Bolton, M. L., Houser, A., & Molinaro, K. (2019). A formal method for assessing the impact of task-based erroneous human behavior on system safety. *Reliability Engineering and System Safety*, *188*, 168-180.

A Formal Method for Assessing the Impact of Task-based Erroneous Human Behavior on System Safety

Matthew L. Bolton[a,*], Kylie A. Molinaro[a], Adam Houser[a]

[a] *University at Buffalo, State University of New York, Department of Industrial and Systems Engineering, Buffalo, NY, USA*

Abstract

Erroneous human behavior is often cited as a major factor to system failure. However, the complexity of the human-automation interaction can make it difficult for engineers to anticipate how erroneous human behavior can contribute to failures. In this work, we introduce a novel method for generating human errors based on the task-based taxonomy of erroneous human behavior. This allows erroneous acts to manifest as divergences from task models. We implement our method using the Enhanced Operator Function Model. We further show how the method can be used with formal system modeling and formal verification with model checking to prove whether or not potentially unanticipated erroneous behavior could contribute to system failures. We evaluate how our method scales and use it to evaluate three case studies: a radiation therapy machine, a pain medication pump, and an Apache helicopter. We discuss these results and explore options for future work.

Keywords: Human error, erroneous human behavior, task analysis, system safety, formal methods.

1. Introduction

Erroneous human behavior, where a human diverges from a normative plan or task [1], is regularly cited as a source of failure in complex systems [1–4]. It contributes to more than 1,000,000 injuries and between 44,000 and 98,000 deaths each year in medicine [5]; roughly 75% of all accidents in general aviation and 50% in commercial aviation [6–8]; one third of unmanned aerial system (UAS) accidents [9]; 90% of automobile crashes [10]; most fratricide incidents in military operations [11]; and high profile disasters like the accident at Three Mile Island [12]. Humans are often blamed for failures associated with erroneous human behaviors. However, the modern perspective on this issue holds that such failures are the result of shortcomings in the design of systems' human-automation interaction and thus not solely the fault of human operators. Unfortunately, the complexity of these systems and the inherent concurrency that exists in human-automation interaction can make it extremely difficult for system designers to predict exactly how erroneous behavior can occur and how it can cause problems.

A growing body of work has been investigating how the exhaustive analysis capabilities of formal methods, such as model checking, can be used to evaluate human-automation interaction [13–15]. In particular, techniques have been developed that enable an analyst to pair models of human task behavior (based on a task analysis) with models of system behavior to determine when normative or erroneous human behavior can play a role in violations of system safety [16–27]. While powerful, these techniques can miss critical human-automation interactions. Methods that only consider normative human behavior neglect the impact of erroneous acts [16–19]. Methods that require analysts to manually include erroneous human behaviors require them to anticipate what erroneous behavior will potentially be problematic [20, 21, 24]. Finally, methods that automatically generate erroneous human behaviors using deviations from tasks are grounded in incompatible genotype- and phenotype-based taxonomies [22, 23, 25–27, 27]. Thus, these approaches predict different types of erroneous human behaviors while missing others [28].

In this paper, we introduce a novel approach to generating erroneous human behavior in formal task analytic behavior models so that model checking can be used to evaluate its impact on system safety. This new method makes use of the task-based taxonomy of erroneous human behavior, which connects the leading phenomenological and genotypical erroneous behaviors based on where erroneous behaviors deviate from a task [29]. By using this taxonomy, our new erroneous behavior generation method offers a more complete means of assessing the impact of erroneous human behavior on system safety. Below we discuss the background necessary for understanding our method. We then describe the method. This is followed by results that assess how the method scales and the ability of the method to detect both known and unknown problems in realistic case studies. The first two cases come from the literature (a pain medication pump [30] and a radiation therapy machine [22]), where legacy phenotype- and genotype-based generation methods where previously employed but could not reproduce the failure conditions found by each other [28]. By using the new generation technique to evaluate these applications, we show our new methods accounts for the erroneous behaviors of the two, legacy, incompatible approaches. The third case, the firing procedures of an Apache helicopter, is unique to this approach and demonstrates

*Corresponding author
Email address: mbolton@buffalo.edu (Matthew L. Bolton)

the ability of the method to discover both known and previously unknown system failures. We ultimately discuss our results and future research directions.

2. Background

Below we cover the necessary background for understanding our method. This includes an overview of formal methods and model checking, how they have been used in erroneous behavior analyses, the Enhanced Operator Function Model (EOFM) task analytic modeling formalism, and the task-based taxonomy of erroneous behavior.

2.1. Formal Methods and Model Checking

Formal methods are techniques and tools for mathematically modeling, specifying, and proving properties about (formally verifying) systems [31]. Formal models mathematically describe the behavior of a target system. Specification properties describe conditions that should always be true in the system. Formal verification is the process of mathematically proving if the formal model satisfies the specifications. Formal methods can be applied in a number of different ways. Early approaches used manual "pen and paper" proofs to show the correctness of computer programs. However, there are now semi-automated and automated theorem proving tools to help analysts perform these proofs [32–35]. One such tool is model checking.

Model checking is an automated approach to formal verification [35]. In this, the formal model describes a system as a state machine: a collection of variables whose values indicate state and transitions between states based on inputs and current state. Specification properties are asserted using model variables, Boolean operators, and other model logic operations (usually temporal logic [36]). For verification, a software tool exhaustively searches through the formal model's statespace. If it finds a violation, the model checker returns a trace through the model (called a counterexample) that shows exactly how the violation occurred. If no violation is discovered, then the model checker successfully proved the system model satisfies the specification.

The major limitation of model checking is scalability [35]. This is because it suffers from the "state explosion problem:" statespace size increases exponentially as concurrent elements are added to it. This can quickly lead to models that are too big (exceed machine memory) or take too long to verify. Because of this, active research is focused on addressing this issue [37].

Formal methods are mostly used to engineer and analyze computer hardware and software [31, 38]. They have also been used to evaluate the safety and reliability of abstract representations of complex and cyberphysical systems [39–43]. Because they are adept at finding problems that arise from interactions between components in complex systems, researchers have been exploring how formal methods (and especially model checking) can be used for human interactive systems [13–15, 44–47]. This includes analyses of interface usability, methods for discovering mode confusion and automation surprise, and analyses for determining the impact of human behavior and perception on system safety and performance [14]. In this work, we are predominantly concerned with how task analytic behavior models have been used with formal methods to evaluate the impact of erroneous human behavior. This is discussed in depth below.

2.2. Formal Methods and Task-based Erroneous Behavior

Task analysis is a systematic process analysts use to describe all of the different ways human operators normatively achieve goals with a system [48–50]. This is commonly documented using a hierarchical task model. Such a model is a collection of individual tasks, where each is represented as a hierarchy of goal-directed activities that decompose into other activities and, at the lowest level, actions. Strategic knowledge (condition logic) and internal operators control when and how activities can execute in relation to each other and the operational environment. Task analytic models are some of the most successful technologies developed by human factors engineers and are thus widely used.

Task models can be interpreted formally. This allows them to be included in larger formal system models that contain a formal description of 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 manually describe normative task models in formal notations as part of larger systems models [51, 52] or translate native task model notations into a formalism in which other system elements are represented [16–19, 21, 53, 54].

Further, investigators have explored how erroneous behaviors can be incorporated into the task models so that their impact on system safety and performance can be evaluated with model checkers. Erroneous behaviors can either be generated manually using patterns or automatically using different theories of erroneous behavior [22, 23, 25–27, 30, 54, 55]. With few exceptions (like the research that has modeled communication errors [26]), these approaches rely on two different taxonomies of erroneous behavior to inform their process. This first is Hollnagel's [1] phenotypes of erroneous behavior, which classifies erroneous behavior based on how they observably deviate from a normative plan of action. Generation methods based on this classification focus of permuting task models to replicate the erroneous phenotypes [20–22, 24, 56, 57]. The second taxonomy concerns the genotypes of erroneous behavior found in Reason's Generic Error Modeling System (GEMS) [2]. In this, erroneous behaviors are classified based on their cognitive causes: slips of attention (problems with the execution of a plan or task), rule-based mistakes (wrong plan), or knowledge-based mistakes (formulating a plan wrong). For work where the human operator is properly trained and knows his or her task, slips are the most relevant. Thus research on replicating erroneous human behavior in task models tends to focus on the generation of slips [23, 25, 27, 30], but can also account for mistakes [54].

While all of these approaches have definite utility, they are limited. First, while genotype-based erroneous behavior generation methods tend to scale linearly with the number of erroneous human behaviors [30], the phenotype-based approaches scale exponentially [22]. This can limit the applicability of the method. Second, erroneous behaviors predicted by the two approaches

are often divergent [28]. In particular, erroneous behaviors generated from the phenomenological perspective tend to be better at evaluating the robustness of system to low-level extraneous actions. Genotype-based methods tend to manifest higher in the task (usually based on failures of attention for task strategic knowledge). Thus, even these formal methods can miss critical system interactions. The recent task-based taxonomy of erroneous human behavior [29] unified the phenomenological and genotypical perspectives and thus offers opportunity for addressing the shortcomings of the previous generation approaches. This taxonomy is discussed next.

2.3. EOFM and the Task-based Erroneous Behavior Taxonomy

The task-based taxonomy of erroneous human behavior is formulated using EOFM. Thus, we discuss EOFM before detailing the taxonomy.

2.3.1. EOFM

EOFM [19, 58] is an XML-based task analytic modeling formalism that represents human behavior as an input/output model. Inputs are system elements that are external to the human, such as interface display components or factors from the environment. Outputs are human actions. The operators' tasks describe how human actions occur based on input and local variables (representing perceptual or cognitive processes).

EOFMs represent a task as a hierarchy of goal-directed activities that decompose into sub-activities and, at the bottom, atomic actions. EOFMs can assert strategic knowledge explicitly as Boolean expressions using input and local variables that assert what must be true for them to start executing (*Precondition*s), repeat (*RepeatCondition*s), or complete (*CompletionCondition*s). Any activity can decompose into one or more other activities or one or more actions. A decomposition operator (Table 1) specifies how many sub-activities or actions can execute and the temporal relationships between them.

Observable, atomic human actions or internal (cognitive or perceptual) actions exist at the bottom of the task hierarchy. Observable actions have three possible behaviors: *AutoReset* actions happen as a single event; *Toggle* actions switch between occurring and not occurring when the action is performed; and *SetValue* actions convey a value that is more complex than occurrence or non-occurrence. Non-observable (internal) actions allow internal behaviors (such as remembering something) to be represented as local variable assignments.

Table 1: Decomposition Operators

Operator	Description
optor_seq	Zero or more of the activities or actions in the decomposition must execute in any order, one at a time.
optor_par	Zero or more of the activities or actions in the decomposition must execute in any order and can execute in parallel.
or_seq	One or more of the activities or actions in the decomposition must execute in any order one at a time.
or_par	One or more of the activities or actions in the decomposition must execute in any order and can execute in parallel.
and_seq	All of the activities or actions in the decomposition must execute in any order, one at a time.
and_par	All of the activities or actions in the decomposition must execute in any order and can execute in parallel.
xor	Exactly one activity or action in the decomposition executes.
ord	All activities or actions must execute in the order they appear in the decomposition.
sync	All actions in the decomposition must execute synchronously.

