Automatically generating specification properties from task models for the formal verification of human-automation interaction

Bolton, ML; Jimenez Enebral, N; van Paassen, MM; Trujillo, M

DOI
10.1109/THMS.2014.2329476

Publication date
2014

Document Version
Final published version

Published in
IEEE Transactions on Human-Machine Systems

Automatically Generating Specification Properties From Task Models for the Formal Verification of Human–Automation Interaction

Matthew L. Bolton, *Member, IEEE*, Noelia Jiménez, Marinus M. van Paassen, *Member, IEEE*, and Maite Trujillo

***Abstract*—Human–automation interaction (HAI) is often a contributor to failures in complex systems. This is frequently due to system interactions that were not anticipated by designers and analysts. Model checking is a method of formal verification analysis that automatically proves whether or not a formal system model adheres to desirable specification properties. Task analytic models can be included in formal system models to allow HAI to be evaluated with model checking. However, previous work in this area has required analysts to manually formulate the properties to check. Such a practice can be prone to analyst error and oversight which can result in unexpected dangerous HAI conditions not being discovered. To address this, this paper presents a method for automatically generating specification properties from task models that enables analysts to use formal verification to check for system HAI problems they may not have anticipated. This paper describes the design and implementation of the method. An example (a pilot performing a before landing checklist) is presented to illustrate its utility. Limitations of this approach and future research directions are discussed.**

***Index Terms*—Formal methods, human–automation interaction (HAI), model checking, system safety, task analysis.**

I. Introduction

HUMAN behavior is a significant contributor to failures in complex systems that depend on human–automation interaction (HAI) [1], [2] for their safe operation. For example, it is a factor in more than 50% of commercial aviation accidents [3], 70% of general aviation accidents [4], many failures in space operations [5], a number of medical error [6], and breakdowns in process control [7]. However, these problems are not necessarily the fault of the human operators. Rather, they arise as a result of complex interactions between system components (human operators, device automation, and conditions in the operational environment) not anticipated by designers [8]. HAI is particularly prone to these issues because of the inherent concurrency between the human operator and the system automation he or she interacts with.

While advances have been made to address these problems [1], [2], failures can still occur because most analyses are incapable of evaluating all of the different conditions under which the supported HAIs occur. However, formal verification techniques, and particularly model checking, offer means of performing such exhaustive analyses.

A. Formal Verification and Model Checking

Formal verification comes from the field of formal methods. Formal methods are mathematically robust techniques and tools for the modeling, specification, and verification of systems [9]. Modeling is concerned with mathematically describing the behavior of a target system, specification properties assert desirable qualities about the system, and verification mathematically proves if the model adheres to the specification. This study uses model checking, a software tool that automatically performs formal verification [10]. In model checking, system behavior is typically modeled as a finite state transition system: a collection of variables and transitions between variable values (states). Specification properties assert desirable attributes about the system usually using model variables and temporal logic [10]. A model checker performs formal verification by exhaustively searching a system's state space to determine if the specification properties hold. If they do, the model checker returns a confirmation. However, if there is a state or sequence of states in the model that violates a property, a counterexample is produced. A counterexample represents a counterproof, which shows exactly how the property was violated. This is typically represented as a sequence of model states, where each state constitutes the full set of model variables and their values at that state, that incrementally lead up to the violation. This can then be examined by an analyst to understand why the failure occurred.

B. Formal Verification and Human–Automation Interaction

Formal verification is typically used in the analysis of computer hardware and software [9], [10]. However, it has also been used to evaluate HAI (see [12] for a review).

Manuscript received September 3, 2013; revised April 17, 2014; accepted May 26, 2014. Date of publication June 25, 2014; date of current version September 12, 2014. This work was supported by ITT AO-6967—Verification Models for Advanced Human–Automation Interaction in Safety Critical Flight Operations from the European Space Agency. The majority of this work was conducted while the first author was an Assistant Professor of industrial engineering with the University of Illinois at Chicago. This paper was recommended by Associate Editor M. Dorneich.

M. L. Bolton is with the Department of Industrial and Systems Engineering, State University of New York at Buffalo, Amherst, NY 14260-2050 USA (e-mail: mbolton@buffalo.edu).

N. Jiménez is with IXION Industry and Aerospace, 28037 Madrid, Spain (e-mail: njimenez@ixion.es).

M. M. van Paassen is with the Faculty of Aerospace Engineering, Delft University of Technology, 2629 HS Delft, The Netherlands (e-mail: m.m.vanpaassen@tudelft.nl).

M. Trujillo is with the European Space Research and Technology Centre, 2201 AZ , Noordwijk, The Netherlands (e-mail: maite.trujillo@esa.int).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/THMS.2014.2329476

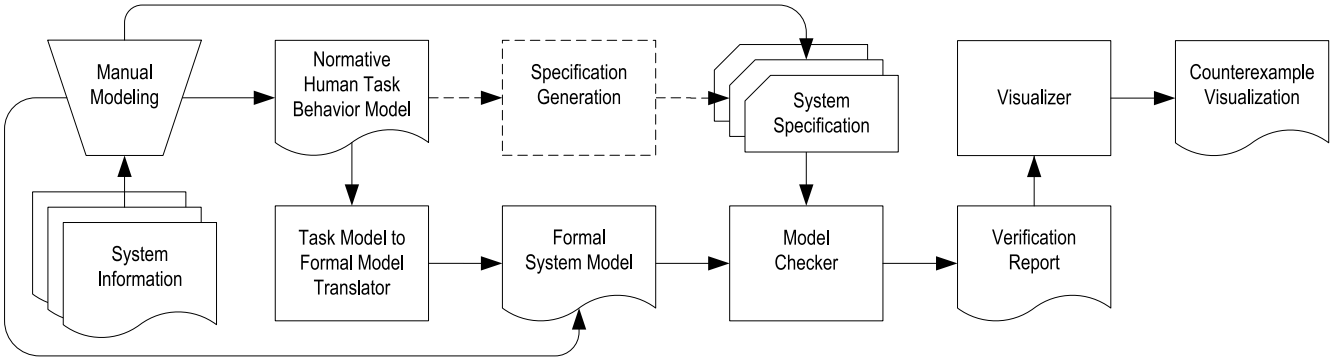


Fig. 1. Formal verification method supported by EOFM. Continuous lines indicate parts of process previously supported by EOFM [11]. Dotted lines represent the novel specification property generation process.

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 [13], [14] and describe the behaviors human operators use to achieve goals with a system. Task analytic models can be produced at different stages of system development, either to represent how people actually interact with a system or how people are expected to interact with a system being developed. They can be used in system engineering in a number of different capacities including human-automation interface creation [15], training development [16], usability analyses [17], [18], and the creation of real-time monitoring systems [19].

Task analytic models can be represented computationally. This allows them to be included in a formal system model containing a formal description of the other relevant system behaviors. Formal verification can then be used to evaluate the impact of both modeled normative and erroneous behavior as well as generated erroneous behavior on system performance and safety. Researchers either represent normative task models manually in formal notations as part of a larger model [20]–[22] or translate native task model notations into a formalism in which other system elements are represented [23]–[29]. Further, researchers have explored how erroneous behavior can be incorporated into the task models so that their impact on system safety and performance can be evaluated with formal verification. Erroneous behaviors can either be manually incorporated into task models using patterns [24], [30]–[33] or automatically generated using different theories of erroneous behaviors [34]–[36].

Given the well-established validity of task analysis and the wide use of the task analytic behavior models they produce, the formal verification techniques that utilize task analytic models are very powerful. They allow analysts to verify that a system's HAI will support safe operation under both normative and erroneous conditions. Further, verification results give analysts a clear indication of what the human operator was doing in discovered problems (visible in the form of counterexamples produced by the analysis tools).

These methods focus on the verification of analyst-created specification properties. Such properties are used to assert desirable system conditions (such as safety properties or performance requirements) using the model variables, usually using a temporal logic [37].

There are limitations to this approach. First, temporal logics can be difficult to learn and interpret [38]. This can result in analysts incorrectly formulating properties. Second, this approach requires that analysts anticipate potentially unsafe conditions and assert their absence as specification properties. Therefore, if analysts fail to anticipate potentially unsafe conditions 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 system's HAI. This means that analysts are using formal verification to look for failures that could be caused by problems with the HAI (among other things) instead of looking for potential HAI sources of failures. Thus, there is a real need for methods that will allow analysts to automatically generate properties that will allow them to check for HAI problems they may or may not have anticipated.

C. Objectives

In the work presented here, we discuss a method that fulfils this need. Specifically, we extend an existing method [11] that supports the formal verification of HAI-dependent systems, which include task analytic behavior models, to automatically generate specification properties. Given the formal structure inherent in hierarchical task analytic models, our method focuses on generating specification properties from the task analytic models themselves (see Fig. 1). While there are many types of HAI properties one might want to check, this study focuses on generating properties that will allow analysts to guarantee that the procedural behavior contained in the task model is compatible with the system. Specifically, this study exploits the fact that formal methods allow task analytic behavior models to be treated as a concurrent process executing with other part of a larger system and thus uses concepts from concurrency as the basis for the generated properties.

