To Appear in the Proceedings of the Sixth IAASS Conference. Montreal: IAASS. 2013

FORMALLY VERIFYING HUMAN-AUTOMATION INTERACTION WITH SPECIFICATION PROPERTIES GENERATED FROM TASK ANALYTIC MODELS

M. L. Bolton(1), **N. Jimenez**(2), **M. M. van Paassen**(3), **and M. Trujillo**(4)

(1)*Department of Mechanical and Industrial Engineering, University of Illinois at Chicago, 2039 Engineering Research Facility, 842 W. Taylor Street, Chicago, Illinois 60607, USA, Email: MBolton@UIC.edu*
(2)*IXION Industry and Aerospace, Calle de Julián Camarillo 21B, 28037 Madrid, Spain, Email: NJimenez@IXION.es*
(3)*Aerospace Engineering, Delft University of Technology, Postbus 5, 2600 AA Delft, The Netherlands, Email: M.M.vanPaassen@TUDelft.nl*
(4)*European Space Research and Technology Centre, Keplerlaan 1, Postbus 299, 2200 AG Noordwijk, The Netherlands, Email: Maite.Trujillo@ESA.int*

ABSTRACT

Human-automation interaction (HAI) is a major contributor to failures in aerospace systems, often due to unexpected interactions between system components. Formal verification is a method of analysis that exhaustively examines all of the interaction in a formal system model to determine if it adheres to desirable specification properties. Task analytic models can be included in formal system models to allow HAI to be evaluated with formal verification. However, previous work required analysts to manually formulate specification properties they wished to check. A practice which can result in unexpected, dangerous HAI conditions not being checked. This paper presents a method for generating specification properties from task models to allow analysts to automatically check for HAI problems they may not have anticipated. The paper describes the design and implementation of the method. An example (a pilot performing a before landing checklist) is presented to illustrate its utility.

1. INTRODUCTION

Human behavior is a major contributor to failures in aerospace systems. It has been a factor in more than 50% of commercial aviation accidents [15], 70% of general aviation accidents [16], and many failures in space operations [18]. These problems arise as a result of complex interactions between system components (human operators, device automation, and conditions in the operational environment) [23]. Human behavior can be particularly problematic because the concurrency between it and the system automation can result in unanticipated human-automation interaction (HAI) [26].

While a number of advances have been made to address this issue [26], analysts can still miss problems because most analysis approaches are incapable of evaluating all of the possible HAIs. However, formal verification techniques, and particularly model checking, offer means of performing such robust analyses.

1.1. Formal Verification and Model Checking

Formal verification comes from the field called formal methods. Formal methods are mathematically robust languages, techniques, and tools for the modeling, specification, and verification of systems [28]. Formal modeling is concerned with mathematically describing the behavior of a target system, specifications assert desirable properties about the system, and verification mathematically proves whether or not the model adheres to the specification. This work uses model checking, a computer software tool capable of automatically performing formal verification [10]. In model checking, system behavior is typically modeled as a finite state transition system: a collection of variables and guarded transitions between variable values (states). Temporal logics [13] are usually used to express specification properties using a combination of model variables, binary logic operators, and temporal operators. A model checker performs formal verification by exhaustively searching a system's statespace to determine if the specification holds. If there is a state in the model that violates the specification, a counterexample is produced which represents a counterproof: a listing of a path of incremental model state transitions that led up to the violation.

1.2. Formal Verification and HAI

Formal verification is typically used in the analysis of computer hardware and software [10, 28]. However, it has also been used to evaluate HAI (see [8] for a review). The method presented here is concerned with formal verification work that uses task analytic models. Task analytic behavior models are a common tool of human factors engineers. They are produced as part of a cognitive task analysis [17, 25] and describe the behaviors human operators use to achieve goals with a system. The meaning of these models can be represented computationally, enabling them to be included in a formal system model containing a formal description of the other relevant system behaviors. Formal verification is then used to evaluate the impact of the modeled behavior (which can be normative or erroneous) on system safety [1–4, 6, 7, 9, 14, 21, 22]. This is a powerful approach because it explicitly models the human behavior, and thus the HAI; gives analysts a clear indication of what the human operator was doing in discovered problems (counterexamples); and uses models of human behavior commonly employed by human factors and systems engineers [7].

These methods tends to focus on the verification of analyst-created properties that assert qualities critical to a system's safe operation. There are limitations to formulating specification properties in this manner. Firstly, temporal logics can be difficult to learn and interpret. This can result in analysts incorrectly formulating properties. Secondly, this approach requires that analyst anticipate potentially unsafe conditions and formulate them as specification properties. Therefore, if an analyst does not anticipate a potentially unsafe condition or problems with HAI, formal verification will give them no insights into those potential failures. Finally, these analyses focus on verifying safety properties, the violation of which could result from the systems HAI. In this respect, analysts are using formal verification to look for specific failure conditions that may be associated with HAI rather than the HAI problems themselves.