EOFMs can be represented visually as tree-like graphs (examples are shown later in Figs. 7 and 10 to 12). Actions are rectangles and activities are rounded rectangles. An activity's decomposition is an arrow labeled with the decomposition operator. The arrow points to a rounded rectangle containing the decomposed activities or actions. Strategic knowledge conditions are triangles and/or arrows connected to the activity that they constrain. These are labeled with the Boolean logic of the condition. A *Precondition* is a yellow, downward triangle; a *CompletionCondition* is a magenta, upward triangle; and a *RepeatCondition* is an arrow recursively pointing to the activity.

EOFMs have formal semantics [19, 60]. This provides an unambiguous, mathematical descriptions of how they execute. For this, every activity and action is treated as a state machine (Fig. 1) that transitions between three execution states: *Ready* (waiting to execute), *Executing*, and *Done*. An activity or action starts in the *Ready* state. It transitions between states based on whether or not the specific Boolean conditions on the labeled transitions (Fig. 1) are true.

The strategic knowledge conditions of an activity (*Precondition*s, *RepeatCondition*s, and *CompletionCondition*s) are used to partially describe when these transitions can occur. However, three additional implicit conditions are also required. These assert whether an activity can start, end, or reset based on the given activity's or action's position in the task. Specifically, a *StartCondition* indicates if an activity can start executing based on the execution states of its parent, its parent's decomposition operator, and its siblings (activities or actions in the same decomposition). An *EndCondition* indicates if an activity or action can end execution based on the execution state of its children (activities or actions the activity decomposes into) and its decomposition operator. Since an action has no children, its *EndCondition* is true when the action has been properly executed. Finally, a *Reset* condition indicates when an activity or action can return to the *Ready* execution state. A *Reset* occurs when an a top-level activity reaches *Done* (and broadcasts a *Reset* to all of its decedents) or is issued to all of an activities dependents when it repeats (this is shown as a transition **With Reset** in Fig. 1). More details on these conditions can be found in [19, 60].

The formal semantics of EOFM are used as the basis for an automated translator (implemented in Java) that interprets the EOFM XML and outputs a formal representation. This can be part of a larger formal system model [19, 60] that the analyst manually implements using the notation of the Symbolic Analysis Laboratory (SAL) [59]. The the model created by the translator contains a module (a sub-model) representing the human task behavior (modules describing the rest of the model elements must be manually described by the analyses). This module has input variables from the XML markup, outputs representing human actions (the type of *SetValue* actions is specified in the XML; all other human actions are Boolean), local variables representing local variables explicitly defined in the XML,

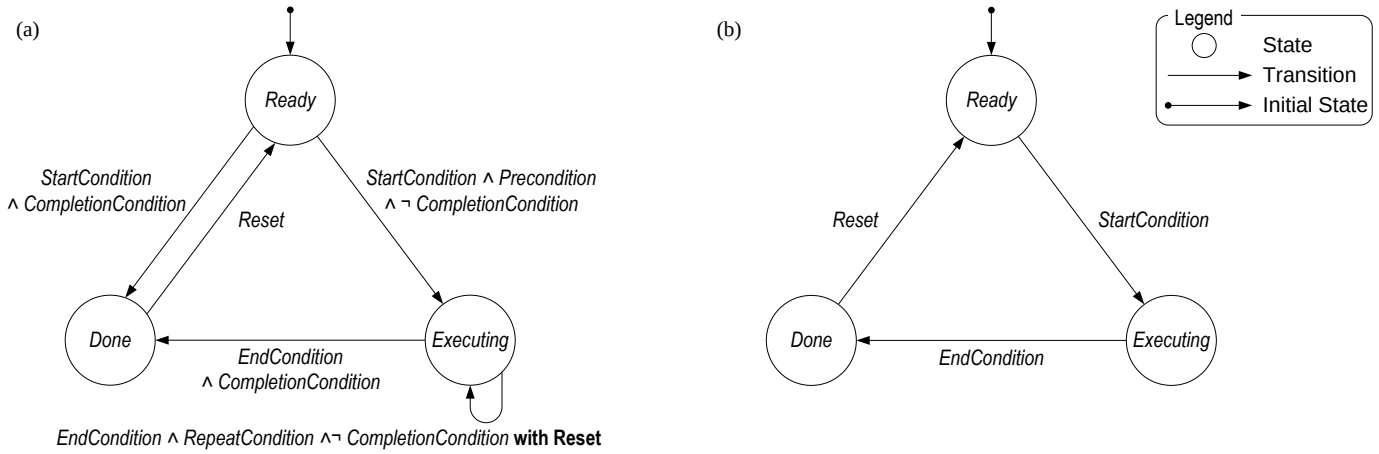

Figure 1: EOFM transition semantics.

```
% Ready to Executing transition for an activity
[] Activity = Ready AND (StartCondition) AND (Precondition)
   AND NOT (CompletionCondition) -->
      Activity' = Executing;
% Ready to Done transition for an activity
[] Activity = Ready AND (StartCondition) AND (CompletionCondition) -->
      Activity' = Done;
% Executing to Executing (a repeat) transition for an activity
[] Activity = Executing AND (EndCondition) AND (RepeatCondition)
   AND NOT (CompletionCondition) -->
      Activity' = Executing;
      % All sub-activities and actions are set to Ready (they are Reset)
      SubAct'   = Ready;
      ...
% Executing to Done transition for an activity
[] Activity = Executing AND (EndCondition)
   AND (CompletionCondition) -->
      Activity' = Done;
```
(a)

```
% Done to Ready transition (Reset) for an activity with no parent
[] Activity = Done -->
      Activity' = Ready;
      % All sub-activities and actions are set to Ready (the are Reset)
      SubAct'   = Ready;
      ...
```
(b)

```
% Ready to Executing transition for a humanaction with autoreset behavior
[] AutoResetAction = Ready AND (StartCondition) AND InterfaceReady -->
      AutoResetAction'       = Executing;
      AutotResetActionValue' = TRUE;
      Submitted'             = TRUE;
```
(c)

```
% Ready to Executing transition for a humanaction with toggle behavior
[] ToggleAction = Ready AND (StartCondition) AND InterfaceReady -->
      ToggleAction'      = Executing;
      ToggleActionValue' = NOT ToggleActionValue;
      Submitted'         = TRUE;
```
(d)

```
% Ready to Executing transition for a humanaction with setvalue behavior
[] SetValueAction = Ready AND (StartCondition) AND InterfaceReady -->
      SetValueAction'      = Executing;
      SetValueActionValue' = Value;
      Submitted'           = TRUE;
```
(e)

```
% Ready to Done (via Executing) transition for a localvariable action
[] LocalVariableAction = Ready AND (StartCondition) -->
      LocalVariableAction' = Done;
      LocalVariable'       = Value;
```
(f)

```
% Executing to Done transition for actions
[] Submitted AND NOT InterfaceReady -->
      Submitted'          = FALSE;
      % All Executing humanactions are set to Done and unchanged otherwise
      Action'             = IF Action = Executing THEN Done ELSE Action ENDIF;
      ...
      % All autoreset actions are set to FALSE (others are unchanged)
      AutoResetAction' = FALSE;
      ...
```
(g)

Figure 2: Patterns of SAL transitions (adapted from [58]) used for activities and actions to transition between their execution states. In SAL notation (see [59]), `[]` indicates the beginning of a nondeterministic transition. This is followed by a Boolean expression representing the transition's guard. The guard ends with a `-->`. This is followed underneath by a series of variable assignments where a `'` on the end of a variable indicates that the variable is being used in the next state. Colors enhance code readability. Comments are green, variables are dark blue, reserved SAL words are light blue, and values are orange. Things in purple represent expressions such activity or action conditions. (a) Shows all of an activity's transitions except for the Reset one from Fig. 1(a). (b) Represents the activity Reset transition from Fig. 1(a). Note that this Reset transition only occurs for activities at the top of a task model hierarchy (ones with no parent). Other resets occur based on the assignments that occur in these types of transitions or in repeat transitions (see (a)). (c)–(f) show the patterns of SAL transitions used for an action to transition from Ready to Executing and/or from Ready to Done with an implied execution in between (see Fig. 1(b)). (c) presents the pattern used for a humanaction with autoreset behavior, while (d) reports the pattern used for a humanaction with toggle behavior. (e) shows the pattern used for a humanaction with setvalue behavior. (f) displays the pattern used for a localvariable action. Note that in (e) and (f), `Value` is used to represent a variable or specific value from the EOFM's XML markup. (g) shows the SAL transition pattern used to handle all action Executing-to-Done transitions. Note that all transitions associated with actions that produce output behavior [(c), (d), (e), and (g)] use two Boolean variables to handle a coordination handshake with the other elements of the formal model (which are manually created by the analyst). An input variable `InterfaceReady` will always be *true* when the interface is ready to receive input. An output variable `Submitted` (which is initialized to *false*) is set to *true* when one or more human actions are performed. More details on the translation process can be found in [58].

and other local variables representing the execution state of the task's activities and actions. Activity and action execution state are defined using a SAL enumerated type with values *Ready*, *Executing*, or *Done* (as per the semantics in Fig. 1). Then, to describe the behavior of the human task, the module explicitly represents all of the edges from the formal semantics in Fig. 1 as nondeterministic, guarded transitions. Figure 2 presents patterns that show how these transitions are formed for each activity and action in a task. A deeper discussion of the EOFM translation process can be found in [58].

2.3.2. *The Task-Based Taxonomy of Erroneous Behavior*

EOFM was used as the basis for the task-based taxonomy of erroneous human behavior [29], a classification system that unifies the phenomenological (what) and genotypical (why) perspectives based on where erroneous behaviors can occur as

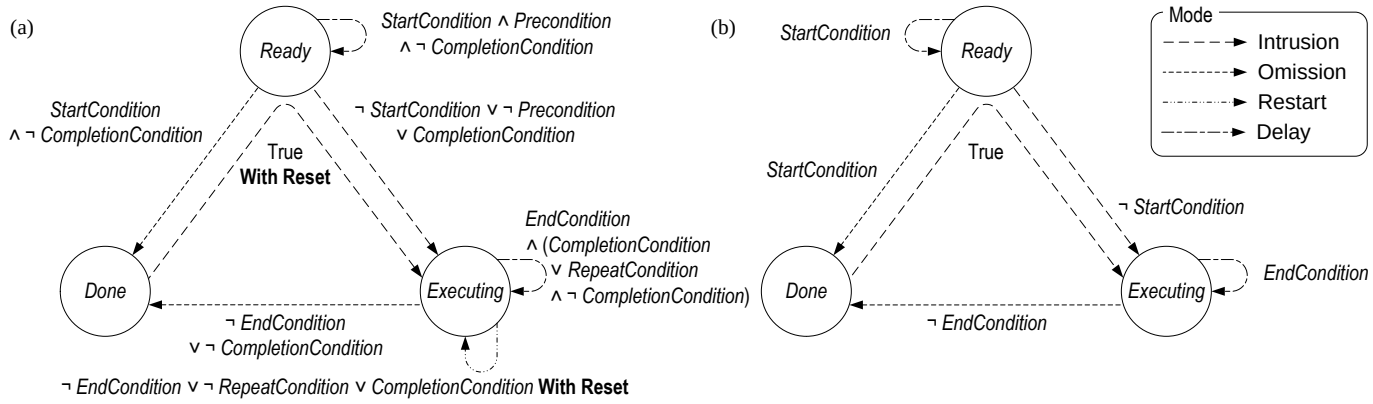

Figure 3: EOFM erroneous transition semantics.

deviations from tasks. The premise behind this is that by knowing where a deviation occurs in a task model, the way the error will manifest (the phenotype of erroneous action [1]), and the information the human operator failed to attend to (the genotype of the slip [2]) will also be apparent.

