Innovations in Systems and Software Engineering (2019) 15:169–190
https://doi.org/10.1007/s11334-019-00333-7

S.I. : NFM2018

Formal modeling and analysis of safety-critical human multitasking

Giovanna Broccia[1] · Paolo Milazzo[1] · Peter Csaba Ölveczky[2]

Received: 1 October 2018 / Accepted: 20 March 2019 / Published online: 8 April 2019

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, such as in an airplane, a spacecraft, or a car, failure to devote sufficient attention to the different tasks could have serious consequences. For example, using a GPS with high cognitive load while driving might take the attention away for too long from the safety-critical task of driving the car. 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: (i) the interaction with a GPS navigation system while driving, (ii) some typical scenarios involving human errors in air traffic control (ATC), and (iii) a medical operator setting multiple infusion pumps simultaneously. We 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, and that working memory overload and distraction may cause an air traffic controller or a medical operator to make critical mistakes.

Keywords Human–computer interaction · Safety-critical systems · Human multitasking · Cognitive models · Real-Time Maude · Model checking

1 Introduction

We often interact with multiple devices or computer systems at the same time. It is now well known that the human brain cannot do many things at once, which means that *human multitasking* amounts to repeatedly shifting 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 takes place 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 [23]. 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. A driver often interacts with the GPS navigation system and/or the infotainment system while driving. Finally, astronauts have to navigate multiple (possibly safety-critical) tasks all the time; e.g., in a docking scenario they need to control speed via RCS rockets while estimating the distance to the docket port, all while dealing with weightlessness and possibly communicating in a foreign language.

This work has been supported by the project "Metodologie informatiche avanzate per l'analisi di dati biomedici" funded by the University of Pisa (PRA 2017_44).

✉ Paolo Milazzo
milazzo@di.unipi.it

Giovanna Broccia
giovanna.broccia@di.unipi.it

Peter Csaba Ölveczky
peterol@ifi.uio.no

[1] Department of Computer Science, University of Pisa, Pisa, Italy

[2] Department of Informatics, University of Oslo, Oslo, Norway

Human multitasking could lead to memory overload (too much information to process/remember), resulting in forgetting/mistaking critical tasks. For example, Lofsky [33] 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, Clark et al. [20] analyze the cause of 139 deaths when using an infusion pump and find that operator distraction caused 67 deaths—much more than the 10 deaths caused by problems with the device itself. Similar figures and examples can be found in the context of aviation [6] and car driving [22].

In addition to memory overload, human multitasking may also lead to cognitive overload when some tasks are too cognitively demanding, which could 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. Likewise, 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 the concurrent tasks. The *cognitive load* of a task is a measure of its complexity in terms of frequency and difficulty of the memory operations it requires to perform [8], and is a crucial parameter when deciding which task gets attention.

In this paper, we propose a formal executable model of human multitasking in safety-critical contexts. The model is specified in Real-Time Maude [38] and is a significant modification and extension of the cognitive framework proposed by Cerone for the analysis of interactive systems [18]. 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 Cerone [18], our framework also captures the limitations of a human's working memory (to enable reasoning about hazards caused by memory and/or cognitive overload) and includes timing features (to analyze, e.g., whether a critical task is ignored for too long).

This paper is a revised and extended version of our conference paper [14], and extends [14] as follows:

- Much more detail about the formal model of human multitasking is provided, including sort definitions and many function definitions.
- We show how to formalize a number of additional features of human multitasking, including *cognitive closure*, *cognitive change*, and the handling of *interface timeouts*. (A cognitive closure arises when a goal has been reached (e.g., the user has received the money from the ATM); a cognitive change corresponds to a change in the mental plan of the user resulting from acquiring knowledge and understanding (e.g., when an air traffic controller understands that two aircraft are getting too close to each other, she adds to her memory a new plan: avoid a collision); and an interface timeout arises when some action can no longer be performed (e.g., when the user has not entered a PIN code into the ATM for too long, she notices a "Session closed" message and abandons the task of withdrawing money from the ATM).
- We add a new case study about air traffic control (ATC), which is based on a study of real ATC operator errors described in [44]. This case study illustrates how our framework can be used to uncover human errors in multitasking that are due to working memory failures.
- The comparison to related work is significantly extended.

Furthermore, this paper also includes a revised version of a third case study, about calibrating and setting multiple infusion pumps at the same time, that originally appeared in [13].

After providing some background on human attention and multitasking and Real-Time Maude in Sect. 2, we present our Real-Time Maude model of safety-critical human multitasking in Sect. 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 Sect. 5 with three case studies: (i) using a GPS navigator while driving; (ii) studying some typical scenarios of human errors in air traffic control (ATC); and (iii) a doctor or nurse calibrating and setting multiple infusion pumps at the same time. We apply model checking to show that in some cases: the cognitive load of the navigator interface could cause the driver to keep the focus away from driving for too long, and the working memory sharing between concurrent tasks can lead to overloading situations causing operator errors in ATC and in the hospital. Finally, Sect. 6 discusses related work, and Sect. 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 [5,37]. It can be differentiated into three separate components [5]: a sensory memory, where information detected by the senses is temporarily stored; a short-term memory (STM), 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 term *working memory* (WM) is often used to refer to the short-term memory. However, some neuro-psychological studies show that the two forms of memory are distinct: the STM is only involved in the short-term storage of information, while WM refers to the cognitive system responsible for temporary holding *and* processing of information. The WM is the component most involved in interactions with computers.

Different hypotheses about the WM agree that it can store a limited amount of items, which, furthermore, decay over time, and that it is responsible for both processing and storage activities. The amount of information—which can be, e.g., digits, letters, words, or other meaningful items—that the WM can hold is 7 ± 2 items [36].

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 elaborate decay-theory and most successful in explaining experimental data is the Time-Based Resource Sharing Model [8]. 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. According to Barrouillet et al. [8], 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 duration of the task.

Several studies show that the attentional mechanisms involved in WM refreshment are also the basis of multitasking. In particular, de Fockert et al. [24] describe 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 [15], 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. The greater the task attractiveness, 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 [15], we use this task attractiveness measure to define an algorithm for simulating human attention. We studied the case of two concurrent tasks, and found that the task more likely to complete first is the one with the highest cognitive load, which is consistent with relevant literature (e.g., [8,24]).

2.2 Real-Time Maude

Real-Time Maude [38,40] is a language and tool that extends Maude [21] to support the formal specification and analysis of real-time systems. The specification formalism is based on *real-time rewrite theories* [39]—an extension of *rewriting logic* [16,34]—, emphasizes *ease* and *generality* of specification, and is particularly useful to model distributed real-time systems in an object-based way. Real-Time Maude specifications are executable under reasonable assumptions, and the tool provides a variety of formal analysis methods, including simulation, reachability analysis, and LTL and timed CTL model checking.

2.2.1 Rewriting logic specification in Maude

A *membership equational logic* (MEL) [35] *signature* is a triple $\Sigma = (K, \sigma, S)$ with K a set of *kinds*, $\sigma = \{\Sigma_{w,k}\}_{(w,k)\in K^* \times K}$ a many-kinded signature, and $S = \{S_k\}_{k\in K}$ a K-kinded family of disjoint sets of sorts. The kind of a sort s is denoted by $[s]$. A Σ-*algebra* A consists of a set A_k for each kind k, a function $A_f : A_{k_1} \times \cdots \times A_{k_n} \to A_k$ for each operator $f \in \Sigma_{k_1 \cdots k_n, k}$, and a subset inclusion $A_s \subseteq A_k$ for each sort $s \in S_k$. The set $T_{\Sigma,k}$ denotes the set of ground Σ-terms with kind k, and $T_\Sigma(X)_k$ denotes the set of Σ-terms with kind k over the set X of kinded variables.

A MEL *theory* is a pair (Σ, E) with Σ a MEL-signature and E a finite set of MEL sentences, which are either conditional equations of the form

$$(\forall X)\ t = t' \ \mathbf{if} \bigwedge_i p_i = q_i \wedge \bigwedge_j w_j : s_j$$

where $t, t' \in T_\Sigma(X)_k$ and $p_i, q_i \in T_\Sigma(X)_{k_i}$ for some kinds $k, k_i \in \Sigma$, or conditional *membership axioms* (stating that a term has a given sort). In Maude, an individual equation in the condition may also be a *matching equation* $p_l := q_l$, which is mathematically interpreted as an ordinary equation. However, operationally, the new variables occurring in the term p_l become instantiated by matching the term p_l against the canonical form of the instance of q_l (see [21] for further explanations). Order-sorted notation $s_1 < s_2$ abbreviates the conditional membership $(\forall x : [s_1])\ x : s_2 \ \mathbf{if}\ x : s_1$. Similarly, an operator dec-

laration $f : s_1 \times \cdots \times s_n \to s$ corresponds to declaring f at the kind level and giving the membership axiom $(\forall x_1 : [s_1], \ldots, x_n : [s_n])\ f(x_1, \ldots, x_n) : s$ **if** $\bigwedge_{1 \le i \le n} x_i : s_i$.

