An Executable Formal Framework for Safety-Critical Human Multitasking

Giovanna Broccia[1], Paolo Milazzo[1], and Peter Csaba Ölveczky[2]

[1] Department of Computer Science, University of Pisa, Pisa, Italy
{giovanna.broccia,milazzo}@di.unipi.it
[2] University of Oslo, Oslo, Norway

Abstract. When a person is concurrently interacting with different systems, the amount of cognitive resources required (cognitive load) could be too high and might prevent some tasks from being completed. When such human multitasking involves safety-critical tasks, for example in an airplane, a spacecraft, or a car, failure to devote sufficient attention to the different tasks could have serious consequences. To study this problem, we define an executable formal model of human attention and multitasking in Real-Time Maude. It includes a description of the human working memory and the cognitive processes involved in the interaction with a device. Our framework enables us to analyze human multitasking through simulation, reachability analysis, and LTL and timed CTL model checking, and we show how a number of prototypical multitasking problems can be analyzed in Real-Time Maude. We illustrate our modeling and analysis framework by studying the interaction with a GPS navigation system while driving, and apply model checking to show that in some cases the cognitive load of the navigation system could cause the driver to keep the focus away from driving for too long.

1 Introduction

These days we often interact with multiple devices or computer systems at the same time. Such *human multitasking* requires us to repeatedly shift attention from task to task. If some tasks are safety-critical, then failure to perform the tasks correctly and timely—for example due to cognitive overload or giving too much attention to other tasks—could have catastrophic consequences.

A typical scenario of safety-critical human multitasking is when a person interacts with a safety-critical device/system while using other less critical devices. For example, pilots have to reprogram the flight management system while handling radio communications and monitoring flight instruments [11]. Operators of critical medical devices, such as infusion pumps, often have to retrieve patient-specific parameters by accessing the hospital database on a different device while configuring the safety-critical device. Finally, a driver often interacts with the GPS navigation system and/or the infotainment system while driving.

Human multitasking could lead to cognitive overload (too much information to process/remember), resulting in forgetting/mistaking critical tasks. For example, [16] reports that during a routine surgery, the ventilator helping the patient to breathe was turned off to quickly take an X-ray without blurring the picture. However, the X-ray jammed, the anesthesiologist went to fix the X-ray but forgot to turn on the ventilator, leading to the patient's death. In another example, [8] analyzes the cause of 139 deaths when using an infusion pump, and finds that operator distraction caused 67 deaths, whereas problems with the device itself only caused 10 deaths. Similar figures and examples can be found in the context of aviation [3] and car driving [29, 10].

In addition to cognitive overload, human multitasking could also lead to ignoring the critical tasks for too long while focusing attention on less critical tasks. For instance, while reprogramming the flight management system, the pilot could miss something important on the flight instruments. If the interface of the virtual clinical folder requires the user's attention for too long, it can cause the operator to make some mistake in the infusion pump setup. An infotainment system that attracts the driver's attention for too long could cause a car accident.

There is therefore a clear need to analyze not only the functionality of single devices (or networks of devices), but also to analyze whether a human can safely use multiple devices/systems at the same time. Such study requires understanding how the human cognitive processes work when interacting with multiple systems and how human attention is directed at the different tasks at hand. In particular, the main cognitive resource to be shared among concurrent tasks is the human *working memory*, which is responsible for storing and processing pieces of information necessary to perform all the concurrent tasks.

In this paper we propose a formal executable model of human multitasking in safety-critical contexts. The model is specified in Real-Time Maude [20]. It is a significant modification and extension of the cognitive framework proposed by Cerone for the analysis of interactive systems [7]. As in that work, our model includes the description of the human working memory and of the other cognitive processes involved in the interaction with a device. The main difference is that Cerone only considered the interaction with a *single* device, whereas we focus on analyzing human multitasking. In contrast to [7], our framework also captures the limitations of a human's working memory (to enabling reasoning about hazards caused by cognitive overload) and includes timing features (to analyze, e.g., whether a critical task is ignored for too long).

After providing some background on human attention and multitasking and Real-Time Maude in Section 2, we present our Real-Time Maude model of safety-critical human multitasking in Section 3. Section 4 explains how Real-Time Maude can be used to analyze prototypical properties in human multitasking. We illustrate our formal modeling and analysis framework in Section 5 by studying the use of a GPS navigator while driving. We apply model checking to show that in some cases: (i) the cognitive load of the navigator interface could cause the driver to keep the focus away from driving for too long, and (ii) the working memory sharing between concurrent tasks can lead to overloading situations causing failures in one of the tasks. Finally, Section 6 discusses related work, and Section 7 gives some concluding remarks.

2 Preliminaries

2.1 Human Selective Attention and Multitasking

Human memory encodes, stores, and holds information, and is one of the cognitive resources most involved in interactions with computers [19, 2]. It can be differentiated into three separate components [2]: a sensory memory, where information detected by the senses is temporarily stored; a short-term store, where sensory information that is given attention is saved; and a long-term store, where information that has been rehearsed through attention in the short-term store is held indefinitely. The short-term store is the component that is most involved in interaction with computers, and is then called the *working memory* (WM).

WM is a cognitive system with a limited capacity responsible for the transient holding, processing, and manipulation of information. Although different hypotheses about the WM have been proposed, they all agree on two important aspects: it can store a limited amount of items, which, furthermore, decay over time, and it is responsible for both processing and storage activities. The amount of information—which can be, e.g., digits, letters, words, or other meaningful units—that can be held in WM is 7 ± 2 items [18].

Maintaining items in the WM requires human attention. Memory items are remembered longer if they are periodically refreshed by focusing on them. Even when performing a single task, in order not to forget something stored in the WM, the task has to be interleaved with memory refreshment. This has been the subject of several psychological theories. The most successful in explaining experimental data is the Time-Based Resource Sharing Model [4]. It introduces the notion of *cognitive load* (CL) as the temporal density of attentional demands of the task being performed. The higher the CL of a task, the more it distracts from refreshing memory items. According to [4], when the frequency of basic activities in a task is constant, the CL of the task equals $\sum a_i n_i / T$, where n_i is the number of task basic activities of type i, a_i represents the difficulty of such activities, and T is the total duration of the task.