The remainder of this paper describes how the method 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), i.e., the logic used for representing

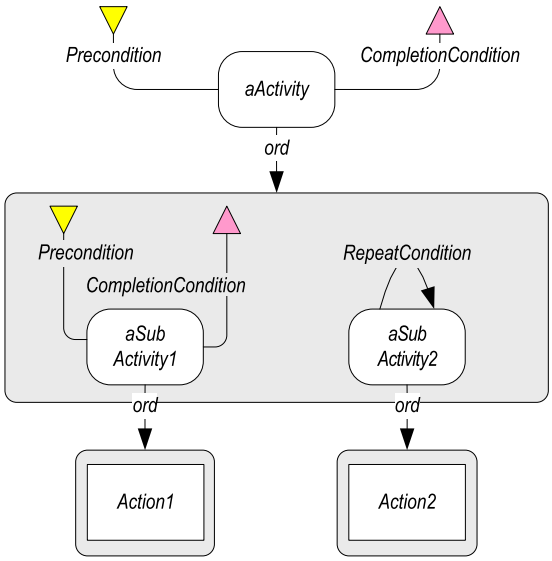


Fig. 2. Example of a task structure from an instantiated EOFM (there can be multiple task structure in a given EOFM) represented in EOFM's visual notation. In this task, an activity (*Activity*) with a precondition and completion condition decomposes into two subactivities using the ordered (*ord*) decomposition operator. *SubActivity1* has both a precondition and completion condition and decomposes into a single action (*Action1*). *SubActivity2* has a repeat condition and also decomposes into a single action (*Action2*).

generated specification properties, 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 avenues of future research.

II. Model Checking Human–Automation Interaction With the Enhanced Operator Function Model

EOFM [29] is an XML-based human task modeling language, derived from the operator function model (OFM) [39], specifically designed to allow task analytic human behavior to be evaluated with formal methods. EOFMs are hierarchical 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 (repeat conditions), and what must be true when it completes execution (completion conditions).

EOFMs can be represented visually as tree-like graphs [40] (see Fig. 2). 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 containing the decomposed activities or actions. Conditions (strategic knowledge) on activities are represented as shapes or arrows (annotated with the condition's 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 that specify how an instantiated EOFM model executes (see Fig. 3). 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 [29] for more details.

EOFM supports formal verification as follows. Analyst-created EOFM task models are automatically translated [29] into the language of the Symbolic Analysis Laboratory (SAL) [41] using the EOFM formal semantics.[1] The transitions associated with activity or action execution states are represented as guarded, asynchronous transitions in a single module. The translated EOFM is then integrated into a larger system model (created manually in the model checker's formalism) using asynchronous composition. A defined architecture allows for a distinction to be made between the human task, mission goals, human–automation interface, automation, and environment [11], [28] in the formal model. A coordination handshake protocol is also employed that uses interleaving to support parallelism between modeled components (see [29] and [42] for more information about the translator). 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 evaluated using EOFM's counter example visualizer [40].

The visualizer uses EOFM's visual notation to help analysts determine why a counterexample was produced. Each step from a counterexample is rendered on a separate page of a document. Each page represents the EOFM task structure executing in that step. The color of the activities and actions in the task indicate their execution state. Activities or actions whose execution state has changed since the previous step are highlighted. Additionally, before generating the visualization, analysts are able to categorize model variables (not associated with representing the human operator task) into categories based on EOFM's for-

[1]Note that the details of the EOFM to SAL translator are beyond the scope of this paper. Please see [29] and [42] for more information on this subject.

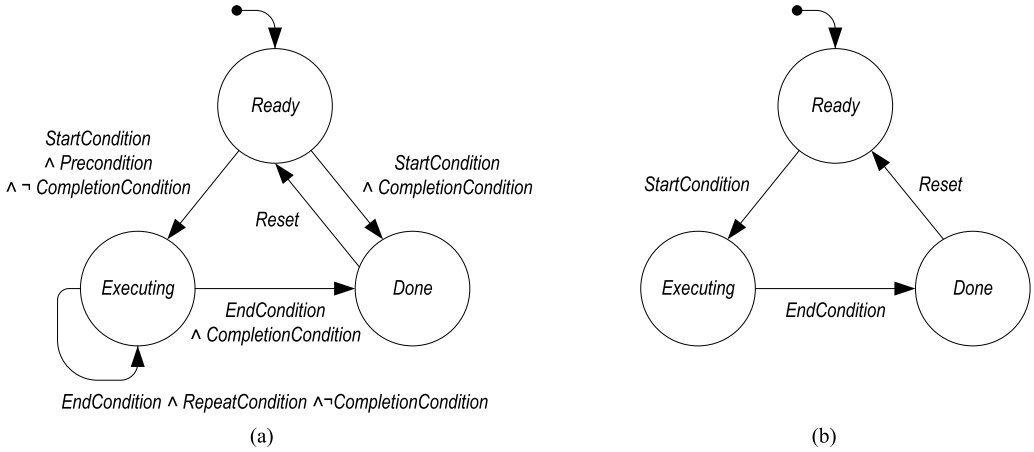


Fig. 3. Formal semantics of an EOFM activity's (a) and action's (b) execution state presented as finite state transition systems [29]. States are circles. Transitions are arrows between states labeled with Boolean expressions. An Arrow starting with a dot points to the initial state.

TABLE I
LTL 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 a 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 (a valid temporally ordered sequence of states) in the model that evaluate to true or false.

mal modeling architecture: environment, automation, human–automation interface, human mission, and other. Then, on each page of the generated visualization, each variable is displayed with its value under its corresponding category. Variables whose values changed from the previous step are highlighted. See [40] for more information.

III. Linear Temporal Logic Specification

Because the formal verification method supported by EOFM (see Fig. 1) uses SAL's symbolic model checker, specification properties must be asserted using LTL. LTL uses propositional variables, Boolean logic operators ($\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow, =, \neq, <, >$, etc.), and temporal operators (see Table I) to assert properties about all paths through a model [10].

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 (it cannot assert that something is true in at least one path through a model). Thus, to conduct an existence proof with a model checker that uses LTL specifications, a negative assertion must be used. 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 (1) 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.

IV. Specification Generation

The computational nature of the EOFM allows one to reason about the execution of task behavior as it interacts with other system elements. Thus, this work focuses on concepts from computational concurrency and uses them to check for properties important for safe HAI design. In computational concurrency, analysts are concerned with a number of different problems [43]. We focus on two: reachability and the prevention of blocking. Reachability is concerned with ensuring that all parts of a concurrent system can be reached. It is important because all elements of a system should be relevant at some time during its execution. If they are not, there is likely something wrong with the implementation or the design of the system. A lack of blocking is also crucial because blocking states prevent a system from making progress towards its goals. Blocking conditions can include deadlock (where a system becomes stuck due to concurrent elements waiting on each other), livelock (where the system becomes stuck in a loop), or starvation (where a process is not able to acquire the resources it needs to satisfy its goals) [44].

These concepts are relevant to a system's HAI because an analyst can use them to reason about how a task model should be executed (performed) in relation to the other concurrent elements of a system. Reachability is of concern because every part of a human operator's task should be relevant in some situation during a system's operation. If not, there may be behaviors in the task model that are never relevant and thus never executed. This could indicate a problem with the task analysis, which

could have far reaching design, training, and analysis implications given the importance of task analyses. It could also reveal a problem with the HAI that prevents desired task behaviors from being performed when they are supposed to. Such conditions are potentially dangerous because if the task model is used in the design of human training, such training would be deficient.

The absence of blocking from task models is also important for safe HAI [45]. From a task perspective, blocking can manifest in two general ways: either a task or tasks are blocked from completing (the task cannot be completed as described in the task model) or all human tasks are prevented from ever executing. Both blocking conditions are potentially problematic for HAI. If a task is not always able to be completed as specified, or the human operator finds him or herself in a situation for which they have no task knowledge, the human operator will have to go off task to fulfill his or her goals. Because such behavior is divergent, it is erroneous [46]. Further, if the divergent human behavior was not factored into system training or design, this could lead to unexpected or dangerous system operating conditions. Additionally, the presence of states that block the human from ever doing anything could represent unanticipated or unexpected system conditions that could produce mode confusion and/or automation surprise, which can be dangerous [47].

Thus, we want to ensure all of the following to guarantee that task models will never have problems executing:

1) Every part of a task should executable in some situation.
2) Tasks should always be able to finish executing.
3) There should never be a situation where no tasks can ever execute.

Ensuring all three of these should guarantee that there are no procedural issues with a system's HAI: every part of a task can be performed in its intended context; if a task is being performed, the human operator will always accomplish his or her goals; and the human will always be able to eventually interact with and achieve goals with the system.

EOFM's semantics (see Fig. 3) enable specification properties to assert qualities about the execution state of task models [48]. Thus, we can use computational concepts to automatically generate LTL specifications that assert all three of the above items and thus allow them to be checked formally.

A. Item 1: Coverage

Item 1 can be addressed using the computational concept of coverage. A coverage criterion represents the extent to which elements of a computational structure are reachable [43]. In this work, we wanted to be able to check that every part of a task model's execution was reachable. For this, we utilized both state coverage and decision coverage. State coverage asserts that every state in a finite state machine should be reachable while decision coverage says that every transition in a finite state machine must be able to occur. Both state and decision coverage apply to EOFM task models through the execution state of its activities and actions (see Fig. 3). Thus, to ensure state coverage, the execution state of every activity and action must be reachable. For decision coverage to be satisfied, every transition between the execution states of every activity and action must be able to occur. We can generate properties that will check for state and decision coverage by reasoning about the execution state of activities and actions using EOFM formal semantics (see Fig. 3).