A Maude module specifies a *rewrite theory* [16,34] of the form $(\Sigma, E \cup A, R)$, where:

- $(\Sigma, E \cup A)$ is a membership equational logic theory specifying the system's state space as an algebraic data type with A a set of equational axioms (such as a combination of associativity, commutativity, and identity axioms), to perform equational deduction with the equations E (oriented from left to right) *modulo* the axioms A, and
- R is a set of *labeled conditional rewrite rules* specifying the system's local transitions, each of which has the form:

$$l : q \longrightarrow r \ \textbf{if} \bigwedge_i p_i = q_i \ \wedge \bigwedge_j w_j : s_j \ \wedge \bigwedge_m t_m \longrightarrow t'_m,$$

where l is a *label*, and q, r are Σ-terms of the same kind.

Intuitively, such a rule specifies a *one-step transition* from a substitution instance of q to the corresponding substitution instance of r, *provided* the condition holds.

We briefly summarize the syntax of Maude (see [21] for more details). Sorts and subsort relations are declared by the keywords `sort` and `subsort`, and operators are introduced with the `op` keyword: `op` f `:` $s_1 \ldots s_n$ `->` s, where $s_1 \ldots s_n$ are the sorts of its arguments, and s is its (value) *sort*. Operators can have user-definable syntax, with underbars '_' marking each of the argument positions, and are declared with the sorts of their arguments and the sort of their result. Some operators can have equational *attributes*, such as `assoc`, `comm`, and `id`, stating, for example, that the operator is associative and commutative and has a certain identity element. Such attributes are then 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 data elements of a sort. The `frozen` attribute declares which argument positions are *frozen*; arguments in frozen positions cannot be rewritten by rewrite rules [21].

There are three kinds of logical statements in the Maude language, *equations*, *memberships* (declaring that a term has a certain sort), and *rewrite rules*, introduced with the following syntax:

- equations: `eq` $u = v$ or `ceq` $u = v$ `if` *condition*;
- memberships: `mb` $u : s$ or `cmb` $u : s$ `if` *condition*;
- rewrite rules: `rl [`l`] :` u `=>` v or `crl [`l`] :` u `=>` v `if` *condition*.

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. The mathematical variables in such statements are either explicitly 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*. Finally, a comment is preceded by '`***`' or '`---`' and lasts till the end of the line.

2.2.2 Object-oriented specification in Real-Time Maude

A Real-Time Maude *timed module* specifies a *real-time rewrite theory* $\mathscr{R} = (\Sigma, E \cup A, R)$ [39], where:

- $(\Sigma, E \cup A)$ contains an equational subtheory $(\Sigma_{TIME}, E_{TIME}) \subseteq (\Sigma, E \cup A)$, satisfying the *TIME* axioms that specifies sort `Time` as the time domain (which can be discrete or dense). Although a timed module is parametric on the time domain, Real-Time Maude provides some predefined modules specifying useful time domains. For example, the modules `NAT-TIME-DOMAIN-WITH-INF` and `POSRAT-TIME-DOMAIN-WITH-INF` define the time domain to be, respectively, the natural numbers and the nonnegative rational numbers. and contain the subsort declarations `Nat < Time` and `PosRat < Time`. The supersort `TimeInf` extends the sort `Time` with an "infinity" value `INF`.
- The rules in R are decomposed into:
  - "ordinary" rewrite rules specifying the system's *instantaneous* (i.e., zero-time) local transitions, and
  - *tick (rewrite) rules* that model the elapse of time in a system, having the form $l : \{t\} \xrightarrow{u} \{t'\}$ **if** *condition*, where t and t' are terms of sort `System`, u is a term of sort `Time` denoting the *duration* of the rewrite, and `{_}` is a built-in constructor of sort `GlobalSystem`.
    In Real-Time Maude, tick rules, together with their durations, are specified using the syntax

    `crl [`l`] :` $\{t\}$ `=>` $\{t'\}$ `in time` u `if` *condition*.

The initial state must be reducible to a term $\{t_0\}$, for t_0 a ground term of sort `System`, using the equations in the specification. The form of the tick rules then ensures uniform time elapse in all parts of a system.

Real-Time Maude is particularly suitable to formally model distributed real-time systems in an object-oriented style. Each term t in a global system state $\{t\}$ is in such cases a term of sort `Configuration` (which is a subsort of `System`), and has the structure of a *multiset* made up of objects and messages. Multiset union for configurations is denoted by a juxtaposition operator `__` (empty syntax) that is declared associative and commutative and having the `none`

multiset as its identity element, so that rewriting is *multiset rewriting* supported directly in Maude.

In object-oriented timed modules one can declare *classes*, *subclasses*, and *messages*. A *class* declaration

$$\texttt{class}\ C\ |\ att_1 : s_1, \ldots, att_n : s_n$$

declares a class C with attributes $att_1, \ldots, att_n$ of sorts $s_1, \ldots, s_n$. An *object* of class C is represented as a term of sort `Object` and has the form

$$< O : C \mid att_1 : val_1, \ldots, att_n : val_n >,$$

where O is the object's identifier, and $val_1, \ldots, val_n$ are its attribute values. A *subclass*, introduced with the keyword `subclass`, inherits all the attributes, equations, and rules of its superclasses. A *message* is a term of sort `Msg`.

The dynamic behavior of concurrent object systems is axiomatized by specifying each of its transition patterns by a rewrite rule. For example, the rewrite rule

```
crl [l] :
    < O1 : C | a1 : x1, a2 : O2, a3 : z, a4 : p >
    < O2 : C | a1 : x2, a2 : O1, a3 : w, a4 : q >
  =>
    < O1 : C | a1 : x1 + w + z, a2 : O2, a3 : z, a4 : p >
    < O2 : C | a1 : x2 + z,     a2 : O1, a3 : w, a4 : q >
  if z <= w
```

defines a family of transitions involving two objects `O1` and `O2` of class `C`, and updates the attribute `a1` of both objects. For example, the new value of the `a1` attribute of object `O2` is the old value of that attribute plus the old value of `O1`'s attribute `a3`. By convention, attributes whose values do not change and do not affect the next state of other attributes or messages, such as `a2` and `a4` in our example, need not be mentioned in a rule. Similarly, attributes whose values influence the next state of other attributes or the values in messages, but are themselves unchanged, such as `a3`, can be omitted from right-hand sides of rules.

2.2.3 Formal analysis in Real-Time Maude

We summarize below some of Real-Time Maude's analysis commands. Real-Time Maude's *timed fair rewrite* command

(tfrew t in time <= τ .)

simulates *one behavior* of the system within time τ from the initial state t. The *timed search* command

(tsearch [n] t =>* *pattern* [such that *condition*] in time <= τ .)

analyzes *all possible behaviors* by using a breadth-first strategy to search for (at most n) states that are reachable from the initial state t within time τ, match the search *pattern*, and satisfy the (optional) search *condition*. The *untimed* search command

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

is similar, but without the time bound. If the arrow `=>!` is used instead of `=>*`, then Real-Time Maude searches for reachable *final* states, that is, states that cannot be further rewritten.

Real-Time Maude's *linear temporal logic (LTL) model checker* checks whether each behavior from an initial state, possibly up to a time bound, satisfies an LTL formula. Real-Time Maude is also equipped with a *timed CTL* model checker to analyze metric temporal logical properties [31,32].

Finally, the command

(find latest t =>* *pattern* [such that *cond*] with no time limit .)

explores all behaviors from the initial state t and finds the longest time needed to reach the desired state (for the *first* time in a behavior).

3 A formal model of human multitasking

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

We model human multitasking in an object-oriented style. The state consists of a number of `Interface` objects, representing the interfaces of the devices/systems with which a 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 (such as, e.g., withdraw cash, find a good rock music radio station, or find the route to Aunt Bertha). We describe a task as a *sequence* of actions that the user performs on the interface to reach some goal. Our notion of task therefore corresponds to the notion of *scenario* in software engineering.

