Task Model Design and Analysis with Alloy[⋆]

Alcino Cunha[1,2][0000−0002−2714−8027], Nuno Macedo[1,3][0000−0002−4817−948X], and Eunsuk Kang[4][0000−0001−7891−6885]

[1] INESC TEC, Porto, Portugal
[2] University of Minho, Braga, Portugal
[3] Faculty of Engineering of the University of Porto, Porto, Portugal
[4] Carnegie Mellon University, Pittsburgh, USA

Abstract. This paper describes a methodology for task model design and analysis using the Alloy Analyzer, a formal, declarative modeling tool. Our methodology leverages (1) a formalization of the HAMSTERS task modeling notation in Alloy and (2) a method for encoding a concrete task model and compose it with a model of the interactive system. The Analyzer then automatically verifies the overall model against desired properties, revealing counter-examples (if any) in terms of interaction scenarios between the operator and the system. In addition, we demonstrate how Alloy can be used to encode various types of operator errors (e.g., inserting or omitting an action) into the base HAMSTERS model and generate erroneous interaction scenarios. Our methodology is applied to a task model describing the interaction of a traffic air controller with a semi-autonomous Arrival MANager (AMAN) planning tool.

Keywords: Task models · HAMSTERS · Interactive system analysis · Alloy · Air traffic control · Arrival manager

1 Introduction

Task models are systematic approaches to describing the activities of a human operator in an interactive computer system. Task models can be used to articulate the operator's goals and means to achieve them, evaluate the usability of an interface, and reason about the impact of operator errors on the system. In safety-critical domains, such as aviation systems and medical devices, where safety failures have been attributed to interaction design [21], formal methods can play an important role in rigorously specifying and verifying task models against desirable interaction properties.

This paper proposes a methodology for specifying and analyzing task models in Alloy, a declarative modeling language based on first-order relational logic [11]. We demonstrate how Alloy can be used to formally specify HAMSTERS [2,15],

[⋆] The work of the first two authors is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project LA/P/0063/2020. The last author was supported in part by the National Science Foundation award CCF-2144860.

a notation for hierarchical task modeling, and analyze various properties about task models using the Alloy Analyzer. Our modeling approach consists of (1) a generic model encoding the semantics of HAMSTERS, (2) instantiation of an application-specific task model on top of this semantic model, and (3) simulation and verification of interaction properties, which, if violated, yield a counterexample that is visualized as a sample interaction scenario. In addition, we show how the basic HAMSTERS model can be extended with a generic error model that captures various types of operator errors (such as inserting or omitting an action), to enable analysis of a task model under erroneous interaction scenarios.

We demonstrate our methodology through the application to a case study on an Arrival MANger (AMAN) tool, an interactive aircraft traffic control software [17]. We have specified a part of the task model for AMAN and checked interaction properties that are important for the traffic controller to successfully carry out their tasks, such as the presence of appropriate visual feedback and deadlock-freeness. In the process, we have identified several flaws in the interaction design as well as the system requirements that are given in the AMAN reference documentation, for some of which we suggest a fix.

As far as we are aware, our work is the first to formalize and analyze task models using Alloy. Although the focus of this paper is on AMAN, our approach is general and should be applicable to other task models in HAMSTERS.

This paper is organized as follows. We begin by introducing our formalization of the HAMSTERS notation in Alloy (Section 2). We then describe an instantiation of the semantic model for specifying the task model for the AMAN tool and the analysis of its interaction properties (Sections 3 and 4). Section 5 explores previous work related to our approach. We conclude with a discussion of the limitations of our approach as well as future work (Section 6).

2 Formalizing HAMSTERS with Alloy

This section presents an Alloy formalization of a subset of the HAMSTERS task model notation, addressing both its structural and behavioral semantics, and a general technique to compose task models with a formal model of the interactive system. Lastly, an extension to model erroneous user behavior is presented.

Like most task modeling notations, HAMSTERS allows the hierarchical decomposition of tasks in a tree-like structure. A key feature of HAMSTERS is that operators are also nodes that define the temporal relationship between sub-tasks, while in the popular ConcurTaskTrees (CTT) notation [19,18] such operators are defined in arcs between the sibling sub-tasks, which can be confusing when different operators are used to decompose a task. Composite tasks have exactly one such child operator node, which can be further decomposed in operator nodes. To simplify our formalization, we will assume that task and operator nodes are always interleaved. This does not limit the expressiveness, since *phantom tasks* can always be added when operator nodes have children operators, as explained in [2]. This allows us to merge composite tasks with the corresponding temporal

```
abstract sig Task {}
abstract sig Atomic extends Task {}
abstract sig Composite extends Task { subtasks : seq Task }
abstract sig Disable, Suspend, Concurrent, Choice, Sequence
             extends Composite {}
one sig Root in Task {}
sig Iterative, Optional, Input in Task {}
// Derived relations
fun parent : Task → Task { ... }
fun succ   : Task → Task { ... }
fact WellFormed {
  // The task model forms a tree
  no Root.parent
  all t : Task - Root | one t.parent
  all t : Task | t not in t.^parent
  // Composite tasks must have at least
  // two (non-duplicate) sub-tasks
  all t : Composite | not lone elems[t.subtasks] and
                      not hasDups[t.subtasks]
  // Choice, disable, and suspend tasks
  // cannot have optional sub-tasks
  all t : Choice + Disable + Suspend | no parent.t & Optional
  ...
}
```

Fig. 1. HAMSTERS structural semantics

operator, and to view a HAMSTERS model as a tree containing only task nodes, composite in branch nodes and atomic in the leaves.