1.3. Objectives

In the work presented here, we discuss a method that addresses these limitations. Specifically, we extend an existing method [3], which supports the formal verification of HAI using task behavior models, to automatically generate specification properties. We would ideally be able to generate properties related to all elements of a system. However, most of the elements of a formal system model are designed to reflect the specific behavior of the target system and thus do not retain enough architectural similarities between applications to make this a reasonable goal. However, given the nature of hierarchical task analytic models, the task models used in these analyses do follow a regular structure and execution pattern. Thus, our method focuses on generating specification properties from the task analytic models themselves (Fig. 1). The property generation process uses the above insight that issues with HAI are problems of concurrency, where generated properties are designed to systematically check for incompatibilities between the concurrent execution of the human task and the other elements of the system. Because task models are represented computationally in formal verification analyses, we can use concepts from computation to reason about their execution in a human-automation interactive system. A model checker can be used to verify the generated properties against the system model to potentially find HAI problems.

The remainder of this document describes how the method (Fig. 1) was realized. We first describe the Enhanced Operator Function Model (EOFM), the task analytic modeling formalism used in this work, and the formal verification analysis method it supports. Linear temporal logic (LTL), the logic used for representing generated specification prosperities, is also described. We then show how concepts from computation can be used with EOFM to generate specification properties capable of finding problems with HAI. We present a simple application of a pilot performing the before landing checklist of an aircraft to demonstrate the use of our approach. Finally, we discuss our results and explore avenues of future research.

2. MODEL CHECKING HAI WITH EOFM

The Enhanced Operator Function Model (EOFM) [9] is an XML-based human task modeling language, derived from the Operator Function Model (OFM) [20], specifically designed to allow task analytic human behavior to be evaluated with formal methods. EOFMs are largely hierarchical and partially heterarchical representations of goal driven activities that decompose into lower level activities, and finally, atomic actions. A decomposition operator specifies the temporal relationships between and the cardinality of the decomposed activities or actions (when they can execute relative to each other and how many can execute). In the application presented here, only the *ord* operator (which asserts that activities or action must execute in a specific order) is used.

EOFMs express strategic knowledge explicitly as conditions on activities. Conditions can specify what must be true before an activity can execute (preconditions), when it can repeat execution (repeat conditions), and what is true when it completes execution (completion conditions).

EOFMs can be represented visually as tree-like graphs [5] (see an example in Fig. 3). Actions are rectangles and activities are rounded rectangles. An activity's decomposition is presented as an arrow, labeled with the decomposition operator, that points to a large rounded rectangle

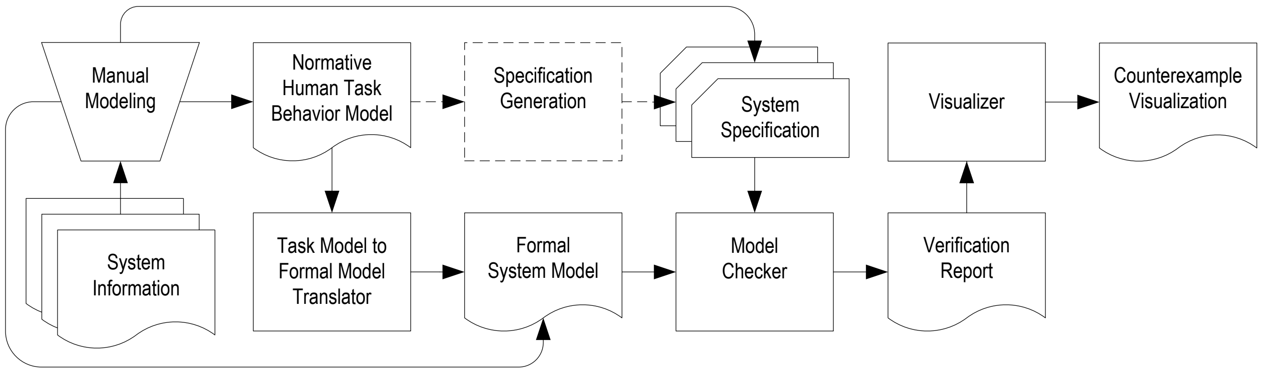

Figure 1. The formal verification method supported by EOFM. Continuous lines indicate parts of process previously supported by EOFM [3]. Dotted lines represent the novel specification property generation process.

containing the decomposed activities or actions. Conditions (strategic knowledge) on activities are represented as shapes or arrows (annotated with the logic) connected to the activity that they constrain. The form, position, and color of the shape are determined by the type of condition. A precondition is a yellow, downward-pointing triangle; a completion condition is a magenta, upward-pointing triangle; and a repeat condition is an arrow recursively pointing to the top of the activity.