In the taxonomy, a human who erroneously diverges from a task will do so by violating the task's formal semantics. Thus, the taxonomy classifies erroneous human behavior based on how the formal semantics are violated. The nature of the violation shows at which activity or action the divergent behavior occurred, what the erroneous behaviors was (its phenotype), and which part of the formal semantics were violated and thus not properly attended to (the error's genotype). Taxonomy classifications are hierarchical. First, the taxonomy distinguishes between erroneous behaviors that originate at the activity or action levels. Task execution can diverge from the formal semantics either through violations of the execution state transition semantics or through incorrect action variable assignments that occur during the action's execution. Thus, the taxonomy further identifies the divergence type associated with an erroneous act. These divergences have limited ways they can manifest (erroneous behavior modes). These represent the next taxonomy level. Within each mode, the task-based taxonomy classifies an erroneous behavior based on the point of divergence from the semantics.

For transition-based behaviors, erroneous behavior modes represent erroneous transitions that can occur between activity and action execution states (see Fig. 3). An intrusion occurs whenever an activity or action transitions to executing when it should not. An omission occurs when an activity transitions to done when it should not. A restart occurs when an activity's execution restarts (it resets and starts executing form the beginning) when it should not. Finally, a delay occurs when an activity or action does not transition out of its current state when it should.

For execution-based erroneous behaviors, the mode is the type of variable assignment (the type of action) performed: a substitution for a *SetValue* action and a misremembrance for local variable assignments.

Each point of divergence is further refined and given meaningful erroneous behavior types based on what the human improperly attended to (for transition-based erroneous behaviors) or improper variable assignment (for execution-based errors).

Table 2: Action-level Erroneous Behavior Types

Assignment	Erroneous Behavior Type
CorrectAction := *IncorrectValue*	Action Value Substitution
IncorrectAction := *CorrectValue*	Action Target Substitution
CorrectVariable := *IncorrectValue*	Value Misremembrance
IncorrectVariable := *CorrectValue*	Target Misremembrance

The original paper on the taxonomy [29] explicitly shows how each of the Hollnagel's phenotypes of erroneous action and each of the slip genotypes from GEMS [2] are associated with deviations from task. However, for the purposes of the discussed research, it is only important to understand the classification of transition-based errors at the level of modes. For execution-based erroneous behaviors, there can be value and target types of substitutions and misremembrances (Table 2). In the former, an incorrect value is assigned to the target variable (the action or the local variable). In the latter, the correct value is assigned to the wrong target (action of local variable).

Note that the closest analog to the task-based taxonomy [29] is the system developed by Fahssi et al. [61]. Fahssi et al. showed how both the phenotypes and genotypes or erroneous behaviors could be systematically identified in task models as part of a human reliability analysis using a guided, manual, systematic process. While similar to the task based taxonomy [29], the system in [61] does not provide a systematic means of permuting the execution semantics of a task model. Thus it was not used in the work presented here.

3. Method

Existing EOFM-based formal verifications techniques [18, 19, 62] were extended to support erroneous behavior generation with the new taxonomy. In our method (Fig. 4), an analyst creates a normative task model as an EOFM. The analyst also constructs a formal system model representing the behavior of the system and environment the human interacts with. The Java-based EOFM to SAL translator uses the EOFM formal semantics (Fig. 1) and the erroneous behavior semantics (Fig. 3 and Table 2) to automatically incorporate both the normative

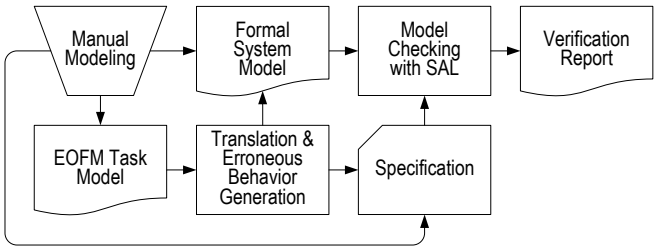

Figure 4: EOFM model checking analysis with the novel erroneous behavior generation technique incorporated into it.

EOFM task behavior and generated erroneous behaviors into the formal system model. The SAL model checker then evaluates whether system specifications created by the analyst or automatically generated from the EOFM task models [62] will always be true. A verification report will indicate if the specifications hold and, if not, show how they were violated with counterexamples. Counterexamples can be inspected using the EOFM counterexample visualizer [63].

The major contribution of this method comes from its new translation and erroneous behavior generation process. This works by modifying the way EOFM formal semantics were interpreted in the translated formal representation of the task behavior model. Specifically, during translation, when the normative transitions for each activity and action are being created using the patterns from Fig. 2, the task-based erroneous behaviors are incorporated into this formal representation by adding transitions that allow for the erroneous transitions in Fig. 3 and assignments from Table 2. Figure 5 shows the patterns that are used for all activities to represent the erroneous behavior modes from Fig. 3(a) in SAL's input language. The corresponding transitions generated for all actions (and replicating the modes from Fig. 3(b)) are shown in Fig. 6(a). The additional SAL transition patterns that are used to generate action-level erroneous behavior types (Table 2) when an action is *SetValue* or a local variable assignment are shown in Fig. 6(b) and (c). These additions enable activities and actions to execute, repeat, stop executing, or assign values incorrectly based on failures of attention in all of the different ways supported by the taxonomy.

A formal model that allows any erroneous behavior to manifest at any given time will create unbounded human behavior that will not be interesting to analysts. Further, including all possible erroneous behaviors in a given analysis will likely only find a particular combination of conditions that cause a failure for a given property. In this situation, an analyst may miss erroneous behaviors that can cause failures in different ways. The method addresses these issues in two different ways. First, the formal model keeps track of the total number of erroneous human behaviors allowed in a given analysis (`Max` in Figs. 5 and 6) and only allow erroneous behaviors to be performed if the maximum was not reached by counting (`Count` in Figs. 5 and 6) the number erroneous behaviors included in any given analysis. Second, the method allows analyst to have full control of which erroneous behaviors are generated. For example, an analyst can choose the default option of generating every erroneous behavior for every activity and action in a task model. However, the analyst can also choose to only generate erroneous behaviors for a given activity or actions (or set of activities and actions) as well as only allow specific modes or types of errors to be generated. Analysts are able to explicitly specify how these options are employed in the EOFM's XML markup prior to translation. The net effect of these features is that, formal verifications conducted with erroneous behavior generation will prove whether or not the modeled system is safe for up to the maximum number of erroneous human behaviors for the subset of analyst allowed erroneous behaviors.

4. Applications and Results

Below we present the case studies considered in our analyses. First we discuss are scalability analyses. This is followed by results of our analyses of legacy systems (the pain medication pump and the radiation therapy machine). Finally, we present our analysis of an Apache helicopter firing procedure.[1]

All of case study verifications were performed on a computer workstation with a 12-core, 3.6 GHz, Intel® Xeon® processor and 128 GB of RAM running Linux Mint 18.1.

4.1. Scalability Analyses

For the scalability tests, analyses were run using the three tasks shown in Fig. 7. These were designed to increase in complexity between tasks based on the number of actions and activities and the number of strategic knowledge conditions ((*C1*, *C2*, *C3*, and *C4*) representing *Precondition*s, *RepeatCondition*s, and *CompletionCondition*s). All used the *optor_par* decompositions because it results in the maximum number of model states [19].

Each task model was translated into a formal model and paired with a system model that updated the state of the strategic knowledge conditions in response to human actions. In all of the tasks, the human must perform a given action (i.e. *Action1*) until its associated condition (i.e. *C1*) is true.

Within each formal model, the maximum number of erroneous human behaviors included (applied to all activities and actions) was varied from 0 to 16. A true specification was verified against each model and the number of visited states and verification times were recorded (see Fig. 8). This shows that the number of model states and verification times all scaled linearly with the maximum number of erroneous human behaviors. This is a significant result because it shows that the erroneous behavior generation method can be applied to complex systems without the risk of exacerbating scalability (an issue that occurs when exponential increases occur).

4.2. Legacy Cases

To demonstrate the ability of this new method to discover situations where erroneous behavior can contribute to failures, we have used it to find problems discovered in older analyses [22, 23]. Specifically, we used the new method to evaluate the safety of a radiation therapy machine [22] (previously evaluated

[1] EOFM and SAL listings of all of the presented case study models can be found at `http://fhsl.eng.buffalo.edu/resources/`.

```
% Erroneous activity Ready to Ready (Delay) transition
[] Activity = Ready AND (StartCondition) AND (Precondition) AND NOT (CompletionCondition) AND (Count <= Max) -->
     Activity' = Ready;
     Count'    = Count + 1;
% Erroneous activity Ready to Done (Omission) transition
[] Activity = Ready AND (StartCondition) AND NOT (CompletionCondition) AND (Count <= Max) -->
     Activity' = Done;
     Count'    = Count + 1;
% Erroneous activity Ready to Executing (Intrusion) transition
[] Activity = Ready AND NOT (StartCondition) AND NOT (Precondition) AND (CompletionCondition) AND (Count <= Max) -->
     Activity' = Executing;
     Count'    = Count + 1;
% Erroneous activity Executing to Executing (Delay) transition
[] Activity = Executing AND (EndCondition) AND ((CompletionCondition) OR (RepeatCondition) AND NOT (CompletionCondition)) AND (Count <= Max) -->
     Activity' = Executing;
     Count'    = Count + 1;
% Erroneous activity Executing to Executing (Restart) transition
[] Activity = Executing AND (NOT (EndCondition) OR NOT (RepeatCondition) OR (CompletionCondition)) AND (Count <= Max) -->
     Activity' = Executing;
     % All sub-activities and actions are set to Ready (they are Reset)
     SubAct'   = Ready;
     ...
     Count'    = Count + 1;
% Erroneous activity Executing to Done (Omission) transition
[] Activity = Executing AND NOT (EndCondition) AND NOT (CompletionCondition) AND (Count <= Max) -->
     Activity' = Done;
     Count'    = Count + 1;
% Erroneous activity Done to Executing (Intrusion) transition
[] Activity = Done AND (Count <= Max) -->
     Activity' = Executing;
     Count'    = Count + 1;
```