2.1 Structural semantics

Figure 1 presents an excerpt of the Alloy formalization of the structural semantics of HAMSTERS. In Alloy, *signatures* are used to declare entities of the domain. Furthermore, its type system supports inheritance: signatures can extend other signatures or be declared abstract, when they cannot contain elements outside one of their extensions. It is also possible to declare *inclusion signatures*, arbitrary subsets of the parent signature that, unlike *extension signatures*, are not required to be disjoint from their siblings. Here we declare an abstract `Task` signature with two extensions, containing the `Atomic` and `Composite` tasks. The latter is further extended by the five HAMSTERS temporal relationships supported in our formalization: `Disable`, `Suspend`, `Concurrent`, `Choice`, and `Sequence`[5]. All

[5] This is known as *Enable* in HAMSTERS, but to avoid confusion with the concept of enabled in the proposed behavioral semantics, we opted to rename it as *Sequence*.

these task types are declared as abstract and will later be extended with signatures denoting the concrete tasks in a specific task model. Finally, a subset singleton signature is declared to denote the `Root` task (in Alloy it is possible to restrict the cardinality of a signature with a multiplicity constraint, in this case **`one`**), as well as two subset signatures marking the `Iterative` and `Optional` tasks. HAMSTERS further classifies tasks according to their nature (for example, distinguishing *User*, *Interactive*, and *System* tasks), which do not affect the behavioral semantics of the model. In our formalization, we identify the *Interactive* user `Input` tasks only, because they will be relevant to some requirements and when considering erroneous execution.

Inside an Alloy signature, it is possible to declare *fields*, relations that map its elements to other entities in the domain. Field `subtasks` of the `Composite` signature relates each composite task with its sub-tasks. Since for some operators, namely `Sequence`, the order of the sub-tasks is relevant, each composite task cannot be related to an arbitrary set of sub-tasks. Recent versions of Alloy allow the declaration of sequences with bounded length with the **`seq`** keyword. Sequences are modeled as mappings from integer indexes to the respective elements, and come equipped with several pre-defined functions and predicates (e.g., `elems` that determines the set of elements in a sequence, or `hasDups` that checks if a sequence contains duplicate elements). In Alloy, parametrized *functions* (keyword **`fun`**) and *predicates* (keyword **`pred`**) can also be declared to define reusable expressions and formulas, respectively. Functions without parameters can be used to define derived constant expressions. In Fig. 1 two derived relations are declared (definitions omitted): `parent`, that relates a task with its parent task, and `succ`, that relates a task with its next sibling task.

Facts can be used to impose assumptions in an Alloy model. These are specified using *Relational Logic*, an extension of *First-Order Logic* with operators that simplify the definition of derived (relational) expressions. The most used operator is the *dot join* composition (`.`), which allows the navigation through fields to obtain related elements. For example, given task `t`, the set of its parent tasks is represented by `t.parent` and the set of its children sub-tasks by `parent.t`. Set operators can also be used, namely intersection (`&`), union (`+`), and difference (`-`). Atomic formulas in Alloy are typically inclusion or multiplicity tests. Operator **`in`** checks if an expression is a subset or equal to another one, and the available multiplicity checks are **`no`** (empty), **`lone`** (at most one element), **`some`** (at least one element), or **`one`** (exactly one element). Atomic formulas can be combined with the standard Boolean operators and quantifiers.

The `WellFormed` fact shown in Fig. 1 ensures that the task model forms a tree. The first constraint forces the root task to have no parents, specified as **`no`** `Root.parent`. The next constraint quantifies over `Task - Root` to ensure that non-root tasks have exactly one parent. The next one uses the *transitive closure* operator (`^`) to ensure that the `parent` relationship is acyclic. Although the static semantics of HAMSTERS is not entirely clear about additional restrictions to the structure of task models, we included several additional ones in `WellFormed`, mostly based on similar restrictions that exist in CTT. For example, we require composite tasks to have at least two (non-duplicate) sub-tasks, and only allow *Optional* sub-tasks in *Concurrent* and *Sequence* composite tasks.

2.2 Behavioral semantics

Since we found no formal description of the HAMSTERS behavioral semantics in the literature, we mainly based our formalization on our experience with the available task model simulators. The most recent version 6 of Alloy [12,5] added explicit support for behavioral specifications, allowing signatures and fields to be declared as mutable (with keyword **`var`**) and adding *Linear Temporal Logic* operators such as **`always`** or **`eventually`**.

Figure 2 presents an excerpt of the Alloy formalization of the behavioral semantics of HAMSTERS. The complexity of the semantics is mainly due to the fact that *Iterative* and sub-tasks of *Suspend* tasks (and consequently, their sub-tasks) can be performed multiple times; and that suspending and disabling tasks can interrupt other tasks at arbitrary points, the former allowing them to eventually resume. We rely on five mutable subsets of `Task` to manage the status of tasks in each state of the execution. We consider atomic tasks to *execute* atomically, and register those already `executed`, and composite tasks to *run* through several states as their sub-tasks are performed, and register the tasks that are `running`. Both executed and finished tasks can be reset if repeatable. Tasks that are `enabled` are those ready to execute (atomic) or run (composite). Tasks that already `finished` executing/running are also registered, as well as those that are `done`, those that have finished and are not repeatable.

The evolution of the set of `executed` atomic tasks is controlled by the first two constraints in the `Behavior` fact. It starts empty and afterwards it is always the case that either it does not change (no task is executed) or it changes according to one of two possible events (specified in separate predicates): either an enabled atomic task is executed and added to `executed` (note that `executed'` denotes the value of `executed` in the next state), or an enabled and already finished repeatable task is reset and it and all its descendant tasks are no longer considered to have been `executed` (the descendants are computed by applying the *reflexive transitive closure* operator ($*$) to `parent`).