Several studies show that the attentional mechanisms involved in WM refreshment are also the basis of multitasking [22, 14, 25, 12]. In particular, [12] describes the roles of the WM, the CL, and attention when executing a "main" task concurrently with a "distractor" task. It is shown that when the CL of the distractor task increases, the interaction with the main task could be impeded.

In [5] we use the cognitive load and two other factors, the task's *criticality level* and *waiting time* (the time the task has been ignored by the user) to define a measure of task attractiveness that we call *task rank.* The higher the task rank, the more likely the user will focus on it. Modeling attention switching based on parameters like CL, criticality level, and waiting time agrees with current understanding of human attention. In [5] we used this task rank to define an algorithm for simulating human attention. We studied the case of two concurrent tasks by varying the task parameters, and found that the task that more likely completes first is the one with the highest cognitive load, which is consistent with relevant literature (e.g., [4, 12]).

2.2 Real-Time Maude

Real-Time Maude [20] extends Maude [9] to support the formal specification and analysis of real-time systems in rewriting logic. Real-Time Maude specifications are executable, and the tool provides a range of formal analysis methods, including simulation, reachability analysis, and temporal logic model checking.

Specification. A Real-Time Maude module specifies a *real-time rewrite theory* [21] $(\Sigma, E \cup A, IR, TR)$, where:

- Σ is an algebraic *signature*; that is, declarations of *sorts*, *subsorts*, and *function symbols*, including a data type for time, which can be discrete or dense.
- $(\Sigma, E \cup A)$ is a *membership equational logic theory* [17], with E a set of (possibly conditional) equations, and A a set of equational axioms such as associativity, commutativity, and identity. $(\Sigma, E \cup A)$ specifies the system's state space as an algebraic data type.
- IR is a set of *labeled conditional rewrite rules* specifying the system's local transitions, each of which has the form[3] $[l] : t \longrightarrow t'$ **if** $\bigwedge_{j=1}^{m} u_j = v_j$, where l is a *label.* Such a rule specifies an *instantaneous transition* from an instance of t to the corresponding instance of t', *provided* the condition holds, i.e., u_i and v_j have the same normal form.
- TR is a set of *tick rewrite rules* l : `{`t`}` $\longrightarrow$ `{`t'`}` **in time** τ **if** *cond*, which specify that going from the *entire* state t to state t' takes τ time units.

We summarize the syntax of Real-Time Maude and refer to [9] for more details. Operators are declared `op` f `:` $s_1 \ldots s_n$ `->` s, and can have user-definable syntax, with underbars '`_`' marking the argument positions. Some operators can have equational *attributes*, such as `assoc`, `comm`, and `id`, stating, respectively, that the operator is associative and commutative and has a certain identity element, so that rewriting is performed *modulo* the declared axioms. Equations and rewrite rules are introduced with, respectively, keywords `eq`, or `ceq` for conditional equations, and `rl` and `crl`. The mathematical variables in such statements are declared with the keywords `var` and `vars`, or are introduced on-the-fly, in which case they have the form *var*`:`*sort* An equation $f(t_1, \ldots, t_n) = t$ with the `owise` (for "otherwise") attribute can be applied to a term $f(\ldots)$ only if no other equation with left-hand side $f(u_1, \ldots, u_n)$ can be applied.

A class declaration `class` C `|` att_1 `:` s_1, ..., att_n `:` s_n declares a class C with attributes att_1 to att_n of sorts s_1 to s_n. An *object* of class C in a given state is represented as a term `<` $O : C \mid att_1 : val_1, ..., att_n : val_n$ `>` of sort `Object`, where O, of sort `Oid`, is the object's *identifier*, and where val_1 to val_n are the current values of the attributes att_1 to att_n. A *message* is a term of sort `Msg`. The state of an object-oriented specification 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. For example, the rewrite rule

---

[3] An equational condition $u_i = v_i$ can also be a *matching equation*, written u_i`:=` v_i, which instantiates the variables in u_i to the values that make $u_i = v_i$ hold, if any.

```
rl [l] :  m(O,w)
          < O : C | a1 : x, a2 : O', a3 : z, a4 : y >
         =>
          < O : C | a1 : x + w, a2 : O', a3 : z, a4 : y >
          dly(m'(O',x), y) .
```

defines a family of transitions in which a message `m`, with parameters `O` and `w`, is read and consumed by an object `O` of class `C`, the attribute `a1` of object `O` is changed to `x + w`, and a new message `m'(O',x)` is generated; this message has a *message delay* `y`, and will become the "ripe" message `m'(O',x)` in `y` time units. Attributes whose values do not change and do not affect the next state of other attributes or messages, such as `a3`, need not be mentioned in a rule. Attributes that are unchanged, such as `a2`, can be omitted from right-hand sides of rules.

Formal Analysis. Real-Time Maude's *timed rewrite* command simulates *one* of the many possible system behaviors from the initial state by rewriting the initial state up to a certain duration. The *search* command

```
(utsearch [[n]] t =>* pattern [such that cond] .)
```

uses a breadth-first strategy to search for (at most n) states that are reachable from the initial state t, match the *search pattern*, and satisfy *cond*. If the arrow `=>!` is used instead of `=>*`, then Real-Time Maude searches for reachable *final* states, that is, states that cannot be further rewritten.

The command `(find latest` t_0 `=>*` *pattern* [`such that` *cond*] `in time <=` *limit* `.)` explores all behaviors from the initial state t_0 and finds the longest time needed to reach the desired state (for the *first* time in a behavior).

Real-Time Maude is also equipped with unbounded and time-bounded *linear temporal logic model checker* which analyzes whether *each* behavior (possible up to some duration) satisfies a linear temporal logic formula, and with a *timed CTL* model checker [15] to analyze *timed* temporal logic properties.

3 A Formal Model of Human Multitasking

This section presents our Real-Time Maude model of human multitasking. We only show parts of the specification, and refer to the full executable specification available at `http://www.di.unipi.it/msvbio/software/HumanMultitasking.html` for more detail.

We model human multitasking in an object-oriented style, where the state consists of a number of `Interface` objects, representing the interfaces of the devices/systems with which the user interacts, and an object of class `WorkingMemory` representing the user's working memory. Each interface object contains a `Task` object defining the task that the user wants to perform on that interface.