EOFM has formal semantics which specify how an instantiated EOFM model executes (Fig. 2). Each activity or action has one of three execution states: waiting to execute (*Ready*), executing (*Executing*), and done (*Done*). An activity or action transitions between each of these states based on its current state; its start condition (*StartCondition* – when it can start executing based on the state of its immediate parent, its parent's decomposition operator, and the execution state of its siblings); its end condition (*EndCondition* – when it can stop executing based on the state of its immediate children in the hierarchy and its decomposition operators); its reset condition (*Reset* – when it can revert to *Ready* based on the execution state of its parents); and, for an activity, the activity's strategic knowledge (the *Precondition*, *RepeatCondition*, and *CompletionCondition*). See [9] for more details.

EOFM supports formal verification using the process shown in Fig. 1. Analyst-created EOFM task models can be automatically translated [9] into the language of the Symbolic Analysis Laboratory (SAL) [11] using the EOFM formal semantics. The translated EOFM can then be integrated into a larger system model (also created manually) using a defined architecture and coordination protocol [4, 9]. Formal verifications of manually created specification properties are performed on this complete system model using SAL's Symbolic Model Checker (SAL-SMC). Any produced counterexamples can be visualized and evaluated using EOFM's visual notation [5].

3. LTL SPECIFICATION

Because the formal verification method supported by EOFM (Fig. 1) uses SAL's symbolic model checker, specification properties must be asserted using linear temporal logic (LTL). LTL uses propositional variables, Boolean logic operators ($\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow, =, \neq, <, >$, etc.), and temporal operators (Tab. 1) to assert properties about all paths through a model [13].

Because LTL can only be used to specify properties about all paths through a model, it cannot be used to positively assert the existence of a desirable system condition that may not exist on all paths. Thus, to conduct an existence proof with a model checker that uses LTL specifications, a negative assertion must be specified. Let ϕ represent a temporal logic proposition that we want to prove exists in a system. Using LTL, we can use the specification

$$\mathbf{G}\neg(\phi) \tag{1}$$

to assert that ϕ should never be true in all paths through the model. If we use a model checker to check Eq. 1

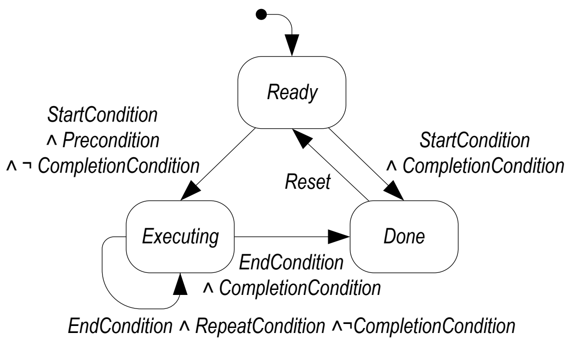

Figure 2. Formal semantics of an EOFM activity's and action's execution state presented as finite state transition systems [9]. States are represented as circles. Transitions appear as arrows between states that are labeled with Boolean expressions. Arrows starting with a dot point to initial states. For actions, the Precondition, RepeatCondition, CompletionCondition, and Ready to Done and Executing to Executing transitions are not used.

against a system model, it will return true if ϕ is never satisfied. However, if ϕ does exist, the model checker will return a counterexample illustrating how ϕ was realized. This trick will be used in specification property generation, where both positive (where the specification asserts the existence of the desirable property) and negative (where the specification asserts the absence of the desirable property) specifications will be used.

4. SPECIFICATION GENERATION

There are a number of properties that a system with well designed HAI will require from its human operator's expected behavior (his or her task). For the purpose of this work, we consider all of the following:

1. Every element of the task should be applicable at some time in the use of the system. If not true, then the task either contains superfluous behaviors or there is a problem with the HAI that prevents all or part of a task from ever becoming relevant.
2. Every task that a human operator attempts should always be finishable. If not true, the task is incapable of always achieving its goals implying problems with the task or deficiencies in other elements of the system.
3. There should never be a situation where the human operator can never perform any task. If not true, there are system states for which human task behavior has not been defined. This could indicate an incomplete description of human task behavior or an unexpected system state.

Because the method supported by EOFM (Fig. 1) treats human task behavior as a computational structure (using the EOFM formal semantics, Fig. 2), task analytic models can be reasoned about computationally. Luckily, all three of the above items we want to ensure map to concepts from computation and can be expressed using linear temporal logic specifications.

Table 1. Linear temporal logic operators.