The value of the four remaining variable subsets is specified by constraints in `Behavior` that define them by set comprehension, with many cases omitted due to space limitations. Set `enabled` only contains tasks whose parent is also enabled, and is further restricted according to the type of the task. For example, an atomic task is only enabled if not yet done and a sub-task of a *Choice* composite task is only enabled if none of its siblings is yet running. The `running` tasks are those not yet done but that already started, that is, have some descendant task that is already done. The set of `finished` tasks is again defined case-by-case. For example, atomic tasks finish immediately when they execute, and *Choice* tasks are considered to be finished when one of their sub-tasks is done. Finally `done` tasks are those that are non-repeatable and that have already finished.

With this formalization of the behavioral semantics it is already possible to check some general properties of task models or validate expected scenarios. In

```
var sig executed, enabled, running, finished, done in Task {}
pred execute [t : Atomic] { // Executing an atomic task
  t in enabled and executed' = executed + t }
pred reset [t : Task] { // Resetting a repeatable task
  t in enabled & (finished - done)
  executed' = executed - *parent.t }
pred nop { executed' = executed }
fact Behavior { no executed and always {
  // Possible events affecting the executed tasks
  nop or (some t : Atomic | execute[t]) or
         (some t : Task | reset[t])
  // The enabled tasks
  enabled = { t : Task | { ...
    // The parent is enabled and if atomic it cannot be done
    t.parent in enabled and (t in Atomic implies t not in done)
    // If inside a choice no sibling can be running
    some t.parent & Choice implies
        no (parent.(t.parent) - t) & running }}
  // The running tasks that already started but are not yet done
  running = { t : Task | t not in done and some ^parent.t & done }
  // The tasks that finished executing
  finished = { t : Task | { ...
    // An atomic task is finished if it already executed
    t in Atomic implies t in executed
    // A choice task is finished if some sub-task is done
    t in Choice implies some parent.t & done }}
  // The non-repeatable tasks that already finished
  done = { t : Task | t in finished - Iterative -
                      (parent.Suspend).succ }
} }
```

Fig. 2. HAMSTERS behavioral semantics

Alloy, **run** commands are used to ask for instances satisfying the specified assumptions and any additional scenario-specific constraints, and **check** commands to verify expected assertions. For decidability reasons, commands are bounded by a user-defined scope that limits the maximum number of elements inside signatures (3 by default) and the maximum number of transitions in the returned instance traces (`10` **steps** by default)[6]. For example, the following command, dubbed `Complete`, generates a task model where the root task is eventually done.

```
run Complete { eventually Root in done } for 1 but 2 steps
```

[6] In this paper we only use the bounded model checking engine of Alloy 6, but the Analyzer also supports unbounded model checking if NuSMV or nuXmv are installed, which is activated with the scope `1..` **steps**.

The defined scope limits the search to task models with at most 1 task and runs with 2 transitions, so the returned instance will be the smallest task model where the goal can be completed as fast as possible, namely one with a single atomic task that is immediately executed. We can also check that task models cannot deadlock, in the sense that while the root goal task is not done, one of the two events (`execute` an atomic task or `reset` a repeatable task) can still occur.

```
pred Deadlock {
  no t : Atomic | t in enabled
  no t : Task | t in enabled and t in finished - done }
check NoDeadlock {
  always (Root not in done implies not Deadlock) } for 6 but 3 seq
```

Note that this assertion will be checked for traces with up to 10 transitions, any possible task model with up to 6 tasks, and where each composite task has at most 3 sub-tasks (due to the scope 3 on `seq`, the size of sequences that are used to model the order of the sub-tasks), an enormous search space that takes 120s to verify with the SAT-based bounded model checking engine (with the Glucose SAT solver) in a commodity 2.3 GHz Intel Core i5 with 16 GB RAM. All commands in the paper were run on the same machine.

2.3 Composing concrete task models with system models

To verify properties of an interactive system where user actions are governed by a task model it is necessary to formally specify the system model and compose it with formal specification of the task model just presented. In Alloy, systems are specified in a style similar to the one used above to control the evolution of `executed` atomic tasks: mutable signatures and fields model the state of the system, and a fact constrains their initial state and which events are possible at each state. Events are typically specified in separate predicates, each with three kinds of formulas: *guards* that specify when is the event enabled, *effects* that specify which mutable structures change and how they change, and *frame conditions* that specify which mutable structures do not change. Each atomic task in the task model should have a corresponding event, and it is necessary to ensure that the former is only enabled when the guard of the latter holds, and that the execution of both is synchronized.

To support this composition, the Alloy HAMSTERS formalization was refined, adding a mutable field to atomic tasks that will determine in which states is the `guard` of the respective system event valid.

```
abstract sig Atomic extends Task { var guard : lone True }
one sig True {}
```

In every state of execution, field `guard` relates each atomic task with at most one element of the singleton signature `True`. Since Alloy has no pre-defined Boolean type this is a simple way to declare a Boolean mutable attribute: given a task `t`, the guard of the respective event is enabled iff `t.guard = True`. When adding specific tasks to the task model, the value of `guard` can be defined with a *signature* *fact*, an assumption defined alongside a signature declaration, which is implicitly universally quantified over all states. The specification of the value of the `enabled` mutable signature in Fig. 2 must also be refined to consider an atomic task enabled only when `t not in done and t.guard = True`.