3.1 Classes

Interfaces. We model an interface as a transition system. Since we follow a user-centric approach, the state of the interface/system is given by what the human *perceives* it to be. For example, I may perceive that an ATM is ready to accept my debit card by seeing a friendly welcoming message on the ATM display, and I can perceive that the machine is not ready for me by seeing chewing gum in the card slot or an "Out of Order" message on the ATM display.

The human's perception of the state of an interface can be represented as a term of the following sort `InterfaceState`:

```
sorts InterfaceState Perception ExpPerception .
subsort Perception < ExpPerception < InterfaceState .
op _for time_ : Perception TimeInf -> InterfaceState [right id: INF] .
op expired : Perception -> ExpPerception .
```

A perception/state may not last forever: after entering my card in the slot, I will only perceive that the ATM is waiting for my PIN code for 8 minutes, after which the ATM will display a message that the transaction is cancelled. The term p `for time` t denotes that the user will perceive p for time t, after which the perception becomes `expired(`p`)`. The transitions of an interface have the form p_1 `--` *action* `-->` p_2. For example, if I perceive that the machine is ready to receive my card, I can perform an action `enterCardInSlot`, and as a result the ATM will display that I should type my PIN code: `ATMready -- enterCard --> typePIN for time 480`. The set of interface transitions are represented as a `;`-separated set of single interface transitions:

```
sorts InterfaceTransition InterfaceTransitions .
subsort InterfaceTransition < InterfaceTransitions .

--- single transition:
op _--_-->_ : Perception DefAction InterfaceState -> InterfaceTransition .
--- sets of transitions:
op noTransition : -> InterfaceTransitions .
op _;_ : InterfaceTransitions InterfaceTransitions ->
                   InterfaceTransitions [assoc comm id: noTransition] .
```

An interface is represented as an object instance of the following class:

```
class Interface | task : Object,      transitions : InterfaceTransitions,
                 previousAction : DefAction, currentState : InterfaceState .
```

where the attribute `transitions` denotes the transitions of the interface; `task` denotes the task object (see below) representing the task that the user wants to perform with the interface; `previousAction` is the previous action performed on the interface (useful for analysis purposes); and `currentState` is (the user's perception of) the state of the device.

Tasks. Instead of seeing a task as a sequence of basic tasks that cannot be further decomposed, we find it more natural to consider a task to be a sequence of subtasks, where each subtask is a sequence of basic tasks. For example, the task of withdrawing money at an ATM may consist of the following sequence of subtasks: insert card; type PIN code; type amount; retrieve card; and, finally, retrieve cash. Some of these subtasks consist of a sequence of basic tasks: the subtask "type PIN code" consists of typing 4 digits and then "OK," and so does the subtask "type amount." We therefore model a task as a '`::`'-separated sequence of subtasks, where each subtask is modeled as a sequence of basic tasks:

```
sorts BasicTask Subtask Task .   subsort BasicTask < Subtask < Task .

--- Subtask is a list of BasicTasks:
op nil : -> Subtask .
op __ : Subtask Subtask -> Subtask [assoc id: nil] .

--- Task is a list of subTasks:
op emptyTask : -> Task .
op _::_ : Task Task -> Task [assoc id: emptyTask] .
```

In a basic task inf_1 `|` p_1 `==>` $action$ `|` inf_2 `duration` τ `difficulty` d `delay` δ, inf_1 is some knowledge, p_1 is a perception (state) of the interface, τ is the time needed to execute the task, and d is a measure of the *difficulty* of the basic task. If the working memory contains inf_1 and I perceive p_1, then I can perform the interface transition labeled *action*, and as a result my working memory forgets inf_1 and stores inf_2. A basic task may not be enabled immediately: you cannot type your PIN code immediately after inserting your card. The ATM first reads your card and does some other processing. The (minimum) time needed before the basic task can be executed is given by the delay δ. This delay could also be the time needed to switch from one task to another. A basic task could be

```
needCash | ATMready ==> enterCard | cardInMachine
   duration 3 difficulty 1/8 delay 0.
```

That is, after performing the action `enterCard` you "forget" that you need cash, and instead store in working memory that the card is in the machine.

Basic tasks are therefore declared

```
op _|_==>_|_duration_difficulty_delay_ : Information Perception DefAction
               Information Time PosRat Time -> BasicTask .
```

As mentioned in Section 2.1, the next task that is given a person's attention is a function of: the cognitive loads of the current subtasks[4], the *criticality level*

---

[4] Since we now consider *structured* tasks and add delays to basic tasks, we redefine the cognitive load of a task to be $\sum \frac{d_i t_i}{t_i + dly_i}$, where d_i, t_i and dly_i denote the difficulty, duration and delay of each basic task i of the *current subtask*. The cognitive load of a task therefore changes every time a new subtask begins, and remains the same throughout the execution of the subtask.

of each task (we focus on safety-critical systems, and the user tends to focus more frequently on safety-critical tasks than on other tasks), and the time that an enabled task has *waited* to be executed. For example, driving a car has a higher criticality level than finding out where to go, which has higher criticality level than finding a good radio station. To compute at each step the "rank" of each task, a task object should contain these values, and is therefore represented as an object instance of the following class `Task`:

```
class Task | subtask : Task,        waitTime : Time,    status : TaskStatus,
             cognitiveLoad : Rat,   criticalityLevel : PosRat .
```

The `subtasks` attribute denotes the *remaining* sequence of subtasks to be performed; `waitTime` denotes how long the next basic task has been enabled; `cognitiveLoad` is the cognitive load of the subtask currently executing; and `criticalityLevel` is the task's criticality level. For analysis purposes, we also add an attribute `status` denoting the "status" of the task, which is either `notStarted`, `ongoing`, `abandoned`, or `completed`.