Figure 5: Patterns of SAL transitions used for representing the erroneous behavior modes for activities. These represent each of the erroneous transitions from Fig. 3(a) as options beyond the normative transitions from Fig. 2(a) and (b). Note that `Max` represents the maximum number of erroneous behaviors allowed in a given analysis and `Count` represents a counter that tracks the number of erroneous behaviors that have occurred. None of these transition can occur unless the count has not reached the maximum (`Count <= Max`). Count is incremented when each transition does occur (`Count' = Count + 1`).

```
% Erroneous action Ready to Ready (Delay) transition
[] Action = Ready AND (StartCondition) AND (Count <= Max) -->
     Action' = Ready;
     Count'  = Count + 1;
% Erroneous action Ready to Done (Omission) transition
[] Action = Ready AND (StartCondition) AND (Count <= Max) -->
     Action' = Done;
     Count'  = Count + 1;
% Erroneous action Ready to Executing (Intrusion) transition
[] Action = Ready AND NOT (StartCondition) AND (Count <= Max) -->
     Action' = Executing;
     Count'  = Count + 1;
% Erroneous action Executing to Executing (Delay) transition
[] Action = Executing AND (EndCondition) AND (Count <= Max) -->
     Action' = Executing;
     Count'  = Count + 1;
% Erroneous action Executing to Done (Omission) transition
[] Action = Executing AND NOT (EndCondition) AND (Count <= Max) -->
     Action' = Done;
     Count'  = Count + 1;
% Erroneous action Done to Executing (Intrusion) transition
[] Action = Done AND (Count <= Max) -->
     Action' = Executing;
     Count'  = Count + 1;
```

(a)

```
% Erroneous Ready to Executing Action Value Substitution
[] SetValueAction = Ready AND (StartCondition) AND InterfaceReady AND (Count <= Max) -->
     SetValueAction'      =  Executing;
     SetValueActionValue' IN {x: ActionType | x /= CorrectValue};
     Submitted'           =  TRUE;
     Count'               =  Count + 1;
% Erroneous Ready to Executing Action Target Substitution
[] SetValueAction = Ready AND NOT (StartCondition) AND InterfaceReady AND (Count <= Max) -->
     SetValueAction'      =  Executing;
     SetValueActionValue' IN {x: ActionType | x /= CorrectValue};
     Submitted'           =  TRUE;
     Count'               =  Count + 1;
% Ready to Executing transition for a humanaction with setvalue behavior
[] SetValueAction = Done AND InterfaceReady AND (Count <= Max) -->
     SetValueAction'      =  Executing;
     SetValueActionValue' IN {x: ActionType | x /= CorrectValue};
     Submitted'           =  TRUE;
     Count'               =  Count + 1;
```

(b)

```
% Erroneous Ready to Done (via Executing) Value Misremembrance
[] LocalVariableAction = Ready AND (StartCondition) AND (Count <= Max) -->
     LocalVariableAction' =  Done;
     LocalVariable'       IN {x: ActionType | x /= CorrectValue};
     Count'               =  Count + 1;
% Erroneous Ready to Done (via Executing) Target Misremembrance with wrong value
[] LocalVariableAction = Ready AND NOT (StartCondition) AND (Count <= Max) -->
     LocalVariableAction' =  Done;
     LocalVariable'       IN {x: ActionType | x /= CorrectValue};
     Count'               =  Count + 1;
% Erroneous Done to Done (via Executing) Target Misremembrance with wrong value
[] SetValueAction = Done AND (Count <= Max) -->
     LocalVariableAction' =  Done;
     LocalVariable'       IN {x: ActionType | x /= CorrectValue};
     Count'               =  Count + 1;
```

(c)

Figure 6: Patterns of SAL transitions used for representing (a) the erroneous behavior modes (from Fig. 3(b)) and (b)–(c) the action-level erroneous behavior types (from Table 2) for actions. The transition patter in (a) are used to represent each of the erroneous transitions from Fig. 3(b) as options beyond the normative transitions from Fig. 2(c)–(g). The transition patterns in (b) are used to represent Action Value Substitution and Action Target Substitution erroneous behavior types from the upper half of Table 2. The transition patterns in (c) are used to represent Value Misremembrance and Target Misremembrance erroneous behavior types from the lower half of Table 2. `Max` and `Count` are as defined in the caption of Fig. 5. Note that in the above, a dark red color indicates a type. Further, the `IN` operator is used to indicate nondeterministic assignment to a variable to its left of a value from a set defined between the curly braces {...} to the right.

with generated erroneous phenotypes [1]) and a patient controlled analgesia pump [23] (a pain medication pump previously evaluated with generated slips [2]).

The new method was applied to these case studies with a maximum of one erroneous human behavior applied to all activities and actions. For both cases it was able to find the originally identified problems. For the radiation therapy machine, this was a condition where the human operator could accidentally select the wrong mode for the machine, correct it too quickly for the machine process, and administer a treatment that irradiates a patient. This analysis visited 702,048 states and took a total of 2.71 seconds.

The pain medication pump analysis found an error where a practitioner could enter an incorrect value into the pump, resulting in the administration of an incorrect prescription. This verification visited 141,2983,528 over 12,811.14 seconds. This is a significant result because the original erroneous behavior generation methods were incompatible and would not find the

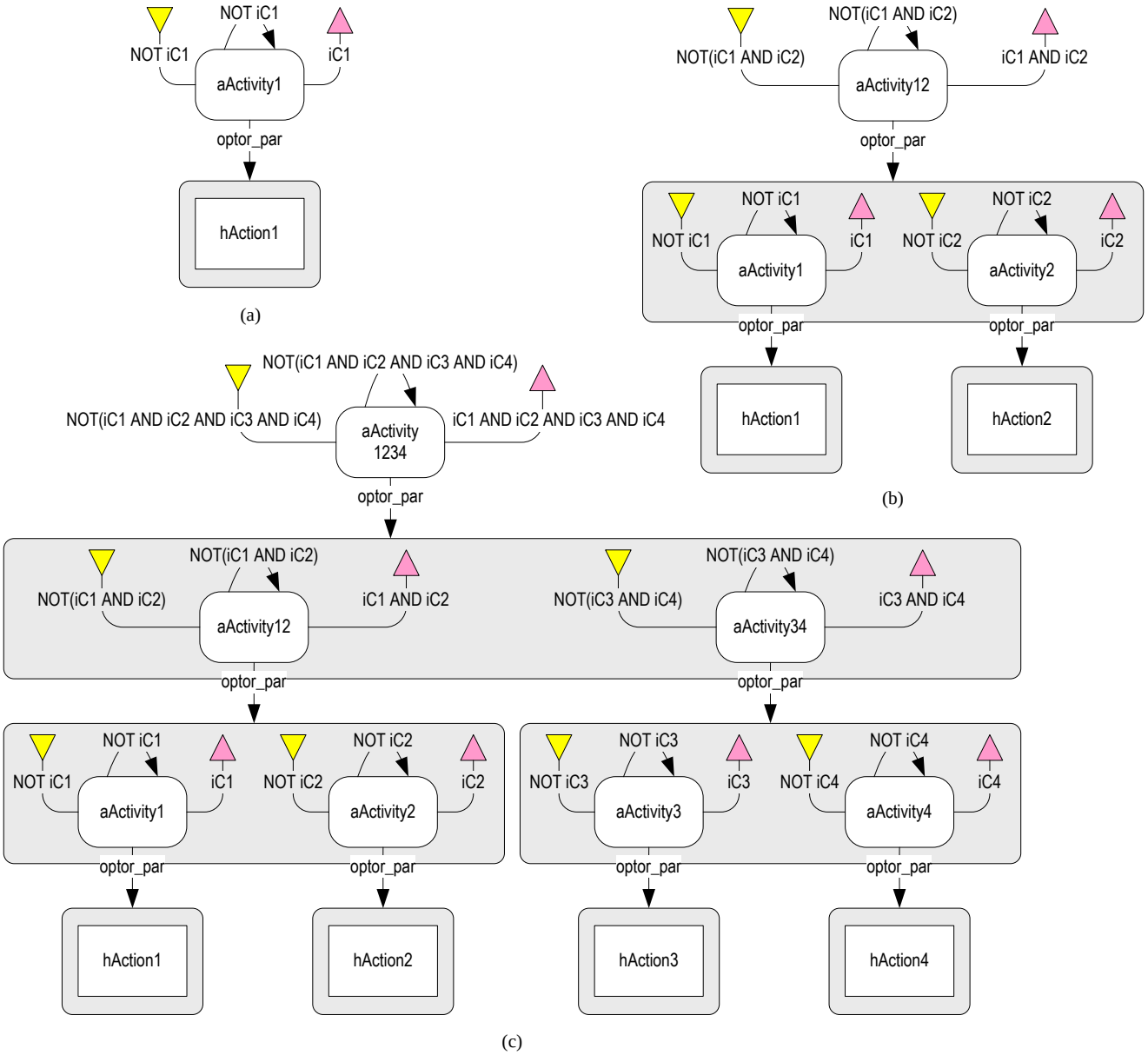

Figure 7: Instantiated EOFM normative task structures used as inputs scalability analyses. Activities begin with the letter "a" and actions do not. (a) The human operator must perform *Action1* until condition *C1* is true. (b) The human operator must perform *Action1* and *Action2* until *C1* and *C2* are true respectively. (c) The human operator must perform *Action1*, *Action2*, *Action3*, and *Action4* until *C1*, *C2*, *C3* and *C4* are true respectively.

issues of the other [28].

Further, for both case studies, we used our new approach to evaluate systems that incorporate the fixes discovered in the original papers: having the radiation therapy machine check power levels before allow treatment administration [22] and requiring prescription review before administration in the pump [23]. For both of these modified models, our method found no situations where a maximum of one erroneous human behavior could irradiate patients or produce incorrect prescription delivery, respectively. For this, the radiation therapy machine verified in 2.74 seconds having visited 685,632 states. The pain medication pump verified in 10,627.58 seconds having visited 1,198,592,964 states. These are positive results for the analyzed applications because they show that the fixes incorporated by the previous efforts are not only robust to the originally considered erroneous behaviors but also the more complete set of erroneous behaviors generated with the new method. More details on each of these applications can be found in [22, 23].

4.3. The Apache Helicopter Case

To see how our method could be used to evaluate erroneous behavior potential in a more complex application, we consider an Apache helicopter procedure for flying between targets, identifying them, and deciding whether to fire on them. Note that information about this application and its tasks came from a publicly available report [64].

