




Modelling Human Reasoning in Practical Behavioural Contexts Using Real-Time Maude

Antonio Cerone¹(✉)  and Peter Csaba Ölveczky²

¹ Department of Computer Science, Nazarbayev University, Nur-Sultan, Kazakhstan
antonio.cerone@nu.edu.kz

² Department of Informatics, University of Oslo, Oslo, Norway
peterol@ifi.uio.no

Abstract. In this paper we present an approach for modelling human reasoning using rewrite systems and we illustrate our approach in the context of human behaviour using a car driving example. Reasoning inference rules and descriptions of human activities are expressed using the Behaviour and Reasoning Description Language (BRDL). The BRDL model is then translated into Real-Time Maude. The object-oriented and equational logic aspects of Maude are exploited in order to define alternative semantic variations of BRDL that implement alternative theories of memory and cognition.

Keywords: Human reasoning · Human behaviour · Formal methods · Rewrite systems · Real-time maude

1 Introduction

One of the main challenges in human-computer interaction (HCI) is that the way humans use devices is not always consistent with the use for which such devices have been designed and built. In fact, although a systematic exploration of the concept of “plausible” behaviour may provide a good baseline for understanding the interaction [5, 11], some forms of “plausible” behaviour emerge only in specific contexts and cannot be predicted *a priori*. Cognitive architectures [12], formal methods [19, 20] and several other approaches, including machine learning and control theory [19], have been used to tackle this problem.

However, cognitive architectures tend to be specialised, each with a specific scope, which is normally academic and seldom practical [12], formal methods are “regarded as requiring too much expertise and effort for day-to-day use, being principally applied in safety-critical areas outside academia” [19, Ch. 7, page 187], and machine learning and control theory focus on the interaction process rather than human behaviour. Moreover, although emulating reasoning

Work partly funded by Seed Funding Grant, Project SFG 1447 “Formal Analysis and Verification of Accidents”, University of Geneva, Switzerland.

is one of the main objectives of some cognitive architectures, past and current efforts in this sense either do not consider human errors or are detached from the practical context of human behaviour [12]. Furthermore, high-level reasoning is not supported by control theory and, although it may emerge using machine learning, the way it emerges cannot be explained.

Our approach builds on the *Behaviour and Reasoning Description Language (BRDL)* [10] and on the use of the Maude rewrite system [16–18] to model the dynamics of human memory and memory processes [8, 9]. The semantics of BRDL is based on a basic model of human memory and memory processes and is adaptable to different cognitive theories. This allows us, on the one hand, to keep the syntax of the language to a minimum, thus making it easy to learn and understand without requiring expertise in mathematics or formal methods and, on the other hand, to use alternative semantic variations to compare alternative theories of memory and cognition. BRDL is equipped with the linguistic constructs to specify reasoning goals, inference rules and unsolved problems. We use rewrite systems [14, 17] to implement such constructs. Specifically, BRDL is translated into Real-Time Maude [16, 18], thus combining human components with the system components that model the environment in which humans operate [9].

Real-Time Maude was used to model and analyse human multitasking by Broccia *et al.* [3, 4], who adopted the initial cognitive framework underlying BRDL [8] and extended it with a number of time-related and other quantitative aspects. In their work, basic activities (also called basic tasks [3, 4, 8]) incorporate non-cognitive aspects, such as the duration and the difficulty of the task, which are interface-dependent outcomes of the interaction process, as well as external aspect, such as the delay in executing the basic activity, which is possibly due to the switching from one task to another. In fact, this was an *ad hoc* extension for modelling human multitasking. In contrast to Broccia *et al.* we model just one time aspect within basic activities, the duration of the mental process, which is the only time aspect characterising the basic activity. Broccia *et al.*, instead, neglect this time aspect.

The rest of the paper is structured as follows. Sections 2 and 3 present overviews of Real-Time Maude and the way it models BRDL syntax, respectively. Section 4 presents the Real-Time Maude implementation of the model of human memory and memory processes that provide the dynamics of BRDL constructs. Section 5 illustrates the rewrite rules to emulate human reasoning and the environment in which humans operate. Section 6 concludes the paper.

2 Real-Time Maude

Real-Time Maude [16, 18] is a formal modeling language and high-performance simulation and model checking tool for distributed real-time systems. It is based on Full Maude, the object-oriented extension of Core Maude, which is the basic version of Maude.

An algebraic equational specification (specifying sorts, subsorts, functions and equations defining the functions) defines the data types in a functional programming style. Labeled rewrite rules `cr1 [l]: t => t' if cond` define local transitions from state t to state t' , and tick rewrite rules `cr1 [l]: {t} => {t'} in time Δ if cond` advance time in the *entire* state t by Δ time units.