3.1 Classes

This section defines the classes in our model and the data types of their attribute values.

3.1.1 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 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 (i.e., the state of the interface the person is interacting with) 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 eight minutes, after which the ATM will display a "Transaction canceled" message. The term p `for time` t denotes that the user will perceive p for time t, after which the perception becomes `expired(`p`)`. The `right id: INF` functional attribute of the `_for time_` operator means that Maude considers p `for time INF` to be *identical* to p.

A interface transition has 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 `enterCard`, and the ATM will then display that I should type my PIN code:

```
ATMready -- enterCard --> typePIN for time 480
```

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.

3.1.2 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] .
```

Each basic task has the form

inf$_1$ `|` p_1 `==>` *action* `|` *inf*$_2$ `duration` τ `difficulty` d `delay` δ

where 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 the *difficulty* of the basic task. If my 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, which 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 declared

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

If the action "performed" in a basic task is `noAction`, then the basic task describes an update of a WM entry that is done *without* interacting with the device. For example, while interacting with the ATM, I may suddenly realize that

I actually do not need any money. We call such an "actionless" basic task a *cognitive basic task*, and it represents an autonomous change in the mental state of the user. The piece of `Information` stored in the WM after the execution of a cognitive basic task has to belong to the subsort `Cognition` (described below).

As mentioned in Sect. 2.1, the next task that is given a person's attention is a function of: the cognitive loads of the current subtasks,[1] the *criticality level* of each task (a person 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. Likewise, if a task has not been given attention for a long time, it should be given attention soon. To compute 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 | subtasks : Task,
             waitTime : Time,
             status : TaskStatus,
             cognitiveLoad : Rat,
             criticalityLevel : PosRat .
```

The `subtasks` attribute denotes the *remaining* sequence of subtasks to be performed; the attribute `waitTime` denotes how long the next basic task has been enabled; the attribute `cognitiveLoad` is a rational number (`Rat`) denoting the cognitive load of the subtask currently executing; and the attribute `criticalityLevel` is a positive rational number (`PosRat`) denoting the task's criticality level. For analysis purposes, we also add an attribute `status` denoting the "status" of the task as a term of the following sort `TaskStatus`:

```
sort TaskStatus .
ops notStarted ongoing completed : -> TaskStatus [ctor] .
```

3.1.3 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 `memory` attribute stores the content of the working memory as a map I_1 `|->` mem_1 `;` `...` `;` I_n `|->` mem_n of sort `Memory`, assigning to each interface I_j the *set* mem_j of items in the memory associated with interface I_j. An element in mem_j is either a *cognition*, a basic piece of *information*, such as, e.g., `cardInMachine`, or a desired *goal* `goal(`*action*`)`. The goal defines the ultimate aim 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 not 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, where the user must provide the application-specific values of `Cognition`, `BasicInfo`, and `Action`:

```
sorts Cognition BasicInfo Action . --- user-defined sorts
sorts Goal Information .
subsorts Cognition Goal BasicInfo < Information .

op goal : Action -> Goal .

sort InfoSet .  --- sets of information elements
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 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:

[1] Since we now consider *structured* tasks and add delays to basic tasks, we redefine the cognitive load of a task to be $\frac{\sum d_i t_i}{\sum t_i + dly_i}$, 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.

– 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).

The rank of a non-empty task is given by the function `rank`, whose definition is shown after the declaration of the variables used in our model:

```
vars DACT DACT2 : DefAction .      var ACT : Action .
vars INF1 INF2 INF3 : Information . var COG2 : Cognition .
vars INF-SET INF-SET2 : InfoSet . var MEMORY : Memory .
vars P P1 : Perception .          var IS : InterfaceState .
vars I I2 : InterfaceId .         vars TASK WM : Oid .
vars T T1 T2 T3 : Time .          vars TI TI2 : TimeInf .
var NZT MIN-DELAY : NzTime .      var CAP : NzNat .
var CL : Rat .                    vars PR PR2 : PosRat .
var OTHER-SUB-TASKS : Task .      var TS : TaskStatus .
vars BASIC-TASKS BTL : Subtask .  var OBJECT : Object .
vars REST ALL-INTERFACES OTHER-INTERFACES : Configuration .
vars NEC1 NEC2 : NEConfiguration .
var TRANSITIONS : InterfaceTransitions .

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) INFO-SET) ; MEMORY)
 = if T2 == 0 then PR2 * CL * (T + 1) else 0 fi .
```

A task which is not yet enabled (the remaining delay `T2` of the first basic task is greater than 0) has rank 0. The `rank` function refines the task rank function in [15], and should therefore be consistent with results in psychology.

The tick rewrite rule in Fig. 1 models the user performing a basic task (if it does not cause memory overload, and the action performed is neither `noAction` nor the goal action) with the interface with the highest rank of all interfaces (`bestRank(...)`).

In this rule, 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 in the subtask, we update the value of `cognitiveLoad` using the following function `cogLoad`:

```
--- Compute a measure of the difficulty of a subtask as
--- (difficulty * duration of each BT):
op difficultyFactor : Subtask -> PosRat .
eq difficultyFactor(nil) =  0 .
eq difficultyFactor((INF | P ==> DACT | INF2
          duration T difficulty PR delay T1) BTL)
 = T * PR + difficultyFactor(BTL) .

--- Compute the total duration of a subtask as
--- (duration + waiting time of each BT):
op subtaskDuration : Subtask -> Time .
eq subtaskDuration(nil) = 0 .
eq subtaskDuration((INF | P ==> DACT | INF2
          duration T difficulty PR delay T1) BTL)
 = (T + T1) + subtaskDuration(BTL) .

--- Compute the cognitive load of a subtask:
op cogLoad : Subtask -> PosRat .
eq cogLoad(BTL)
 = if BTL == nil then 0
   else difficultyFactor(BTL) / subtaskDuration(BTL) fi .
```

The first conjuncts in the condition of the rewrite rule in Fig. 1 say that the rule can only be applied when the action performed is neither the goal action nor `noAction`. Since `INF1` 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 duration of this tick rule is the duration `NZT` of the executing basic task. During that time, every other task idles: the "perception timer" and the remaining delay of the first basic task are decreased according to elapsed time, and the waiting time is increased if the basic task is enabled. The definition of the `idle` operator is given in Fig. 2.

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 :
```

```
crl [interacting] :
    {OTHER-INTERFACES
     < I : Interface |
               task :
          < 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 (DACT =/= noAction) /\ (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))) .
```

Fig. 1 Rewrite rule modeling a basic interaction of the user with the interface with the highest rank

```
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) .
```

Fig. 2 Definition of the `idle` operator

```
                (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 similar rule removes an arbitrary item from the memory associated with the current interface if the memory does not store any item for another interface.

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 and the function `minDelay` returns the minimum of all delays among all first basic tasks of each task in the configuration and it is defined as follows:

```
op minDelay : NEConfiguration -> TimeInf .
eq minDelay(OBJECT) = delay(OBJECT) .
eq minDelay(NEC1 NEC2)
 = min(minDelay(NEC1), minDelay(NEC2)) .
```

In the above rules, we did not reach our goal with the interface. The following rule `closure` 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  ...
```

In the above rule `closure`, when the goal action ACT (e.g., `takeCash`) for the interface `I` is performed, we forget everything associated to the interface `I`, except the memory item (if any) `INF2` (e.g., `feelingRich`) resulting from having performed the goal action ACT. Furthermore, we remove all the remaining subtasks from the task associated to the interface `I`, and set the `status` of that task to `completed`.

As mentioned, a person may change cognition ("mental state") without interacting with a device, or may acquire knowledge through a cognitive process. For instance, a user could understand that the ATM is out of order just looking at an "out of order" message without interacting with it, and he/she could change his/her mental state by deciding to use another ATM. This can be modeled by a cognitive basic task, which performs `noAction` and updates the WM without synchronizing with any interface transition. The following rule models the execution of a cognitive basic task:

```
crl [cognitive] :
    {OTHER-INTERFACES
     < I : Interface | task :
       < TASK : Task | subtasks :
               ((INF1 | P1 ==> noInfo | COG2
                 duration NZT difficulty PR delay 0)
               BASIC-TASKS)
                :: OTHER-SUB-TASKS,
                     cognitiveLoad : CL,
                     status : TS > >
     < WM : WorkingMemory | memory : MEMORY
                  ; (I |-> INF-SET INF1 goal(ACT)),
                        capacity : CAP >}
  =>
    {idle(OTHER-INTERFACES, NZT)
     < I : Interface | task : < TASK : Task |  ... > >
     < WM : WorkingMemory | memory : MEMORY
                  ; (I |-> INF-SET COG2 goal(ACT)) >}
  in time NZT
    if ....
```

The following rule concerns only the interface: 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 key 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.