Operator	Usage	Interpretation
Global	$\mathbf{G}\,\psi$	ψ will always be true.
Ne**X**t	$\mathbf{X}\,\psi$	ψ is true in all of the next states.
Future	$\mathbf{F}\,\psi$	ψ is eventually true in future state.
Until	$\phi\,\mathbf{U}\,\psi$	ϕ will be true until ψ is true.

Note. ϕ and ψ are propositions about either a state or path in the model that can evaluate to either true or false, where a path is a valid temporally ordered sequence of states.

4.1. Item 1: Coverage

Item 1 refers to the computational concept of coverage. A coverage criterion represents the extent to which elements of a computational structure are reachable [24]. In this work, we are concerned with state coverage. State coverage asserts that every state in a finite state machine is reachable. State coverage applies to EOFM task models through the execution state of its activities and actions (Fig. 2). Thus, to ensure state coverage, the execution state of every activity and action must be reachable.

To ensure that state coverage is maintained, properties are generated to check that every activity and action in a task model is capable of reaching each of the three execution states (Fig. 2). Because every activity and action automatically starts in the *Ready* state, there is no need to check that it is reachable. However, we must generate properties to specify that *Executing* and *Done* are reachable. Because these will be reachability properties, they must be asserted with the pattern from Eq. 1. To check that every activity and action in a task model can execute, we generate a property (called *Act Executability*) of the form shown in Eq. 2 (Tab. 2) for each activity or action in the task structure. To check that every activity or action can reach the *Done* state (a property we call *Act Completability*), we generate a property of the form in Eq. 3 (Tab. 2) for each activity or action in an instantiated EOFM. Since both *Act Executability* and *Act Completability* are negatively asserted, this means that if the *Executing* and *Done* states are reachable, Eqs. 2 and 3 will produce counterexamples when checked. If *Executing* and/or *Done* are **not** reachable, the model checker will return true.

4.2. Item 2: Starvation

While knowing that a task can execute is useful, we also need to ensure that it will always finish (item 2). Using computation terminology, this means that we never want our task models to suffer starvation: an inability to gain the necessary resources to finish [27]. To check that a task will never starve, we generate a property called *Act Inevitable Completability* (Eq. 4; Tab. 2) for each activity and action. This asserts that, when the act is executing, it must eventually finish. Because *Act Inevitable Completability* is asserted positively, a model checker will return true if it is satisfied. Otherwise, a counterexample will illustrate how a state or cycle was reached from which the associated act could not stop executing.

4.3. Item 3: Liveness

Liveness describes a condition where something desirable will eventually occur [10]. For HAI-dependent systems, we want to ensure that there is never a situation where the human operator can never perform a task (item 3). To check this, we generate a property of the form in Eq. 5 called *Task Liveness* (Tab. 2). *Task Liveness* is asserted positively. Thus, a model checker will return a confirmation if it is satisfied. Otherwise, a counterexample will show how the violation occurred.

4.4. Accounting for Uninteresting States

In checking *Act Inevitable Completability* and *Task Liveness*, it is conceivable that there could be states in the model where either is violated in ways that are not interesting to analysts. For example, a model may have an "end state" at or after which these properties would not be important. To accommodate this, the analyst will need to check modified forms of these properties. Thus, we can reformulate Eqs. 4 or 5 as

$$\psi \Rightarrow \left(\bigwedge_{\phi \in \Phi}^{\forall \Phi} \neg \phi \right) \tag{6}$$

where ψ is the original specification (either Eq. 4 or 5), Φ is the set of expressions representing states an analysts wishes to exclude, and ϕ is a particular state excluding expression. This can be interpreted as: if ψ is true, then all of the expressions in Φ are false.

4.5. Implementation

The EOFM to SAL translator was modified to automatically generate all of the properties from Tab. 2 for use in model checking with SAL.

5. APPLICATION

To illustrate how this method can be used to find problems in a HAI-dependent aerospace system, we present an application: a pilot attempting to perform the before landing checklist (previously discussed in [6]). In this application, a pilot is performing an instrument approach where he or she is navigating the aircraft to the runway using vertical guidance (the glide slope). The vertical position of the aircraft relative to the glideslope is displayed with a moving diamond on the glideslope indicator. When the aircraft is nearing the glideslope, the diamond becomes "alive", moving towards the center of the display. The diamond will first pass through the "two dot" and then the "one dot" positions. When the aircraft is on the glideslope, the diamond is at the capture position.

To land safely, the pilot performs the before landing checklist. Herein this means [12]: (a) the ignition must be set to override to allow for engine restart should the engine quit; (b) the landing gear must be down; (c) the spoilers should be armed; and (d) the flaps should be extended to the appropriate flap setting (first 25° and then 40°) to slow the aircraft and prevent stalling.