We briefly summarize the syntax of Real-Time Maude and refer to Ölveczky's work [16, 18] for more details. Maude *equational logic* supports declaration of *sorts*, with keyword `sort` for one sort, or `sorts` for many. A sort A may be specified as a subsort of a sort B by `subsort A < B`. Operators are introduced with the `op` and `ops` keywords: `op f : s1 ... sn -> s`. They can have user-definable syntax, with underbars '`_`' marking the argument positions. Some operators can have equational *attributes*, such as `assoc`, `comm`, and `id`, stating that the operator is associative, commutative and has a certain identity element, respectively. Such attributes are used by the Maude engine to match terms *modulo* the declared axioms. An operator can also be declared to be a constructor (`ctor`) that defines the carrier of a sort. Equations and rewrite rules are introduced with, respectively, keywords `eq`, or `ceq` for conditional equations, and `r1`, or `cr1` for conditional rules. The mathematical variables in such statements are declared with the keywords `var` and `vars`, or can be introduced on the fly in a statement without being declared previously, in which case they have the form `var:sort`. An equation $f(t_1, \dots, t_n) = t$ with the `owise` (for “otherwise”) attribute can be applied to a subterm $f(\dots)$ only if no other equation with left-hand side $f(u_1, \dots, u_n)$ can be applied.

A declaration `class C | att1 : s1, ..., attn : sn` declares a class C with attributes att_1 to att_n of sorts s_1 to s_n . An *object* of class C is represented as a term `<O : C | att1 : val1, ..., attn : valn>` of sort `Object`, where O , of sort `Objid`, is the object's *identifier*, and where val_1 to val_n are the current values of the attributes att_1 to att_n . The state is a term of sort `Configuration`, and is a *multiset* of objects and messages. Multiset union is denoted by an associative and commutative juxtaposition operator, so that rewriting is *multiset rewriting*.

Real-Time Maude specifications are executable, and the tool provides a variety of formal analysis methods. The *timed rewriting* command (`tfrew t in time <= timeLimit .`) simulates *one* of the system behaviours by rewriting the initial state t up to duration *timeLimit*.

3 Behaviour and Reasoning Description Language

The Behaviour and Reasoning Description Language (BRDL) [10] originates from and extends the *Human Behaviour Description Language (HBDL)* introduced in previous work [8, 9]. HBDL aims at the modelling of automatic and deliberate human behaviour while interacting with an environment consisting of heterogeneous physical components. It requires reasoning and problem solving aspects to be modelled explicitly in a procedural way, whereby the reasoning process and the problem solution are explicitly described with the language. BRDL, instead, is equipped with the linguistic constructs to specify reasoning

goals, inference rules and unsolved problems. It is then the cognitive engine that implements the language to emulate the reasoning and problem solving processes.

BRDL is based on Atkinson and Shiffrin's *multistore model* of human memory [1]. This model is characterised by three stores between which various forms of information flow: *sensory memory*, where information perceived through the senses persists for a very short time, *short-term memory (STM)*, 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. 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 [2]. We consider a further decomposition of LTM: *semantic memory*, which refers to our *knowledge* of the world and consists of the *facts* that can be *consciously* recalled, and *procedural memory*, which refers to our *skills* and consists of *rules* and *procedures* that we *unconsciously* use to carry out tasks, particularly at the motor level.

BRDL has a concise, appealing syntax, which is presented elsewhere [10]. In order to show how BRDL is translated to Maude, in this section we introduce an ASCII, verbose version of the syntax, as it is implemented in Maude. Both HDBL and BRDL describe human behaviour through the manipulation of three kinds of entities:

perceptions are sensed in the environment and enter human input channels;
actions are performed by the human on the environment;
cognitive information consists in the items we store in our STM, including information retrieved from the LTM, goals, recent perceptions or planned actions.

3.1 BRDL Entities and Cognitive Control

BRDL entities are modelled with Maude using the following sort structure.

```
sorts Perception Action Cognition BasicItem Item Goal .
subsorts Cognition Perception Action < BasicItem < Item .
subsort Goal < Item .
```

where *Perception*, *Action* and *Cognition* model perceptions, actions and cognitive information, respectively. Sort *Item* models anything that can be stored in STM and sort *BasicItem* is its subsort that excludes goals (from sort *Goal*). All these entities may also be elements of sets that define further sorts as follows:

```
subsorts Perception < PerceptionSet < BasicItemSet .
subsorts Cognition < CognitionSet < BasicItemSet .
subsorts Action < ActionSet < BasicItemSet .
```

```

subsort BasicItem < BasicItemSet .
subsorts EmptyItemSet < PerceptionSet CognitionSet ActionSet
      < BasicItemSet < ItemSet .
subsort Item < ItemSet .
op none : -> EmptyItemSet [ctor] .
op _;_ : BasicItemSet BasicItemSet ->
      BasicItemSet [ctor assoc comm id: none] .
op _;_ : PerceptionSet PerceptionSet ->
      PerceptionSet [ctor assoc comm id: none] .
op _;_ : ActionSet ActionSet ->
      ActionSet [ctor assoc comm id: none] .
op _;_ : ItemSet ItemSet -> ItemSet [ctor ditto] .

```

We use semicolon “;” as the general operator to add elements or subsets to a set, starting from an empty set (*none* in this case).

We extend *Perception* to *DefPerception* and *Action* to *DefAction* by including as default values *noPerception* and *noAction* to model the absence of perception and action, respectively.

```

sorts DefPerception DefAction .
subsort Perception < DefPerception . subsort Action < DefAction .
op noAction : -> DefAction [ctor] .
op noPerception : -> DefPerception [ctor] .