It is worth remarking that although each task is a sequence of basic tasks, and at every step, the task with the best rank is given attention next, a model may still exhibit nondeterminism, since:

– If two or more transitions of an `Interface` object are defined for the current state and action (possibly leading to different next states), one of them is chosen nondeterministically (see, e.g., rule `interacting`).

```
{initializeCognLoad(
 < wm : WorkingMemory | memory : interface_1 |-> goal(action_1) otherItems_1 ; ... ;
                                  interface_n |-> goal(action_n) otherItems_n ,
                        capacity : capacity >
 < interface_1 : Interface | task :
   < task_1 : Task | subtasks : (b_{1_1} ... b_{1_{l_1}}) :: ... :: (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 >)}
```

Fig. 3 Initial state

- At a certain stage, more than one interface may have the same best rank, in which case the task to be given attention is selected nondeterministically among those best-ranked tasks (see rule `interacting`).
- If memory becomes overloaded, the memory item that is forgotten is selected nondeterministically (see, e.g., rule `interactingForgetSomethingOtherInterface`).

To analyze whether the desired properties hold for a set of interfaces/tasks, we therefore need to analyze all possible behaviors that may nondeterministically take place from the initial state. To do so, we use Real-Time Maude reachability analysis, and analyze whether it possible to reach a (possibly final) state in which a desired property is violated.

4.1 Initial states

The initial state should have the form described in Fig. 3, 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 present 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 `cognitiveLoad` attributes of all interfaces are initialized by the `initializeCognLoad` function, which computes the cognitive load of the first subtask of each task.

4.2 Model checking the properties

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[2]:

```
(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`:

```
(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.

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

5 Case studies

This section illustrates the use of our modeling and analysis framework with three safety-critical multitasking case studies: (i) a person who interacts with a GPS navigation device while driving; (ii) an operator of an air traffic control system; and (iii) a doctor or nurse calibrating and starting multiple infusion pumps that inject drugs intravenously into a patient.

All the executable Real-Time Maude models of the case studies, with analysis commands, are available at http://www.di.unipi.it/msvbio/software/HumanMultitasking.html.

In each of the three case studies we provide what we think are plausible task sets, with appropriate durations, delays, and difficulty levels of the basic tasks, and appropriate criticality levels of the tasks. However, the purpose is to illustrate our framework, which can be used to analyze human multitasking of a given set of tasks. We do not claim that the tasks that we provide are "correct" with correct values of the various parameters. As further discussed in Sect. 7, some parameter values, like durations and delays, could be obtained experimentally, whereas the difficulty level might be a more subjective value.

5.1 Interacting with a GPS device while driving

Our first case study deals with driving while interacting with a navigation system. We have two interfaces: a car and a 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 removing 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 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) ,
    task : < driving : Task | ... >,     --- see above
    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 : < findDestination :  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 Sect. 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:

```
No solution
```

However, when the driver only types four characters, the command returns a bad state:

```
Solution 1
   T:Time --> 7
   ...
```

Here, the driver types the last two characters and finds the destination in the long list without turning her 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 the driver to complete the `turnLeft` action:

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

Real-Time Maude shows that the left turn is completed at time 24:

```
Result:
{< car : Interface | currentState : curveRight,
    previousAction : turnLeft,
    task : < driving : Task | cognitiveLoad : 1/24,
             criticalityLevel : 3/5, status : ongoing,
             subtasks :…, waitTime : 0 >,
    transitions :… >
 < gps : Interface | currentState : gpsReady,
    previousAction : chooseAddress,
    task : < findDestination : Task | cognitiveLoad : 3/5,
             criticalityLevel : 3/10, status : completed,
             subtasks : emptyTask, waitTime : 0 >,
    transitions : … >
 < wm : WorkingMemory | capacity : 5,
    memory :  car |-> keyInserted goal(pickKey) ;
              gps |-> noInfo >}
in time 24
```

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:

```
Result:  { ... } in 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 every 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:Oid : 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` is not performed, and we leave the key in the car. The same command with our "standard" model of GPS interaction does not find any final state with an uncompleted task pending.

5.2 Air traffic control operators

Air traffic control (ATC) operators are personnel responsible for monitoring and controlling air traffic. They usually work in ATC centers and control towers on the ground, by monitoring the position, speed, altitude, and route of aircraft

in their assigned sector, visually and by radar. In addition, they also deal with radio communication with pilots to give them instructions and to receive useful information about the flights, which they report on *flight progress strips* (FPSs or strips). Such FPS are paper strips used to record basic information for each aircraft, such as call sign, aircraft type, destination, altitude, planned route, flight level (FL), etc. Controllers update strips dynamically as they control the associated aircraft through their sectors. One of the main tasks of ATC operators is to avoid flight collisions; i.e., to avoid that the distance between aircraft goes below a minimum prescribed distance. When this happens, they say that the aircraft violates *separation*. Air traffic controllers can also transfer an aircraft to the next sector controller when they are too busy, or assign one of their tasks to an assistant.

Despite the availability of advanced radar and technological support, strips and other aids, ATC operators heavily rely on working memory, by encoding, storing and retrieving information recently perceived about aircraft and the environment, such as pilot requests, information about the flights, weather reports, and so on [44].

5.2.1 Modeling ATC tasks

We focus on the following three tasks that controllers have to carry out concurrently:

1. *Monitoring a radar sector* to: (i) keep the distance between aircraft under control and avoid possible collisions; (ii) move an aircraft to the next sector controller when they are too busy; and (iii) visually perceive new information about flights.
2. *Managing pilots' calls* to update information about flights on strips.
3. *Checking that an assistant carries out assigned tasks*.

For the first of these tasks, we model a different monitoring task and consequently, a different radar interface, for each critical zone of the radar sector to be monitored. The monitoring task is essentially a set of three different subtasks; we show each subtask of this task separately:

i. Controlling parts of the screen, possibly adding information about flights to the memory:

```
< monitoring : Task | subtasks :
  ((noInfo | Screen1 ==> lookAtScreen1 | noInfo
             duration 4 difficulty 4/10 delay 0)
   (noInfo | Screen2 ==> lookAtScreen2 | updFL
             duration 4 difficulty 4/10 delay 0)
   (noInfo | Screen3 ==> lookAtScreen3 | noInfo
             duration 4 difficulty 4/10 delay 0)) ::
  …
```

This subtask consists of three basic tasks of looking at different parts of the screen. The second basic task models that, while looking at a section of the screen, the ATC operator notices that one of the aircrafts has changed its flight level, and the operator adds this information (`updFL`) to her memory.

ii. Monitoring a possible collision:

```
  … ::
((noInfo | Screen3 ==> noAction | possibleClsn1
             duration 3 difficulty 4/10 delay 0)
 (possibleClsn1 | Collision1 ==> monitorClsn1 | noInfo
             duration 4 difficulty 5/10 delay 0)) ::
  …
```

This subtask consists of two basic tasks. The first is a cognition basic task (the "action performed" is `noAction`) which models that the ATC operator, while looking at the screen, understands that a collision could happen. She then adds the cognition `possibleClsn1` to her memory. She therefore changes her mental plan by recovering such a cognition from her working memory and monitors the possible collision in the second basic task.

iii. Moving an aircraft to the next sector controller, activated by a cognition about the presence of too many aircrafts on the screen:

```
… ::
((noInfo | Screen4 ==> lookAtScreen4 | noInfo
           duration 4 difficulty 4/10 delay 0)
(noInfo | Screen4 ==> noAction | movingAircraft
           duration 3 difficulty 4/10 delay 0)
(movingAircraft | Screen5 ==> move | noInfo
           duration 4 difficulty 5/10 delay 0)),
waitTime : 0,
status : notStarted,
criticalityLevel : 8/10,
cognitiveLoad : 0 >
```

This subtask consists of three basic tasks. The first one is explained above. The second one is a cognition which models the ATC operator realizing that there are too many aircrafts in her sector, and she then adds the cognition `movingAircraft` to her memory. In the third basic task, she retrieves this cognition from her memory and moves an aircraft to the next sector controller. (The end of the code above shows the remaining attributes of the `Task` object `monitoring`.)

The criticality level of the monitoring tasks could vary, depending on the number of aircraft present in the sector or their type: some of them could require little active control, such as overflights, "lows and slows," and aircraft on the pilots' own navigation or on a radar route [44].

The radar interface associated with such monitoring task is defined by the following `Interface` object:

```
< radar : Interface |
     transitions :
        (Screen1 -- lookAtScreen1 --> Screen2) ;
        (Screen2 -- lookAtScreen2 --> Screen3) ;
        (Screen3 -- lookAtScreen3 --> Collision1) ;
        (Collision1 -- monitorClsn1 --> Screen4) ;
        (Screen4 -- lookAtScreen4 --> Screen5) ;
        (Screen5 -- move --> stop),
     task : < monitoring : Task | ... >, --- see above
     previousAction : noAction,
     currentState : Screen1 >
```

For the second task above, we model a radio communication task, and consequently a radio interface, for each communication with a different pilot. It consists of a sequence of subtasks modeling the pilot's calls and the updating of strips with the information received, and is formalized by the following `Task` object `radioCommunication`:

```
< radioCommunication : Task | subtasks :
 ((noInfo | call1 ==> communicating1 | updatingAltitude
                   duration 3 difficulty 4/10 delay 7))
 ::
 ((updatingAltitude | strip1 ==> updating1 | noInfo
                   duration 2 difficulty 3/10 delay 0))
 ::
 ((noInfo | call2 ==> communicating2 | updatingRoute
                   duration 3 difficulty 4/10 delay 5))
 ::
 ((updatingRoute | strip2 ==> updating2 | noInfo
                   duration 2 difficulty 3/10 delay 0))
 ::
 ((noInfo | call3 ==> communicating3 | updatingFL
                   duration 3 difficulty 4/10 delay 2))
 ::
 ((updatingFL | strip3 ==> updating3 | noInfo
                   duration 2 difficulty 3/10 delay 0))
  waitTime : 0,
  status : notStarted,
  criticalityLevel : 4/10,
  cognitiveLoad : 0 >
```

For example, first subtask

```
(noInfo | call1 ==> communicating1 | updatingAltitude
      duration 3 difficulty 4/10 delay 7)
```

models a call from a pilot, who tells the ATC operator about an update of the altitude (`updatingAltitude`), and the following addition of this information to the ATC operator's memory. The operator then uses this information in the second subtask

```
(updatingAltitude | strip1 ==> updating1 | noInfo
      duration 2 difficulty 3/10 delay 0)
```

where the operator retrieves the information from memory and updates the altitude on the flight's strip.

The criticality level of the radio communication task could vary according to the type of aircraft the controller is receiving information from. The radio interface associated to such a communication task is formalized by the following object:

```
< radio : Interface |
     transitions :
        (call1 -- communicating1 --> strip1) ;
        (strip1 -- updating1 --> call2) ;
        (call2 -- communicating2 --> strip2) ;
        (strip2 -- updating2 --> call3) ;
        (call3 -- communicating3 --> strip3) ;
        (strip3 -- updating3 --> stop),
     task : < radioCommunication : Task | ... >,
     previousAction : noAction,
     currentState : call1 >
```

Finally, the third task, checking that an assistant carries out an assignment, is formalized by the following object:

```
< checkingAssignedTask : Task | subtasks :
    ((noInfo | assistantReady ==> checkingTask | noInfo
           duration 3 difficulty 3/10 delay 0)),
     waitTime : 0,
     status : notStarted,
     criticalityLevel : 3/10,
     cognitiveLoad : 0 >
```

We model the assistant as an interface with which the controller has to interact: he/she is formalized by the following `Interface` object:

```
< assistant : Interface |
     transitions :
        (assistantReady -- checkingTask --> stop),
     task : < checkingAssignedTask : Task | ... >,
     previousAction : noAction,
     currentState : assistantReady >
```

5.2.2 Analyzing urgency

As in the car/GPS example, some of the tasks are not only characterized by high criticality but also by urgency. If two aircrafts violate separation, the controller has to monitor the identified conflict situation as soon as she perceives it. The fact that many such tasks may have to be performed at the same time could lead to a dangerous situation where some urgent actions are not performed when they should.

To analyze urgency, we model a situation where an air traffic controller monitors her radar sector while communicating with a pilot via radio. During the monitoring activity, the ATC operator finds three possible collisions (we model three subtasks with actions `monitorClsn1`, `monitorClsn2`, and `monitorClsn3`); however, the radio communication task might prevent him/her from monitoring such collisions at a specific time. We check the longest time needed for the controller to complete all `monitorClsn`X actions:

```
Maude> (find latest {initState1} =>*
     {< radar : Interface | previousAction : monitorClsn >
      REST:Configuration }
    with no time limit .)
```

The result of this analysis shows that the `monitorClsn1` action is completed at time 15, the `monitorClsn2` action

is completed at time 42, and the `monitorClsn3` action is completed at time 57. However, the same analysis with an initial state with just the radar interface object and task shows that when the ACT operator is not distracted, she finishes the `monitorClsn2` action at time 50—8 time units *later* than in the multitasking scenario—and the `monitorClsn3` action completes at time 72, i.e., 15 time units later than in the multitasking scenario.

5.2.3 Analyzing memory failures due to multitasking

Analysis of interviews with air traffic controllers in [44] indicate that memory errors are associated with working memory overload and distraction. We focus on three kinds of errors presented in the report by modeling, simulating, and analyzing them, and we try to give a plausible explanation of these errors in term of cognitive causes: *prospective memory failure*, *retrospective memory failure*, and *forgetting temporary information*.

Prospective memory is the form of memory involved in remembering to perform a planned action. Sixteen errors presented in [44] involve prospective memory failure.

To analyze prospective memory failures, we model a situation where the controller monitors three different zones in her radar sector while communicating with two different pilots and checking that an assistant carries out an assigned task. The checking assignment task has a delay of 20 time units, since controller plans to perform this task in the future.

The initial state of the working memory is

```
< wm : WorkingMemory | capacity: 7,
          memory : (radar1 |-> goal(lookAtScreen3)) ;
                   (radar2 |-> goal(lookAtScreen12)) ;
                   (radar3 |-> goal(lookAtScreen7)) ;
                   (radio1 |-> goal(updating1)) ;
                   (radio2 |-> goal(updating2)) ;
                   (assistant |-> goal(checkingTask)) >
```

We check if all tasks are guaranteed to be completed:

```
Maude > (utsearch [1] initState1 =>!
           {< I:InterfaceId : Interface | task :
               < T:Oid : Task | status : TS:TaskStatus,
                                A:AttributeSet > >
            REST:Configuration }
         such that TS:TaskStatus =/= completed .)
```

and we find a bad state: the goal associated to the assistant interface is deleted from memory and the controller cannot complete the checking assignment task.

Retrospective memory is the memory of people, words, or events encountered or experienced in the past. Three of the errors presented in [44] involve retrospective memory failure: controllers lose track of task progress since they forget the action previously performed. Task interruptions have disruptive effects on task performance, although several studies show that they have different consequences when performed at different moments [1,28].

To analyze retrospective memory failures, we model a situation where the air traffic controller monitors the radar and decides to move an airplane to the next controller's sector when she perceives that she is too busy. At the same time, she has to answer three different calls from pilots and annotate the information received by them on strips.

We show that interrupting the monitoring task at different moments can have different consequences. We model two initial states, `initState1` and `initState2`. In the first one, the controller receives three calls after a delay of 7, 8 and 9 time units, respectively; in the second one the controller receives three calls after a delay of 8, 9 and 10 time units, respectively. The difficulties of the radio communication tasks have been set in order to have the same cognitive load for each task.

We check whether all tasks are guaranteed to complete for both initial states. Our analysis found an undesired state for `initState1`, and no such state for `initState2`. In the first case the main task is interrupted after the operator decides to transfer the aircraft: adding new information from calls lead to memory overload which resulted in forgetting this decision. In the second case, the monitoring is not interrupted after the controller's decision to move the airplane.

Many of the errors reported in [44] concern forgetting temporary information. Some of these errors involve forgetting about the presence of aircrafts that require little active control. To analyze such errors, we model a situation where the air traffic controller monitors different zones of the radar screen at the same time. Some of these zones have highly critical situations to monitor, while one of them have aircrafts that require low control and thus have lower criticality. We show that the controller can forget the less critical sector because she is distracted by other sectors, and too much information is added to her memory.

We model five interfaces representing the different zones of the screen. Each zone has a different cognitive load and a different criticality level, depending on the number of flights and the type of aircrafts flying there.

We analyze whether all tasks are guaranteed to complete. The usual command shows that the task associated with the zone with low criticality, representing a zone with flights which require little control, does not complete because that task's goal is deleted from the working memory. We check whether the task associated with that zone is at least started with commands:

```
Maude> (find latest {initState1} =>*
   {< zone2 : Interface | previousAction : lookAtScreen4 >
    REST:Configuration }
   with no time limit .)
```

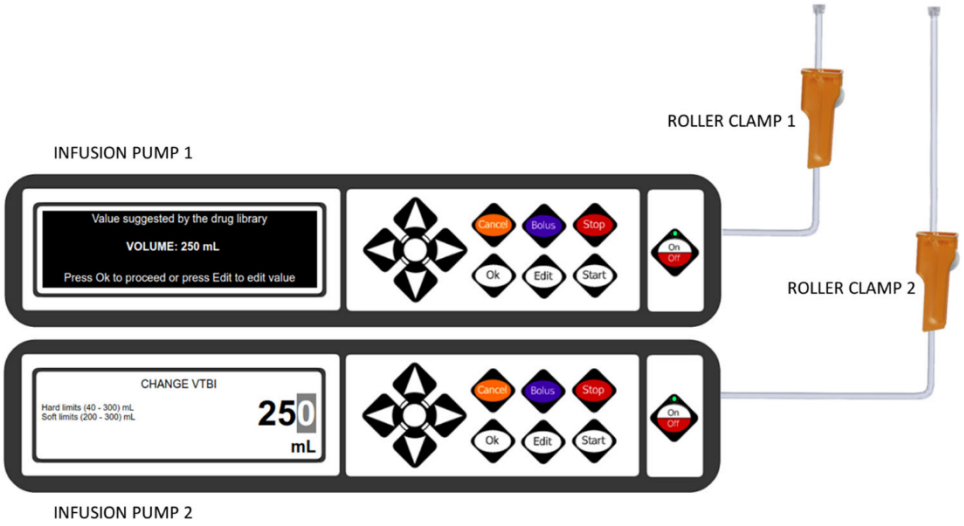

Fig. 4 Example scenario with two infusion pumps

and

```
Maude> (find latest {initState1} =>*
    {< radar : Interface | previousAction : lookAtScreen12 >
     REST:Configuration }
   with no time limit .)
```

The first command shows that the first action of the task is performed at time 33, while the second command shows that the last action of the task is never performed.

5.3 Concurrent configuration of infusion pumps

This section shows the application of our framework to a case study based on an experiment described in [7], where users were asked to interact with two medical devices. The aim of the experiment was to study multitasking strategies adopted by clinicians, to assess whether particular strategies could induce omission errors, for example forgetting to perform a procedural step required to complete the task.

The original experiment involved the use of two simulated infusion pumps (see Fig. 4). Infusion pumps are medical devices routinely used in hospitals to inject fluids (e.g., drugs or nutrients) into the bloodstream of a patient in precise amount and at controlled rates. The devices under consideration provide a front panel with a display and a number of buttons used by clinicians to configure, operate, and monitor the pump. To set up an infusion pump, clinicians are usually required to perform five main steps:

1. Read infusion parameters, typically volume to be infused (vtbi) and infusion duration or infusion rate, from a prescription chart.
2. Enter the infusion parameters using the data entry system provided by the pump.
3. Connect the pump to the patient using a "giving set" (a transparent plastic tube with a needle at one end, and a bag with fluid at the other end).
4. Open the roller clamp to allow the fluid to circulate.
5. Start the infusion.

Table 1 A multitasking strategy for setting up two infusion pumps

Time	Prescription chart	Pump 1	Pump 2
1	read vtbi1		
2	read vtbi2		
3		enter vtbi1	
4			enter vtbi2
5	read time1		
6		enter time1	
7		open clamp1	
8	read time2		
9			enter time2
10			open clamp2
11		start infusion1	
12			start infusion2

Intensive care patients may be connected to more than one infusion pump at the same time. When multiple infusion pumps need to be configured, clinicians may choose to interleave the steps necessary for setting up the pumps. This is usually done to optimize cognitive resources (e.g., memory load), or time (e.g., to perform operations on one pump while waiting that the other pump executes an operation) [7].

Different multitasking strategies may produce different memory loads. One possible multitasking strategy for setting up the two pumps is shown in Table 1. The question we consider is "What is the capacity of the working memory needed to ensure that all tasks are successfully completed,

using a particular multitasking strategy?" An answer to this question could help manufacturers design devices that are simpler to use. It could also be used by hospitals to develop better training material for clinicians. Academic researchers could also benefit, e.g., to test cognitive hypotheses before running an experimental study.

5.3.1 Model

Our approach is to first model the concurrent interaction with two infusion pumps. This model is then used to estimate expected memory load, by checking whether all tasks can be successfully completed when the capacity of the working memory is X. By varying this parameter X, we can find the smallest capacity necessary. To experiment with different multitasking strategies, we use different values for the difficulty and duration parameters of the basic tasks.

The model includes interfaces representing each pump. The concurrent tasks relate to the procedure for setting vtbi and time values for the two pumps. To set the values, clinicians must read and memorize the values provided by the prescription chart, and then use the pumps' data entry system to enter the values.

The task for setting up *Pump 1* is specified as follows (the task for *Pump 2* is analogous):

```
< settingPump1 : Task | subtasks :
 ((noInfo | prescriptionFormVtbiP1 ==> noAction | vtbi300
                duration 1 difficulty 2/10 delay 0)
  (vtbi300 | setVTBIP1 ==> type300 | noInfo
                duration 1 difficulty 2/10 delay 0))
 ::
 ((noInfo | prescriptionFormTimeP1 ==> noAction | time3
                duration 1 difficulty 2/10 delay 0)
  (time3 | setTimeP1 ==> type3 | noInfo
                duration 1 difficulty 2/10 delay 0))
 ::
 ((clampOpeningP1 | clampP1 ==> openClampP1 | noInfo
                duration 1 difficulty 2/10 delay 0))
 ::
 ((noInfo | infusionReadyP1 ==> startInfusionP1 | noInfo
                duration 1 difficulty 2/10 delay 0)) >
```

This task consists of six basic tasks, grouped into four subtasks:

1. Read and memorize the vtbi value for *Pump 1* from the prescription chart;
2. Enter vtbi in *Pump 1*;
3. Read and memorize the infusion duration for *Pump 1* from the prescription chart;
4. Enter infusion duration in *Pump 1*;
5. Open clamp 1;
6. Start infusion.

The basic task

```
(noInfo | prescriptionFormVtbiP1 ==> noAction | vtbi300
            duration 1 difficulty 2/10 delay 0)
```

models a cognitive basic task: the operator reads from the prescription chart the vtbi value for the pump 1, she finds out that the value to be inserted is 300 and inserts into her working memory the cognition `vtbi300`.

The infusion pump interface associated to such task is defined by the following `Interface` object:

```
< pump1 : Interface |
     transitions :
        (setVTBIP1 -- type300 --> setTimeP1) ;
        (setTimeP1 -- type3 --> clampP1) ;
        (clampP1 -- openClampP1 --> infusionReadyP1) ;
        (infusionReadyP1 -- startInfusionP1
             --> newInterfaceStateP1),
     task : < settingPump1 : Task | ... >, --- see above
     previousAction : noAction,
     currentState : setVTBIP1 >
```

5.3.2 Analysis

To analyze whether with working memory capacity X, we can complete all tasks successfully, the initial state of the `WorkingMemory` object `wm` is