Working Memory. The working memory is used when interacting with the interfaces, and can only store a limited number of information items. We model the working memory as an object of the following class:

```
class WorkingMemory | memory : Memory,  capacity : NzNat .
```

where `capacity` denotes the maximal number of elements that can be stored in memory at any time. The attribute `memory` stores the content of the working memory as a map I_1 `|->` mem_1 `; ... ;` I_n `|->` mem_n, assigning to each interface I_j the *set* mem_j of items in the memory associated to interface I_j. An element in mem_j is either a *cognition*, a basic piece of *information*, such as `cardInMachine`, or a desired *goal* `goal(`*action*`)`. The goal defines the goal of the interaction with the interface, which is to end up performing some final action, such as `takeCash`. Cognitions are more of a mental state (want to withdraw money, or do want to do so?), and can change without interacting with an interface, whereas basic information cannot. The data type `Memory` specifying this map is defined as follows:

```
sorts Goal Information .  subsorts Cognition Goal BasicInfo < Information .

op goal : Action -> Goal .

sort InfoSet .   subsort Information < InfoSet .
op noInfo : -> Information .
op __ : InfoSet InfoSet -> InfoSet [assoc comm id: noInfo] .

sort Memory .
op noMemory : -> Memory .
op _|->_ : InterfaceId InfoSet -> Memory .
op _;_ : Memory Memory -> Memory [assoc comm id: noMemory] .
```

For example, the working memory of a person p who wants to drive to X and likes to listen to NPR could be:

```
< p : WorkingMemory | capacity : 7,
                      memory : car |-> goal(parkAtX) ;
                               gps |-> XlivesInaddr goal(pushFindWay) ;
                               radio |-> NPRIsButton3 goal(pushButton(3)) >
```

3.2 Dynamic Behavior

We formalize human multitasking with rewrite rules that specify how attention is directed at the different tasks, and how this affects the working memory. In short, whenever a basic task is enabled, attention is directed toward the task/interface with the highest *task rank*, and a basic task/action is performed on that interface.

The rank of each task is a function of:

- the cognitive load of the "current" subtask, which is a function of the durations and difficulty levels the basic tasks in the subtask;
- the criticality level of the task; and
- the time that the task has been waiting (i.e., enabled being executed).

This rank of a task is defined as follows[5]:

```
op rank : NEConfiguration Memory -> PosRat .
eq rank(< I : Interface | task :
          < TASK : Task | subtasks : ((INF1 | P1 ==> DACT | INF2 duration
                                        NZT difficulty PR delay T2)  BTL)
                                      :: OTHER-SUB-TASKS,
                          waitTime : T, cognitiveLoad : CL,
                          criticalityLevel : PR2 > >,
        (I |-> goal(ACT) INF-SET) ; MEMORY)
 = if T2 == 0 then PR2 * CL * (T + 1) else 0 fi .

eq rank(< I : Interface |  >,  MEMORY) = 0 [owise] .
```

As the definition shows, a task without a goal, or one which is not yet enabled (the remaining delay `T2` of the first basic task is greater than 0) is 0. This `rank` function refines the task rank function in [5], and should therefore be consistent with results in psychology.

The following tick rewrite rule models the user performing a basic task (if it does not cause memory overload, and the action performed is not the goal action) with the interface with the highest rank of all interfaces (`bestRank(...)`):

```
crl [interacting] :
    {OTHER-INTERFACES
     < I : Interface | task :
```

---

[5] We do not show the variable declarations, but follow the convention that variables are written in all capital letters.

```
     < TASK : Task | subtasks : ((INF1 | P1 ==> DACT | INF2 duration NZT
                                   difficulty PR delay 0) BASIC-TASKS)
                                 :: OTHER-SUB-TASKS,
                waitTime : T1,           cognitiveLoad : CL,
                criticalityLevel : PR2,   status : TS >,
            transitions : (P1 -- DACT --> (P2 for time TI2)) ; TRANSES,
            currentState : (P1 for time TI), previousAction : DACT2 >
   < WM : WorkingMemory | memory : MEMORY ; (I |-> INF1  goal(ACT) INF-SET),
                         capacity : CAP >}
=>
  {idle(OTHER-INTERFACES, NZT)
   < I : Interface | task :
     < TASK : Task | subtasks : (if BASIC-TASKS =/= nil
                                  then (BASIC-TASKS :: OTHER-SUB-TASKS)
                                  else OTHER-SUB-TASKS fi),
                waitTime : 0,
                status : (if TS == notStarted then ongoing else TS fi),
                cognitiveLoad : (if BASIC-TASKS =/= nil then CL else
                                     cogLoad(first(OTHER-SUB-TASKS)) fi) >,
            currentState : (P2 for time TI2),  previousAction : DACT >
   < WM : WorkingMemory | memory : MEMORY ; (I |-> INF2 goal(ACT) INF-SET) >}
 in time NZT
   if assess(DACT2,P1) =/= danger /\ (DACT =/= ACT)
      /\ card(MEMORY ; (I |-> INF2 goal(ACT) INF-SET)) <= CAP
      /\ rank(< I : Interface | >,
              (MEMORY ; (I |-> INF1 goal(ACT) INF-SET)))
         == bestRank(< I : Interface | >  OTHER-INTERFACES,
                     (MEMORY ; (I |-> INF1  goal(ACT) INF-SET))) .
```

The user perceives that the state of interface `I` is `P1`. The next basic task can be performed if information `INF1` is associated with this interface in the user's working memory, and the interface is (perceived to be) in state `P1`. The user then performs the basic task labeled `DACT`, which leads to a new item `INF2` stored in working memory, while `INF1` is forgotten. This rule is only enabled if the remaining delay of the basic task is 0 and the user has a goal associated with this interface. If the basic task performed is the last basic task in the subtask, we must also recompute the value of `cognitiveLoad` to be the cognitive load of the next subtask.

The first conjuncts in the condition say that the rule can only be applied when the user does not assess a danger in the current situation and when the action performed is not the goal action. Since `INF1` and/or `INF2` could be the empty element `noInfo`, the rule may increase the number of items stored in working memory (when `INF1` is `noInfo`, but `INF2` is not). The third conjunct in the condition ensures that the resulting knowledge does not exceed the capacity of the working memory. The last conjunct ensures that the current interface should be given attention: it has the highest rank among all the interfaces.