Table 2. Task Model Specification Properties to Support Safe Human-automation Interaction

Name:	Act Executability	
Description:	A given activity or action should be able to execute.	
Formulation:	$\mathbf{G}\neg(Act = Executing)$	(2)
Interpretation of a Confirmation:	× There are no conditions where *Act* can ever execute.	
Interpretation of a Counterexample:	✓ There are conditions where *Act* can execute.	
Name:	Act Completability	
Description:	A given activity or action should be able to be done.	
Formulation:	$\mathbf{G}\neg(Act = Done)$	(3)
Interpretation of a Confirmation:	× There are no conditions where *Act* can ever be done.	
Interpretation of a Counterexample:	✓ There are conditions where *Act* can be done.	
Name:	Act Inevitable Completability	
Description:	Every activity or action that is executing must eventually finish.	
Formulation:	$\mathbf{G}\left((Act = Executing) \Rightarrow \mathbf{F}(Act \neq Executing)\right)$	(4)
Interpretation of a Confirmation:	✓ *Act* can always finish executing.	
Interpretation of a Counterexample:	× There is a least one condition where *Act* can never finish executing.	
Property Name:	Task Liveness	
Description:	There should never be a situation where no activity can ever execute.	
Formulation:	$\mathbf{G}\neg\left(\mathbf{F}\left(\mathbf{G}\left(\bigwedge_{Act\in RootActivities}^{\forall RootActivities} Act \neq Executing\right)\right)\right)$	(5)
Interpretation of Confirmation:	✓ There is never be a situation where no activity can ever execute.	
Interpretation of Counterexamples:	× There is a situation where no activity can ever execute.	

Note. *Act* represents the execution state of a given activity or action from an instantiated EOFM. *RootActivities* represents the set of all top level activities. A ✓ and × indicate if the associated verification outcome is desirable or undesirable respectively.

Spoilers are retractable plates on the wings that, when deployed, slow the aircraft and decrease lift. A pilot can arm the spoilers for automatic deployment using a lever. If spoilers are not used, the aircraft can overrun the runway. If spoilers are deployed too early, the aircraft loses lift and could have a hard landing. Premature deployment can occur due to mechanical issues. Arming the spoilers before the landing gear has been lowered, before the landing gear doors have fully opened, or during landing can result in automatic premature deployment [12]. For this reasons, pilots wait to arm the spoilers until after landing gear has been deployed and the landing gear doors have completely opened (in our example, this can take between 10 and 18 seconds due to variability in the hydraulics).

5.1. Task Modeling

The task behavior for performing the before landing checklist was instantiated as an EOFM (Fig. 3). This task model assumes the pilot can observe the value of all of the following: the glideslope indicator (*GSIndicator*), the ignition indicator light (whether or not the ignition has been overridden; *IgnitionLight*), the angle of the aircraft's flaps as indicated on the flaps gauge (*FlapsGuage*), if the landing gear is down (*ThreeGearLights*), and if the landing gear doors are opening (*GearDoorLight*).

This model was converted into SAL's input language using the translator [9] with the newly added automatic specification generation. The specification generation process produced 34 properties: 33 properties representing *Act Executability*, *Act Completability*, and *Act Inevitable Completability* for the 11 activities and actions in the model; and one property representing *Task Liveness* for the entire EOFM instances.

5.2. Modeling the Rest of the System

To complete the formal system model, formal representations were created for the system operational environment, the aircraft's automation, and its human-automation interface.[1]

The system's operational environment was represented abstractly as the relative distance (*Position*) of the aircraft from the capture position on the glideslope. The aircraft starts at a position where the glideslope diamond is not alive. The aircraft proceeds to the capture position and begins to descend on the glideslope, a process that will take 18 seconds. Thus, the relative position of the aircraft from the initial position is discretized into intervals (0 to 18) where the aircraft passes from one interval to the next in one second. The model was set to meet a dedicated end state when it starts to descend (at position 18).

The formal model of the device automation represented the functionality of the aircraft's ignition, landing gear, spoilers, and landing gear doors. In the models the ignition starts out not in override, the landing gear starts out un-deployed, the spoilers unarmed, and the landing gear doors closed. The state of these properties could change in response to human actions received from the human-device interface or other environmental or internal system

[1]Due to space considerations, full details of these models are not reported. More information can be found in [6]. Full models can be found at `http://www.sys.uic.edu/resources/`