In this application, an Apache helicopter is expected to fly over a given region. Bisecting this region laterally is a national

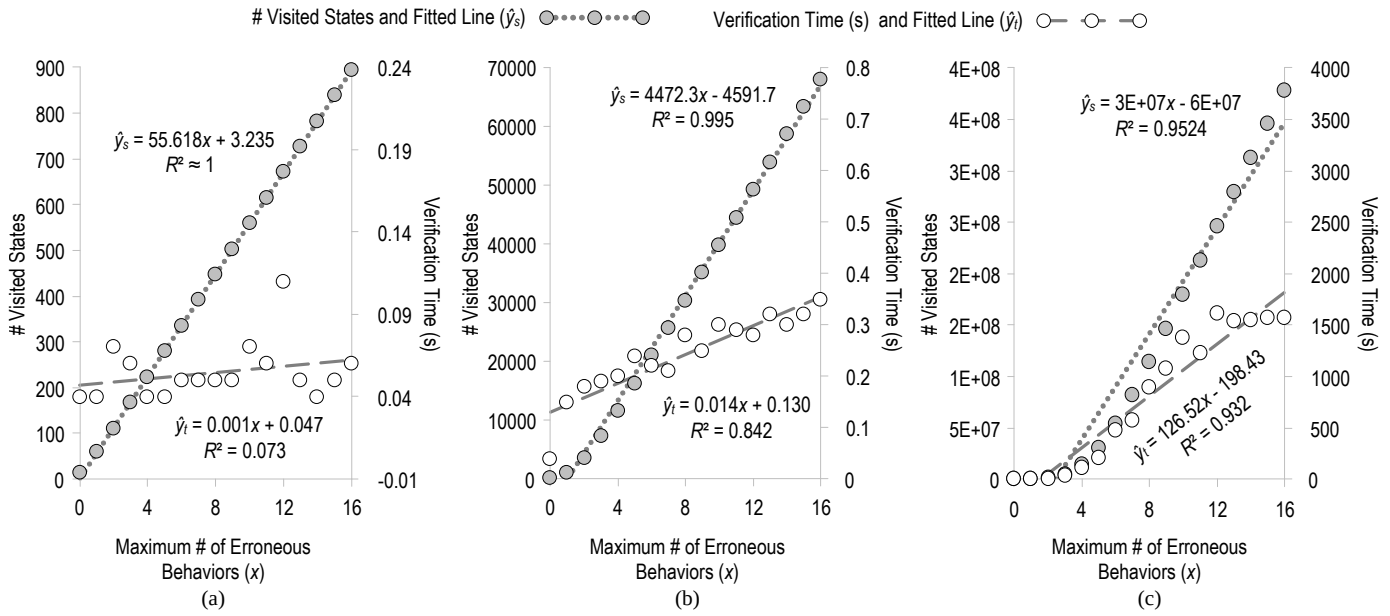

Figure 8: Scalability test results. These show that the number of visited states and the total verification time increase linear with the maximum number of erroneous behaviors included in the model checking analyses that used each of the respective task models [(a), (b), and (c)] from Fig. 7.

border. Below the border are friendly targets. Above it are hostile targets. The Apache helicopter pilots are expected to fly around this region to inspect targets, locate new targets, receive new locations overs the radio, identify encountered targets (as friendly or hostile by calling in over the radio), and, if a target is hostile, fire on it.

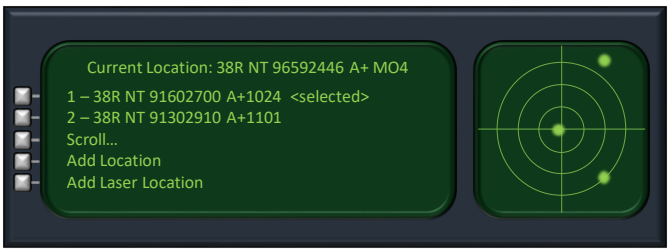

Figure 9: An Apache helicopter flight computer. This displays the helicopter's current location. The pilot can select a location to fly to, scroll through entered locations, manually add a location, or add a location by using an onboard laser for identifying the position of a something in the environment.

To assist pilots in their task, they have a flight computer (Fig. 9) that allows them to store locations for future reference. In our version of this system, two locations are displayed on the screen at a time and there can be up top four total locations in the computer. The pilot can scroll between options. He or she can also manually add locations to the computer that are received over the radio. Locations can further be added by using a laser to identify the position of an object in the environment.

The EOFM tasks for normatively controlling the helicopter are shown in Figs. 10 to 12. This includes the task for initiating flight (Fig. 10), the task for entering a new location from the radio (Fig. 11(a)), the task for identifying a new target that has been spotted by the pilot (Fig. 11(b)), and the task for engaging the target (Fig. 12).

When these tasks were converted into SAL using our new translator (initially with a maximum of 0 erroneous behaviors). The SAL version was paired with other formal representations of the system. This contained the flight state of the helicopter (iFlyState) which can be Landed (if the aircraft is not flying), EnRoute (if it is flying), or Hovering (if it is investigating and/or dealing with a potential target). The model also represents information available to the human operator through the flight computer: the current location of the aircraft (iCurrentLocation) which can be either above or below the border; as well as the locations currently displayed on the interface based on the current scroll location (iDisplayLocation1 and iDisplayLocation2); the set of all locations in the computer; and the selected location. This model is also capable of allowing a new radio location to become available (iNewRadioLocationAvailable, which can become true nondeterministically when the aircraft if flying), the value of this communicated to the pilot (iNewRadioLocation), and whether or not a request confirmation is given over the radio (iRadioConfirmation). Further, when an aircraft is flying, the aircraft can nondeterministically encounter a new target.

For this model, we wanted to check that no friendly fire incidents ever occurred. This was formulated in linear temporal logic (the specification language of SAL) as:

$$\mathbf{G}\neg \left(\begin{array}{l} \text{hFireWeapons} \\ \wedge \text{iTargetPresent} \\ \wedge \mathit{IsBelow}(\text{iCurrentLocation}) \end{array} \right). \qquad (1)$$

This can be interpreted as: globally ($\mathbf{G}$) it should never be true that the pilot fires the weapons when a target is present with the current location is below the border.

We formally verified the model with the specification in (1) both with normative human behavior and with erroneous human behavior generated using the new approach. Because these anal-

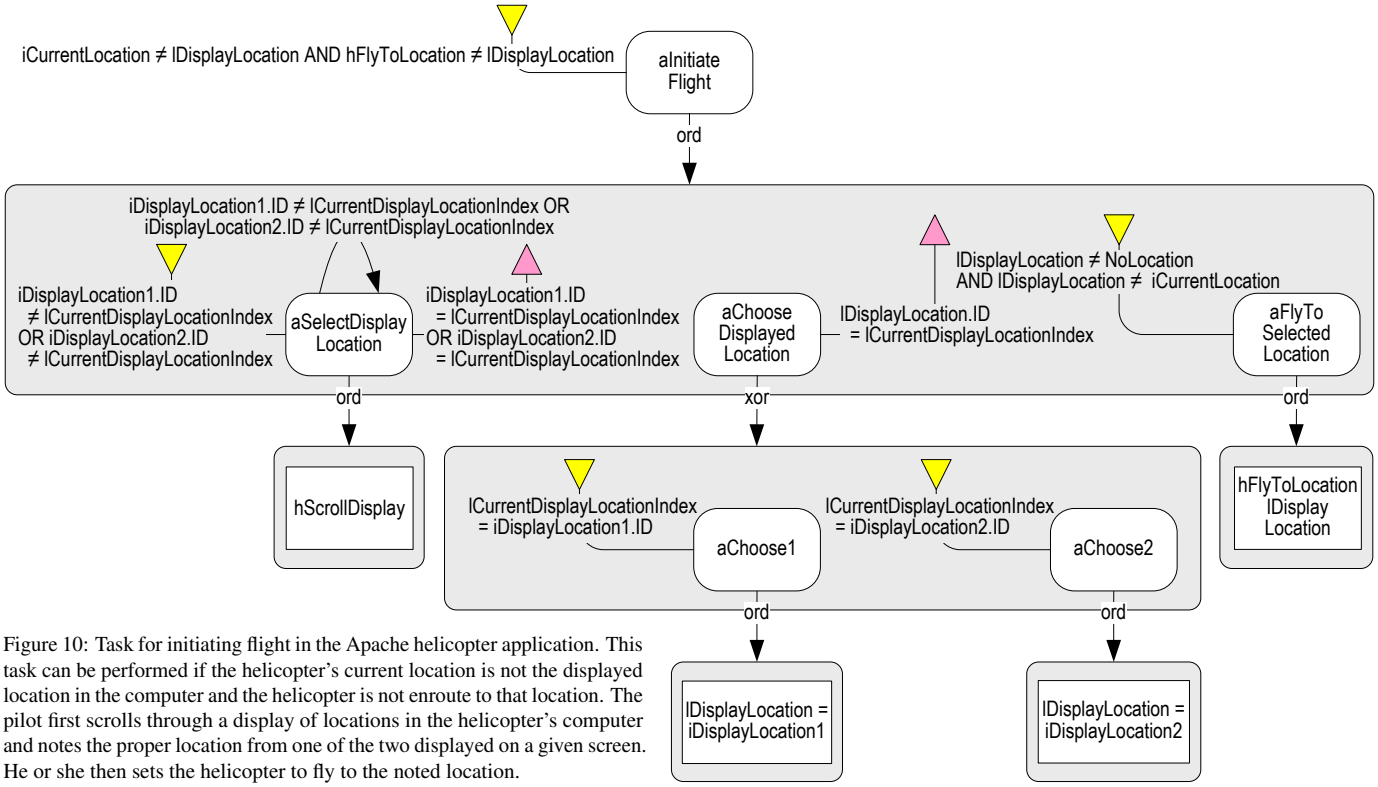

Figure 10: Task for initiating flight in the Apache helicopter application. This task can be performed if the helicopter's current location is not the displayed location in the computer and the helicopter is not enroute to that location. The pilot first scrolls through a display of locations in the helicopter's computer and notes the proper location from one of the two displayed on a given screen. He or she then sets the helicopter to fly to the noted location.