Finally, to ensure the synchronization of the execution, a call to predicate `execute[t]` should be included in the specification of the system event corresponding to atomic task `t`. Since the guard will already be checked by this predicate, the specification of the event needs only to specify the effects and frame conditions. If an event can be triggered by multiple tasks, the call to `execute` can be replaced by the disjunction of multiple calls.

Suppose, for example, that an interactive system consisted only of two atomic tasks executed in sequence. Its specification would look as follows.

```
... // state declaration
one sig Goal extends Sequence {} { subtasks = 0→Task1 + 1→Task2 }
one sig Task1 extends Atomic {} { guard = True iff ... } // guard
one sig Task2 extends Atomic {} { guard = True iff ... } // guard
fact System {
  ... // initial state
  always (event1 or event2) }
pred event1 {
  execute[Task1]
  ... } // effects and frame
pred event2 {
  execute[Task2]
  ... } // effects and frame
```

Notice in the signature fact of the `Goal` *Sequence* task that the order of the subtasks is specified by stating which sequence index is mapped to each of them.

2.4 Adding erroneous behavior

In safety-critical interactive applications, it is often important to verify if expected properties still hold even in presence of user errors, i.e., interactions that do not conform to the defined task model. We will focus on user errors while executing *Input* tasks in *Sequence*, namely omission or duplication of required input tasks or performing them in a different order, although the impact of other kinds of errors could be explored with similar approaches. To account for user errors, our formalization needs to be further refined. First a subset signature of `Atomic` is added containing the `Erroneous` tasks, which in the `WellFormed` fact are further restricted to be `Input` tasks whose parent is a `Sequence` task. A mutable field `log` is also added to the `Sequence` composite tasks to record the actual sequence of tasks that was executed (which might differ from the sequence specified in `subtasks` due to user errors). This will allow us to later detect which errors occurred in an execution.

```
sig Erroneous in Atomic {}
fact WellFormed { ...
```

```
  all t : Erroneous | t in Input and some t.parent & Sequence }
abstract sig Sequence extends Composite { var log : seq Task }
```

The formalization of the behavioral semantics also needs to be adapted to allow user errors. In particular, the specification of `enabled` is changed to consider an atomic erroneous task enabled even if already executed. The conjunct that concerns atomic tasks is changed to the following:

```
t in Atomic implies
  (t in Erroneous or t not in done) and t.guard = True
```

Likewise, the restrictions for a sub-task of a *Sequence* to be enabled (omitted in Fig. 2) is relaxed to allow the execution of erroneous tasks out of order.

Finally, the specification of `execute`, `reset`, and `nop` is also changed to consider their effect on the `log` mutable field. For example, `reset` should clear the `log` of the reset task and all its descendant tasks and maintain the `log` of the remaining sequence tasks.

```
pred reset [t : Task] {
  ...
  all x : *parent.t | no x.log’
  all x : Sequence - *parent.t | x.log’ = x.log }
```

3 The AMAN case study

In this section we will show how the presented HAMSTERS formalization and system composition technique can be applied to a case study related to air traffic control[7], namely a semi-autonomous *Arrival MANager* (AMAN) [17].

3.1 Task model

In [17], the interaction of the air traffic controller (ATCo) with AMAN is described by a HAMSTERS task model. This task model includes numerous perceptive user tasks that have no direct impact on the interaction with the system, so in this section we will consider a simplified version, presented in Fig. 3, that includes mainly *Interactive* (both input and/or output) and *System* tasks. We will not describe the graphical notation of HAMSTERS (see [15]) but we believe it will be easy to grasp given the description bellow.

The ATCo task of managing the *Landing Sequence* (LS) can be suspended every 10s by the AMAN autonomous activity, which is a sequence of 3 system tasks where updated information about the planes is received by the radar and a new LS is computed and displayed. In abstract terms an LS is an assignment of planes to landing time slots, that respects some safety requirements concerning the separation of planes. The current LS is displayed in a user interface that

[7] The full HAMSTERS and AMAN Alloy models are available at `https://github.com/nmacedo/HAMSTERS-Alloy`.

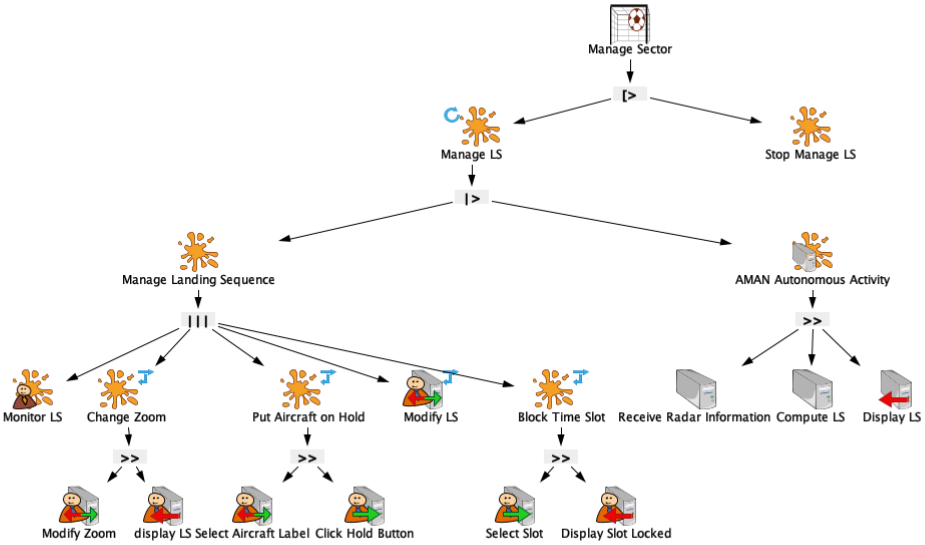

Fig. 3. Simplified AMAN task model (only *Interactive* and *System* tasks)

shows each plane as a label next to the assigned slot in a timeline. While monitoring the LS, the ATCo may concurrently modify the zoom level of this timeline, meaning that only a prefix of the LS is shown at any time. The ATCo may also modify the LS manually, select planes to put on hold, or block some time slots where no plane can be assigned. Finally, this iterative (and interactive) activity of managing the LS can be terminated at any time by the disabling stop task.