```
< wm : WorkingMemory |
     memory :
       (pump1 |-> goal(startInfusionP1) clampOpeningP1);
       (pump2 |-> goal(startInfusionP2) clampOpeningP2),
     capacity : X >
```

The initial content of the WM consists of the two task goals (i.e., starting the infusion), and two memory items to remember to open the roller clamps before starting the infusion (`clampOpeningP1` and `clampOpeningP2`).

Real-Time Maude is then used to check whether a given WM capacity X is sufficient to achieve the goal. This helps to obtain a quantitative evaluation of the complexity of the task (in terms of memory load) and to identify situations where the multitasking strategy could exceed the WM capacity of the operator.

The exact same `utsearch` command as in Sect. 5.2.3 is then used to check whether, from the given initial state, it is possible to reach a *final* state where all tasks have not completed successfully. By experimenting with different parameter values, the model checker finds interleaving strategies where the user is not able to complete the tasks when the capacity of the WM is set to 5. One such example is given in Table 1: with WM capacity set to 5, the user can perform correctly the concurrent tasks up to *enter time 2* (i.e., an omission error occurs for action *open clamp 2*). If the WM capacity is set to 6, on the other hand, the analysis shows that the user is always able to reach the goal successfully, with any multitasking strategy.

The results of our analysis are in line with the experimental results in [7] and provide an explanation to the omission error in terms of CL.

5.3.3 Modeling and analyzing a redesigned interface

To check whether a design solution could be adopted to reduce memory load, the pump design was modified using the Next-Action Cueing technique [19]. A set of cues is presented in the user interface of the system at appropriate moments, to remind the operator what action should be performed next. For example, when the clamp needs to be opened, the operator does not need to retrieve this information from WM if there is a visual cue on the pump screen that indicates what needs to be done (e.g., a simple message "OPEN CLAMP" on the display of the pump).

This new design incorporated into the model, by introducing a cognitive basic task in the subtask for setting up an infusion pump: perceiving the cue will trigger the activation and execution of a certain action.

```
< settingPump1 : Task | subtasks :
 ((noInfo | prescriptionFormVtbiP1 ==> noAction | vtbi300
                  duration 1 difficulty 2/10 delay 0)
  (vtbi300 | setVTBIP1 ==> type300 | noInfo
                  duration 1 difficulty 2/10 delay 0))
 ::
 ((noInfo | prescriptionFormTimeP1 ==> noAction | time3
                  duration 1 difficulty 2/10 delay 0)
  (time3 | setTimeP1 ==> type3|noInfo
                  duration 1 difficulty 2/10 delay 0))
 ::
 ((noInfo | clampP1 ==> noAction| clampOpeningP1
                  duration 1 difficulty 2/10 delay 0)
  (clampOpeningP1 | clampP1 ==> openClampP1 | noInfo
                  duration 1 difficulty 2/10 delay 0))
 ::
 ((noInfo | infusionReadyP1 ==> startInfusionP1 | noInfo
                  duration 1 difficulty 2/10 delay 0)) >
```

The basic task