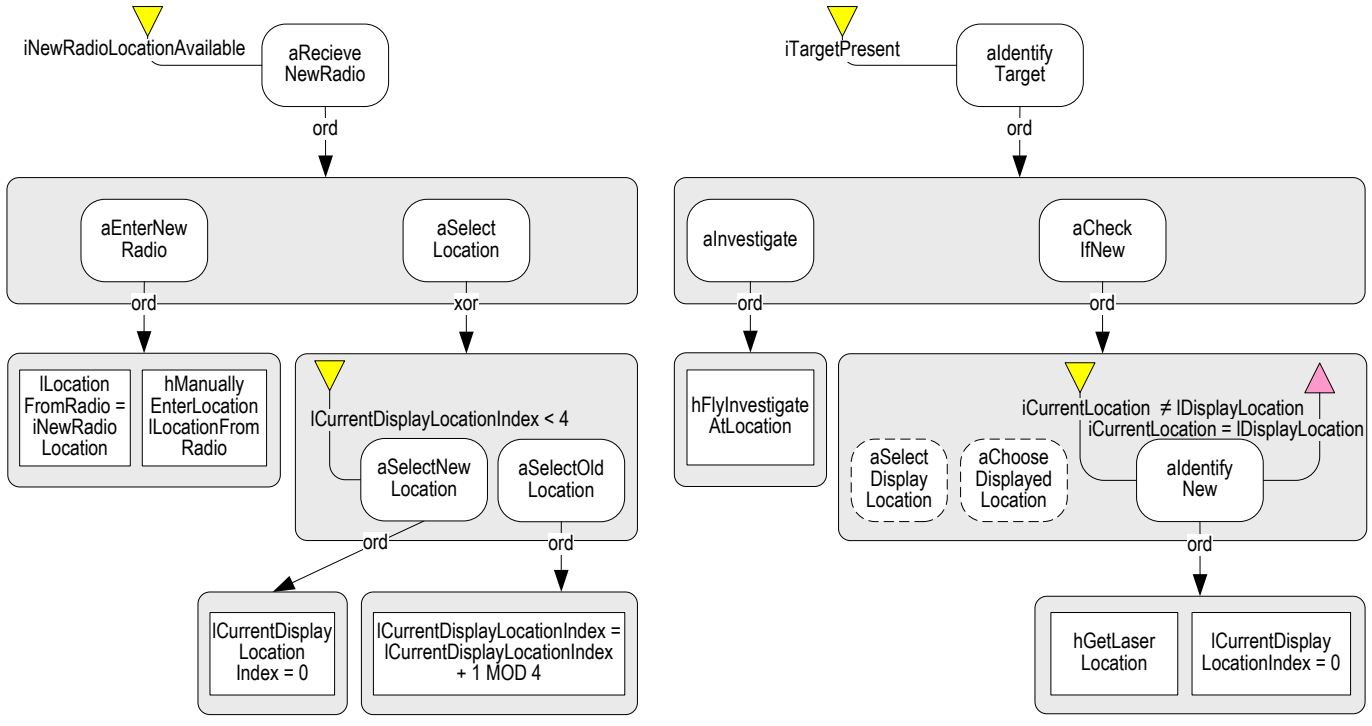

Figure 11: (a) Task from the Apache helicopter application showing how a human pilot receives a new location over the radio, manually enters the location into the computer (which was inserted immediately after the current location), and then selects this location. (b) Task from the Apache application showing how a human pilot responds to the identification of a target in the environment. To do this, the pilot first hovers the aircraft by investigating at the location. The pilot then identifies if the target is new based on the target locations in the computer. If the target is new, the pilot uses the helicopter's laser to obtain the target's location. Note that in the above activities with dotted lines are defined elsewhere (in this case Fig. 10).

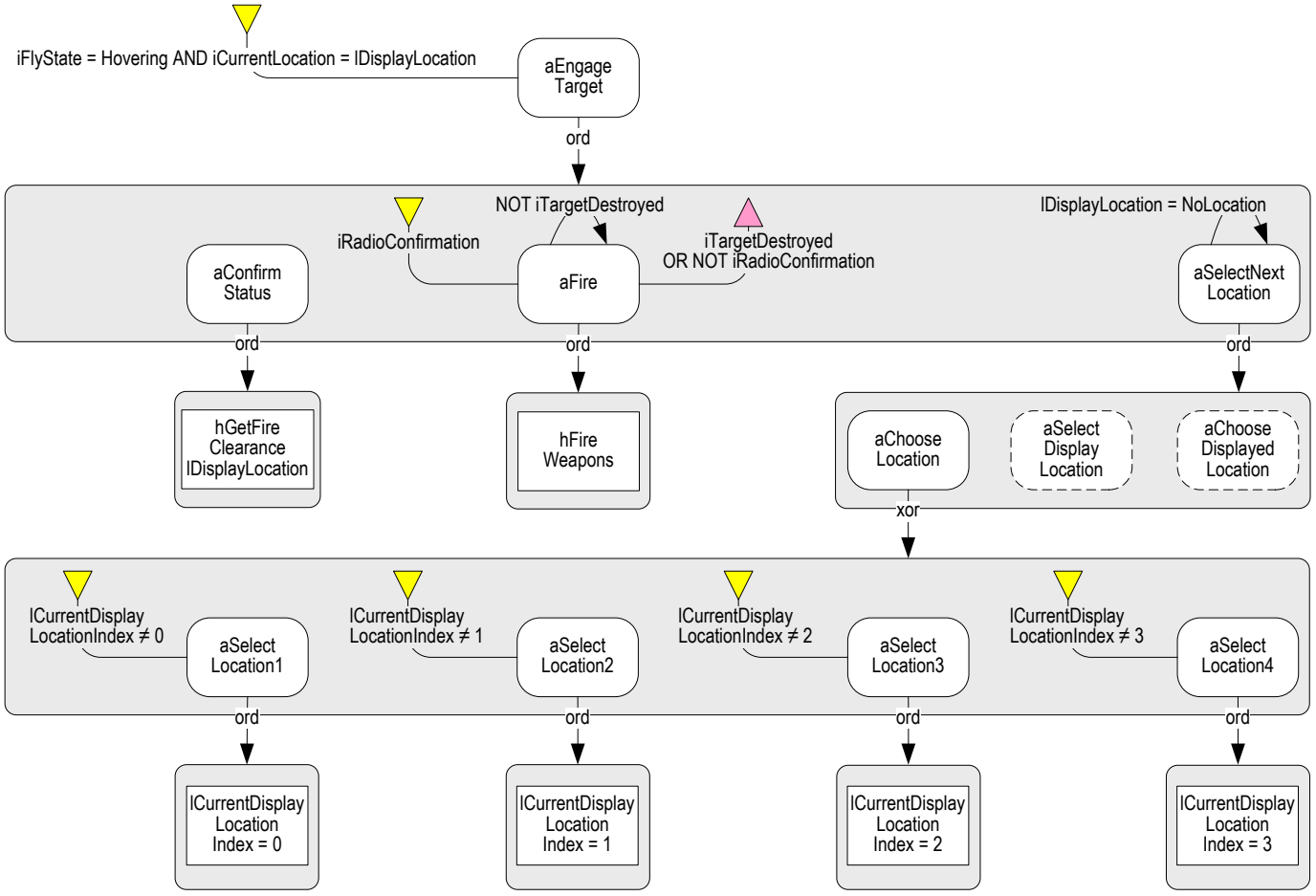

Figure 12: Task from the Apache application showing how a human pilot engages a target. If the helicopter is hovering and the helicopter is at the selected location, the pilot can engage the helicopter. This is done by first confirming whether or not to fire on the target by reading the displayed location over the radio. If a confirmation was received, the pilot can fire on the target until it is destroyed. The pilot must then find and select a new target to fly to.

yses strove to both replicate documented failures [64] and find new problems, we took a systematic approach to the erroneous behavior analyses. In particular, we exploited the feature of our method to selectively generate all possible erroneous behaviors for each activity and action between evaluations (all possible erroneous behavior that could manifest at a single activity or action were considered in given analyses). This resulted in 56 separate verifications. In total, these took just over 34 hours to complete and visited a sum total of 3,261,110,619 states. The average verification took 36.44 minutes and visited 58,234,118.2 states. The model without any erroneous human behaviors verified to true, thus it was not possible to commit friendly fire when the pilot behaved normatively. However, the other verifications found 39 different erroneous behaviors that could lead to friendly fire (see Table 3). While this sounds very concerning, an examination of the counterexamples associated with each found that many of the errors manifested in one of nine types of failure (thus each entry in Table 3 is associated with and clustered under a given "Note"). While all of these erroneous behaviors are possible and potentially dangerous, the behavior seen under note *E* is of particular interest. This is because a situation where a pilot gets clearance by remembering the wrong value (which could be another or previously entered location) matches the profile of actual fratricide incidents that have occurred with Apache helicopters [64]. The other failure conditions do not have analogues in the literature or accident reports. However, they do exhibit face validity and are thus worthy of future investigation.

5. Discussion and Conclusions

In this work, we introduced a novel approach to generating erroneous human behavior based on the task-based taxonomy of erroneous human behavior [29]. This approach can be used in formal verification analyses that contain information about both the normative human task behavior and the system the human is interacting with to predict how both anticipated and unanticipated erroneous behaviors can contribute to system failures. This work is significant for several reasons.

First, because the new method is based on the task-based taxonomy of erroneous human behavior, it is more complete than previous erroneous human behavior generation techniques [22, 23, 25, 27, 30, 54] that were based on incompatible phenotypes and genotypes. This capability was illustrated in the case study analyses, where our new method was able to find

Table 3: Verification Results for the Apache Helicopter Application

Activity / Action	Time (s)	# Visited states	Erroneous Behavior	Note
NA	1113.01	18,488,464	NA	[A]
aInitiateFlight	13.18	683,140	Executing-to-Done Omission	[B]
aSelectDisplayLocation from aInitiatelight	13.05	745,977	Done-to-Executing Intrusion	[C]
hScrollDisplay from aInitiatelight	10.87	409,451	Ready-to-Executing Intrusion	
aChooseDisplayedLocation from aInitiatelight	11.55	508,949	Done-to-Executing Intrusion	
aChoose1 from aInitiatelight	11.5	478,907	Ready-to-Executing Intrusion	
aChoose2 from aInitiatelight	10.28	378,551	Ready-to-Executing Intrusion	
aFlyToSelectedLocation	10.16	438,422	Done-to-Executing Intrusion	
aRecieveNewRadio	12.98	909,744	Ready-to-Executing Intrusion	
aEnterNewRadio	12.05	557,998	Ready-to-Executing Intrusion	
aInvestigate	10.87	482,260	Done-to-Executing Intrusion	
hFlyInvestigateAtLocation	11.14	348,919	Done-to-Executing Intrusion	
aSelectDisplayLocation from aIdentifyTarget	12.25	713,170	Done-to-Executing Intrusion	
hScrollDisplay from aIdentifyTarget	10.89	418,830	Ready-to-Executing Intrusion	
aChooseDisplayedLocation from aIdentifyTarget	11.62	443,296	Done-to-Executing Intrusion	
aChoose1 from aIdentifyTarget	11.35	466,720	Ready-to-Executing Intrusion	
aChoose2 from aIdentifyTarget	10.35	427,537	Ready-to-Executing Intrusion	
aIdentifyNew	11.39	599,971	Done-to-Executing Intrusion	
hGetLaserLocation	11.49	469,858	Ready-to-Executing Intrusion	
aSelectNextLocation	88.22	1,630,1277	Ready-to-Executing Intrusion	
aSelectDisplayLocation from aSelectNextLocation	14.22	964,988	Ready-to-Executing Intrusion	
hScrollDisplay from aSelectNextLocation	10.79	446,885	Ready-to-Executing Intrusion	
aChooseDisplayedLocation from aSelectNextLocation	11.49	496,445	Ready-to-Executing Intrusion	
aChoose1 from aSelectNextLocation	11.04	484,291	Ready-to-Executing Intrusion	
aChoose2 from aSelectNextLocation	10.4	485,383	Ready-to-Executing Intrusion	
aFire	6.44	12,642	Ready-to-Executing Intrusion	[D]
hFireWeapons	6.53	10,107	Ready-to-Executing Intrusion	
lDisplayLocation from aInitiatelight under aChoose1	9.9	1,896,915	Misremembrance	[E]
lDisplayLocation from aInitiatelight under aChoose2	9.54	1,895,321	Misremembrance	
lDisplayLocation from aIdentifyTarget under aChoose1	10.24	1,945,480	Misremembrance	
lDisplayLocation from aIdentifyTarget under aChoose2	9.21	1,949,899	Misremembrance	
lDisplayLocation from aSelectNextLocation under aChoose1	11.04	1,950,732	Misremembrance	
lDisplayLocation from aSelectNextLocation under aChoose2	10.63	1,957,855	Misremembrance	
hFlyToLocation	9.81	527,089	SetValue ValueSubstitution	[F]
hManuallyEnterLocation	11.51	1,047,300	SetValue Value Substitution	[G]
aIdentifyTarget	13.26	864,789	Executing-to-Done Omission	[H]
aCheckIfNew	13.97	944,480	Executing-to-Done Omission	
aEngageTarget	6.83	31,797	Ready-to-Executing Intrusion	[I]
aConfirmStatus	9.6	270,931	Ready-to-Done Omission	[J]
hGetFireClearance	8.91	198,275	Value Substitution	[K]