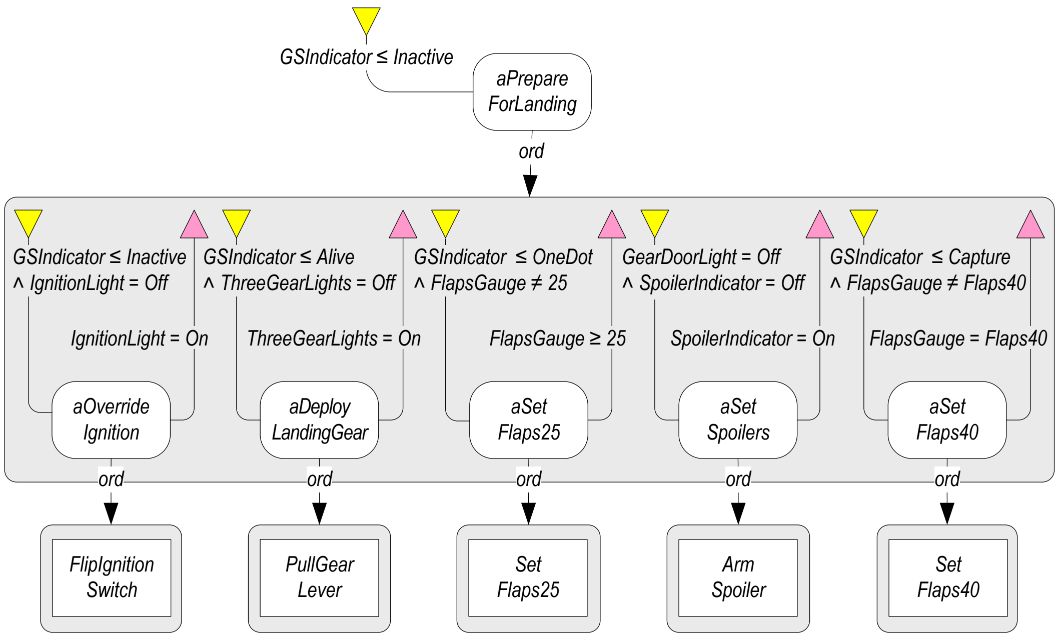

Figure 3. Visualization of the EOFM task model for the before landing checklist.

conditions. While most changes to pilot actions were effectively instantaneous (taking less than a second) landing gear doors would take between 10 and 18 seconds to fully transition from closed to open after landing gear deployment was initiated.

The formal model of the human-automation interface represented the state of the flightdeck controls and indicator lights associated with arming the spoilers, the landing gear doors, the landing gear, the flaps, the glideslope indicator, and the ignition. The human-automation interface would receive human actions (flipping the ignition switch, pulling the landing gear lever, setting the flaps, and arming the spoilers) and have them affect the behavior of the automation. As the state of the automation and environmental variables changed, the indicators associated with the iginition (*IgnitionLight*), spoilers (*SpoilerIndicator*), landing gear doors (*GearDoorLight*), landing gear (*ThreeGearLights*), flaps (*FlapsGuage*), and glideslope indicator (*GSIndicator*) would update to reflect these changes.

5.3. Formal Verification

Formal verification was performed on a PC laptop running Windows 7 and Cygwin 1.7 with an Intel i5-2467M CPU and 4 gigabytes of RAM. Using SAL's symbolic model checker, it took 14.6 seconds of total execution time to verify all 34 generated properties.

For all but four of the generated properties, the desirable verification results were obtained. Every *Act Executability* and *Act Completability* property returned the desired counterexample indicating that every part of the task model (Fig. 3) was capable of being reached (state coverage was achievable). However, three of the *Act Inevitable Completability* properties did not evaluate to true indicating that, for three acts in the task model (*aSetFlaps40*, *aSetSpoilers*, *aPrepareForLanding*), there were states where they would never finish executing. Further, the one *Task Liveness* property also produced a counterexample, the undesired outcome.

Examining the counterexamples for the failures of *Act Inevitable Completability* (using the EOFM counterexample visualization [5]) revealed a common problem. In all three, the landing gear doors took between 16, 17, and 18 seconds to fully deploy. This appeared to create a situation where the aircraft would start descending (the end state for the model) having just performed the action for setting the flaps to 40° (for the *aSetFlaps40* property) or arming the spoilers (for the *aSetSpoilers* property). For the failure of *aPrepareForLanding*'s property, the aircraft started descending before the human operator could arm the spoilers.

All three of these properties failed because the model reached the end state of the model before all the necessary activities could finish executing. More importantly, all three of these failures could constitute serious problems for the safe operation of the aircraft. For all of them, the aircraft reaches the capture position without having the flaps properly set. This could result in the aircraft going too fast as it starts to descend. The failure of *aSetSpoilers*' property constitutes a situation where spoilers are armed while the aircraft is just starting to descend, a condition that could lead to premature spoiler deployment. Finally, the counterexample associated with *aPrepareForLanding*'s property revealed a situation where the aircraft did not have its spoilers armed as it started to descend. If unnoticed by the pilot, this could lead to the aircraft overrunning the runway.