To ensure that state coverage is satisfied, properties are generated to check that every activity and action in a task model is capable of reaching each of the three execution states (see Fig. 3). 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 (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 (2) (see Table II) 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 (3) (see Table II) for each activity and action. Since both act executability and act completability are negatively asserted, if the *Executing* and *Done* states are reachable, (2) and (3) will produce counterexamples when checked. If *Executing* and/or *Done* are **not** reachable, the model checker will return true.

To check for decision coverage, properties are generated to verify that every transition between activity and action execution states can occur. There are five possible transitions between activity execution states (there are three for actions; Fig. 3), and a specification property is generated to check that each is possible using a negative assertion. Thus, if a counterexample is generated when the property is checked, then the associated transition is possible. The ability of an action or activity to transition from *Ready* to *Executing* (a property called act startability) is asserted using (4). The ability of an activity to transition from *Ready* to *Done* was checked with the act skippability property (5). Act repeatability, the ability of an activity to transition from *Executing* to *Executing* via a repeat condition was asserted with (6). The ability of an activity or action to reset (transition from *Done* to *Ready*) was called act resetability and asserted with (7). Finally, the ability of an activity or action to transition from *Executing* to *Done*, was asserted using act finishability (8).

Note that act skippability (5) and act repeatability (6) are only relevant for activities because actions lack the associated transitions in their formal semantics (see Fig. 3). Further, act skippability is only applicable for an activity if it has a completion condition because one is required for the *Ready* to *Done* transition. Similarly, act repeatability is only appropriate when an activity has a repeat condition.

B. Item 2: Starvation

While knowing that all parts of a task can execute is useful, we also need to ensure that, once a task is executing, it will always finish (item 2). Computationally, this means that we never want task model starvation: a blocking condition where something is unable to gain the necessary resources to finish [44]. To check that a task will never starve, we generate a property called act inevitable completability (9) (see Table II) for

TABLE II
TASK MODEL SPECIFICATION PROPERTY PATTERNS FOR MODEL CHECKING SYSTEM HAI

Name:	Act Executability	
Description:	A given activity or action should be able to execute.	
Formulation:	$\mathbf{G}\neg(Act = Executing)$	(2)
Interpretation of Confirmation:	× There are no conditions where Act can ever execute.	
Interpretation of 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 Confirmation:	× There are no conditions where Act can ever be done.	
Interpretation of Counterexample:	✓ There are conditions where Act can be done.	
Name:	Act Startability	
Description:	A given activity or action should be able to transition from *Ready* to *Executing*.	
Formulation:	$\mathbf{G}\neg(Act = Ready \wedge \mathbf{X}(Act = Executing))$	(4)
Interpretation of Confirmation:	× There are no conditions where Act will transition from *Ready* to *Executing*.	
Interpretation of Counterexample:	✓ There are conditions where Act can transition from *Ready* to *Executing*.	
Name:	Act Skippability	
Description:	A given activity or action should be able to transition from *Ready* to *Done* if it has a completion condition.	
Formulation:	$\mathbf{G}\neg(Act = Ready \wedge \mathbf{X}(Act = Done))$	(5)
Interpretation of Confirmation:	× There are no conditions where Act will transition from *Ready* to *Done*.	
Interpretation of Counterexample:	✓ There are conditions where Act can transition from *Ready* to *Done*.	
Name:	Act Repeatability	
Description:	A given activity or action should be able to transition from *Executing* to *Executing* if it has a repeat condition.	
Formulation:	$\mathbf{G}\neg((Act = Executing \wedge EndCondition) \wedge \mathbf{X}(Act = Executing \wedge \neg EndCondition))$	(6)
Interpretation of Confirmation:	× There are no conditions where Act will repeat.	
Interpretation of Counterexample:	✓ There are conditions where Act can repeat.	
Name:	Act Resetability	
Description:	A given activity or action should be able to transition from *Done* to *Ready* if it has a completion condition.	
Formulation:	$\mathbf{G}\neg(Act = Done \wedge \mathbf{X}(Act = Ready))$	(7)
Interpretation of Confirmation:	× There are no conditions where Act will transition from *Done* to *Ready*.	
Interpretation of Counterexample:	✓ There are conditions where Act can transition from *Done* to *Ready*.	
Name:	Act Finishability	
Description:	A given activity or action should be able to transition from *Executing* to *Done* if it has a completion condition.	
Formulation:	$\mathbf{G}\neg(Act = Executing \wedge \mathbf{X}(Act = Done))$	(8)
Interpretation of Confirmation:	× There are no conditions where Act will transition from *Executing* to *Done*.	
Interpretation of Counterexample:	✓ There are conditions where Act can transition from *Executing* to *Done*.	
Name:	Act Inevitable Completability	
Description:	Every activity or action that is executing must eventually finish.	
Formulation:	$\mathbf{G}\,((Act = Executing) \Rightarrow \mathbf{F}(Act \neq Executing))$	(9)
Interpretation of Confirmation:	✓ Act can always finish executing.	
Interpretation of 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:	$\mathbf{G}\neg(\mathbf{F}(\mathbf{G}(\bigwedge_{Act \in RootActivities}^{\forall RootActivities} Act \neq Executing)))$	(10)
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. Equation numbers in the table continue from the equation numbers in the text. Act represents the execution state of a given activity or action. $RootActivities$ represents the set of all top level activities. A ✓ and × indicate if the associated verification outcome is desirable or undesirable respectively.

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.

C. Item 3: Liveness

Inevitable completability properties will allow an analyst to check that a system is not blocking a task from completing. However, it will not allow analysts to discover if the human operator is prevented from ever interacting with the system (from ever performing a task). Thus, we also want to check that the system itself never blocks the human operator from performing any tasks (item 3). From a computational perspective, this means we want to ensure liveness: that something desirable will always eventually happen [10]. Put another way, we want to ensure that there are no blocking states where tasks cannot execute. To check this, we again create properties that reason about the execution state of EOFM activities. This generated property, which we call task liveness, takes the form shown in (10) (see Table II). This asserts that it should never be true that the model will reach a future state where all of the root activities in an EOFM (the topmost activities of each task structure) never execute. Task liveness is asserted positively. Thus, a model checker

will return a confirmation if it is satisfied. Otherwise, a counterexample should show how a violating livelock or deadlock state was reached.

D. Accounting for Uninteresting States

In checking act inevitable completability and task liveness, it is conceivable that the model checker will identify states where either is violated in ways uninteresting to analysts. For example, a model may have an "end state" at or after which the properties are not be important. To accommodate this, the analyst will need to check modified forms of these properties. To do this, an analyst can manually reformulate (9) or (10) as

$$\psi \Rightarrow \left(\bigwedge_{\phi \in \Phi}^{\forall \Phi} \neg \phi \right) \tag{11}$$

where ψ is the original specification, and Φ is the set of expressions representing states an analyst wishes to exclude.

E. Implementation

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

F. Property Relationships and Interpreting Verification Results

If all of the generated properties produce the expected verification results, the relationships between the properties are such that the system's HAI is guaranteed to be well designed for the procedural behavior contained in the EOFM. Specifically, the properties collectively ensure that: every part of a task is relevant and doable in some situation (2)–(8); once a task is being performed, its goals will always be accomplished (9); and the human operator will always be able to eventually interact with the system (10).

However, should properties be violated, the interrelationships between properties may make result interpretations difficult. Thus, the following describes how verification results can be examined to identify the source of discovered problems.

1) Task Liveness: The verification results of a task liveness property (10) is probably the easiest to interpret because there will only be one such property for a given model and it is asserted positively. Thus, if task liveness is not satisfied, the model checker will produce a counterexample. EOFM's counterexample visualizer [40] can then be used to diagnose why the failure occurred. Task liveness is different from act inevitable completability in that task liveness checks situations where tasks cannot be performed, while act inevitable completability detects situations where tasks cannot be finished. Task liveness is also different from the coverage-based properties in that the latter check that there is at least one situation where each part of a task can execute. As such, even when the coverage-based properties are satisfied, there could be system states where no tasks can execute.

2) Act Inevitable Completability: Like task liveness, act inevitable completability (9) is asserted positively. Thus, violations will produce counterexamples that can be analyzed using the visualizer [40]. Unlike task liveness, there are multiple act inevitable completability properties. Given the hierarchical nature of EOFM and the fact that inevitable completability properties are generated for each activity and action, it is possible that a single problem will result in multiple failures of inevitable completability specifications. Given the nature of EOFM's formal semantics, the failure of a given action or activity to complete will inherently prevent the activity, from which it is decomposed, from completing its execution. Thus, if multiple failures occur in a given task structure, analysts should find the failure (or failures) that occurs for the act that is the lowest in the task's hierarchy and focus their evaluation there. As failures of inevitable completability are addressed, the model can be iteratively reverified against all of the inevitable completability properties until all of the discovered problems have been eliminated.

There are also two features of act inevitable completability that analysts should consider when interpreting results. Firstly, because these properties use the implication operator ($\Rightarrow$), they can be vacuously true. This means that, for a given activity or action Act, if $Act = Executing$ is never true, Act's inevitable completability property will verify to true. Thus, it is critical that analysts check Act's act executability property to ensure that an affirmative verification of Act's inevitable completability property is not caused by vacuity. Secondly, act completability can be satisfied (produce the desired counterexample) without satisfying act inevitable completability and, due to vacuous truth, act inevitable completability can be satisfied without act completability being satisfied. Thus, it is important that analysts check both.