As explained in Section 2.3, formalizing a concrete task model in Alloy requires declaring singleton extensions of the appropriate types, and for composite tasks specifying the order of the respective sub-tasks. It is also necessary to specify the *Iterative*, *Optional*, and *Interactive Input* tasks. Figure 4 presents a snippet of this formalization for the AMAN.

3.2 Interactive system model

Since we are interested in exploring and analyzing design alternatives, our model of the AMAN system will be purposely abstract, as desired in such an early phase of the development so that the conducted analysis applies to many possible different implementations. The structures that characterize such abstract view of the AMAN state are presented in Fig. 5. Two immutable signatures are declared to represent the `Planes` and the different time `Slots`. The latter are totally ordered using the pre-defined module `util/ordering`. Mutable field `slot` represents the current LS, associating each plane to at most one time slot. Mutable field `label` represents the labels currently displayed on screen. Labels are not explicitly modeled, so `label` directly associates slots with the plane shown in the label. The state also includes six mutable subset signatures of `Plane` and/or

```
one sig ManageSector extends Disable {} {
  subtasks = 0→ManageLS + 1→StopManageLS }
one sig StopManageLS extends Atomic {}
one sig ManageLS extends Suspend {} {
  subtasks = 0→ManageLandingSequenceLS +
            1→AMANAutonomousActivity }
one sig AMANAutonomousActivity extends Sequence {} {
  subtasks = 0→ReceiveRadarInformation +
            1→ComputeLS + 2→DisplayLS }
one sig ReceiveRadarInformation, ComputeLS, DisplayLS
        extends Atomic {}
...
fact { Iterative = ManageLS
       Optional  = ChangeZoom + PutAircraftOnHold +
                   ModifyLS + BlockTimeSlot
       Input     = ModifyLS + ModifyZoom + SelectAircraftLabel +
                   ClickHoldButton + SelectSlot }
```

Fig. 4. Alloy formalization of the AMAN task model

```
open util/ordering[Slot]
sig Plane { var slot : lone Slot }
sig Slot { var label : set Plane }
var sig radar in Plane {}
var sig displayed, blocked in Slot {}
one var sig zoom in Slot {}
var sig holding, selected in Plane+Slot {}
```

Fig. 5. An abstract view of the AMAN state

`Slot`: the planes currently detected by the `radar`; the slots in the timeline prefix currently `displayed`; the `blocked` slots; the singleton selected `zoom` level, here represented by the last slot of the timeline that should be displayed; the planes that are `holding` and the slots where the label already displays the plane as `holding` (note that there is always a delay between computing or modifying the LS and updating the displayed information); and, finally, the slots or plane labels currently `selected` (either to block or put on hold, respectively).

Given these state declarations we can define the guards that enable the execution of the atomic tasks. For example, we could enable '*Stop Manage LS*' to occur only when there are no planes detected by the radar.

```
one sig StopManageLS extends Atomic {} { guard = True iff no radar }
```

Other atomic tasks that are guarded in our model are:

- '*Modify Zoom*' requires that there are at least two time slots.

- '*Select Aircraft Label*' requires non-holding planes to be currently displayed.
- '*Click Hold Button*' requires one plane label to be selected.
- '*Modify LS*' requires that there is some displayed plane label (to drag to a different position in the LS) and at least one free non-blocked slot on display.
- '*Select Slot*' requires that some non-blocked slot is currently displayed.
- '*Display Slot Locked*' requires one slot to be selected.

To give another example of the formalization of such guards, consider the one assigned to the '*Modify LS*' task.

```
some Slot.label and some displayed - Plane.slot - blocked
```

Finally the interactive system behavior should be formalized following the methodology described in Section 2.3. For each atomic task one system event should be specified, including a call to the execution of the respective task to ensure the proper synchronization between the task and the system. The same event may be associated to multiple tasks, which is the case of the two distinct '*Display LS*' tasks in the AMAN task model. To give an example of an event specification consider the '*Compute LS*' *System* task.

```
pred computeLS {
  execute[ComputeLS]
  // effects
  slot'.Slot = radar and no Plane.slot' & blocked
  all s : Slot | lone slot'.s
  Plane <: holding' = holding & radar
  // frame conditions
  radar' = radar and label' = label and ... }
```

Here we completely abstract the way the LS is computed, again leaving room for many different implementations. The only restrictions imposed on the new value of `slot` are that all planes detected by the radar are assigned a non-blocked slot and that at most one plane is assigned to each slot. Holding planes that are no longer detected by the radar are also removed from the `holding` set. All remaining mutable relations keep their value.

4 Analysis of the AMAN design

Design analysis should include both validation and verification. Our validation focused on exploring specific execution scenarios to rule out possible conflicts or inconsistencies in the interactive system specification. Then, verification checked some of the requirements listed in the case study documentation [17].