To test whether these failures were the result of the delay in the opening of the landing gear doors, the model was iteratively modified across multiple verification runs to see if there was a minimum delay that would remove these failures from the model. This showed that when the delay was 14 seconds or less, all three of these properties returned the desirable verification outcome.

Finally, an examination of the counterexample associated with the failure of the *Task Liveness* specification revealed that no tasks would ever execute once the aircraft reached the capture position. This is not surprising given that this constituted the end state of the model. Thus, to ensure *Task Liveness* held in all other states, we used the pattern shown in Eq. 6 to create the specification

$$\begin{aligned}\mathbf{G}\neg(\mathbf{F}(\mathbf{G}(aPrepareForLanding \neq Executing))) \\ \Rightarrow \neg(AircraftPosition = EndPosition).\end{aligned} \tag{7}$$

When this was checked, it verified to true. This indicated that the pilot would always be able to perform a task in all but the end state of the model.

6. DISCUSSION

In the work presented here, we have extended the formal verification method supported by EOFM (Fig. 1) with a novel means of generating specification properties from task models. By exploiting the computational representation of a task in a formal model, the method is capable of detecting if parts of task models are never used, if parts of them will never finish, and if there are situations where no tasks can ever be performed; all properties indicative of problems with HAI. The automatic nature of the method is also advantageous in that: (a) there is no risk of specification properties being manually misformulated; (b) analysts need not anticipate all of the HAI problems associated with the generated properties to check for them; and (c) specification properties are represented in terms of the task models meaning they detect problems with the system's HAI rather than just potentially problematic automation conditions.

The utility of the method was demonstrated with a realistic example: a pilot performing a before landing checklist. In this application, several problems with HAI were discovered and the results of the analysis were used to investigate the source of the failures. Impressively, the method discovered the problems associated with the pilot no adjusting the aircraft's flaps in time, something not discovered in previous analyses using this model [6].

It is important to note that the method is still compatible with the traditional verification analyses supported by EOFM [2–4, 6, 7, 9]. Thus, if an analyst has specific system safety conditions that he or she wants to check, that option is still available.

Despite its successes, the method does have some limitations that should be addressed in future work.

6.1. Scalability

All analyses that use model checking suffer from the state explosion problem [10], where the statespace of a model grows exponentially as the number of concurrent elements are added to the system model. This can increase the amount of time required to verify a model and, in the worst case, result in a model to big to verify on a given computer. Because our method relies on the verification of multiple specification properties, for large models, this could result in excessively long analysis times. Future work should investigate the scalability of the method and explore how additional methods can be used to manage formal model complexity [19].

6.2. Interpreting Results

The counterexample visualizer supported by EOFM [5] can help analysts evaluate single counterexamples. However, because a large number of properties must be verified with our method, analyst may find it difficult to interpret verification results. Future work should investigate options for assisting analysts in results interpretation.

6.3. Model Checker Limitations

In preparing the application for this paper, a number of problems with SAL were revealed. Specifically, the verification of *Act Inevitable Completability* and *Task Liveness* prosperities could result in counterexamples illustrating artificial model loops (infinite state cycles where no variable values changed) or failure to detect deadlock states that violated the properties (SAL uses a separate checker for finding deadlock states). To address these issues, the model was formulated to both eliminate these uninteresting cycles and allow the end state to recursively cycle, preventing end state deadlock.

There may be several ways of addressing these problems. SAL is open source so it may be possible to modify it to remove these issues. Tools capable of helping analysts create models that do not have the noted limitations could be developed. Finally, other model checkers will likely not have these problems and the EOFM-supported method could be adapted to work with these. Future work should explore these options.

6.4. Method Extensions

There are other coverage criteria and computation concepts [24] that could be used to reason about the execution of task models. Future work should identify which of these are capable of providing analysts with insights important to HAI and use them to generate additional specification properties.

ACKNOWLEDGEMENT

This work was supported by Verification Models for Advanced Human-Automation Interaction in Safety Critical Flight Operations from the European Space Agency.