3) Coverage-Based Properties: Because the coverage-based properties (2)–(8) are asserted negatively, they will produce a confirmation from the model checker when they fail. Thus, additional care must be taken in interpreting their verification results. The following sections discuss each of the coverage-based property types and describe how the interrelationships between them can be used to interpret model checker outputs. In all cases, it is assumed that there are no problems with a given act's start, end, and reset conditions. This is because these are implemented by the EOFM to SAL translator [29] and have been validated to be working as intended [42].

a) Act executability: If an act executability property (2) fails to return a counterexample, this can imply one of two things. The act may never have a chance to execute because another activity or action is preventing it from being reached. Alternatively, if the act is an activity, its strategic knowledge may be over constraining when it can execute. Thus, an analyst should first look at the other checked properties to see if an activity or action that must execute before the given act is either incapable of executing or completing. Otherwise, the analyst should check act startability.

b) Act completability: A failure of act completability (3) (a failure to produce a counterexample) could occur because: the act in question can never execute, it has a descendent that is blocking its execution from completing, or its completion condition is never satisfied. The first condition can be checked

by evaluating the act executability, the second by evaluating the act executability and act completability of the act's descendants, and the third by evaluating act finishability.

c) Act startability: If act executability fails (does not produce a counterexample), act startability (4) will invariably fail as well. If the analysis of the act executability indicates that the failure lies with the given act, then a failure of act startability will indicate that the precondition of the act is never satisfied or that the completion condition is always satisfied when the precondition is satisfied (see Fig. 3).

d) Act skippability: If an act is not skippable (5), the act's completion condition will never be satisfied when it is in the *Ready* state. It will be at the analyst's discretion to determine if he or she cares that the act is not skippable.

e) Act repeatability: An unrepeatable act (6) is one in which the repeat condition is never satisfied.

f) Act resetability: Act resetability (7) will be violated for a given act when an activity it descends from fails to reset or repeat. Thus, for diagnostics, an analyst should look for failures of act resetability and act repeatability for activities the act descends from. Note that a failure of a top level activity to reset will never occur unless the analyst specifically modifies the formal model to produce such behavior [42].

g) Act finishability: Act finishability (8) can fail based on the execution of other acts the same way that act completability can. However, if the failure has been isolated to the given act, a failure of act finishability will indicate that the act's completion condition can never be satisfied.

V. Application

To illustrate how this method can be used to find problems in an HAI-dependent aerospace system, we present an application: a pilot attempting to perform the before landing checklist (a slightly modified version of a model previously discussed in [49]). 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 [50]: 1) the ignition must be set to override to allow for engine restart should it quit; 2) the landing gear must be down; 3) the spoilers should be armed; and 4) the flaps should be extended to the appropriate flap setting (first 25° and then 40°) to slow the aircraft and prevent stalling.

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 [51]. If deployed too early, the aircraft loses lift and could have a hard landing [52]. 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 [50]. For these reasons, pilots wait to arm the spoilers until after the landing gear has been deployed and the landing gear doors have completely opened (in our example, this can take between 10 and 18 s due to variability in the condition of the hydraulic system).[2]

A. Task Modeling

The task behavior for performing the before landing checklist was instantiated as an EOFM (see Fig. 4). 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 via the flaps gauge (*FlapsGuage*), if the landing gear is down (*ThreeGearLights*), and if the landing gear doors are opening (*GearDoorLight*).

If the ignition light is OFF, the pilot can override the ignition (*aOverrideIgnition*) before the glideslope indicator is alive. Once it is alive and the three gear lights are off, the pilot can then deploy the landing gear (*aDeployLandingGear*). At a glideslope indicator reading of one dot, the pilot can set the flaps to 25°. The spoilers can be armed (*aSetSpoilers*) if the spoiler indicator and landing gear doors lights are off. At the capture position, the flaps can be set to 40° (*aSetFlaps40*).

This model was converted into SAL's input language using the translator [29], and the properties form Table II were automatically created using the newly added automatic specification generation feature. While the original XML instantiation contained 92 lines of XML markup, the translated version contained 184 lines of SAL code. This raw translated version was slightly modified to ensure that the formal system model never reached a deadlock state, erroneously reset when the model reached its end state (when the aircraft was descending to the runway), or could produce any undesired infinite loops.

The specification generation process produced 72 properties. Sixty properties represented the different reachability properties associated with coverage (act executability, completability, inevitable completability, startability, skippability, resetability, and finishability) for each activity and action in the task model. Eleven act inevitable completability properties were produced, one for each activity and action. Finally, one property representing task liveness was created.

B. 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. Note that these were created directly in the model checker's formal notation [41] and are, for the sake of readability, presented here as finite state transition systems.

[2]Spacial constraints prevent a more thorough description of this application. Additional information can be found in [49]. A full listing of the instantiated EOFM and complete formal model with generated properties can be found at http://fhsl.eng.buffalo.edu/resources/.

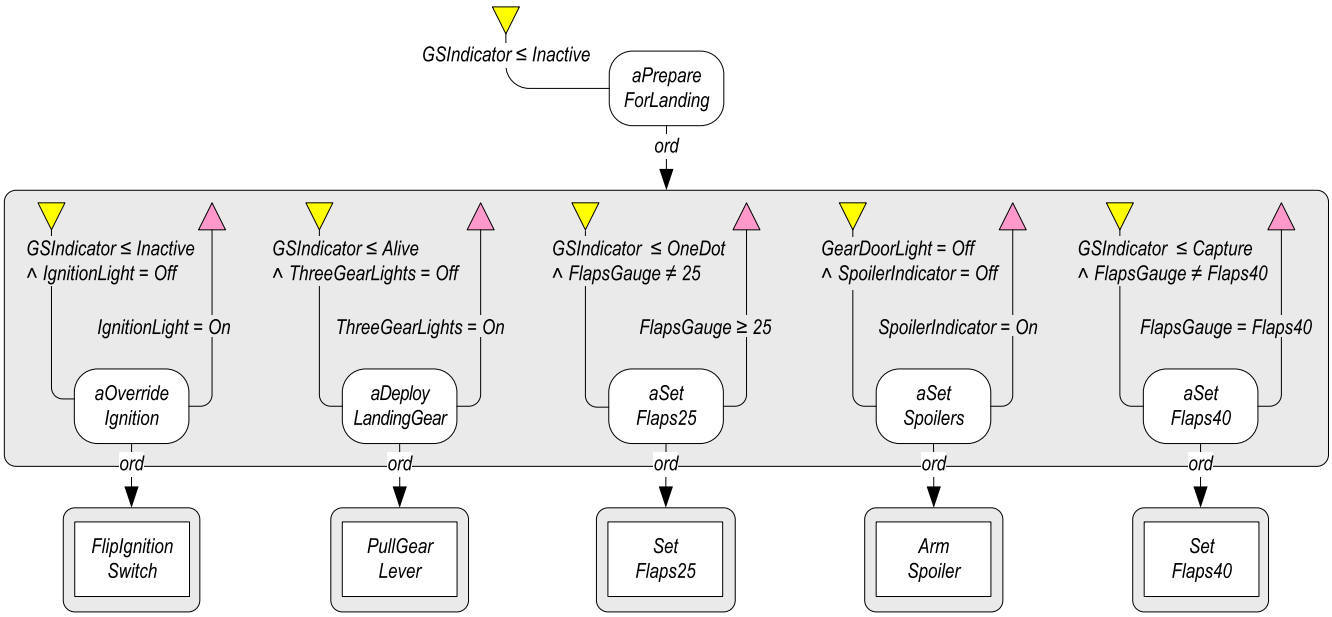


Fig. 4. Visualization of the EOFM task model for the before landing checklist [49].

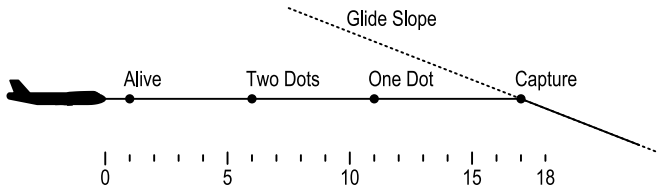


Fig. 5. *Position* of the aircraft relative to the glideslope, where distance is measured in seconds.

1) Environment: The system's operational environment was represented abstractly as the relative distance (*Position*) of the aircraft from the capture position on the glideslope (see Fig. 5). 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 s. Thus, the relative position of the aircraft from the initial position is discretized into integer values (0 to 18) where the aircraft passes from one value to the next in 1 s. The model was set to meet a dedicated end state when it starts to descend (at position 18, the end state of the model).

2) Automation: The formal model of the device automation (see Fig. 6) 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 undeployed, 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 conditions. While most changes to pilot actions were effectively instantaneous (taking less than a second) landing gear doors would take between 10 and 18 s to fully transition from closed to open after landing gear deployment was initiated.

C. Human–Automation Interface

The formal model of the human–automation interface (see Fig. 7) 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 ignition (*IgnitionLight*), spoilers (*SpoilerIndicator*), landing gear doors (*GearDoorLight*), landing gear (*ThreeGearLights*), flaps (*FlapsGuage*), and glideslope indicator (*GSIndicator*) would update to reflect these changes.