```
(noInfo | clampP1 ==> noAction | clampOpeningP1
    duration 1 difficulty 2/10 delay 0)
```

models the cognitive basic task mentioned above: the operator, by looking at the pump interface, notices a signal and understands that she has to open the clamp. She then inserts into her working memory the cognition `clampOpeningP1`, which she uses in the following basic task to perform the action `openClampP1` with the `pump1` interface.

Analysis of this new version of the task indicates that, for all possible interleaving strategies, the user is always able to complete the tasks as long as the capacity of the WM is at least 5.

6 Related work

There has been some work on applying "computational models" to study human attention and multitasking. The ACT-R architecture [3], an executable rule-based framework for modeling cognitive processes, has been applied to study, e.g., the effects of distraction by phone dialing while driving [41] and the sources of errors in aviation [17]. Recent versions of ACT-R handle human attention in accordance with the theory of concurrent multitasking proposed in [42]. The theory describes concurrent tasks that can interleave and compete for resources. Cognition balances task execution simply by favoring least recently processed tasks. Additional parameters, such as the criticality level or the cognitive load of the task, are not taken into account.

Other computational models for human multitasking include the *salience, expectancy, effort and value* (*SEEV*) model [49] and the *strategic task overload management* (*STOM*) model [47]. The SEEV model is specifically designed to describe (sequential) visual scanning of an instrument panel, where each instrument may serve different tasks. The model has been validated against data collected by performing experiments with real users using the BORIS robotic simulator [45] developed for the training of aerospace professionals. The multitasking paradigm underlying SEEV is different from the one we consider in this paper, which is not *sequential scanning* but *voluntary task switching* [4].

The STOM model, that can be seen as an evolution of the SEEV model, is closer to our work. Like our framework, STOM deals with voluntary task switching and uses a multi-attribute approach to predict the decision of the user switching from one task to another. Which attributes to consider and how they should be weighted in the model are still open research questions. In its original formulation, STOM used four attributes for each task: difficulty, priority, interest, and salience [47]. This version of the model was validated against data of real users interacting with the MATB II simulator [43,48]. More recently, new attributes have been considered, such as the time of tasks [46]. Some of the attributes considered in STOM are similar to parameters we consider in our framework. Task difficulty, for instance, is also used in our framework, although at the level of basic tasks rather than of whole tasks (our tasks are structured). Moreover, the priority attribute is somehow similar to our task criticality level. However, a fundamental difference between STOM and our framework is that we model attention switching at a lower level of abstraction, namely, by describing the underlying cognitive processes related to the WM and the CL. This allows us to base our approach on the well-established theories described in Sect. 2.1, which have been validated through several experiments on both adults and children [8–11,24] and through neuroimaging [24], a technique which allows studying the activity of different brain areas as well as specific brain functions. Another important difference between STOM and our framework is that STOM does not explicitly model the user's working memory, its capacity limit, and its action mechanism. Modeling the WM enables us to analyze memory issues such as memory over-

load, retrospective memory failure, and prospective memory failure, which are some of the main problems in human multitasking. Finally, STOM does not provide a formal model that can be subjected to different formal analysis methods to analyze whether the model satisfies desired properties.

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, Gelman et al. [25] model a pilot and the flight management system (FMS) of a civil aircraft and use WMC simulation and SAL model checking to study *automation surprises* (i.e., the system works as designed but the pilot is unable to predict or explain the behavior of the aircraft). The occurrence of the automation surprise is studied by checking the reachability of an undesired state, where the mental state of the pilot differs from the actual state of the airplane. In [30] the WMC simulator and GALE, an optimization method for complex models, are applied to study the *continuous descent approach* (*CDA*), a continuous nonstop descent in which only one request for landing is needed. The papers [26,27] propose a formal model for reasoning about excessive task load and concurrency issues that can lead to errors in human-machine interaction with complex systems. Task load is a measure of the number of tasks a user is expected to perform at a given time, and has been shown to be a good indicator of user mental workload in the avionics domain. In [29], the PVS theorem prover and the NuSMV model checker are used to find the potential source of automation surprises in a flight guidance system. In contrast to our work, none of these formal frameworks deal with multitasking.

We discuss the differences with the formal cognitive framework proposed in [18] in the introduction.

Finally, as mentioned in Sect. 2.1, in [15], 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 psychological literature.

7 Concluding remarks

We have presented an executable formal framework for human multitasking in Real-Time Maude which supports the modeling, simulation, and formal model checking analysis of a human interacting with multiple interfaces.

Our framework is able to analyze different problems in human multitasking, including errors caused by user distraction, cognitive overload, and memory overload. We focus on safety-critical multitasking, and therefore include criticality levels of tasks in our framework.

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 three case studies from different application domains.

Modeling human behavior is a complex task. Although the task ranking procedure used to model attention switching is consistent with studies in psychology, we have been working on fine-tuning our model by performing experiments with real users. In collaboration with psychologists, we have devised a web application to administer a test in which users were asked to interact with two concurrent tasks: a "main" critical task, and a "distractor" task [12]. Through these experiments, we identified different typologies of users; these results should be used to formalize the behavior of these typologies.

Although we used what we think are plausible task sets in our case studies, they were not defined based on known task sets. In general, some parameter values of a task (e.g., duration and delay) are measurable, whereas others, like difficulty and criticality, are not. It is, for example, possible to measure how long it takes for the user to push a button on the GPS interface, or how long it takes for the GPS to process the inserted address and enable the next basic action. But it seems hard or impossible to measure the difficulty of turning the steering wheel, since difficulty is a subjective parameter.

The proposed framework should be further developed in different directions. At the moment, our tasks are sequences of subtasks (which again are sequences of basic tasks) representing single *scenarios*, and the task switching algorithm is essentially deterministic. On the one hand, the framework should be extended to a probabilistic setting, where a user directs attention to different tasks with certain probabilities. Such probabilistic real-time models could then be subjected to statistical model checking analysis using tools such as PVeStA [2]. On the other hand, our framework should also allow taking into account possible multitasking *strategies*.

Finally, we should compare our framework with related models of human multitasking on selected case studies and apply it to other safety-critical multitasking applications.

Acknowledgements We thank Paolo Masci for his contributions to the infusion pump case study and the anonymous reviewers for their very helpful comments on an earlier version of this paper.

References

1. Adamczyk PD, Bailey BP (2004) If not now, when? The effects of interruption at different moments within task execution. In: Proceedings of the SIGCHI conference on human factors in computing systems. ACM, pp 271–278
2. AlTurki M, Meseguer J (2011) PVeStA: a parallel statistical model checking and quantitative analysis tool. In: CALCO'11, LNCS, vol 6859. Springer
3. Anderson JR, Matessa M, Lebiere C (1997) ACT-R: a theory of higher level cognition and its relation to visual attention. Hum Comput Interact 12(4):439–462
4. Arrington CM, Logan GD (2005) Voluntary task switching: chasing the elusive homunculus. J Exp Psychol Learn Mem Cogn 31(4):683–702
5. Atkinson RC, Shiffrin RM (1968) Human memory: a proposed system and its control processes. In: Spence KW, Spence JT (eds) Psychology of learning and motivation, vol 2. Academic Press, London, pp 89–195
6. Australian Transport Safety Bureau: Dangerous distraction. Safety Investigation Report B2004/0324 (2005)
7. Back J, Cox A, Brumby D (2012) Choosing to interleave: human error and information access cost. In: SIGCHI conference on human factors in computing systems, CHI'12. ACM, pp 1651–1654
8. Barrouillet P, Bernardin S, Camos V (2004) Time constraints and resource sharing in adults' working memory spans. J Exp Psychol Gen 133(1):83–100
9. Barrouillet P, Bernardin S, Portrat S, Vergauwe E, Camos V (2007) Time and cognitive load in working memory. J Exp Psychol Learn Mem Cogn 33(3):570
10. Barrouillet P, Camos V (2001) Developmental increase in working memory span: resource sharing or temporal decay? J Mem Lang 45(1):1–20
11. Barrouillet P, Gavens N, Vergauwe E, Gaillard V, Camos V (2009) Working memory span development: a time-based resource-sharing model account. Dev Psychol 45(2):477
12. Broccia G (2019) A formal framework for modelling and analysing safety-critical human multitasking. Ph.D. thesis, University of Pisa (in preparation)
13. Broccia G, Masci P, Milazzo P (2018) Modeling and analysis of human memory load in multitasking scenarios. In: Proceedings of the SIGCHI symposium on engineering interactive computing systems (EICS 2018). ACM, pp 9:1–9:7
14. Broccia G, Milazzo P, Ölveczky PC (2018) An executable formal framework for safety-critical human multitasking. In: NASA formal methods (NFM 2018), LNCS, vol 10811. Springer
15. Broccia G, Milazzo P, Ölveczky PC (2018) An algorithm for simulating human selective attention. In: SEFM 2017 collocated workshops, LNCS, vol 10729. Springer
16. Bruni R, Meseguer J (2006) Semantic foundations for generalized rewrite theories. Theor Comput Sci 360(1–3):386–414
17. Byrne MD, Kirlik A (2005) Using computational cognitive modeling to diagnose possible sources of aviation error. Int J Aviat Psychol 15(2):135–155
18. Cerone A (2016) A cognitive framework based on rewriting logic for the analysis of interactive systems. In: SEFM 2016, LNCS, vol 9763. Springer
19. Chung PH, Byrne MD (2008) Cue effectiveness in mitigating post-completion errors in a routine procedural task. Int J Hum Comput Stud 66(4):217–232
20. Clark T et al (2006) Impact of clinical alarms on patient safety. Technical report, ACCE Healthcare Technology Foundation
21. Clavel M, Durán F, Eker S, Lincoln P, Martí-Oliet N, Meseguer J, Talcott C (2007) All About Maude, LNCS, vol 4350. Springer
22. Dingus TA, Guo F, Lee S, Antin JF, Perez M, Buchanan-King M, Hankey J (2016) Driver crash risk factors and prevalence evaluation using naturalistic driving data. Proc Natl Acad Sci 113(10):2636–2641
23. Dismukes R, Nowinski J (2007) Prospective memory, concurrent task management, and pilot error. In: Attention: from theory to practice. Oxford University Press
24. de Fockert JW, Rees G, Frith CD, Lavie N (2001) The role of working memory in visual selective attention. Science 291(5509):1803–1806
25. Gelman G, Feigh KM, Rushby JM (2014) Example of a complementary use of model checking and human performance simulation. IEEE Trans Hum Mach Syst 44(5):576–590
26. Houser A, Ma LM, Feigh K, Bolton ML (2015) A formal approach to modeling and analyzing human taskload in simulated air traffic scenarios. In: Complex systems engineering (ICCSE). IEEE, pp 1–6
27. Houser A, Ma LM, Feigh KM, Bolton ML (2018) Using formal methods to reason about taskload and resource conflicts in simulated air traffic scenarios. Innov Syst Softw Eng 14(1):1–14
28. Iqbal ST, Bailey BP (2005) Investigating the effectiveness of mental workload as a predictor of opportune moments for interruption. In: CHI'05 extended abstracts on Human factors in computing systems. ACM
29. Joshi A, Miller SP, Heimdahl MPE (2003) Mode confusion analysis of a flight guidance system using formal methods. In: Digital avionics systems conference (DASC'03). IEEE
30. Krall J, Menzies T, Davies M (2016) Learning mitigations for pilot issues when landing aircraft (via multiobjective optimization and multiagent simulations). IEEE Trans Hum Mach Syst 46(2):221–230
31. Lepri D, Ábrahám E, Ölveczky PC (2013) A timed CTL model checker for Real-Time Maude. In: CALCO'13, LNCS, vol 8089. Springer
32. Lepri D, Ábrahám E, Ölveczky PC (2015) Sound and complete timed CTL model checking of timed Kripke structures and real-time rewrite theories. Sci Comput Program 99:128–192
33. Lofsky AS (2005) Turn your alarms on!. APSF Newslett 19(4):43
34. Meseguer J (1992) Conditional rewriting logic as a unified model of concurrency. Theor Comput Sci 96:73–155
35. Meseguer J (1998) Membership algebra as a logical framework for equational specification. In: Proceedings of the WADT'97, LNCS, vol 1376. Springer
36. Miller GA (1956) The magical number seven, plus or minus two: some limits on our capacity for processing information. Psychol Rev 63(2):81–97
37. Miller GA, Galanter E, Pribram KH (1986) Plans and the structure of behavior. Adams Bannister Cox, New York
38. Ölveczky PC (2014) Real-Time Maude and its applications. In: WRLA 2014, LNCS, vol 8663. Springer (2014)
39. Ölveczky PC, Meseguer J (2002) Specification of real-time and hybrid systems in rewriting logic. Theor Comput Sci 285:359–405
40. Ölveczky PC, Meseguer J (2007) Semantics and pragmatics of Real-Time Maude. High Order Symb Comput 20(1–2):161–196
41. Salvucci DD (2001) Predicting the effects of in-car interface use on driver performance: an integrated model approach. Int J Hum Comput Stud 55(1):85–107
42. Salvucci DD, Taatgen NA (2008) Threaded cognition: an integrated theory of concurrent multitasking. Psychol Rev 115(1):101–130
43. Santiago-Espada Y, Myer RR, Latorella KA, Comstock JR Jr (2011) The multi-attribute task battery II (MATB-II) software for human performance and workload research: a user's guide. NASA tech. rep.
44. Shorrock ST (2005) Errors of memory in air traffic control. Saf Sci 43(8):571–588

45. Todd BK, Fischer J, Falgout J, Schweers J (2013) Basic operational robotics instructional system. NASA tech. rep.
46. Wickens CD, Gutzwiller RS (2017) 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. SAGE Publications, pp 757–761
47. Wickens CD, Gutzwiller RS, Santamaria A (2015) Discrete task switching in overload: a meta-analyses and a model. Int J Hum Comput Stud 79:79–84
48. Wickens CD, Gutzwiller RS, Vieane A, Clegg BA, Sebok A, Janes J (2016) Time sharing between robotics and process control: validating a model of attention switching. Hum Factors 58(2):322–343
49. Wickens CD, Sebok A, Li H, Sarter N, Gacy AM (2015) Using modeling and simulation to predict operator performance and automation-induced complacency with robotic automation: a case study and empirical validation. Hum Factors 57(6):959–975