REFERENCES

1. Aït-Ameur, Y., Baron, M., and Girard, P. (2003). Formal validation of HCI user tasks. In *Proceedings of the International Conference on Software Engineering Research and Practice*, pages 732–738, Las Vegas. CSREA Press.
2. Bolton, M. L. (ND). Automatic validation and failure diagnosis of human-device interfaces using task analytic models and model checking. *Computational and Mathematical Organization Theory*. DOI 10.1007/s10588-012-9138-6.
3. Bolton, M. L. and Bass, E. J. (2009). A method for the formal verification of human interactive systems. In *Proceedings of the 53rd Annual Meeting of the Human Factors and Ergonomics Society*, pages 764–768, Santa Monica. HFES.
4. Bolton, M. L. and Bass, E. J. (2010a). Formally verifying human-automation interaction as part of a system model: Limitations and tradeoffs. *Innovations in Systems and Software Engineering: A NASA Journal*, 6(3):219–231.
5. Bolton, M. L. and Bass, E. J. (2010b). Using task analytic models to visualize model checker counterexamples. In *Proceedings of the 2010 IEEE International Conference on Systems, Man, and Cybernetics*, pages 2069–2074, Piscataway. IEEE.
6. Bolton, M. L. and Bass, E. J. (2012). Using model checking to explore checklist-guided pilot behavior. *International Journal of Aviation Psychology*, 22(4):343–366.
7. Bolton, M. L., Bass, E. J., and Siminiceanu, R. I. (2012). Using phenotypical erroneous human behavior generation to evaluate human-automation interaction using model checking. *International Journal of Human-Computer Studies*, 70(11):888–906.
8. Bolton, M. L., Bass, E. J., and Siminiceanu, R. I. (2013). Using formal verification to evaluate human-automation interaction in safety critical systems, a review. *IEEE Transactions on Systems, Man and Cybernetics: Systems*, 43(3):488–503.
9. Bolton, M. L., Siminiceanu, R. I., and Bass, E. J. (2011). A systematic approach to model checking human-automation interaction using task-analytic models. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 41(5):961–976.
10. Clarke, E. M., Grumberg, O., and Peled, D. A. (1999). *Model checking*. MIT Press, Cambridge.
11. De Moura, L., Owre, S., and Shankar, N. (2003). The SAL language manual. Technical Report CSL-01-01, Computer Science Laboratory, SRI International, Menlo Park.
12. Degani, A. (2004). *Taming HAL: Designing interfaces beyond 2001*. Macmillan, New York.
13. Emerson, E. A. (1990). Temporal and modal logic. In van Leeuwen, J., Meyer, A. R., Nivat, M., Paterson, M., and Perrin, D., editors, *Handbook of Theoretical Computer Science*, chapter 16, pages 995–1072. MIT Press, Cambridge.
14. Fields, R. E. (2001). *Analysis of Erroneous Actions in the Design of Critical Systems*. PhD thesis, University of York, York.
15. Kebabjian, R. (2012). Accident statistics. `http://www.planecrashinfo.com/cause.htm`. Accessed 3/25/2013.
16. Kenny, D. J. (2011). 22nd joseph t. nall report: General aviation accidents in 2010. Technical report, AOPA Air Safety Institute.
17. Kirwan, B. and Ainsworth, L. K. (1992). *A Guide to Task Analysis*. Taylor and Francis, London.
18. Maluf, D. A., Gawdiak, Y. O., and Bell, D. G. (2005). On space exploration and human error: A paper on reliability and safety. In *Proceedings of the 38th Annual Hawaii International Conference on System Sciences*, pages 79–84, Piscataway. IEEE.
19. Mansouri-Samani, M., Pasareanu, C. S., Penix, J. J., Mehlitz, P. C., OMalley, O., Visser, W. C., Brat, G. P., Markosian, L. Z., and Pressburger, T. T. (2007). Program model checking: A practitioners guide. Technical report, Intelligent Systems Division, NASA Ames Research Center, Moffett Field.
20. Mitchell, C. M. and Miller, R. A. (1986). A discrete control model of operator function: A methodology for information display design. *IEEE Transactions on Systems Man Cybernetics Part A: Systems and Humans*, 16(3):343–357.
21. Palanque, P. A., Bastide, R., and Senges, V. (1996). Validating interactive system design through the verification of formal task and system models. In *Proceedings of the IFIP TC2/WG2.7 Working Conference on Engineering for Human-Computer Interaction*, pages 189–212, London. Chapman and Hall, Ltd.
22. Paternò, F., Santoro, C., and Tahmassebi, S. (1998). Formal model for cooperative tasks: Concepts and an application for en-route air traffic control. In *Proceedings of the 5th International Conference on the Design, Specification, and Verification of Interactive Systems*, pages 71–86, Vienna. Springer.
23. Reason, J. (1990). *Human Error*. Cambridge University Press, New York.
24. Sandler, C., Badgett, T., and Thomas, T. M. (2004). *The Art of Software Testing*. John Wiley & Sons.
25. Schraagen, J. M., Chipman, S. F., and Shalin, V. L. (2000). *Cognitive Task Analysis*. Lawrence Erlbaum Associates, Inc., Philadelphia.
26. Sheridan, T. B. and Parasuraman, R. (2005). Human-automation interaction. *Reviews of human factors and ergonomics*, 1(1):89–129.
27. Silberschatz, A., Galvin, P. B., and Gagne, G. (2009). *Operating system concepts*. J. Wiley & Sons.
28. Wing, J. M. (1990). A specifier's introduction to formal methods. *Computer*, 23(9):8, 10–22, 24.