D. Expected Results

All of the coverage-based properties were expected to produce the desired result (a counterexample) with two exceptions. Because the task model was modified to avoid a reset, all of the act resetability properties were expected to verify to true (indicating that the all of the activities and actions would never reset). Additionally, because the pilot was expected to perform all of the activities and actions to ensure that the aircraft was ready for landing, all of the act skippability properties were expected to verify to true (indicating that none of the activities were skippable). All of the act inevitable completability properties and the task liveness property were expected to produce the desired result (verify to true).

E. Formal Verification Results

Formal verification was performed on a PC workstation running Linux Mint 14 with a quad-core, 3.3-GHz Intel XEON processors and 64 gigabytes of RAM. Using SAL's symbolic

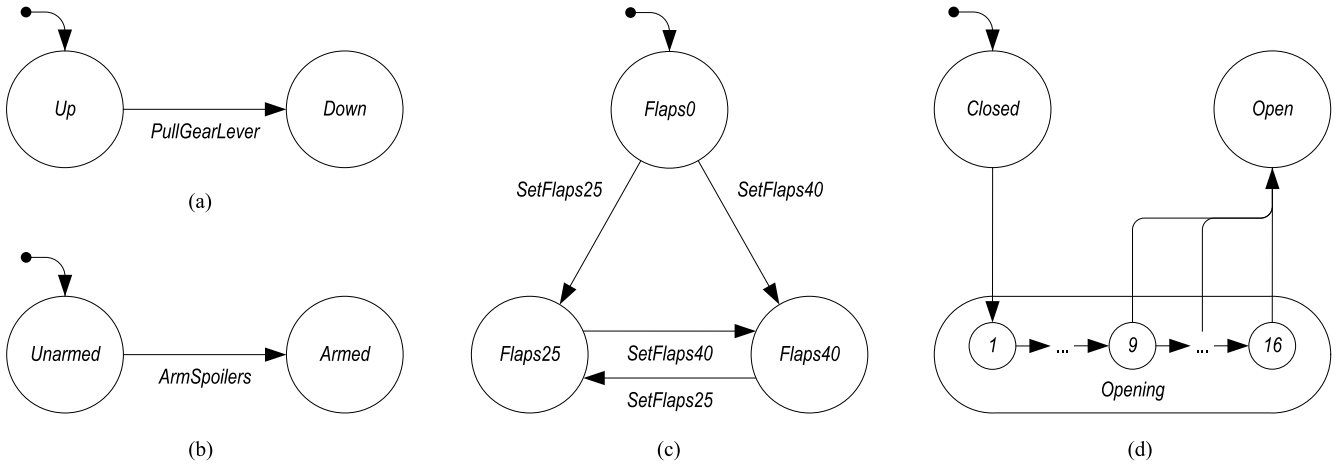


Fig. 6. State transition representation of the formal model of the system's automation. (a) Landing gear. (b) Spoilers. (c) Flaps. (d) Landing gear doors, where the number on the opening states indicates how long, in seconds, it takes for the landing gear to open. In the presented model, the doors can take between 10 and 18 s to open (note that the transition to *Open* takes 1 s to occur).

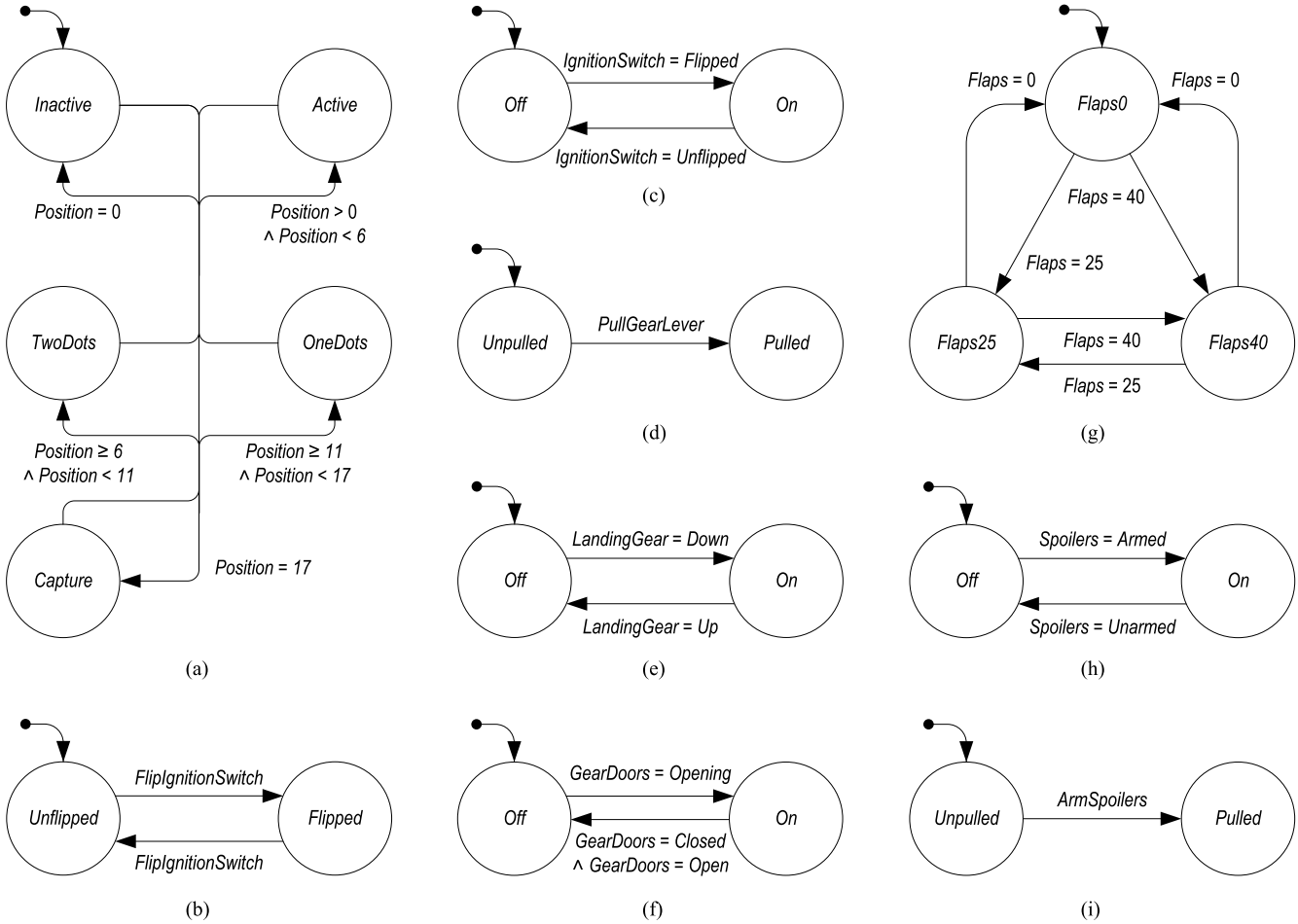


Fig. 7. State transition representation of the human–automation interface's formal model. (a) Glideslope indicator. (b) Ignition switch. (c) Ignition light. (d) Landing gear lever. (e) Three landing gear lights. (f) Gear doors light. (g) Flaps gauge. (h) Spoiler arming lever. (i) Spoiler indicator light.

model checker, it took 22.21 s of total execution time to verify all 72 generated properties with 1535 listed as the maximum number of visited states for a single verification.

For all but four of the generated properties, the expected verification results were obtained. Every coverage-based specification property produced the expected result indicating that, for all but act resetability and act skipability, every part of the task model (see Fig. 4) and its associated transitions between execution states was reachable. However, three of the act inevitable completability properties did not evaluate to true indicating that, for *aSetFlaps40*, *aSetSpoilers*, and *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 [40]) revealed a common problem. In all three, the landing gear doors took between 16, 17, and 18 s 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 pilot could arm the spoilers.

All three of these properties failed because the model reached the end state before all the necessary activities could finish executing. More importantly, all of these failures constitute serious problems for the safe operation of the aircraft. In 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 [53]. The failure of *aSetSpoilers*' property illustrates a situation where spoilers are armed while the aircraft is just starting to descend, a condition that could lead to premature spoiler deployment [50]. 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 [51] or accidental premature deployment [52].

F. Design Exploration

To test whether these failures were the result of the delay in the opening of the landing gear doors, the delay was iteratively reduced between multiple verification runs to see if there was a minimum delay that would remove these failures from the model. When the delay was 14 s or less, all three of these properties returned the desirable verification outcomes.

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 (11) to create the specification

$$\mathbf{G}\neg(\mathbf{F}(\mathbf{G}(aPrepareForLanding \neq Executing))) \Rightarrow \neg(AircraftPosition = EndPosition). \quad (12)$$

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 artificial end state of the model.

VI. Discussion

In the work presented here, we extended the formal verification method supported by EOFM with a novel means of generating specification properties from task models (see Fig. 1). By exploiting its computational representation of task models, the method is capable of detecting if parts of task models are never used, if there are situations where a task cannot be completed, and if there are times 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: 1) there is no risk of specification properties being manually misformulated; 2) analysts need not anticipate all of the HAI problems associated with the generated properties to check for them; and 3) specification properties are represented in terms of the task models and thus detect problems with the system's HAI rather than specific failures that can arise from the HAI.

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 not adjusting the aircraft's flaps in time, something not discovered in previous analyses [49].