The rule is a tick rule; its duration is the duration `NZT` of the basic task being executed. During that time, every other task idles: its “perception timer” and remaining delay of the first basic task are decreased according to elapsed time, and its waiting time is increased if the basic task is enabled:

```
op idle : Configuration Time -> Configuration [frozen (1)] .

eq idle(none, T) = none .
eq idle(< I : Interface | task :
          < TASK : Task | subtasks : ((INF1 | P1 ==> DACT | INF2 duration NZT
                                       difficulty PR delay T2) BASIC-TASKS)
                                     :: OTHER-SUB-TASKS,
                          waitTime : T3 >,
            currentState : IS > REST, T)
 = < I : Interface | task :
     < TASK : Task | subtasks : ((INF1 | P1 ==> DACT | INF2 duration NZT
                           difficulty PR delay (T2 monus T)) BASIC-TASKS)
                            :: OTHER-SUB-TASKS,
                      waitTime : T3 + (T monus T2) >,
            currentState : idle(IS, T) > idle(REST, T) .

eq idle(< I : Interface | task : < TASK : Task | subtasks : emptyTask,
                                                 waitTime : T3 >,
                          currentState : IS > REST, T)
 = < I : Interface | task : < TASK : Task | waitTime : 0 >,
                     currentState : idle(IS, T) > idle(REST, T) .

eq idle(< WM : WorkingMemory | >   REST, T)
 =  < WM : WorkingMemory | >   idle(REST, T) .

op idle : InterfaceState TimeInf -> InterfaceState .
eq idle(P1 for time TI, T)
 = if T < TI then P1 for time (TI monus T) else expired(P1) fi .
eq idle(expired(P1), T) = expired(P1) .
```

If performing the basic task would exceed the capacity of the memory, some other item in the memory is nondeterministically forgotten, so that items associated to the current interface are only forgotten if there are no items associated to other interfaces. (This is because maintaining information in working memory requires the user's attention, and user attention is on the current task, so it is more natural that items of the other tasks are forgotten first.) The following rule shows the case when an item for a different interface is erased from memory. Since a mapping is associative and commutative, *any* memory item `INF3` associated with *any* interface `I2` different from `I` could be forgotten. This rule is very similar to the rule above, and we only show the differences:

```
crl [interactingForgetSomethingOtherInterface] :
    { ... < I : Interface | task : < TASK : Task |  ... >  ... >
     < WM : WorkingMemory | memory : (I |-> INF1 goal(ACT) INF-SET) ;
                                     (I2 |-> INF3 INF-SET2) ; MEMORY,
                            capacity : CAP >}
```

```
  =>
    { ... < I : Interface | task : < TASK : Task | ... > ... >
     < WM : WorkingMemory | memory : (I |-> INF2 goal(ACT) INF-SET) ;
                                     (I2 |-> INF-SET2) ; MEMORY >}
    in time NZT
     if  ... /\ card((I |-> INF2 goal(ACT) INF-SET)
                    ; (I2 |-> INF3 INF-SET2) ; MEMORY) > CAP /\ ...
```

A third very similar rule removes an arbitrary item from the working memory associated with the current interface if the memory does not store any item for any other interface (see our online executable specification for details).

If each "next" basic task has a remaining delay, then time advances until the earliest time when the delay of some basic task reaches 0:

```
crl [tickAllIdling] :
    {ALL-INTERFACES
     < WM : WorkingMemory | memory : MEMORY ; (I |-> goal(ACT) INF-SET) >}
   =>
    {idle(ALL-INTERFACES, MIN-DELAY)
     < WM : WorkingMemory | >} in time MIN-DELAY
    if MIN-DELAY := minDelay(ALL-INTERFACES) .
```

where `MIN-DELAY` is a variable of a sort `NzTime` of non-zero time values.

In the above rules, we did not reach our goal with the interface. The following rule treats the case then the action `ACT` performed is our goal action. Again, this rule is quite similar to the above rules, so some parts are replaced by '. . .':

```
crl [closure] :
    {OTHER-INTERFACES
     < I : Interface | task :
       < TASK : Task | subtasks : ((INF1 | P1 ==> ACT | INF2 duration NZT
                                    difficulty PR delay 0) BASIC-TASKS)
                                  :: OTHER-SUB-TASKS >,
           transitions : (P1 -- ACT --> (P2 for time TI2)) ; TRANSES,
           currentState : (P1 for time TI) >
     < WM : WorkingMemory | memory : MEMORY ; (I |-> INF-SET INF1 goal(ACT)) >}
  =>
    {idle(OTHER-INTERFACES, NZT)
     < I : Interface | task :
       < TASK : Task | subtasks : emptyTask,  waitTime : 0,
                       cognitiveLoad : 0, status : completed >, ... >
     < WM : WorkingMemory | memory :  MEMORY ; (I |-> INF2) >}
    in time NZT  if  ...
```

The rules `interacting` could only be applied when the user's assessment of her current situation was different from `danger`; if (s)he assesses that (s)he is in danger, (s)he abandons the current task for the interface. We refer to the executable specification for the description of this rule.

As mentioned, a person may change cognition ("mental state") without interacting with a device, or may acquire knowledge through a cognitive process:

```
crl [cognitive] :
    {OTHER-INTERFACES
     < I : Interface | task :
       < TASK : Task | subtasks : ((COG1 | P1 ==> DACT | COG2 duration NZT
                                    difficulty PR delay 0)
                                   BASIC-TASKS) :: OTHER-SUB-TASKS,
                       cognitiveLoad : CL,  status : TS > >
     < WM : WorkingMemory | memory : MEMORY ; (I |-> INF-SET COG1 goal(ACT)),
                            capacity : CAP >}
  =>
    {idle(OTHER-INTERFACES, NZT)
     < I : Interface | task : < TASK : Task |  ... > >
     < WM : WorkingMemory |
           memory : MEMORY ; (I |-> INF-SET COG2 goal(ACT, GT)) >}
   in time NZT   if ....
```

Our last rule concerns only the interface. As mentioned, sometimes the interface state comes with a timer (e.g., the ATM only waits for a PIN code for eight minutes). When this timer expires, an instantaneous rule changes the interface state (e.g., display “Ready” when the machine has waited too long for the PIN):

```
rl [timeout] :
   {REST
    < I : Interface | transitions : (expired(P1) -- DACT --> IS) ; TRANSES,
                      currentState : expired(P1) >}
   =>
   {REST < I : Interface | currentState : IS, previousAction : DACT >} .
```

4 Analyzing Safety-Critical Human Multitasking

This section explains how Real-Time Maude can be used to analyze whether a human is able to perform a given set of tasks successfully. In particular, we focus on the following potential problems that could happen when multitasking:

1. A critical task may be ignored for too long because attention is given to other tasks. For example, it is not good if a driver does not give attention to driving for 15 seconds because (s)he is focusing on the infotainment system.
2. A task, or a crucial action in a task, is not completed on time, since too much attention has been given to other tasks. For example, a pilot should finish all pre-flight tasks before taking off, and a driver should have entered the destination in the GPS before the first major intersection is reached.
3. Other tasks' concurrent use of working memory may cause the user to forget/misremember memory items that are crucial to complete a given task.