[A] No erroneous behavior occurred.
[B] When initiating flight, the pilot fails to fly to the new location and gets confirmation to fire on a target based on the location he or she should have flown to.
[C] The pilot replaces getting clearance to fire (hGetFireClearance) with an action from an erroneously executing activity or action.
[D] The pilot erroneously fires his or her weapons.
[E] The pilot spuriously and erroneously remembers the wrong location and uses this to get clearance to fire on a target.
[F] The pilot flies to the incorrect location and gets confirmation to fire based on the location he or she should have flown to.
[G] The pilot manually enters an incorrect value into the flight computer, flies to it, and gets fire clearance based on the values he or she should have entered.
[H] The pilot fails to acquire a laser location on a target and erroneously gets permission to fire on the target based on the location he or she was previously flying to.
[I] Erroneously beginning the engagement process and getting permission to fire based on the last remembered location.
[J] Skipping the confirmation/clearance activity and firing based on a previously received clearance.
[K] Getting fire clearance by erroneously saying the wrong location.

the same system problems that could only be discovered with either the phenotype- or genotype-based techniques [28]. The technique was also able to discover a real failure condition in a new case study (the Apache Helicopter application) as well as multiple previously unknown failure scenarios. This is an important contribution because it means that this approach will enable analysts to both find more unexpected problems than previously possible and, if no problems are discovered, have an unprecedented confidence in the reliability of their system.

Second, the scalability analyses show that the method's verification time increases linearly with the maximum number of considered erroneous human behaviors. This is a very positive result because it means that method scalability will be less of a restrictive issues when using our method. This is also an

important result because previous phenotype-based generation methods [22] scaled exponentially with the maximum number of erroneous human behavior. Thus the new method offers more complete erroneous behavior generation with better scalability. This will allow the reliability benefits of using our method to be available to many more systems.

Third, the ability to selectively apply erroneous behavior generation across the entire task model, at each activity or action, or for specific erroneous behaviors is also a contribution. Previous methods either focused on manually incorporating specific erroneous behavior [20, 21, 24, 24, 56, 57] or look at all possible erroneous behaviors [22, 23, 25–27, 30, 54, 55]. By giving analysts more options, our method can enable either of these analyses. It will also allow analysts to find more potential erroneous behaviors than allowed by just these two approaches. This is particularly well demonstrated in the Apache helicopter application, where the application of the generation to each activity and action between analyses enable the discovery of 39 different erroneous behaviors that could contribute to system failures. This capability should further allow the analyses to scale even more effectively to complex systems, where generation of all possible erroneous behaviors in a given verification could limit method applicability.

Finally, the analyses of the Apache helicopter application are themselves a contribution. Specifically, the method was able to discover the types of conditions that have actually resulted in fratricide (Note *E* from Table 3). This provide evidence that the method is valid. The method also discovered nine other plausible, but previously unknown, general types of erroneous human behaviors that could contribute to friendly fire. Future work should investigate if these erroneous behaviors could actually result in friendly fire in a real Apache helicopter. Identifying and correcting these errors could thus significantly reduce the potential for future fratricide incidents.

Our developments also suggest avenues of future research. We discuss these below.

5.1. Team Communication and Coordination

Previous EOFM research determined how to extend it to support verification of human-human communication and coordination [58, 65]. These developments were then developed to allow miscommunication to to be automatically accounted for in verification analyses both without [26, 55] and with other types of generated erroneous behaviors [25, 66]. The EOFM communication and coordination extensions are fully compatible with the semantics of EOFM used in the presented work. Further, the miscommunication generation method uses the same concepts as action value substitutions and action target substitutions (Table 2) in the new method. Thus, the new erroneous behavior generation method should be capable of generating erroneous behaviors and miscommunication in team tasks that require human-human collaboration and communication. This should be investigated in future work.

5.2. Cognitive Models of Erroneous Human Behavior

This work was concerned with task-based formal erroneous behavior generation techniques for accounting for erroneous human behavior. However, there have been a number of researchers who have explored how formal models based on cognitive architectures can be used to determine how erroneous behaviors can arise and cause system problems (see reviews in [13, 15]). Future work should investigate how our erroneous behavior generation method compares with these techniques and how they scale relative to each other.

5.3. Accounting for the Stochasticity of Erroneous Behaviors

By virtue of using traditional, symbolic model check, the method presented in this work considers all of the possible erroneous behaviors supported by the taxonomy equally. Thus, the method does not account for the effect different environmental or social factors could have for making some errors more probable than others. Results by Zheng et al. [67] have shown that it is possible to predict rates of erroneous human behavior based on such factors using a combination of probabilistic model checking [68] and human reliability analysis [69, 70]. Future work should investigate how these developments could be integrated with the task-based taxonomy of erroneous behavior [29] to enable analysts to account for the probabilities of different erroneous acts occurring in verification analyses.

5.4. Further Scalability Improvements

In previous developments, we showed how the scalability of the formal EOFM representation could be substantially improved by "flattening" it [60]: representing the execution state of each activity in terms of its descendant actions. However, this improvement was conceptually incompatible with the presented erroneous behavior generation method because it relies on each activity and action erroneously transitioning at each possible activity or action state. If it were possible to account for the erroneous behavior transitions (and the counting of them) in the "flattened" model, this would significantly improve method scalability. This should be the subject of future research.

5.5. Model Validity

Beyond scalability, the other major challenge for model-checking-based analyses is model validity. If the verified model does not accurately capture the behavior of the target system, the results of the analyses could be meaningless. This has the potential to be a significant limitation of the method presented in this paper because it relies on the analyst or modeler both formally modeling the environment and automation of the system and conducting an accurate task analysis. Both of these activities require an extensive knowledge of the target systems and significantly effortful observational analyses.

Fortunately, there are methods for addressing this. Specifically, model checking produces counterexamples when specification failures are discovered. This gives analysts a means of testing whether discovered issues are actually manifest in a system. Further, parallel work [71] has investigated how EOFM can be used with automated test case generation, a means of using formal models to produce collections of tests with coverage guarantees. Such test can be used to validate that systems and humans behave in ways captured by the model.

Additionally, by giving analysts a precise means of describing and reasoning about task analytic behavior, EOFM has the potential to take some of the art and guess work out of task analysis. Future should investigate how EOFM can be used to influence the task analysis process and potentially automate portions of it to help address validity issues that could arise from incomplete or inaccurate task analyses.

5.6. Additional Applications

In this paper, we only considered three applications: two medical devices and the Apache helicopter. These applications enabled us to both check that the method could find existing problems and, in the case of the Apache helicopter, discover new potential failure conditions with high face validity. Unfortunately, the lack of availability of an Apache helicopter prevented us from directly testing for the discovered failures with the actual system. However, there are many other potential applications in medicine, aviation, energy systems, cybersecurity, and defense where this limitation may not be present. Further, the considered cases were all existing systems, not systems being designed or implemented. The real power of our method would be realized when using our method early in the design process to anticipate and address problems. Future work should investigate other safety-critical applications where our approach can be applied to so that its benefits can be assessed throughout the systems engineering life cycle.

Acknowledgement

The work presented here was supported by grant W911NF-15-1-0474 "Young Investigator Program (8.5): Preventing Complex Failures of Human Interactive Systems with Erroneous Behavior Generation and Robust Human Task Behavior Patterns" by the Army Research Office / Army Research Lab.

References

[1] Hollnagel, E.. The phenotype of erroneous actions. International Journal of Man-Machine Studies 1993;39(1):1–32.

[2] Reason, J.. Human Error. New York: Cambridge University Press; 1990. ISBN 0521314194.

[3] Sheridan, T.B., Parasuraman, R.. Human-automation interaction. Reviews of Human Factors and Ergonomics 2005;1(1):89–129.

[4] Perrow, C.. Normal Accidents: Living with High-risk Technologies. Princeton: Princeton University Press; 1999.

[5] Kohn, L.T., Corrigan, J., Donaldson, M.S.. To Err is Human: Building a Safer Health System. Washington: National Academy Press; 2000.

[6] Kenny, D.J.. 26th Joseph T. Nall report: General aviation accidents in 2014. Tech. Rep.; AOPA Foundation; 2015.

[7] Kebabjian, R.. Accident statistics. planecrashinfo.com; 2016. URL: `http://www.planecrashinfo.com/cause.htm`; accessed 3/14/2016.

[8] Javaux, D.. Human error, safety, and systems development in aviation. Reliability Engineering & System Safety 2002;2(75):115–119.

[9] Manning, S.D., Rash, C.E., LeDuc, P.A., Noback, R.K., McKeon, J.. The role of human causal factors in US Army unmanned aerial vehicle accidents. Tech. Rep. 2004-11; USA Army Research Laboratory; 2004.

[10] NHTSA, . National motor vehicle crash causation survey: Report to congress. Tech. Rep. DOT HS 811 059; National Highway Traffic Safety Administration; Springfield; 2008.

[11] Office of Technology Assessment, . Who goes there: Friend or foe. Tech. Rep. OTA-ISC-537; Congress, US; Washington, DC; 1993.

[12] Le Bot, P.. Human reliability data, human error and accident models—illustration through the three mile island accident analysis. Reliability Engineering & system safety 2004;83(2):153–167.

[13] Bolton, M.L., Bass, E.J., Siminiceanu, R.I.. Using formal verification to evaluate human-automation interaction in safety critical systems, a review. IEEE Transactions on Systems, Man and Cybernetics: Systems 2013;43(3):488–503.

[14] Bolton, M.L.. Novel developments in formal methods for human factors engineering. In: Proceedings of the Human Factors and Ergonomics Society Annual Meeting. SAGE Publications Sage CA: Los Angeles, CA; 2017, p. 715–717.

[15] Weyers, B., Bowen, J., Dix, A., Palanque, P., editors. The Handbook of Formal Methods in Human-Computer Interaction. Berlin: Springer; 2017.

[16] Paternò, F., Santoro, C.. Integrating model checking and HCI tools to help designers verify user interface properties. In: Proceedings of the 7th International Workshop on the Design, Specification, and Verification of Interactive Systems. Berlin: Springer; 2001, p. 135–150.

[17] Aït-Ameur, Y., Baron, M.. Formal and experimental validation approaches in HCI systems design based on a shared event B model. International Journal on Software Tools for Technology Transfer 2006;8(6):547–563.