It is important to note that the method is still compatible with the traditional verification analyses supported by EOFM [11], [28], [29], [34], [48], [49]. 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, listed below, that should be addressed in future work.

A. Additional Applications

Although simple, the application presented here illustrates how our method could be used to find unanticipated problems with HAI. However, the application does not cover all of the different issues that can arise during HAI. Further, the primary issue (the timing of the aircraft's doors and its impact on spoilers) was known a priori. Thus, different problems are likely to be discovered when the method is applied to other realistic systems with or without known HAI issues. Future work should explore how the method can be applied to different human interactive systems to develop a more comprehensive understanding of its strengths and weaknesses.

B. Methodological Considerations

1) Use in Design and Analysis: Formal verification is not widely used in the design and analysis of human-automation interactive systems. This will limit the use of the presented method in the immediate future. This lack of use is partially due to the challenges associated with learning and interpreting formal methods [38]. EOFM and the presented methods are

advantageous in this respect in that they allow human factors analysts to create task models using notations they are familiar with and automatically generate specification properties that they can check. Further, the counterexample visualizer [40] helps them interpret verification results using the human operator's task. However, the presented method still requires the analyst to model all of the other system elements the human interacts with in the model checker's input language. This will clearly limit its use. A number of researchers have investigated how to create formal models of human–automation interfaces that have been designed graphically or constructed with more widely used design environments [26], [54]–[56], while others have investigated how to create formalisms that are intuitive to human factors engineers [57]–[60] to model interfaces. Future work should investigate how these methods can be integrated with EOFM and incorporated into more widely used model-based engineering environments.

2) Scalability: All analyses that use model checking suffer from the state explosion problem [10], where the state space 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 too big to verify on a given computer. This phenomenon has been observed with formal models utilizing EOFM, where model state spaces grow exponentially with the size of the task behavior contained in a given EOFM [34], [42]. Because the property generation method presented here does not add to model complexity, it does not directly influence the feasibility of the formal verification in terms of state space. However, any reductions in the complexity of the state space used to represent EOFMs would improve the applicability of the method. Because our method relies on the verification of multiple specification properties, for large models, this could result in excessively long analysis times. Thus, future work should investigate means for improving the scalability of the method in terms of both state space size and verification time.

3) Results Interpretation: The counterexample visualizer supported by EOFM [40] and the guidelines reported here (see Section IV-F) can help analysts evaluate verification results. However, because a large number of properties must be verified with our method, analysts may find it difficult to interpret verification results for more complex applications. Future work should investigate options for assisting analysts in results interpretation: helping them understand the connections between the different properties and helping them compare the analysis results between properties based on these connections.

4) 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 properties 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. Alternatively, we could have used deadlock checker synergistically with the formal verification analyses, where we would manually inspect each deadlock state to determine if it violated task liveness or inevitable completability.

There may be several ways of addressing these problems. SAL is open source so it may be possible to modify it and remove these restrictions. Tools capable of helping analysts create models that do not have the noted issues could be developed. A wrapper tool could also be created to help analysts evaluate whether deadlock states (found using SAL's deadlock checker) could impact checked properties. Finally, the EOFM-supported method could be adapted to work with model checking environments that don't have these limits. Future work should explore these options.

C. Property Generation Extensions

The method introduced in this paper has shown that it is possible to automatically generate specification properties from task models capable of checking a system's HAI. However, there are many other computational and HAI concepts that could be generated to make the presented method more complete. This possibility is explored below.

1) Other Specification Formalisms: Specification properties in SAL's symbolic model checker are formulated using LTL, thus this was the logic used for property generation. However other model checkers support different specification languages that can offer different expressive power. For example, computation tree logic (CTL) is used by a number of model checkers and allows properties too be positively asserted about the existence of states. Such expressiveness would allow all of the coverage-based properties to be asserted positively with CTL rather than negatively as they are with LTL. Future work should explore how different specification languages could be used to express the properties generated here as well as enable the generation of different properties.

2) Additional Computation Concepts: There are other coverage criteria and computation concepts [43], [44] that could be used to reason about the execution of task models in a larger formal system model. For example, condition coverage asserts that every subexpression in a Boolean assertion is both true and false at some point in execution and parameter value coverage asserts that all values of different runtime parameters be used. Alternatively, computer scientists often look to eliminate race conditions from concurrent system: where the desired outcome depends on a particular temporal sequence of events that cannot be guaranteed. Future work should identify which of these concepts provide analysts with insights important to HAI and use them to generate additional specifications from task models.

3) Accounting for Erroneous Human Behavior: Although not discussed here, EOFM supports the ability to automatically generate erroneous human behavior and include it in the formal representation of the task analytic behavior model [34], [35]. It is conceivable that specification properties could be used to prove properties about the different generated erroneous behaviors. Future work should investigate this.

4) Accounting for Human Cognition and Performance: There are many cognitive factors that can influence HAI beyond

the procedural knowledge of task models. Work by Rukšėnas *et al.* [61]–[64] have explored how different cognitive concepts can be included in formal verification analyses. If these sophisticated cognitive representations could be integrated into EOFM, human attention, monitoring behavior, information processing, decision making, workload, and/or other human performance factors could to be considered in EOFM property generation and formal verification analyses. Future work should investigate such an extension.

5) Accounting for Interface and Automation States: The method presented here generates properties designed to evaluate how well the task behavior contained in an EOFM supports a system's HAI. While these analyses can give analysts valuable insights, they can miss issues related to the system's automation or human-automation interface. For example, the method will give analysts no insights into the existence of interface states that are unreachable by the human operator's task or exhibit poor usability properties.

Several researchers [65], [66] have developed different coverage criteria for human–computer and human–automation interfaces. These could be used to generate specification properties for formal interface models for use in formal verification analyses. By including such properties in formal verification analyses along with the task-based specifications discussed here, analysts should be able to gain additional insights into the HAI of their system.

Further, a body of work has investigated different ways of using formal methods to evaluate human-automation interface designs without considering models of the human task behavior. In these approaches, a formal model of a human-automation interface is verified against specifications that represent desirable usability properties (see [12]). Campos and Harrison [67] identified four related categories of properties that could be expressed in temporal logic and thus formally verified, and multiple researchers have defined property patterns within these categories [55], [67]–[69]. Reachability properties assert qualities about whether and when interface conditions can be attained [55], [68], [69]. Visibility properties describe how feedback to human actions should manifest [69], [70]. Task-related properties specify things that concern the completion of task goals and their ability to be undone [68]–[70]. Reliability describes properties that qualify safe and reliable performance including behavioral consistency, the absence of deadlock, and the lack of hidden mode changes [55], [68], [71], [72]. Tools have been developed that allow these properties to be automatically generated from interface models [55], [73]. Further, work by Campos *et al.* [74] have explored how these types of properties can be used to while reasoning about how information resource constraints on human actions can impact system usability. Given that the analysis method discussed in this paper uses a human–automation interface as part of its formal system model, it should be compatible with usability property generation methods and formal verification analyses. Thus, future work should investigate how these different approaches can be used synergistically to provide even more complete HAI analyses.

6) Including Time and Other Continuous Quantities: The application presented here was highly dependent on the timing of different system events. However, the model represented time using an integer counter. This is because the current version of EOFM does not support the realistic representation of time as a real numbered continuous quantity. This is largely due to the limitations of model checking, where the majority of model checkers (including SAL-SMC) can only evaluate systems with a finite number of discrete states. This limits the types of analyses that can be performed with the current version of the method. Clearly, the application presented in this paper would be much more realistic if time was modeled in higher fidelity. Luckily, progress is being made in this area. For example, one of the tools in SAL is its infinite bounded model checker [75], which supports the modeling of real time using timed and time-out automata [76]. Future work should investigate how time can be incorporated into models utilizing EOFM. Such an extension could also open up possibilities for generating and verifying properties specifically related to the timing of task performance.

Other continuous quantities and dynamics can be challenging to include in formal verification analyses as well. This is not only because they are continuous, but because nonlinear arithmetic of variables cannot generally be handled by model checkers. Those that study hybrid system modeling and verification (see, for example, [77]–[79]) have developed a number of abstraction and verification approaches that can account for these types of operations. However, EOFM does not currently support these. Future work should investigate how hybrid system modeling can be used synergistically with EOFM so that they can be considered in EOFM-supported specification property generation and verification analyses.

REFERENCES

[1] T. B. Sheridan and R. Parasuraman, "Human-automation interaction," *Rev. Human Factors Ergonomics*, vol. 1, no. 1, pp. 89–129, 2005.

[2] R. Parasuraman, T. B. Sheridan, and C. D. Wickens, "A model for types and levels of human interaction with automation," *IEEE Trans. Syst., Man Cybern. A, Syst., Humans*, vol. 30, no. 3, pp. 286–297, May 2000.

[3] R. Kebabjian. (2012). Accident statistics. [Online]. Available: http://www.planecrashinfo.com/cause.htm

[4] D. J. Kenny, "22nd Joseph T. Nall report: General aviation accidents in 2010," AOPA Air Safety Inst., Frederick, MD, USA, Tech. Rep., 2011.

[5] D. A. Maluf, Y. O. Gawdiak, and D. G. Bell, "On space exploration and human error: A paper on reliability and safety," in *Proc. 38th Annu. Hawaii Int. Conf. Syst. Sci.*, 2005, pp. 79–84.