```

Only relevant perceptions are transferred, possibly after some kind of processing, to the STM using *attention*, a selective processing activity that aims to focus on one aspect of the environment while ignoring others. *Explicit attention* is associated with our goal in performing a task. It focusses on goal-relevant stimuli in the environment. *Implicit attention* is grabbed by sudden stimuli that are associated with the current mental state or carry emotional significance. Inspired by Norman and Shallice [15], we consider two levels of cognitive control:

automatic control

fast processing activity that requires only *implicit attention* and is carried out outside awareness with no conscious effort implicitly, using rules and procedures stored in the procedural memory;

deliberate control

processing activity triggered and focussed by *explicit 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.

In order to model automatic and deliberate control as well as reasoning, we introduce the following sorts and operations.

```

sorts Automatism KnowledgeDomain .
op automatism : KnowledgeDomain -> Automatism [ctor] .
op goal : KnowledgeDomain BasicItemSet -> Goal [ctor] .
op infer : KnowledgeDomain -> Inference [ctor] .

```

We define automatic behaviour in terms of a specific knowledge domain (sort `KnowledgeDomain` and operation `automatism`). Automatic behaviour is driven by the knowledge domain, which gives a focus to implicit attention.

Deliberate behaviour is driven by a goal, which not only depends on the knowledge domain but also on a representation of the goal achievement. This representation may be given by a combination of perceptions, actions and cognitive information. For example,

- during the interaction with an ATM (automatic teller machine) with the goal of withdrawing cash, we achieve the goal when we perform the action of collecting the cash;
- if our goal is to switch a light on, we achieve the goal when we perceive the light being on;
- if our goal is to solve a mathematical puzzle, we achieve the goal when the solution is represented by the cognitive information in our STM.

Inference is driven by the knowledge domain on which we are reasoning.

3.2 Basic Activities

Human behaviour is modelled in BRDL (and HTDL) as a set of *basic activities*, defined through the following sorts and operations

```

sorts AutomaticActivity DeliberateActivity Knowledge .
op _:_>|_-->_|>duration_ : Automatism BasicItemSet DefPerception
    DefAction ItemSet Time -> AutomaticActivity . [ctor]
op _:_>|_-->_|>duration_ :Goal BasicItemSet DefPerception
    DefAction ItemSet Time -> DeliberateActivity [ctor] .
op _:_>|_-->_|>duration_ : Inference BasicItemSet
    ItemSet Time -> Knowledge [ctor] .

```

An automatic basic activity within a given knowledge domain *domain* is modelled in BRDL and HTDL as

```

automatism(domain) : info1 >| perception --> action |> info2 duration d

```

where *info1* is the triggering cognitive information in the STM, *perception* is the triggering perception, *action* is the performed action, *info2* is a new cognitive information stored in the STM, and *d* is the duration of the mental processing. Symbol “>|” denotes that *info1* is removed from the STM and “|>” denotes that *info2* is stored in the STM. Using derived operations (i.e. not defined as constructors but through equations) we have the following syntactic sugar

```

automatism(domain) : info1 | perception -->
    action |> info2 duration d

```

where *info1* acts as a trigger but is not is removed from STM, and

```

automatism(domain) : info | info1 >| perception -->
    action|> info2 duration d

```

where the union $info;info1$ acts as a trigger but only $info1$ is removed from STM.

A deliberate basic activity within a given knowledge domain $domain$ is modelled in BRDL and HTDL as

```
goal(domain, info) : info1 >| perception --> action |> info2 duration d
```

where $info$ is the information denoting the achievement of the goal.

An inference within a given knowledge domain $domain$ is modelled in BRDL as

```
inference(domain) : info1 >|-->|> info2 duration d
```

where $info1$ is the premise and $info2$ is the consequence.

Syntactic sugar for deliberate basic activities and inferences is defined similarly to automatic basic activities.

Procedural memory is modeled as the sort ProcMem, which is a set of automatic basic activities

```
sort ProcMem . subsort AutomaticActivity < ProcMem .
op emptyPM : -> ProcMemory [ctor] .
op _;_ : ProcMemory ProcMemory -> ProcMemory
[ctor assoc comm id: emptyPM] .
```

Semantic memory is modeled by two sort, sort ActivMem, which is a set of deliberate basic activities,

```
sort ActivMem . subsort DeliberateActivity < ActivMem .
op emptyAM : -> ActivMem [ctor] .
op _;_ : ActivMem ActivMem -> ActivMem [ctor assoc comm id: emptyASM] .
```

and sort InferMem , which is a set of inferences,

```
sort InferMem . subsort Knowledge < InferMem .
op emptyIM : -> InferMem [ctor] .
op _;_ : InferMem InferMem -> InferMem [ctor assoc comm id: emptyIM] .
```

3.3 Zebra Crossing Example

As an example to illustrate these forms of human behaviour and reasoning, let us consider car driving. The knowledge domain is given by constant operation

```
op driving : -> KnowledgeDomain [ctor] .
```

Automatic control is essential in properly 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 that requires explicit attention. For example, the learner has to explicitly pay attention to the other cars, the pedestrian walking on the footpath, who may be ready to walk across the road, the presence of zebra crossings, traffic lights, road signals, etc. These are goals that drive explicit attention. Moreover, the information gathered through this process has to be deliberately used to achieve goals (deliberate control), which continuously emerge while driving as a learner.