The initial state should have the form

```
{initializeCognLoad(
 < wm : WorkingMemory | memory : interface1 |-> goal(action1) otherItems1 ; ... ;
                                 interfacen |-> goal(actionn) otherItemsn,
```

```
                         capacity : capacity >
 < interface_1 : Interface | task :
   < task_1 : Task | subtasks : (b_{1_1} ... b_{1_l}) :: ... :: (b_{1_{m_1}} ... b_{1_{m_j}}),
                    waitTime : 0, cognitiveLoad : 0, criticalityLevel : cl_1,
                    status : notStarted >
      transitions : trans_1, previousAction : noAction, currentState : perc_1 >
 ...
 < interface_n : Interface | task :
   < task_n : Task | subtasks : ..., waitTime : 0, cognitiveLoad : 0,
                    criticalityLevel : cl_n, status : notStarted >
      transitions : trans_n, previousAction : noAction, currentState : perc_n >)}
```

where: $interface_k$ is the name of the k-th interface; $task_k$ is the task to be performed with/on $interface_k$; $b_{k_{i_j}}$ is the j-th basic task of the i-th subtask of $task_k$; cl_k is the criticality level of $task_k$; $trans_k$ are the transitions of $interface_k$; $action_k$ is the goal action to be achieved with $interface_k$; $otherItems_k$ are other items initially in the memory for $interface_k$; $perc_k$ is the initial perception ("state") of $interface_k$; and *capacity* is the number of items that can be stored in working memory. The function `initializeCognLoad` initializes the `cognitiveLoad` attributes by computing the cognitive load of the first subtask of each task.

The first key property to analyze is: Is it possible that an (enabled) task t is ignored continuously for at least time Δ? This property can be analyzed in Real-Time Maude as follows, by checking whether it is possible to reach a "bad" state where the `waitTime` attribute of task t is at least Δ:[6]

```
(utsearch [1] initialState =>*
  {REST:Configuration  < I:InterfaceId : Interface | task :
                         < t : Task | waitTime : T:Time, A:AttributeSet > >}
  such that T:Time >= Δ .
```

where the variable `REST:Configuration` matches the other objects in the state.

The second key property is checking whether a certain task t is guaranteed to finish before time T. This can be analyzed using Real-Time Maude's `find latest` command, by finding the longest time needed to reach status `completed`:

```
(find latest initialState =>*
  {REST:Configuration  < I:InterfaceId : Interface | task :
                         < t : Task | status : completed, A:AttributeSet > >}
  with no time limit .)
```

We can also use the `find latest` command to find out the longest time needed for a task t to complete the specific action *act*:

```
(find latest initialState =>*
  {REST:Configuration  < I:InterfaceId : Interface | previousAction : act >}
  with no time limit .)
```

We can analyze whether it is guaranteed that a task t will be completed by searching for a "bad" *final* state where the status of the task is not `completed`:

---

[6] The variable `A:AttributeSet` captures the other attributes in *inner* objects.

```
(utsearch [1] initialState =>!
  {REST:Configuration  < I:InterfaceId : Interface | task :
                     < t : Task | status : TS:TaskStatus, A:AttributeSet > >}
  such that TS:TaskStatus =/= completed .)
```

If we want to analyze whether it is guaranteed that *all* tasks can be completed, we just replace *t* in this command with a variable `I2:TaskId`.

If a safety-critical task cannot be completed, or completed in time, we can check whether this is due to the task itself, or the presence of concurrent "distractor" tasks, by analyzing an initial state *without* the distractor tasks.

5 Example: Interacting with a GPS Device while Driving

This section illustrates the use of our modeling and analysis framework with an example of a person who interacts with a GPS navigation device while driving.

We have two interfaces: the car and the navigation system. The task of driving consists of the three subtasks (i) start driving, (ii) drive to destination, and (iii) park and leave the car. The first subtask consists of the basic tasks of inserting the car key, turning on the ignition, and start driving; subtask (ii) describes a short trip during which the driver wants to perform a basic driving action at most every three time units; and subtask (iii) consists of stopping the car and remove the key when we have arrived at the destination. The driving task can be formalized by the following `Task` object:

```
< driving : Task | subtasks :
((noInfo | carOff ==> insertKey | keyInserted duration 1 difficulty 3/10 delay 0)
 (noInfo | carOn ==>  turnKey | noInfo duration 1 difficulty 2/10 delay 0)
 (noInfo | carReady ==> startDrive | noInfo duration 1 difficulty 2/10 delay 2)) ::
((noInfo | straightRoad ==> straight | noInfo duration 1 difficulty 1/10 delay 3)
 (noInfo | straightRoad2 ==> straight | noInfo duration 1 difficulty 1/10 delay 3)
 (noInfo | curveLeft ==> turnLeft | noInfo duration 1 difficulty 4/10 delay 3)
 (noInfo | curveRight ==> turnRight | noInfo duration 1 difficulty 2/10 delay 3)
 (noInfo | straightRoad3 ==> straight | noInfo duration 1 difficulty 1/10 delay 3)
 (noInfo | straightRoad4 ==> straight | noInfo duration 1 difficulty 1/10 delay 3)) ::
((noInfo | destination ==> stopCar | noInfo duration 2 difficulty 2/10 delay 2)
 (keyInserted | carStopped ==> pickKey | noInfo duration 2 difficulty 1/10 delay 0)),
waitTime : 0, status : notStarted, criticalityLevel : 6/10, cognitiveLoad : 0 >
```

The interface of the car is formalized by the following `Interface` object:

```
< car : Interface | transitions :
  (carOff -- insertKey --> carOn) ; (carReady -- startDrive --> straightRoad) ;
  (carOn -- turnKey --> carReady) ; (straightRoad -- straight --> straightRoad2) ;
  (straightRoad2 -- straight --> curveLeft) ; (curveLeft -- turnLeft --> curveRight) ;
  (curveRight -- turnRight --> straightRoad3) ;
  (straightRoad3 -- straight --> straightRoad4) ;
  (straightRoad4 -- straight --> destination) ; (destination -- stopCar --> carStopped) ;
  (carStopped -- pickKey --> carOff) ; (carReady -- noAction --> carOff),
 task : ... , previousAction : noAction, currentState : carOff >
```

For the GPS navigator, we assume that to enter the destination the user has to type at least partially the address. The navigator then suggests a list of possible destinations, among which the user has to select the right one. Therefore, the GPS task consists of three subtasks: (i) start and choose city; (ii) type the initial k letters of the desired destination; and (iii) choose the right destination among the options given by the GPS.