[6] L. T. Kohn, J. Corrigan, and M. S. Donaldson, *To Err is Human: Building a Safer Health System*. Washington, DC, USA: Natl. Acad. Press, 2000.

[7] J. M. O'Hara, J. C. Higgins, W. S. Brown, R. Fink, J. Persensky, P. Lewis, J. Kramer, A. Szabo, and M. A. Boggi, "Human factors considerations with respect to emerging technology in nuclear power plants," United States Nuclear Regulatory Commission, Washington, DC, USA, Tech. Rep. NUREG/CR-6947, 2008.

[8] J. Reason, *Human Error*. New York, NY, USA: Cambridge Univ. Press, 1990.

[9] J. M. Wing, "A specifier's introduction to formal methods," *Computer*, vol. 23, no. 9, pp. 8–23, 1990.

[10] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.

[11] M. L. Bolton and E. J. Bass, "A method for the formal verification of human interactive systems," in *Proc. 53rd Annu. Meet. Human Factors Ergon. Soc.*, Santa Monica, CA, USA, 2009, pp. 764–768.

[12] M. L. Bolton, E. J. Bass, and R. I. Siminiceanu, "Using formal verification to evaluate human-automation interaction in safety critical systems, a review." *IEEE Trans. Syst., Man, Cybern. A, Syst., Humans*, vol. 43, no. 3, pp. 488–503, 2013.

[13] B. Kirwan and L. K. Ainsworth, *A Guide to Task Analysis*. New York, NY, USA: Taylor & Francis, 1992.

[14] J. M. Schraagen, S. F. Chipman, and V. L. Shalin, *Cognitive Task Analysis*. Philadelphia, PA, USA: Lawrence Erlbaum Assoc., Inc., 2000.

[15] F. Paternò, "Model-based design of interactive applications," *Intelligence*, vol. 11, no. 4, pp. 26–38, 2000.

[16] R. W. Chu, C. M. Mitchell, and P. M. Jones, "Using the operator function model and OFMspert as the basis for an intelligent tutoring system: Towards a tutor/aid paradigm for operators of supervisory control systems," *IEEE Trans. Syst., Man Cybern. A, Syst., Humans*, vol. 25, no. 7, pp. 1054–1075, Jul. 1995.

[17] A. Lecerof and F. Paternò, "Automatic support for usability evaluation," *IEEE Trans. Softw. Eng.*, vol. 24, no. 10, pp. 863–888, Oct. 1998.

[18] B. E. John and D. E. Kieras, "Using GOMS for user interface design and evaluation: Which technique?" *ACM Trans. Comput.-Human Interact.*, vol. 3, no. 4, pp. 287–319, Dec. 1996.

[19] E. J. Bass, S. T. Ernst-Fortin, R. L. Small, and J. Hogans, "Architecture and development environment of a knowledge-based monitor that facilitate incremental knowledge-base development," *IEEE Trans. Syst., Man, Cybern. A, Syst., Humans*, vol. 34, no. 4, pp. 441–449, Jul. 2004.

[20] S. Basnyat, P. Palanque, B. Schupp, and P. Wright, "Formal socio-technical barrier modelling for safety-critical interactive systems design," *Safety Sci.*, vol. 45, no. 5, pp. 545–565, 2007.

[21] S. Basnyat, P. A. Palanque, R. Bernhaupt, and E. Poupart, "Formal modelling of incidents and accidents as a means for enriching training material for satellite control operations," presented at the Joint ESREL 2008 17th SRA-Europe Conf., London, U.K., 2008.

[22] E. L. Gunter, A. Yasmeen, C. A. Gunter, and A. Nguyen, "Specifying and analyzing workflows for automated identification and data capture," in *Proc. 42nd Hawaii Int. Conf. Syst. Sci.*, Los Alatimos, CA, USA, 2009, pp. 1–11.

[23] P. A. Palanque, R. Bastide, and V. Senges, "Validating interactive system design through the verification of formal task and system models," in *Proc. IFIP TC2/WG2.7 Working Conf. Eng. Human-Comput. Interact.*, London, U.K., 1996, pp.189–212.

[24] R. E. Fields, "Analysis of erroneous actions in the design of critical systems," Ph.D. dissertation, Dept. Comput. Sci., York Univ. , York, U.K., 2001.

[25] Y. Aït-Ameur, M. Baron, and P. Girard, "Formal validation of HCI user tasks," in *Proc. Int. Conf. Softw. Eng. Res. Pract.*, Las Vegas, NV, USA, 2003, pp. 732–738.

[26] Y. Aït-Ameur and M. Baron, "Formal and experimental validation approaches in HCI systems design based on a shared event B model," *Int. J. Softw. Tools Technol. Transfer*, vol. 8, no. 6, pp. 547–563, 2006.

[27] F. Paternò and C. Santoro, "Integrating model checking and HCI tools to help designers verify user interface properties," in *Proc. 7th Int. Workshop Design, Specification, Verification Interact. Syst.*, Berlin, Germany, 2001, pp. 135–150.

[28] M. L. Bolton and E. J. Bass, "Formally verifying human-automation interaction as part of a system model: Limitations and tradeoffs," *Innovations Syst. Softw. Eng.: NASA J.*, vol. 6, no. 3, pp. 219–231, 2010.

[29] M. L. Bolton, R. I. Siminiceanu, and E. J. Bass, "A systematic approach to model checking human-automation interaction using task-analytic models," *IEEE Trans. Syst., Man, Cybern. A, Syst., Humans*, vol. 41, no. 5, pp. 961–976, Sep. 2011.

[30] R. Bastide and S. Basnyat, "Error patterns: Systematic investigation of deviations in task models," in *Task Models and Diagrams for Users Interface Design*. Berlin, Germany: Springer, 2007, pp. 109–121.

[31] M. L. Bolton, E. J. Bass, and R. I. Siminiceanu, "Using formal methods to predict human error and system failures," presented at the 2nd Int. Conf. Appl. Human Factors Ergon., Las Vegas, NV, USA, 2008.

[32] M. L. Bolton and E. J. Bass, "Formal modeling of erroneous human behavior and its implications for model checking," in *Proc. Sixth NASA Langley Formal Methods Workshop*, Hampton, VA, USA, 2008, pp. 62–64.

[33] F. Paternò and C. Santoro, "Preventing user errors by systematic analysis of deviations from the system task model," *Int. J. Human-Comput. Studies*, vol. 56, no. 2, pp. 225–245, 2002.

[34] M. L. Bolton, E. J. Bass, and R. I. Siminiceanu, "Using phenotypical erroneous human behavior generation to evaluate human-automation interaction using model checking," *Int. J. Human-Comput. Studies*, vol. 70, no. 11, pp. 888–906, 2012.

[35] M. L. Bolton and E. J. Bass, "Generating erroneous human behavior from strategic knowledge in task models and evaluating its impact on system safety with model checking," *IEEE Trans. Syst., Man Cybern. Syst.*, vol. 43, no. 6, pp. 1314–1327, Nov. 2013.

[36] M. L. Bolton and E. J. Bass, "Evaluating human-human communication protocols with miscommunication generation and model checking," in *Proc. Fifth NASA Formal Methods Symp. Moffett Field: NASA Ames Res. Center*, 2013, pp. 48–62.

[37] E. A. Emerson, "Temporal and modal logic," in *Handbook of Theoretical Computer Science*, J. van Leeuwen, A. R. Meyer, M. Nivat, M. Paterson, and D. Perrin, Eds., Cambridge, MA, USA: MIT Press, 1990, ch. 16, pp. 995–1072.

[38] C. Heitmeyer, "On the need for practical formal methods," in *Proc. 5th Int. Symp. Formal Tech. Real-Time Fault-Tolerant Syst.*, 1998, pp. 18–26.

[39] C. M. Mitchell and R. A. Miller, "A discrete control model of operator function: A methodology for information display design," *IEEE Trans. Syst. Man Cybern. A, Syst. Humans*, vol. SMCA-16, no. 3, pp. 343–357, May. 1986.

[40] M. L. Bolton and E. J. Bass, "Using task analytic models to visualize model checker counterexamples," in *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, 2010, pp. 2069–2074.

[41] L. De Moura, S. Owre, and N. Shankar, "The SAL language manual," *Comput. Sci. Lab., SRI Int.*, Menlo Park, CA, USA, Tech. Rep. CSL-01-01, 2003.

[42] M. L. Bolton, "Using task analytic behavior modeling, erroneous human behavior generation, and formal methods to evaluate the role of human-automation interaction in system failure," Ph.D. dissertation, Dept. Syst. Inf. Eng., Univ. Virginia, Charlottesville, VA, USA, 2010.

[43] C. Sandler, T. Badgett, and T. M. Thomas, *The Art of Software Testing*. New York, NY, USA: Wiley, 2004.

[44] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*. New York, NY, USA: Wiley, 2009.

[45] A. Degani and M. Heymann, "Formal verification of human-automation interaction," *Human Factors*, vol. 44, no. 1, pp. 28–43, 2002.

[46] E. Hollnagel, "The phenotype of erroneous actions," *Int. J. Man-Mach. Studies*, vol. 39, no. 1, pp. 1–32, 1993.

[47] N. B. Sarter and D. D. Woods, "How in the world did we ever get into that mode? Mode error and awareness in supervisory control," *Human Factors*, vol. 37, no. 1, pp. 5–19, 1995.