For instance, let us define perceptions, actions and cognitive information of a driver dealing with a zebra crossing as follows:

```
ops static moving ped zebra : Oid -> Perception [ctor] .
ops stop go : Action [ctor] .
ops givenWayPed waitForPed leftZebraCrossing : -> Cognition [ctor] .
```

The role of such constructors will be explained later in this section.

A learner's perception of an approaching zebra crossing, normally by seeing a road signal, either a horizontal or vertical one, triggers the storage of the cognitive representation of this perception in STM. We may model this instance of explicit attention as

```
goal(driving,zebra) : none | zebra --> noAction |> zebra duration d1
```

where `zebra` denotes the perception of the zebra crossing and occurs three times for modelling, from left to right: the achievement of the goal of explicitly perceiving the presence of the zebra crossing, the actual perception and the representation of the perception in STM. There is no resultant action since here we are modelling attention.

When also pedestrians ready to cross are perceived, the cognitive representation of this perception is stored in STM.

```
goal(driving,ped) : none | ped --> noAction |> ped duration d2
```

Once the cognitive representations of perceptions `zebra` and `ped` are in STM, if the car is moving and the driver is (cognitively) aware of it (modelled by `moving` in the STM), this composite mental state triggers the retrieval of the following inference, which models the road code rule concerning zebra crossings:

```
inference(driving) :
  moving ; zebra ; ped |-->|> goal(driving,givenWayPed) duration d3
```

Retrieving the rule results in adding goal `goal(driving,givenWayPed)` to the STM without removing `moving`, `zebra` and `ped`. Such a goal dictates the prescribed behaviour of giving way to pedestrians (whose achievement is denoted by `givenWayPed`). This behaviour is 'implemented' by the human as modelled by the following deliberate basic activity:

```
goal(driving,givenWayPed) :
  none | none --> stop |> waitForFree duration d4
```

where `stop` is the action of stopping the car and `waitForFree` denotes the driver's mental state of waiting for the zebra crossing to be free.

Once automaticity in driving is acquired, the driver is no longer aware of low-level details and resorts to implicit attention to perform them (automatic control). In general, also an expert driver always starts driving with a precise goal in mind, which normally is that of reaching a specific destination, possibly as a subgoal of the reason for reaching it. Although such a goal is kept in the driver's STM, most driving activities are carried out under automatic control, with no need to retrieve the learned rules. Therefore, the behaviour of an expert driver is modelled as follows:

```
automatism(driving) : none | zebra --> noAction |> zebra duration d1
automatism(driving) : none | ped --> noAction |> ped duration d2
```



```

automatism(driving) :
  moving ; zebra |> ped --> stop |> ped ; waitForFree duration d3
automatism(driving) :
  moving ; ped |> zebra --> stop |> zebra ; waitForFree duration d3

```

The first two automatic activities model implicit attention, which results in the storage of the perception of zebra crossing and pedestrians, respectively. The last two automatic activities model the automatic reaction to the perception of pedestrian in combination with the awareness of the presence of a zebra crossing or the perception of zebra crossing in combination with the awareness of the presence of pedestrian, depending on which perception occurs first.

We can note that automatic behaviour is more efficient than deliberate behaviour for the following reasons:

- there are no goals in STM to drive explicit attention (low cognitive load);
- there is an immediate reaction to perceptions, when in the appropriate mental state (faster reaction);
- there is no recourse to inference (decreased access to LTM).

4 Dynamics of BRDL Models

We model the structure of the human memory using the following Real-Time Maude class.

```

class Human | cognitiveLoad : Nat,
             shortTermMemory : TimedItemSet,
             inferSemMem : InferMem,
             activSemMem : ActivMem,
             procMem : ProcMem .

```

The STM is modelled by attribute `shortTermMemory` with `cognitiveLoad` being its current load, the semantic memory by the two attributes `inferSemMem` and `activSemMem` and the procedural memory by the single attribute `procMem`.

4.1 STM Model with Real-Time Maude

The limited capacity of the STM requires the presence of a mechanism to empty it when the stored information is no longer needed. In fact, information in the STM decays very quickly, normally in less than one minute, unless it is reinforced through maintenance rehearsal. To implement STM decay, we need to associate a time to the elements of sort `Item`

```

sorts TimedItem TimedItemSet .
subsort TimedItem < TimedItemSet .
op _decay_ : Item Time -> TimedItem [ctor] .
op emptyTIS : -> TimedItemSet [ctor] .
op _;_ : TimedItemSet TimedItemSet -> TimedItemSet
        [ctor assoc comm id: emptyTIS] .
op maxDecayTime : -> Time .
eq maxDecayTime = 20000 .

```

Therefore the STM is modelled as an element of sort `TimedItemSet`, the set of elements of sort `TimedItem`. A piece of information in the STM is associated with a *decay time*, which is initialised to the *maximum decay time* (`maxDecayTime`, for example set to 20000 ms) when the information is stored in the STM. Then decay time decreases along with the passage of time. A piece of information disappears from the STM once its decay time has decreased to 0.

Additionally, every time a goal is achieved, a process called *closure* may determine a subconscious removal of information from the STM: the information used to complete the task is likely to be removed from the STM, since it is no longer needed. Therefore, when closure occurs, a piece of information may disappear from the STM even before its decay time has decreased to 0. Conversely, maintenance rehearsal resets the decay time to the maximum decay time.