If the user types the entire address of the destination, the navigator returns a short list of possible matches; if (s)he types fewer characters, the navigator returns a longer list, making it harder for the user to find the right destination. We consider two alternatives: (1) the driver types 13 characters and then searches for the destination in a short list; and (2) the driver types just four characters and then searches for the destination in a longer list. The GPS task for case (1) is modeled by the following `Task` object:

```
< findDestination : Task | subtasks :
((noInfo | gpsReady ==> typeSearchMode | noInfo duration 1 difficulty 1/10 delay 0)) ::
((noInfo | chooseCity ==> selectCity | noInfo duration 2 difficulty 5/10 delay 2)) ::
((noInfo | typing1 ==> typeSomething | noInfo duration 1 difficulty 3/10 delay 3)
 (noInfo | typing2 ==> typeSomething | noInfo duration 1 difficulty 3/10 delay 0)
 ...
 (noInfo | typing13 ==> pushSearchBtn | noInfo duration 1 difficulty 3/10 delay 0)) ::
((noInfo | searching ==> chooseAddress | noInfo duration 2 difficulty 2/10 delay 0)),
waitTime : 0, status : notStarted, criticalityLevel : 3/10, cognitiveLoad : 0 >
```

Case (2) is modeled similarly, but with only four typing actions before pushing the search button. In that case, the last basic task (choosing destination from a larger list) has duration 5 and difficulty $\frac{6}{10}$.

The GPS interface in case (1) is defined by the following `Interface` object:

```
< gps : Interface | transitions :
  (gpsReady -- typeSearchMode --> chooseCity) ; (chooseCity -- selectCity --> typing1) ;
  (typing1 -- typeSomething --> typing2)  ; (typing2 -- typeSomething --> typing3)  ;
  ...
  (typing13 -- pushSearchBtn --> searching) ; (searching -- chooseAddress --> gpsReady),
 task : ... , previousAction : noAction, currentState : gpsReady >
```

The initial state of the working memory is

```
< wm : WorkingMemory | capacity : 5, memory : (car |-> goal(pickKey)) ;
                                             (gps |-> goal(chooseAddress)) >
```

We use the techniques in Section 4 to analyze our models, and first analyze whether an enabled driving task can be ignored for more than six seconds:

```
Maude> (utsearch [1] {initState} =>* {< car : Interface | task :
              < driving : Task | waitTime : T:Time, A:AttributeSet > >
              REST:Configuration} such that T:Time > 6 .)
```

Real-Time Maude finds no such bad state when the driver types 13 characters. However, when the driver only types four characters, the command returns a bad state: the driver types the last two characters and finds the destination in the long list without turning his attention to driving in-between.

Sometimes even a brief distraction can be dangerous. For example, when the road turns, a delay of three time units in making the turn could be dangerous. We check the longest time needed for driver to complete the `turnLeft` action:

```
Maude> (find latest {initState} =>*
          {REST:Configuration < car : Interface | previousAction : turnLeft >}
         with no time limit .)
```

Real-Time Maude shows that the left turn is completed at time 21. However, the same analysis with an initial state *without* the GPS interface object and task shows that an undistracted driver finishes the left turn at time 17.

Finally, to analyze potential memory overload, we modify the GPS task so that the driver must remember the portion of address already written: a new item is added to the working memory after each three characters typed.

We then check whether all tasks are guaranteed to be completed in this setting, by searching for a *final* state in which some task is not completed:

```
Maude> (utsearch [1] {initState2} =>! {< I:InterfaceId : Interface | task :
           < T:TaskId : Task | status : TS:TaskStatus, A:AttributeSet > >}
           REST:Configuration} such that TS:TaskStatus =/= completed .)
```

This command finds such an undesired state: `keyInserted` could be forgotten when the driver must remember typing; in that case, the goal action `pickKey` cannot be performed, and we leave the key in the car. The same command with our "standard" model of GPS interaction does not return any final state with an uncompleted task pending.

6 Related work

There has been some work on applying "computational models" to study human attention and multitasking. In [23] the ACT-R architecture, an executable rule-based framework for modeling cognitive processes, is applied to study the effects of distraction by phone dialing while driving. However, when the study was undertaken, ACT-R did not include a model of human attention. To let the two tasks interleave, the authors had to explicitly define in each task when the attention has to move to the other task. In [6] the ACT-R architecture was used to study sources of aviation errors, again without taking multitasking properly into account. The authors of [23] later developed a theory of concurrent multitasking [24] and consequently extended the ACT-R architecture. The new theory describes concurrent tasks that can interleave and compete for resources. Cognition balances task execution by favoring least recently processed tasks [13].

Other computational models for human multitasking include the *salience, expectancy, effort and value (SEEV)* model [27] and the *strategic task overload management (STOM)* model [28, 26]. Both have been validated against data collected by performing experiments with real users using simulators. Although dealing with human multitasking, the SEEV and STOM models are specifically designed to describe (sequential) visual scanning of an instrument panel, where each instrument may serve different tasks. The multitasking paradigms underlying SEEV and STOM are different from the one we consider in this paper, which is not *sequential scanning* but *voluntary task switching* [1].

The above systems (and other similar approaches) have all been developed in the context of cognitive psychology and neuroscience research. They do not provide what computer scientists would call a formal model, but are typically based on some mathematical formulas and an implementation (in Lisp in the case of ACT-R) that supports only simulation. In contrast, we provide a formal model that can be not only simulated, but also subjected to a range of formal analyses, including reachability analysis and timed temporal logic model checking.