4.1 Scenario exploration

Alloy's **`run`** commands can be passed arbitrary formulas as constraints that must hold in the generated instances, which allows the user to loosely specify interesting scenarios. For instance, to inspect a scenario where eventually a time slot is displayed as holding, one could write the following command

Scenario	Description	Steps	Time
`Complete`	Goal is completed	2	3.2
`NotComplete`	Tasks keep running and goal is never completed	6	24.7
`AllExecute`	All atomic tasks are executed at least once	15	82.8
`SomeHolding`	Some plane label is displayed as holding	9	23.2
`OmitError`	An input task is erroneously omitted	11	42.6
`RepeatError`	An input task is erroneously repeated	4	7.1
`ReorderError`	A sequence task is executed in the wrong order	12	47.2

Table 1. Generated scenarios, time in seconds

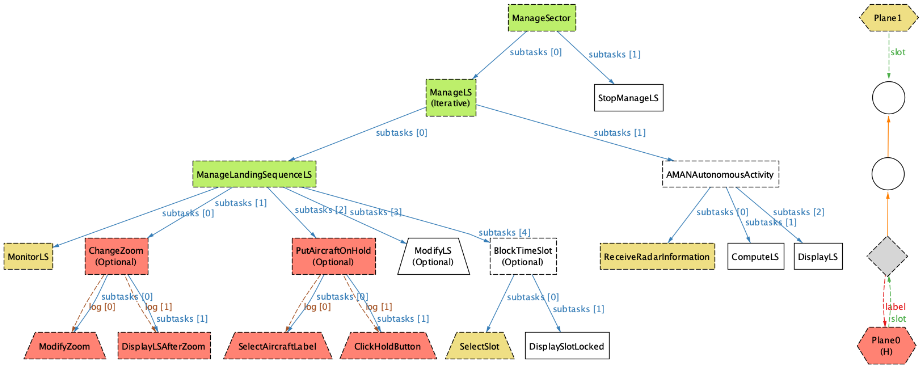

Fig. 6. Alloy theme for the AMAN case study in a state of `SomeHolding`

```
run SomeHolding {
  no Erroneous and eventually (some Slot & holding)
} for 3 but 5 seq, 20 steps
```

which would present an AMAN execution trace with at most 20 states and at most 3 planes and slots. For a more complex scenario, suppose that we wish to inspect erroneous executions, identified through a finished *Sequence* task where sub-tasks are missing in the log (**`seq/Int`** denotes the available **`seq`** indexes).

```
run OmitError { eventually (
  some st : Sequence | st in finished
  some i : seq/Int, x : Task | st.subtasks = insert[st.log,i,x])
} for 3 but 5 seq, 20 steps
```

Table 1 summarizes the tested scenarios, including the minimal number of steps needed to generate them and the running time (in seconds). All commands were run with the default scope of 3 except for 5 **`seq`** and 20 **`steps`**. The first 3 commands refer only to HAMSTERS concepts (such as completing the goal, as shown in Section 2.2), and could be applied to other concrete task models.

Alloy depicts generated instances graphically, showing one transition of the trace at a time (two panes, with the pre- and post-state). The user can then navigate along the trace to inspect other states. To ease the visualization custom themes can be defined. We developed such a theme for the AMAN model, and

Fig. 6 shows an advanced state of a trace returned by `SomeHolding`. The position of the elements was manually positioned for better understanding (unfortunately the Alloy visualizer does not ensure that arcs do not overlap). Concerning the task model (left-hand side), tasks are colored according to their status: enabled atomic tasks in yellow, running composite tasks in green, and done tasks in red. *Iterative* and *Optional* tasks are marked with a label. Concerning the interactive system (right-hand side), LS time slots are shown with circles, with those displayed in the screen colored gray. The selected zoom level is shown as a lozenge, blocked slots as a double circle, and selected slots with an 'X' label. Planes are shown as hexagons, colored yellow if detected by the radar and red if holding; if the holding status is being displayed an 'H' is added to the plane label.

Similarly to simulators, the visualizer provides different scenario exploration operations which allow the user to iterate through alternative traces that conform to the executed command. For instance, the '*New Config*' operation searches for an alternative static configuration (here, the existing planes and slots) and '*New Fork*' for an alternative transition in the selected state (here, executing a different task). These operations were used extensively during validation.

4.2 Requirement verification

After thorough validation, we verified some desirable properties. Note that our approach focused on the analysis of the user interaction rather than an implementation of an AMAN system, so not all requirements from [17] are verifiable.

In the first phase we consider only interactions with the AMAN system that conform to the task model (i.e., without user errors). We started by specifying simple **`check`** commands that were expected to be false due to the delay between the tasks, such as whether holding planes are always within the radar (`HoldingInRadar`) or whether planes in the LS are always shown in the screen (`LabelsInLS`). Alloy indeed showed these to be false, providing counter-examples that can be visualized and explored likewise the scenarios in Section 4.1.

