–232. Springer, 2018.

[Cer20] Antonio Cerone. Behaviour and reasoning description language (BRDL). In Javier Camara and Martin Steffen, editors, *SEFM 2019 Collocated Workshops (CIFMA)*, LNCS 12226, pages 137–153. Springer, 2020.

[CFG07] M.C. Caschera, F. Ferri, and P. Grifoni. Multimodal interaction systems: information and time features. *International Journal of Web and Grid Services*, 3(1):82–99, 2007.

[CLC05] Antonio Cerone, Peter Lindsay, and Simon Connelly. Formal analysis of human-computer interaction using model-checking. In *Proc. of SEFM 2005*, pages 352–361. IEEE, 2005.

[CM21] Antonio Cerone and Diana Murzagaliyeva. Information retrieval from semantic memory: BRDL-based knowledge representation and Maude-based computer emulation. In *SEFM 2020 Collocated Workshops (CIFMA)*, LNCS 12524, pages 159–175. Springer, 2021.

[CMN83] S.K. Card, T.P Moran, and A. Newell. *The Psychology of Human-Computer Interaction*. Laurence Erlbaum, 1983.

[CO20] Antonio Cerone and Peter Csaba Ölveczky. Modelling human reasoning in practical behavioural contexts using Real-Time Maude. In *FM'19 Collocated Workshops (FMIS)*, LNCS 12232, pages 424–442. Springer, 2020.

[CP21] Antonio Cerone and Graham Pluck. A formal model for emulating the generation of human knowledge in semantic memory. In *From Data to Models and Back (DataMod 2020)*, LNCS 12611, pages 104–122. Springer, 2021.

[De 15] Raquel Araujo De Oliveira. *Formal Specification and Verification of Interactive Systems with Plasticity : Applications to Nuclear-Plant Supervision*. PhD thesis, University of Grenoble, 2015.

[DFAB04] Alan Dix, John Finlay, Gregory Abowd, and Russel Beale. *Human-Computer Interaction*. Pearson Education, 3rd edition, 2004.

[Dix91] Alan John Dix. *Formal Methods for Interactive Systems*. Academic Press, 1991.

[GdHvV91] G.C van der Veer G. de Haan and J.C. van Vliet. Formal modelling techniques in human-computer interaction. *Acta Psychologica*, 78:27–67, 1991.

[Hei07] Rüdiger Heimgärtner. Cultural differences in human computer interaction: Results from two online surveys. In *Open Innovation. Proc. of the 10th Symposium for Information Science*, pages 145–157, 2007.

[IBCB91] Ioanna Iacovides, Ann Blandford, Ann Cox, and Jonathan Back. How external and internal resources influence user action: the case of infusion device. *Cognition, Technology and Work*, 18(4):793–805, 1991.

[Joh97] C. Johnson. Reasoning about human error and system failure for accident analysis. In *Proc. of INTERACT 1997*, pages 331–338. Chapman and Hall, 1997.

[Kir90] B. Kirwan. Human reliability assessment. In *Evaluation of Human Work*, chapter 28. Taylor and Francis, 1990.

[KT18] Iuliia Kotseruba and John K. Tsotsos. 40 years of cognitive architectures: core cognitive abilities and practical applications. *Artificial Intelligence Review*, 2018.

[LC02] Peter Lindsay and Simon Connelly. Modelling erroneous operator behaviours for an air-traffic control task. In *Third Australasian User Interfaces Conference (AUIC2002)*, pages 43–54. Australian Computer Society, 2002.

[Lev95] N. G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

[LLR17] J.E. Lairs, C. Lebiere, and P.S. Rosembloom. A standard model for the mind: towards a common computational framework across artificial intelligence, cognitive science, neuroscience, and robotics. *AI Magazine*, 38:13–26, 2017.

[Mac05] C. Mach. *Knowledge and Error*. Reidel, 1905. English translation, 1976.

[Mil56] G. A. Miller. The magical number seven, plus or minus two: Some limits on our capacity to process information. *Psychological Review*, 63(2):81–97, 1956.

[MPF+16] C. Martinie, P. Palanque, R. Fahssi, J. P. Blanquart, C. Fayollas, and C. Seguin. Task model-based systematic analysis of both system failures and human errors. *IEEE Trans. Human-Mach. Syst.*, 46(2):243–254, 2016.

[MRO+15] Paolo Masci, Rimvydas Rukšėnas, Patrick Oladimeji, Abigail Cauchi, Andy Gimblett, Yunqiu Li, Paul Curzon, and Harold Thimbleby. The benefits of formalising design guidelines: a case study on the predictability of drug infusion pumps. *Innovations Syst. Softw. Eng.*, 11(2):73–93, 2015.

[NS72] A. Newel and H.A. Simon. *Human Problem Solving*. Prentice-Hall, 1972.

[NS86] Donald A. Norman and Tim Shallice. Attention to action: Willed and automatic control of behavior. In *Consciousness and Self-Regulation*, volume 4 of *Advances in Research and Theory*. Plenum Press, 1986.

[Ö17] Peter Csaba Ölveczky. *Designing Reliable Distributed Systems — A Formal Methods Approach Based on Executable Modeling in Maude*. Springer, 2017.

[OKBH18] Antti Oulasvirta, Per Ola Kristensson, Xiaojun Bi, and Andrew Howes, editors. *Computational Interaction*. Oxford University Press, 2018.

[PAT19] PAT: Process Analysis Toolkit. User manual (online version). http://pat.comp.nus.edu.sg/wp-source/resources/OnlineHelp/htm/index.htm, 1 Dec 2019.

[PBP97] P. Palanque, R. Bastide, and F. Paterno. Formal specification as a tool for objective assessment of safety-critical interactive systems. In *Proc. of INTERACT 1997*, pages 323–330. Chapman and Hall, 1997.

[PRS17] Jennifer Preece, Yvonne Rogers, and Helen Sharp. *Interaction Design — beyond human-computer interaction*. Wiley, 5th edition, 2017.

[RCB08] Rimvydas Rukšėnas, Paul Curzon, and Ann E. Blandford. Modelling rational user behaviour as games between an angel and a demon. In *Sixth IEEE International Conference on Software Engineering and Formal Methods*, pages 355–364. IEEE Computer Society, 2008.

[RCBB14] R. Rukšėnas, P. Curzon, A. E. Blandford, and J Back. Combining human error verification and timing analysis: A case study on an infusion pump. *Form. Asp. of Comput.*, 26:1033–1076, 2014.

[Rea90] James Reason. *Human Error*. Cambridge University Press, 1990.

[Sam10] Alexei V. Samsonovic. Toward a unified catalog of implemented cognitive architectures. In *Proceedings of the 1st Annual Meeting on Biologically Inspired Cognitive Architectures (BICA 2010)*, pages 195–244. IOS Press, 2010.

[SBBW09] Li. Su, Howard Bowman, Philip Barnard, and Brad Wyble. Process algebraic model of attentional capture and human electrophysiology in interactive systems. *Form. Asp. of Comput.*, 21(6):513–539, 2009.

[Thi07] Harold Thimbleby. *Press On*. MIT Press, 2007.

[WBDP17] Benjamin Weyers, Judy Bowen, Alan Dix, and Philippe Palanque, editors. *The Handbook of Formal Methods in Human-Computer Interaction*. Springer, 2017.