[48] M. L. Bolton, "Automatic validation and failure diagnosis of human-device interfaces using task analytic models and model checking," *Comput. Math. Organization Theory*, vol. 19, pp. 288–312, 2013.

[49] M. L. Bolton and E. J. Bass, "Using model checking to explore checklist-guided pilot behavior," *Int. J. Aviation Psychology*, vol. 22, no. 4, pp. 343–366, 2012.

[50] A. Degani, *Taming HAL: Designing Interfaces Beyond 2001*. New York, NY, USA: Macmillan, 2004.

[51] NTSB, "Runway overrun during landing, American Airlines flight 1420, McDonnell Douglas MD-82, N215AA, Little Rock, Arkansas, June 1, 1999," Natl. Transp. Safety Board, Washington, DC, USA, Tech. Rep. NTSB/AAR-01/02, 2001.

[52] Aviation Safety Network and Flight Safety Foundation. (2014). "Accident description: Air Canada flight 621," Flight Safety Foundation, Available: http://aviation-safety.net/database/record.php?id=19700705-0

[53] NTSB, "Crash during approach to landing Empire Airlines flight 8284, Avions de Transport Régional Aerospatiale Alenia ATR 42-320, N902FX Lubbock, Texas, January 27, 2009," Natl. Transp. Safety Board, Washington, DC, USA, Tech. Rep. NTSB/AAR-11/02, 2011.

[54] R. Bastide, D. Navarre, and P. Palanque, "A tool-supported design framework for safety critical interactive systems," *Interacting Comput.*, vol. 15, no. 3, pp. 309–328, 2003.

[55] J. C. Campos and M. D. Harrison, "Interaction engineering using the ivy tool," in *Proc. 1st ACM SIGCHI Symp. Eng. Interactive Comput. Syst.*, 2009, pp. 35–44.

[56] M. B. Dwyer, Robby, O. Tkachuk, and W. Visser, "Analyzing interaction orderings with model checking," in *Proc. 19th IEEE Int. Conf. Automated Softw. Eng.*, 2004, pp.154–163.

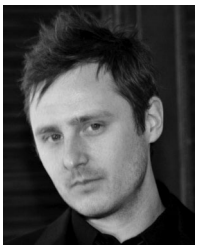
[57] M. Feary, "Automatic detection of interaction vulnerabilities in an executable specification," in *Proc. 7th Int. Conf. Eng. Psychol. Cognitive Ergonomics*, 2007, pp. 487–496.

[58] L. Sherry and M. Feary, "Improving the aircraft cockpit user-interface: Using rule-based expert system models," *PC AI*, vol. 15, no. 6, pp. 21–25, 2001.

[59] A. Degani, A. Gellatly, and M. Heymann, "HMI aspects of automotive climate control systems," in *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, 2011, pp.1795–1800.

[60] F. Paterno, "A theory of user-interaction objects," *J. Visual Languages Comput.*, vol. 5, no. 3, pp. 227–249, 1994.

[61] P. Curzon, R. Rukšėnas, and A. Blandford, "An approach to formal verification of human-computer interaction," *Formal Aspects Comput.*, vol. 19, no. 4, pp. 513–550, 2007.
[62] R. Rukšėnas, P. Curzon, J. Back, and A. Blandford, "Formal modelling of cognitive interpretation," in *Proc. 13th Int. Workshop Design, Specification, Verification Interactive Syst.*, 2007, pp. 123–136.
[63] R. Runšėkas, J. Back, P. Curzon, and A. Blandford, "Formal modelling of salience and cognitive load," in *Proc. 2nd Int. Workshop Formal Methods Interactive Syst.*, 2008, pp. 57–75.
[64] R. Rukšenas, J. Back, P. Curzon, and A. Blandford, "Verification-guided modelling of salience and cognitive load," *Formal Aspects Comput.*, vol. 21, no. 6, pp. 541–569, 2009.
[65] H. Reza, S. Endapally, and E. Grant, "A model-based approach for testing GUI using hierarchical predicate transition nets," in *Proc. Fourth Int. Conf. Inf. Technol.*, 2007, pp. 366–370.
[66] L. Duan, A. Hofer, and H. Hussmann, "Model-based testing of infotainment systems on the basis of a graphical human-machine interface," in *Proc. 2nd Int. Conf. Advances Syst. Testing Validation Lifecycle*, 2010, pp. 5–9.
[67] J. C. Campos and M. Harrison, "Formally verifying interactive systems: A review," in *Proc. Fouth Int. Eurographics Workshop Design, Specification, Verification Interactive Syst.*, 1997, pp. 109–124.
[68] G. D. Abowd, H. Wang, and A. F. Monk, "A formal technique for automated dialogue development," in *Proc. 1st Conf. Designing Interactive Syst.*, 1995, pp. 219–226.
[69] F. Paternò, "Formal reasoning about dialogue properties with automatic support," *Interacting Comput.*, vol. 9, no. 2, pp. 173–196, 1997.
[70] J. C. Campos and M. D. Harrison, "Systematic analysis of control panel interfaces using formal tools," in *Proc. 15th Int. Workshop Design, Verification Specification Interactive Syst.*, 2008, pp. 72–85.
[71] A. Joshi, S. P. Miller, and M. P. Heimdahl, "Mode confusion analysis of a flight guidance system using formal methods," in *Proc. 22nd Digital Avionics Syst. Conf.*, Oct. 2003, pp. 2.D.1-1–2.D.1-12.
[72] N. G. Leveson, L. D. Pinnel, S. D. Sandys, S. K, and J. D. Reese, "Analyzing software specifications for mode confusion potential," presented at the *Workshop Human Error Syst. Develop.*, Glasgow, U.K., 1997.
[73] K. Loer and M. D. Harrison, "An integrated framework for the analysis of dependable interactive systems (IFADIS): Its tool support and evaluation," *Automated Softw. Eng.*, vol. 13, no. 4, pp. 469–496, 2006.
[74] J. C. Campos, G. Doherty, and M. D. Harrison, "Analysing interactive devices based on information resource constraints," *Int. J. Human-Comput. Studies*, vol. 72, no. 3, pp. 284–297, 2014.
[75] L. Moura, S. Owre, H. Rue, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari, "SAL 2," in *Computer Aided Verification* (ser. Lecture Notes in Computer Science). Berlin, Germany: Springer, 2004, vol. 3114, pp. 496–500.
[76] B. Dutertre and M. Sorea, "Timed systems in SAL," SRI Int., Menlo Park, CA, USA, Tech. Rep. SRI-SDL-04-03, 2004.
[77] M. Oishi, I. Mitchell, A. Bayen, C. Tomlin, and A. Degani, "Hybrid verification of an interface for an automatic landing," in *Proc. 41st IEEE Conf. Decision Control*, 2002, pp. 1607–1613.
[78] M. Oishi, I. Hwang, and C. Tomlin, "Immediate observability of discrete event systems with application to user-interface design," in *Proc. 42nd IEEE Conf. Decision Control*, 2003, pp. 2665–2672.
[79] E. J. Bass, K. M. Feigh, E. Gunter, and J. Rushby, "Formal modeling and analysis for interactive hybrid systems," presented at the *4th Int. Workshop Formal Methods Interactive Syst. Potsdam*, Germany, 2011.



Matthew L. Bolton (S'05–M'10) received the M.S. and Ph.D. degrees in systems engineering from the University of Virginia, Charlottesville, VA, USA, in 2006 and 2010, respectively.

He is currently an Assistant Professor with the Department of Industrial and Systems Engineering, State University of New York at Buffalo, Amherst, NY, USA. His research focuses on the development of tools and techniques that use human performance modeling and formal methods to analyze, design, and evaluate safety-critical systems.

Noelia Jiménez received the M.S. degree in physics sciences from Complutense University, Madrid, Spain, in 1994.

She currently works at IXION Industry and Aerospace, Madrid, in the Ground Segment area devoted to M&C systems, Virtual Control Centers, and Collaborative Systems. She is a Project Manager of IXION M&C systems for European Space Agency and works as an expert consultant on M&C architectures and HMI for R&D projects for the development of ground control centers for autonomous vehicles, combining support for 3-D visualization and new HMI-multimodal interfaces. She is the Manager of the internal Advanced User Interfaces Laboratory, IXION.

Marinus M. van Paassen (M'08) received the M.Sc. and Ph.D. degrees from the Delft University of Technology, Delft, The Netherlands, in 1988 and 1994, respectively, for studies on the neuromuscular system of pilot arms in manual control.

He is an Associate Professor in aerospace engineering with the Delft University of Technology, working on human–machine interaction and aircraft simulation. His work on human–machine interaction ranges from studies of perceptual processes and human manual control to complex cognitive systems. In the latter field, he applies Cognitive Systems Engineering analysis (Abstraction Hierarchy, Multilevel Flow Modeling) and Ecological Interface Design to the work domain of vehicle control.

Maite Trujillo received the M.Sc. degree in decision modeling and information systems and the Ph.D. degree in electronic and computer engineering from Brunel University, London, U.K, and the M.B.A. degree in international management from the Raj Soin College of Business, Dayton, OH, USA. She also holds Professional Certificates in Strategic Decision and Risk Management from Stanford University, Palo Alto, CA, USA, and Aviation Safety and Accident Investigation from Embry-Riddle Aeronautical University, Daytona Beach, FL, USA.

She manages the independent verification compliance of space debris mitigation and re-entry requirements for ESA projects and leads the human rating of future crewed transportation systems.