On the formal methods side, we discuss the differences with the formal cognitive framework proposed in [7] in the introduction.

Finally, as mentioned in Section 2.1, in [5] we propose a task switching algorithm for non-structured tasks that we extend in the current paper. That work does not provide a formal model, but is used to demonstrate the agreement of our modeling approach with relevant psychological literature.

7 Concluding Remarks

In this paper we have presented for the first time a formal executable framework for safety-critical human multitasking. The framework enables the simulation and model checking in Real-Time Maude of a person concurrently interacting with multiple devices of different degrees of safety-criticality. Task switching is modeled trough a task ranking procedure which is consistent with studies in psychology. We have shown how Real-Time Maude can be used to automatically analyze prototypical properties in safety-critical human multitasking, and have illustrated our framework with a simple example.

As part of future work, we will in the near future perform experiments in collaboration with psychologists to refine our model. We should also apply our framework on real safety-critical case studies.

References

1. Arrington, C.M., Logan, G.D.: Voluntary task switching: chasing the elusive homunculus. Journal of Experimental Psychology: Learning, Memory, and Cognition 31(4), 683–702 (2005)
2. Atkinson, R.C., Shiffrin, R.M.: Human memory: A proposed system and its control processes. In: Spence, K.W., Spence, J.T. (eds.) Psychology of Learning and Motivation, vol. 2, pp. 89–195. Academic Press (1968)
3. Australian Transport Safety Bureau: Dangerous distraction. Safety Investigation Report B2004/0324 (2005)

4. Barrouillet, P., Bernardin, S., Camos, V.: Time constraints and resource sharing in adults' working memory spans. Journal of Experimental Psychology: General 133(1), 83–100 (2004)
5. Broccia, G., Milazzo, P., Ölveczky, P.C.: An algorithm for simulating human selective attention. In: DataMod 2017. LNCS (2017), in press
6. Byrne, M.D., Kirlik, A.: Using computational cognitive modeling to diagnose possible sources of aviation error. The International Journal of Aviation Psychology 15(2), 135–155 (2005)
7. Cerone, A.: A cognitive framework based on rewriting logic for the analysis of interactive systems. In: SEFM 2016. LNCS, vol. 9763, pp. 287–303. Springer (2016)
8. Clark, T., David, Y., Baretich, M., Bauld, T., Dickey, D., Gieras, I., et al.: Impact of clinical alarms on patient safety. Tech. rep., ACCE Healthcare Technology Foundation (2006)
9. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude, LNCS, vol. 4350. Springer (2007)
10. Dingus, T.A., Guo, F., Lee, S., Antin, J.F., Perez, M., Buchanan-King, M., Hankey, J.: Driver crash risk factors and prevalence evaluation using naturalistic driving data. Proceedings of the National Academy of Sciences 113(10), 2636–2641 (2016)
11. Dismukes, R., Nowinski, J.: Prospective memory, concurrent task management, and pilot error. In: Attention: From Theory to Practice, pp. 225–236. Oxford University Press Great Britain (2007)
12. de Fockert, J.W., Rees, G., Frith, C.D., Lavie, N.: The role of working memory in visual selective attention. Science 291(5509), 1803–1806 (2001)
13. Kushleyeva, Y., Salvucci, D.D., Lee, F.J.: Deciding when to switch tasks in time-critical multitasking. Cognitive Systems Research 6(1), 41–49 (2005)
14. Lavie, N., Hirst, A., De Fockert, J.W., Viding, E.: Load theory of selective attention and cognitive control. Journal of Experimental Psychology: General 133(3), 339–354 (2004)
15. Lepri, D., Ábrahám, E., Ölveczky, P.C.: Sound and complete timed CTL model checking of timed Kripke structures and real-time rewrite theories. Science of Computer Programming 99, 128–192 (2015)
16. Lofsky, A.S.: Turn your alarms on! APSF Newsletter: The Official Journal of the Anesthesia Patient Safety Foundation 19(4), 43 (2005)
17. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: Proc. WADT'97, LNCS, vol. 1376, pp. 18–61. Springer (1998)
18. Miller, G.A.: The magical number seven, plus or minus two: some limits on our capacity for processing information. Psychological Review 63(2), 81–97 (1956)
19. Miller, G.A., Galanter, E., Pribram, K.H.: Plans and the Structure of Behavior. Adams Bannister Cox (1986)
20. Ölveczky, P.C.: Real-Time Maude and its applications. In: WRLA 2014. LNCS, vol. 8663. Springer (2014)
21. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-Order and Symbolic Computation 20(1-2), 161–196 (2007)
22. Pashler, H.: Dual-task interference in simple tasks: data and theory. Psychological Bulletin 116(2), 220–244 (1994)
23. Salvucci, D.D.: Predicting the effects of in-car interface use on driver performance: An integrated model approach. International Journal of Human-Computer Studies 55(1), 85–107 (2001)
24. Salvucci, D.D., Taatgen, N.A.: Threaded cognition: an integrated theory of concurrent multitasking. Psychological Review 115(1), 101–130 (2008)

25. Wickens, C.D.: Processing resources and attention. In: Multiple-task Performance, pp. 3–34. Taylor & Francis (1991)
26. Wickens, C.D., Gutzwiller, R.S.: The status of the strategic task overload model (STOM) for predicting multi-task management. In: Proceedings of the Human Factors and Ergonomics Society Annual Meeting. vol. 61, pp. 757–761. SAGE Publications (2017)
27. Wickens, C.D., Sebok, A., Li, H., Sarter, N., Gacy, A.M.: Using modeling and simulation to predict operator performance and automation-induced complacency with robotic automation: a case study and empirical validation. Human factors 57(6), 959–975 (2015)
28. Wickens, C.D., Gutzwiller, R.S., Vieane, A., Clegg, B.A., Sebok, A., Janes, J.: Time sharing between robotics and process control: Validating a model of attention switching. Human factors 58(2), 322–343 (2016)
29. Young, K.L., Salmon, P.M.: Examining the relationship between driver distraction and driving errors: A discussion of theory, studies and methods. Safety science 50(2), 165–174 (2012)