In order for a goal with `BIS` as parameter of sort `BasicItemSet` to be achieved

- the entire cognitive information included in `BIS` has to be in STM;
- one of the perceptions (if any) has to be the trigger of the occurring basic activity (which may be automatic or deliberate);
- one of the actions (if any) has to be performed by the occurring basic activity.

This is implemented by operations

```
op removeTime : TimedItemSet -> ItemSet .
op goalAchieved : Goal ItemSet DefPerception DefAction -> Bool .
```

where operation `removeTime` removes the time from the elements of the STM and operation `goalAchieved` returns `true` if the goal is achieved.

It is not fully understood how closure works. We can definitely say that once the goal is achieved, it is removed from the STM. However, it is not clear what happens to the information that was stored in STM in order to achieve the goal. We said at the end of Sect. 3.1 that if an ATM is used with the goal of withdrawing cash, the goal is achieved when the user collects the cash delivered by the ATM [8]. However, old ATM interfaces (some still in activity) deliver the cash before returning the card to the user. There is then the possibility that the user collects the cash and, feeling the goal achieved, abandons the interaction forgetting to collect the card. This cognitive error is known as *post-completion error* [6,7,13]. It could be explained by the loss of the information that was stored in STM, when the user inserted the card in the ATM, as a reminder to collect the card at a later stage. In fact, such a loss of information is the result of the closure due to the achievement of the goal when the user collects the cash.

In practice, however, a user interacting with an old ATM interface does not always forget the card. This may be explained by assuming that the likelihood to forget the card depends on the user's cognitive load. Therefore we define the following thresholds

```
op closureThresholdLow : -> Nat . eq closureThresholdLow = 4 .
op closureThresholdHigh : -> Nat . eq closureThresholdHigh = 6 .
```

and force closure to occurs if the cognitive load is at least `closureThresholdHigh` and prevent its occurrence if the cognitive load is less than `closureThresholdLow`. In all other cases closure may occur non-deterministically.

Finally, a piece of information may also non-deterministically disappear from the STM when the STM has reached its maximum capacity and it is needed to make space for the storage of new information. This is implemented by allowing the STM to temporarily exceed its capacity, thus reaching an unstable state in which the only applicable rule is

```

crl [forgetSomethingIfSTMfull] :
  < H : Human | shortTermMemory : (ITEM decay NZT) ; STM,
    cognitiveLoad : CL >
=>
  < H : Human | shortTermMemory : STM,
    cognitiveLoad : sd(CL, 1) >
  if CL > stmCapacity .

```

where `sd` is the symmetric difference between natural numbers.

4.2 Model of the Environment

A specific environment with which the human interacts is defined as an object of class

```

class Environment | state : TimedEnvState,
  transitions : EnvTransitions .

```

The `state` attribute characterises the environment and its time aspects by means of the following sort structure

```

sort EnvState .
sorts TimedEnvState ExpiringEnvState TimedEnvStateSet .
subsort EnvState < ExpiringEnvState < TimedEnvState < TimedEnvStateSet .
op _expiring'in_ : EnvState Time -> ExpiringEnvState [ctor] .
op _expired : EnvState -> ExpiringEnvState [ctor] .
op _in'time_ : ExpiringEnvState Time -> TimedEnvState [ctor right id: 0]
var STATE : EnvState .
eq STATE expiring in 0 = STATE expired .
op noEnvState : -> TimedEnvStateSet [ctor] .
op _;_ : TimedEnvStateSet TimedEnvStateSet -> TimedEnvStateSet
  [ctor assoc comm id: noEnvState] .

```

where

- sort `EnvState` of environmental states is user-defined and application-specific;
- sort `ExpiringEnvState` add a *life time* to the environmental state;
- sort `TimedEnvState` add a *delay time* to the (possibly expiring) environmental state.

Note that 0 is right identity in the construction of timed environmental states out of expiring environmental states. Thus a timed environmental state with delay 0 is actually an expiring environmental state (with no delay). Moreover, expiring environmental states are characterised by a postfix constructor `expired` in order to determine different transitions with respect to the non-expired states.

The sort `EnvTransitions` models environmental transitions as follows:

```
sort EnvTransitions .
sort EnvTransition .
subsort EnvTransition < EnvTransitions .
op noTrans : -> EnvTransitions [ctor] .
op _-->_ : ExpiringEnvState TimedEnvState -> EnvTransition [ctor] .
op _--_-->_ : EnvState Action TimedEnvState -> EnvTransition [ctor] .
op _;_ : EnvTransitions EnvTransitions -> EnvTransitions
      [ctor assoc comm id: noTrans] .
```

Obviously interactions (`_--_-->_`) are associated with actions, internal actions (`_-->_`) are not.

The sort `EnvTransitions` is populated through the user-defined, application-specific operation

```
op transitions : Cid Oid -> EnvTransitions .
```

where `Cid` is a class identifier and `Oid` is an object identifier.

States of the environment may be observable by humans. Such observability is modelled as

```
op observability : ExpiringEnvState -> PerceptionSet .
eq observability(STATE expired) = none .
eq observability(STATE expiring in NZT) = observability(STATE) .
```

with the rest of operation `observability` user-defined and application-specific.