Regarding the documented safety requirements [17], **Req5** – stating that labels cannot overlap in the LS – holds for our system (`NoOverlap`, specified as `all s : Slot | lone s.label`). We faced some difficulties when formalizing **Req6** – that no planes can be moved to blocked slots. Although it seems to restrict the action of "moving a label", we tried to interpret it as an invariant on the state of the system. The AMAN described in [17] allows planes to be in blocked slots until the next AMAN iteration executes. Thus, a simple interpretation of **Req6** (`NoLabelsBlockedA`, specified as `no p : Plane | some label.p & blocked`) is obviously false. In fact, **Req6** must state that the inconsistency is temporary, and that the plane will eventually be moved from the blocked slot (`NoLabelsBlockedB`):

```
always eventually (no p : Plane | some label.p & blocked)
```

This means that **Req6** is a *liveness* property – eventually a desirable state will be reached – that will be trivially false without imposing *fairness* constraints – that the model cannot stutter indefinitely if there are tasks left to do. Our

HAMSTERS formalization provides two predicates, `WF` and `SF`, that the user can call to enforce *weak* and *strong* fairness on the execution of tasks, respectively, according to standard semantics. For instance, `WF` states that atomic tasks cannot be permanently enabled without being executed, and that *Iterative* tasks cannot be permanently waiting to be reset for another iteration. For `NoLabelsBlockedB` to hold, it suffices to enforce weak fairness, so that the AMAN autonomous activity eventually updates the LS and recovers consistency.

Regarding interaction requirements [17], it is clear that not all task sequences allowed by the task model are feasible in our formalized interactive system, as seems to be required by **Req14**. It suffices to consider the pre-conditions on the events of the composed system. Instead, we explored other properties related to the availability of tasks. We were able to check that our AMAN task model never deadlocks – it never reaches a state without enabled tasks and the root goal still not completed (`NoDeadlock`). We were also able to check that whenever a non-holding plane is being displayed, it will eventually be possible to select it, another liveness property requiring strong fairness (`SelectAvailable`).

Lastly, for the automation requirements [17] we focused on **Req9** – that all inputs from the ATCo must have some graphical feedback. This can be formalized as a liveness property (`Feedback`) such as

```
all t : Input | always (execute[t] implies eventually DisplayChanges)
```

where `DisplayChanges` tests whether there were any changes in the variables related to the AMAN display. Interestingly, even when enforcing fairness, this property does not hold. After inspecting the returned counter-example it became clear why: if the ATCo selects a plane to put on hold but modifies the zoom level before clicking the hold button, he may never get visual feedback about the plane changing to the holding status. One may try to add an additional pre-condition to '*Click Hold Button*' to require the selected plane to be visible. This would fix `Feedback` but introduce other problems: the non-visible plane cannot be unselected, breaking availability properties like `SelectAvailable`. We also tried to change the interactive system by having '*Click Hold Button*' automatically zoom out to show the selected plane; but another counter-example is found where the autonomous AMAN task starts, before the ATCo presses hold, that no longer detects the plane in the radar and does not display it to the ATCo. The most direct way we could envision to fix this issue requires several changes: enforcing the pre-condition mentioned above on '*Click Hold Button*'; enforcing a pre-condition on '*Select Aircraft Label*' to forbid the selection of planes already selected; and breaking down '*Put Aircraft On Hold*' to not force every '*Select Aircraft Label*' to be followed by a '*Click Hold Button*'. This allows '*Select Aircraft Label*' to be executed again to unselect a previously selected plane, finally guaranteeing that all our assertions hold.

Table 2 summarizes the checked assertions, with the minimal number of steps of the counter-examples for the invalid ones and the running time (in seconds). All commands were run with the default scope of 3, 5 **`seq`** and 20 **`steps`**.

The last issue we addressed was the robustness of our system against ATCo errors. To this purpose, we ran all **`check`** commands again but this time allowing

Assertion	**Description**	**Steps**	**Time**
`HoldingInRadar`	All holding planes are detected by the radar	8	19.2
`LabelsInLS`	All the displayed labels are part of the LS	7	15.6
`NoOverlap`	Labels should not overlap (**Req5**)	unsat	114.6
`NoLabelsBlockedA`	No labels in blocked slots (**Req6**)	7	15.6
`NoLabelsBlockedB`	Labels will not stay in blocked slots (**Req6**)	unsat	539.9
`NoDeadlock`	The composed system does not deadlock (**Req14**)	unsat	90.8
`SelectAvailable`	Selecting planes to hold is always possible (**Req14**)	unsat	567.6
`Feedback`	Input tasks always have some feedback (**Req9**)	12	132.1

Table 2. Verified assertions (without erroneous tasks), time in seconds

erroneous tasks as described in Section 2.4. Interestingly, all commands that held remain valid even in this scenario. This means that our very simple formalization of an AMAN system is resistant to user errors, in the sense that an ATCo acting outside the task model does not break the consistency of the system.

5 Related work

Despite our best attempts, we were unable to find a publicly available document describing the formal semantics of HAMSTERS. We reverse-engineered the semantics by interacting with the given HAMSTERS simulator and also referring to the semantics of CTT [19,18], a predecessor to HAMSTERS, when appropriate. In particular, our recursive definition to determine the enabled set of tasks seems to be very similar to the recursive algorithm implemented in CTTE [16], the most popular CTT design and simulation tool. Although we believe that our formalization is reasonable, it is possible that it differs from what the designers of the HAMSTERS notation intended; the outcome of our analysis is thus also contingent on the fidelity of our model.

The composition (or *coupling*) of task models and system models, to show the consistency of the prescribed interaction model, has been proposed before. In particular, techniques have been proposed for the co-execution of HAMSTERS task models with Petri Net system models [1] or with actual interactive applications [13]. These techniques allow only to validate the consistency of the coupled system, while our Alloy-based technique can also be used to also verify requirements with model checking.

The consistency of task models and interactive applications can also be checked by first generating scenarios from task models to be latter run in the target application. For HAMSTERS, a technique for scenario generation has been proposed [6] that first generates a state machine from a task model, and then uses standard graph traversals to generate possible sequence of tasks that can complete the goal. To keep the number of generated scenarios under control, this work was latter extended with a technique to manipulate the task model prior to generating the state machine, with the goal of guiding the generation of scenarios to those more relevant for the system under analysis [7]. Our Alloy specification of HAMSTERS task models could also be used for directly generating relevant scenarios, since, as shown in Section 4.1, `run` commands can be used to generate scenarios satisfying given constraints.