[18] Bolton, M.L., Bass, E.J.. Formally verifying human-automation interaction as part of a system model: Limitations and tradeoffs. Innovations in Systems and Software Engineering: A NASA Journal 2010;6(3):219–231.

[19] Bolton, M.L., Siminiceanu, R.I., Bass, E.J.. A systematic approach to model checking human-automation interaction using task-analytic models. IEEE Transactions on Systems, Man, and Cybernetics, Part A 2011;41(5):961–976.

[20] Bastide, R., Basnyat, S.. Error patterns: Systematic investigation of deviations in task models. In: Task Models and Diagrams for Users Interface Design. Berlin: Springer; 2007, p. 109–121.

[21] Fields, R.E.. Analysis of erroneous actions in the design of critical systems. Ph.D. thesis; University of York; York; 2001.

[22] Bolton, M.L., Bass, E.J., Siminiceanu, R.I.. Generating phenotypical erroneous human behavior to evaluate human–automation interaction using model checking. International Journal of Human-Computer Studies 2012;70(11):888–906.

[23] Bolton, M.L., Bass, E.J.. Generating erroneous human behavior from strategic knowledge in task models and evaluating its impact on system safety with model checking. IEEE Transactions on Systems, Man and Cybernetics: Systems 2013;43(6):1314–1327.

[24] Bolton, M.L., Bass, E.J.. Formal modeling of erroneous human behavior and its implications for model checking. In: Proceedings of the Sixth NASA Langley Formal Methods Workshop. Hampton: NASA Langley Research Center; 2008, p. 62–64.

[25] Pan, D., Bolton, M.L.. Properties for formally assessing the performance level of human-human collaborative procedures with miscommunications and erroneous human behavior. International Journal of Industrial Ergonomics 2018;63:75–88.

[26] Bolton, M.L.. Model checking human–human communication protocols using task models and miscommunication generation. Journal of Aerospace Information Systems 2015;12(7):476–489.

[27] Barbosa, A., Paiva, A.C., Campos, J.C.. Test case generation from mutated task models. In: Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems. ACM; 2011, p. 175–184.

[28] Bolton, M.L.. 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. thesis; University of Virginia; Charlottesville; 2010.

[29] Bolton, M.L.. A task-based taxonomy of erroneous human behavior. International Journal of Human-Computer Studies 2017;108:105–121.

[30] Bolton, M.L., Bass, E.J.. Evaluating human-automation interaction using task analytic behavior models, strategic knowledge-based erroneous human behavior generation, and model checking. In: Proceedings of the IEEE International Conference on Systems Man and Cybernetics. Piscataway: IEEE; 2011, p. 1788–1794.

[31] Wing, J.M.. A specifier's introduction to formal methods. Computer 1990;23(9):8, 10–22, 24.

[32] Shankar, N., Owre, S., Rushby, J.M., Stringer-Calvert, D.W.J.. PVS Prover Guide. Computer Science Laboratory, SRI International; Menlo Park, CA; 1999.

[33] Kaufmann, M., Moore, J.S., Manolios, P.. Computer-Aided Reasoning: An Approach. Norwell, MA, USA: Kluwer Academic Publishers; 2000.

[34] Bertot, Y., Castéran, P., Huet, G.i., Paulin-Mohring, C.. Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions. Texts in theoretical computer science; Berlin, New York: Springer; 2004.

[35] Clarke, E.M., Grumberg, O., Peled, D.A.. Model checking. Cambridge: MIT Press; 1999.

[36] Emerson, E.A.. Temporal and modal logic. In: van Leeuwen, J., Meyer, A.R., Nivat, M., Paterson, M., Perrin, D., editors. Handbook of Theoretical Computer Science; chap. 16. Cambridge: MIT Press; 1990, p. 995–1072.

[37] Mansouri-Samani, M., Pasareanu, C.S., Penix, J.J., Mehlitz, P.C., O'Malley, O., Visser, W.C., et al. Program model checking: A practitioner's guide. Tech. Rep.; Intelligent Systems Division, NASA Ames Research Center; Moffett Field; 2007.

[38] Thomas, M.. The role of formal methods in achieving dependable software. Reliability Engineering & System Safety 1994;43(2):129–134.

[39] Bolbot, V., Theotokatos, G., Bujorianu, M.L., Boulougouris, E., Vassalos, D.. Vulnerabilities and safety assurance methods in cyber-physical systems: A comprehensive review. Reliability Engineering & System Safety 2019;182:179–193.

[40] Sharvia, S., Papadopoulos, Y.. Integrating model checking with HiP-HOPS in model-based safety analysis. Reliability Engineering & System Safety 2015;135:64–80.

[41] Gribaudo, M., Horváth, A., Bobbio, A., Tronci, E., Ciancamerla, E., Minichino, M.. Fluid petri nets and hybrid model-checking: A comparative case study. Reliability Engineering & System Safety 2003;81(3):239–257.

[42] Wu, D., Zheng, W.. Formal model-based quantitative safety analysis using timed coloured petri nets. Reliability Engineering & System Safety 2018;176:62–79.

[43] Bozzano, M., Cimatti, A., Katoen, J.P., Katsaros, P., Mokos, K., Nguyen, V.Y., et al. Spacecraft early design validation using formal methods. Reliability Engineering & System Safety 2014;132:20–35.

[44] Rushby, J.. Using model checking to help discover mode confusions and other automation surprises. Reliability Engineering & System Safety 2002;75(2):167–177.

[45] Bredereke, J., Lankenau, A.. Safety-relevant mode confusions–modelling and reducing them. Reliability Engineering & System Safety 2005;88(3):229–245.

[46] Javaux, D.. A method for predicting errors when interacting with finite state systems. How implicit learning shapes the user's knowledge of a system. Reliability Engineering & System Safety 2002;75:147–165.

[47] Herbert, L., Hansen, Z.. Restructuring of workflows to minimise errors via stochastic model checking: An automated evolutionary approach. Reliability Engineering & System Safety 2016;145:351–365.

[48] Kirwan, B., Ainsworth, L.K.. A Guide to Task Analysis. London: Taylor and Francis; 1992.

[49] Schraagen, J.M., Chipman, S.F., Shalin, V.L.. Cognitive Task Analysis. Philadelphia: Lawrence Erlbaum Associates, Inc.; 2000.

[50] Moray, N., Sanderson, P.M., Vicente, K.J.. Cognitive task analysis of a complex work domain: A case study. Reliability Engineering & System Safety 1992;36(3):207–216.

[51] Basnyat, S., Palanque, P., Schupp, B., Wright, P.. Formal socio-technical barrier modelling for safety-critical interactive systems design. Safety Science 2007;45(5):545–565.

[52] Gunter, E.L., Yasmeen, A., Gunter, C.A., Nguyen, A.. Specifying and analyzing workflows for automated identification and data capture. In: Proceedings of the 42nd Hawaii International Conference on System Sciences. Los Alatimos: IEEE Computer Society. ISBN 978-0-7695-3450-3; 2009, p. 1–11.

[53] Palanque, P.A., Bastide, R., Senges, V.. 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. London: Chapman and Hall; 1996, p. 189–212.

[54] Martinie, C., Palanque, P., Fahssi, R., Blanquart, J.P., Fayollas, C., Seguin, C.. Task model-based systematic analysis of both system failures and human errors. IEEE Transactions on Human-Machine Systems 2016;46(2):243–254.

[55] Bolton, M.L., Bass, E.J.. Evaluating human-human communication protocols with miscommunication generation and model checking. In: Proceedings of the Fifth NASA Formal Methods Symposium. Moffett Field: NASA Ames Research Center. Moffett Field: NASA Ames Research Center; 2013, p. 48–62.

[56] Bolton, M.L., Bass, E.J., Siminiceanu, R.I.. Using formal methods to predict human error and system failures. In: Proceedings of the 2nd International Conference on Applied Human Factors and Ergonomics. Las Vegas: Applied Human Factors and Ergonomics International; 2008, p. CD–ROM.

[57] Paternò, F., Santoro, C.. Preventing user errors by systematic analysis of deviations from the system task model. International Journal of Human-Computer Studies 2002;56(2):225–245.

[58] Bolton, M.L., Bass, E.J.. Enhanced Operator Function Model (EOFM): A Task Analytic Modeling Formalism for Including Human Behavior in the Verification of Complex Systems; chap. 13. Cham: Springer International Publishing; 2017, p. 343–377.

[59] De Moura, L., Owre, S., Shankar, N.. The SAL language manual. Tech. Rep. CSL-01-01; Computer Science Laboratory, SRI International; Menlo Park; 2003.

[60] Bolton, M.L., Zheng, X., Molinaro, K., Houser, A., Li, M.. Improving the scalability of formal human–automation interaction verification analyses that use task-analytic models. Innovations in Systems and Software Engineering 2017;13(1):1–17.

[61] Fahssi, R., Martinie, C., Palanque, P.. Enhanced task modelling for systematic identification and explicit representation of human errors. In: Human-Computer Interaction – INTERACT 2015. Cham: Springer International Publishing; 2015, p. 192–212.

[62] Bolton, M.L., Jimenez, N., van Paassen, M.M., Trujillo, M.. Automatically generating specification properties from task models for the formal verification of human-automation interaction. IEEE Transactions on Human-Machine Systems 2014;44:561–575.

[63] Bolton, M.L., Bass, E.J.. Using task analytic models to visualize model checker counterexamples. In: Proceedings of the 2010 IEEE International Conference on Systems, Man, and Cybernetics. Piscataway: IEEE; 2010, p. 2069–2074.

[64] United States, . Operation Desert Storm - Apache Helicopter Fratricide Incident: Report to the Chairman, Subcommittee on Oversight and Investigations, Conmittee on Energy and Commerce, House of Representatives. Tech. Rep. GAO/OSI-93-4; United States General Accounting Office; 1993.

[65] Bass, E.J., Bolton, M.L., Feigh, K., Griffith, D., Gunter, E., Mansky, W., et al. Toward a multi-method approach to formalizing human-automation interaction and human-human communications. In: Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics. Piscataway: IEEE; 2011, p. 1817–1824.

[66] Pan, D., Bolton, M.L.. A formal method for evaluating the performance level of human-human collaborative procedures. In: International Conference on Cross-Cultural Design. Springer; 2015, p. 186–197.

[67] Zheng, X., Bolton, M.L., Daly, C., Feng, L.. A formal human reliability analysis of a community pharmacy dispensing procedure. In: Proceedings of the Human Factors and Ergonomics Society Annual Meeting. Los Angeles: Sage; 2017, p. 728–732.

[68] Kwiatkowska, M., Norman, G., Parker, D.. PRISM 4.0: Verification of probabilistic real-time systems. In: International conference on computer aided verification. Springer; 2011, p. 585–591.

[69] Hollnagel, E.. Cognitive Reliability and Error Analysis Method (CREAM). Amsterdam: Elsevier; 1998.

[70] Sun, Z., Li, Z., Gong, E., Xie, H.. Estimating human error probability using a modified CREAM. Reliability Engineering & System Safety 2012;100:28–32.

[71] Li, M.. Formal methods for user experience evaluation and testing. Ph.D. thesis; University at Buffalo, State University of New York; Buffalo; 2018.