4.3 Zebra Crossing Environment

In order to define the behaviour of the environment for the example in Sect. 3.3, we need two environments, one to model the car behaviour and one to model the zebra crossing behaviour. Both car and zebra crossing have a location, which is variable for the car and fixed for the zebra crossing. They also need to have additional state components to characterise whether the car is moving or is static and whether the zebra crossing has pedestrians or is free.

Environment and Observability. If we assume to have only one human, one car and one zebra crossing

```
ops driver1 car1 zebra1 : -> Oid [ctor] .
```

then the environmental state is defined as follows:

```

sorts Location AdditionalState .
ops atInit atZebra atFinal : -> Location [ctor] .
ops hasPed isFree isMoving isBraking isStatic : -> AdditionalState [ctor] .
op state : Location AdditionalState -> EnvState [ctor] .

```

The meanings of the operations that define locations and additional state components are obvious. An environmental state consists of a location and an additional state.

The observability operation is defined as follows:

```

eq observability(state(LOC,isStatic)) = static .
eq observability(state(LOC,isMoving)) = moving .
eq observability(state(zebra1,AS)) = zebra .
eq observability(state(zebra1,isFree)) = zebra ; noPed .
eq observability(state(zebra1,hasPed)) = zebra1 ; ped .

```

Transition System. The environmental transition systems are defined as

```

class Car . subclass Car < Environment .
var C : Oid .
eq transitions(Car, C) =
  (state(atInit,isMoving) --> state(atZebra, isMoving)
                                expiring in 1 in time 30000) ;
  (state(atZebra,isMoving) -- stop(C) --> state(atZebra, isBraking)) ;
  (state(atZebra,isBraking) --> state(atZebra, isStatic) in time 2000) ;
  (state(atZebra,isStatic) -- go(C) --> state(atZebra, isMoving)) ;
  (state(atZebra,isMoving) --> state(atFinal, isMoving)
                                expiring in 1 in time 30000) ;
  (state(atFinal,isMoving) -- stop(C) --> state(atFinal, isBraking)) ;
  (state(atFinal,isBraking) --> state(atFinal, isStatic in time 2000) .

```

for the car, and

```

class Zebra . subclass Zebra < Environment .
var Z : Oid .
eq transitions(Zebra, Z) =
  (state(Z,isFree) expired --> state(Z, hasPed) expiring in 5000) ;
  (state(Z,hasPed) expired --> state(Z, isFree) expiring in 20000) .

```

for the zebra crossing.

The timings mean that the car takes time 30000 to move between two consecutive locations and time 2000 to brake, being in an unstable state until these times are elapsed and, once stable, expiring immediately (in time 1) if not taken, and that there are pedestrian crossing every 25000 time units (25000 = 20000 + 5000) who take time 5000 to cross.

Initial Configuration. Let us consider a driver who has already acquired a general automatism in driving, in which implicit attention controls the storage of information in STM, but still needs to perform inferences to apply road code rules. The initial configuration of the overall system is

```

op init : -> Configuration .
eq init = < cerone : Human |
    cognitiveLoad : 2,
    shortTermMemory : emptyTIS,
    proceduralMemory :
    (automatism(driving) : none | moving --> noAction |> moving duration 1) ;
    (automatism(driving) : none | static --> noAction |> static duration 1) ;
    (automatism(driving) : none | zebra --> noAction |> zebra duration 1) ;
    (automatism(driving) : none | ped --> noAction |> hasPed duration 1) ;
    (automatism(driving) : none | freePed --> noAction |> freePed duration 1),
    knowledge :
    (infer(driving) : (moving ; zebra ; hasPed) |-->|>
        goal(driving,givenWayPed) duration 10) ;
    (infer(driving) : (static ; zebra ; freePed) |-->|>
        goal(driving,leftZebraCrossing) duration 10),
    activity :
    (goal(driving,givenWayPed) :
        (moving ; zebra ; hasPed) | noPerception -->
            stop(car1) |> waitForPed) duration 10) ;
    (goal(driving,leftZebraCrossing) :
        (zebra ; waitForPed) > (static ; freePed) | noPerception -->
            go(car1) |> none duration 10)
>
< zebra1 : Zebra | transitions : transitions(Zebra, zebra1),
    state : state(zebra1,zebraPed) expiring in 5000
>
< car1 : Car | transitions : transitions(Car, car1),
    state : state(initLoc,moving)
> .

```

5 Rewrite Rules

At the end of Sect. 4.1 we have introduced the `forgetSomethingIfSTMfull` rewrite rule. In this section we illustrate three more rewrite rules: `internal`, `reasoning` and `timePassing`. Other rewrite rules not presented here involve the automatic and deliberated activities, including special cases such as implicit and explicit attention, which are characterised by the presence of perception and absence of action, and cognition, which are characterised by the absence of both perception and action. Such rules are duplicated for the closure and non-closure cases.

5.1 Internal Action Rewrite Rule

Internal actions are modelled by the following rewrite rule.

```

cr1 [internal] :
    {< E : Environment | >

```

```

    REST}
=>
  {< E : Environment | state : TESTATE >
  REST}
if ALL-TESTATES := fireTransitions(< E : Environment | >)
  /\ TESTATE ; OTHER-TESTATES := ALL-TESTATES .