Techniques to analyze the impact of user errors on task models have also been studied before [3,9,10]. In particular, an extension of the HAMSTERS notation has been proposed to explicitly describe possible user errors [10]. Our formalization only handles some of the user errors that can described in this extension, namely slips and lapses in sequence tasks. Methods have been proposed for analyzing the impact of user errors on CTT [20] or HAMSTERS task models [14]. Unlike these manual techniques, our Alloy-based technique allows the automatic analysis of the impact of user errors in HAMSTERS task models.

6 Conclusion

This paper introduced a technique for task model design with Alloy, that enables the automatic validation and verification of the coupling of a HAMSTERS task model with a state-based system model. The proposed technique also allows the automatic analysis of the impact of user errors. The technique was applied to the AMAN case study, helping us identify and propose fixes to flaws in the interaction design and in the system requirements.

Although this paper mainly focused on the application of the proposed technique to the AMAN case study, we believe that it has other potential utilities. First, our semantic model could be used as a backend for other tools that rely on the HAMSTERS notation – for example, by augmenting the HAMSTERS simulator with an ability to automatically generate sample scenarios or verify properties. By leveraging the capability of the Alloy Analyzer to exhaustively enumerate a set of instances, our approach could also be used to generate test cases from a task model and execute them to evaluate the underlying application. Finally, we plan to apply our approach to analyze task models in other safety-critical domains, such as medical devices [8] and automotive systems [4].

References

1. Barboni, E., Ladry, J.F., Navarre, D., Palanque, P., Winckler, M.: Beyond modelling: An integrated environment supporting co-execution of tasks and systems models. In: EICS. pp. 165–174. ACM (2010)
2. Ben Amor, M.: Hamsters: A new task model for interactive systems. Master's thesis, University of Namur (2009)
3. Bolton, M.L., Bass, E.J., Siminiceanu, R.I.: Generating phenotypical erroneous human behavior to evaluate human–automation interaction using model checking. International Journal of Human-Computer Studies **70**(11), 888–906 (2012)
4. Bolton, M.L., Siminiceanu, R.I., Bass, E.J.: A systematic approach to model checking human–automation interaction using task analytic models. IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans **41**(5), 961–976 (2011)

5. Brunel, J., Chemouil, D., Cunha, A., Macedo, N.: The Electrum Analyzer: Model checking relational first-order temporal specifications. In: ASE. pp. 884–887. ACM (2018)
6. Campos, J.C., Fayollas, C., Martinie, C., Navarre, D., Palanque, P., Pinto, M.: Systematic automation of scenario-based testing of user interfaces. In: EICS. pp. 138–148. ACM (2016)
7. Campos, J.C., Fayollas, C., Gonçalves, M., Martinie, C., Navarre, D., Palanque, P., Pinto, M.: A more intelligent test case generation approach through task models manipulation. Proceedings of the ACM on Human-computer Interaction **1**(EICS), 1–20 (2017)
8. Campos, J.C., Harrison, M.: Modelling and analysing the interactive behaviour of an infusion pump. Electronic Communications of the EASST **45** (2011)
9. Cerone, A., Lindsay, P.A., Connelly, S.: Formal analysis of human-computer interaction using model-checking. In: SEFM. pp. 352–362. IEEE Computer Society (2005)
10. Fahssi, R., Martinie, C., Palanque, P.: Enhanced task modelling for systematic identification and explicit representation of human errors. In: INTERACT. LNCS, vol. 9299, pp. 192–212. Springer (2015)
11. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, revised edn. (2016)
12. Macedo, N., Brunel, J., Chemouil, D., Cunha, A., Kuperberg, D.: Lightweight specification and analysis of dynamic systems with rich configurations. In: SIGSOFT FSE. pp. 373–383. ACM (2016)
13. Martinie, C., Navarre, D., Palanque, P., Fayollas, C.: A generic tool-supported framework for coupling task models and interactive applications. In: EICS. pp. 244–253. ACM (2015)
14. Martinie, C., Palanque, P., Fahssi, R., Blanquart, J.P., Fayollas, C., Seguin, C.: Task model-based systematic analysis of both system failures and human errors. IEEE Transactions on Human-Machine Systems **46**(2), 243–254 (2015)
15. Martinie, C., Palanque, P.A., Winckler, M.: Structuring and composition mechanisms to address scalability issues in task models. In: INTERACT. LNCS, vol. 6948, pp. 589–609. Springer (2011)
16. Mori, G., Paternò, F., Santoro, C.: CTTE: Support for developing and analyzing task models for interactive system design. IEEE Transactions on Software Engineering **28**(8), 797–813 (2002)
17. Palanque, P., Campos, J.C.: AMAN case study (2022)
18. Paterno, F.: Model-based design and evaluation of interactive applications. Springer (1999)
19. Paternò, F., Mancini, C., Meniconi, S.: ConcurTaskTrees: A diagrammatic notation for specifying task models. In: INTERACT. IFIP Conference Proceedings, vol. 96, pp. 362–369. Chapman & Hall (1997)
20. Paterno, F., Santoro, C.: Preventing user errors by systematic analysis of deviations from the system task model. International Journal of Human-Computer Studies **56**(2), 225–245 (2002)
21. Thimbleby, H.: Fix IT: How to see and solve the problems of digital healthcare. Oxford University Press (2021)