```

The rule makes use of the `fireTransitions` operation, which is defined as follows:

```

op fireTransitions : Configuration -> TimedEnvStateSet .
eq fireTransitions(< E : Environment |
  state : ESTATE,
  transitions : (ESTATE --> TESTATE) ; TRANSES > REST ) =
  TESTATE ; fireTransitions( < E : Environment |
  state : ESTATE,
  transitions : TRANSES > REST ) .
eq fireTransitions( REST ) = noEnvState [owise] .

```

The `fireTransitions` operation returns the set of the environmental states generated by the firing of the enabled internal transitions. In the `internal` rewrite rule, such a set is assigned to variable `ALL-TESTATES`, which is matched to `TESTATE ; OTHER-TESTATES`, thus giving the rewritten state `TESTATE`.

5.2 Reasoning Rewrite Rule

Reasoning is modelled by the following rewrite rule.

```

crl [reasoning] :
  {< H : Human |
  cognitiveLoad : CL,
  shortTermMemory : (TIS1 ; TIS2),
  inferMem : (infer(KD) : BIS >| -->|> IS duration T) ; KNOW >
  REST}
=>
  {< H : Human |
  cognitiveLoad : card(NEW-STM),
  shortTermMemory : NEW-STM,
  inferMem : (infer(KD) : BIS >| -->|> IS duration T) ; KNOW >
  idle(REST, T)}
in time T
if BIS == removeTime(TIS1)
  /\ CL < closureThresholdHigh /\ CL <= stmCapacity
  /\ NEW-STM := addTime(BIS ; IS, maxDecayTime)) ; idle(TIS2,T) .

```

In addition to operation `removeTime` introduced in Sect. 4.1, the rule makes use of

- the `addTime` operation, which transforms the untimed sets `BIS` and `IS` into a timed set to be added to the STM;
- the `idle` operation, which models the passage of a given time by decrementing each element of sort `TimedItemSet` of the STM and, for each environment component, the delay and expiration times of the `state` attribute, which is of sort `TimedEnvState`, if positive.

Note that the decay time of the premises in `BIS` is set to the maximum decay time because the use of `BIS` in the inference is an implicit maintenance rehearsal of its timed version `TIS1`.

Let us consider the zebra crossing example introduced in Sects. 3.3 and 4.3. When moving, `zebra` and `ped` are stored in the STM, the road code rule concerning zebra crossing (from Sect. 4.3)

```
inference(driving) :
    moving ; zebra ; ped |-->|> goal(driving,givenWayPed) duration d3
```

is retrieved, thus enabling the application of Maude `reasoning` conditional rule with

```
BIS = moving ; zebra ; ped    and    IS = goal(driving,givenWayPed)
```

The new goal `goal(driving,givenWayPed)` is then added to the STM and triggers the following deliberate basic activity, stored in LTM, which implement the road code rule (from Sect. 4.3):

```
goal(driving,givenWayPed) :
    none | none --> stop |> waitForFree duration d4
```

Such a rule dictates the action of stopping the car (`stop`) and the storage of `waitForFree` in the STM.

5.3 Time Passing Rewrite Rule

```
cr1 [timePassing] :
    {CONFIG}
=>
    {idle(CONFIG,1)}
    in time 1
    if nothingEnabled(CONFIG) .
```

where operation `nothingEnabled` is defined as

```
op nothingEnabled enablingSTM : Configuration -> Bool .
eq nothingEnabled(CONFIG) = (fireTransitions(CONFIG) == noEnvState)
    and (enablingSTM(CONFIG)) == false .
```

and operation `enablingSTM` checks whether the configuration has an object of class `Human` whose STM either exceeds the maximum cognitive load or is enabling an inference rule, an automatic basic activity or a deliberate basic activity. In this way the `timePassing` rewrite rule may be applied only if no other rewrite rule can be applied.

6 Conclusion and Future Work

We have presented a translation of BRDL into Real-Time Maude. In previous work [8,9], a subset of BRDL, the *Human Behaviour Description Language (HBDL)*, was implemented using Core Maude. However, that untimed implementation was limited to automatic and deliberate behaviour powered by a very simple, fixed short-term memory model, with a minimalist, inflexible approach to closure and without decay. Reasoning and problem solving aspects had to be modelled explicitly in a procedural way in a limited, unstructured environment consisting of just one component.

BRDL, instead, is equipped with the linguistic constructs to specify reasoning goals, inference rules and unsolved problems. These linguistic constructs, extensively described in our previous work [10] can be used to model human behaviour in a natural way from the point of view of a psychologist or cognitive scientist. The Real-Time Maude implementation of BRDL presented in this paper provides an engine capable to emulate the human reasoning specified by such constructs, but its knowledge is not needed to use BRDL. Moreover, the object-oriented and real-time aspects of Maude allow us to overcome the limitation of previous work [8] and carry out the implementation of the time aspects envisaged in recent work [9].

Moreover, our work differentiates itself from the work by Broccia *et al.* [3,4] in several respects:

- we have implemented, using Real-Time Maude, a language for the high-level, general description of human behaviour and reasoning (BRDL), whereas the work by Broccia *et al.* is restricted to the modelling and analysis of human multitasking;
- our modelling approach clearly separate human cognition from its environment, with all interaction aspects emerging through the composition of the human component with the operating environment, whereas the framework developed by Broccia *et al.* explicitly incorporates in the human component interaction aspects, such as task duration and difficulty, and delay due to external constraints, such as the presence of other tasks;
- we model the duration of the mental processing, a time aspect that has not been considered by Broccia *et al.*;
- Broccia *et al.* adopt the same minimalist, inflexible approach to closure introduced in Cerone's previous work [8], in which all cognitive information used to achieve the goal is removed independently of the cognitive load, whereas, in our approach, we may use and compare several forms of closure and use thresholds on cognitive load to control the application of closure;
- our components are eager, namely the time passing rewrite rule may be applied only if no other rewrite rule can be applied.

As future work we plan to implement BRDL problem solving constructs [10] and use the model checking capabilities of Real-Time Maude to extend the untimed analysis approach used in previous work [8] to the formal verification of timed properties.

References

1. Atkinson, R.C., Shiffrin, R.M.: Human memory: a proposed system and its control processes. In: Spense, K.W. (ed.) *The Psychology of Learning and Motivation: Advances in Research and Theory II*, pp. 89–195. Academic Press (1968)
2. Atkinson, R.C., Shiffrin, R.M.: The control of short-term memory. *Sci. Am.* **225**(2), 82–90 (1971)
3. Broccia, G., Milazzo, P., Ölveczky, P.C.: An executable formal framework for safety-critical human multitasking. In: Dutle, A., Muñoz, C., Narkawicz, A. (eds.) *NFM 2018*. LNCS, vol. 10811, pp. 54–69. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-77935-5_4
4. Broccia, G., Milazzo, P., Ölveczky, P.C.: Formal modeling and analysis of safety-critical human multitasking. *Innov. Syst. Softw. Eng.* 169–190 (2019). <https://doi.org/10.1007/s11334-019-00333-7>
5. Butterworth, R., Blandford, A.E., Duke, D.: Demonstrating the cognitive plausibility of interactive systems. *Form. Asp. Comput.* **12**, 237–259 (2000)
6. Byrne, M.D., Bovair, S.: A working memory model of a common procedural error. *Cogn. Sci.* **21**, 31–61 (1997)
7. Byrne, M.D., Davis, E.M.: Task structure and postcompletion error in the execution of a routine procedure. *Hum. Factors* **48**, 627–638 (2006)
8. Cerone, A.: A cognitive framework based on rewriting logic for the analysis of interactive systems. In: De Nicola, R., Kühn, E. (eds.) *SEFM 2016*. LNCS, vol. 9763, pp. 287–303. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41591-8_20
9. Cerone, A.: Towards a cognitive architecture for the formal analysis of human behaviour and learning. In: Mazzara, M., Ober, I., Salaün, G. (eds.) *STAF 2018*. LNCS, vol. 11176, pp. 216–232. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-04771-9_17
10. Cerone, A.: Behaviour and reasoning description language (BRDL). In: *SEFM 2019 Collocated Workshops*. LNCS. Springer (2019, in press)
11. Harrison, M.D., Campos, J.C., Rukšėnas, R., Curzon, P.: Modelling information resources and their salience in medical device design. In: *EICS 2016*, pp. 194–203. ACM (2026)
12. Kotseruba, I., Tsotsos, J.K.: 40 years of cognitive architectures: core cognitive abilities and practical applications. *Artif. Intell. Rev.* **53**(1), 17–94 (2018). <https://doi.org/10.1007/s10462-018-9646-y>
13. Li, S.W., Blandford, A., Cairns, P., Young, R.M.: The effect of interruptions on postcompletion and other procedural errors: an account based on the activation-based goal memory model. *J. Exp. Psychol. Appl.* **14**, 314–328 (2008)
14. Martí-Oliet, N., Meseguer, J.: Rewriting logic: roadmap and bibliography. *Theoret. Comput. Sci.* **285**(2), 121–154 (2002)
15. Norman, D.A., Shallice, T.: Attention to action: willed and automatic control of behaviour. In: *Consciousness and Self-Regulation, Advances in Research and Theory*, vol. 4. Plenum Press (1986)
16. Ölveczky, P.C.: Real-time maude and its applications. In: Escobar, S. (ed.) *WRLA 2014*. LNCS, vol. 8663, pp. 42–79. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-12904-4_3
17. Ölveczky, P.C.: *Designing Reliable Distributed Systems*. UTCS. Springer, London (2017). <https://doi.org/10.1007/978-1-4471-6687-0>
18. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time-Maude. *High-Order Symb. Comput.* **20**(1–2), 161–196 (2007)

19. Oulasvirta, A., Kristensson, P., Bi, X., Howes, A. (eds.): Computational Interaction. Oxford University Press, Oxford (2018)
20. Weyers, B., Bowen, J., Dix, A., Palanque, P.: Erratum to: the handbook of formal methods in human-computer interaction. In: Weyers, B., Bowen, J., Dix, A., Palanque, P. (eds.) The Handbook of Formal Methods in Human-Computer Interaction. HIS, pp. E1–E3. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-51838-1_21