Validating Human-device Interfaces with Model Checking and Temporal Logic Properties Automatically Generated from Task Analytic Models

Matthew L. Bolton
San José State University Research Foundation
NASA Ames Research Center
Moffett Field, CA 94035
matthew.l.bolton@nasa.gov

Keywords:
Formal methods, model checking, task analysis, temporal logic, validation, human-automation interaction

ABSTRACT: *When evaluating designs of human-device interfaces for safety critical systems, it is very important that they be valid: support the goal-directed tasks they were designed to facilitate. Model checking is a type of formal analysis that is used to mathematically prove whether or not a model of a system does or does not satisfy a set of specification properties, usually written in a temporal logic. In the analysis of human-automation interaction, model checkers have been used to formally verify that human-device interface models are valid with respect to goal-directed tasks encoded in temporal logic properties. All of the previous work in this area has required that analysts manually specify these properties. Given the semantics of temporal logic and the complexity of task analytic behavior models, this can be very difficult. This paper describes a method that allows temporal logic properties to be automatically generated from task analytic models created early in the system design process. This allows analysts to use model checkers to validate that modeled human-device interfaces will allow human operators to successfully perform the necessary tasks with the system. The use of the method is illustrated with a patient controlled analgesia pump programming example. The method is discussed and avenues for future work are described.*

1. Introduction

Human-device interfaces (HDIs) for safety critical systems must support the human work they were intended to enable, otherwise they may not allow the system to be operated safely. Because of this, HDIs should be validated in order to determine whether they allow human operators to perform the necessary tasks. While traditional human factors techniques provide means of accomplishing this, they are limited in that they are not exhaustive and thus there are conditions they might miss. Analysis techniques found in formal methods, particularly formal verification via model checking, can help address this problem.

1.1 Formal methods and model checking

Formal methods are a set of languages and techniques for the modeling, specification and verification of systems (Wing, 1990). Model checking is an automated approach used to verify that a formal model of a system (usually of computer software or hardware) satisfies a set of desired properties (a specification) (Clarke, Grumberg, & Peled, 1999). A formal model describes a system as a set of variables and transitions between variable values (states). Specification properties are typically represented in a temporal logic (see Emerson, 1990) where the variables that describe the formal system model are used to construct propositions. Verification is the process of proving that the system meets the properties in the specification. Model checking performs this process automatically by exhaustively searching a system's state space in order to determine if these criteria hold. If there is a violation, an execution trace is produced (a counterexample or witness). This depicts a model state (a valuation of the model's variables) corresponding to a specification violation along with a list of the incremental model states that led up to it.

Model checking has traditionally been used to find problems in computer hardware and software applications (Clarke et al., 1999). However, some researchers have used model checking to investigate issues related to human-automation interaction (HAI) (see Bolton, 2010 for a survey). Such techniques have an advantage over traditional HAI analysis methods in that they allow analysts to consider all of the possible conditions in a model in order to find any that may be problematic. Of particular relevance to this paper is the research concerned with model checking human-device interfaces (HDIs).

1.2 Model checking human-device interfaces

In using model checking to evaluate HDIs, analysts must first formally model their target HDI with any relevant underlying system or automation behavior. There are a num-

ber of different tools for doing this (see Bolton, 2010), but all follow in the tradition of Parnas (1969) in that they represent HDIs as finite state transition systems.

With a formal model of a HDI, analysts can formulate desirable properties using a temporal logic and check that the model of the HDI adheres to them using a model checker (Abowd, Wang, & Monk, 1995; Campos & Harrison, 1997, 2008; Paternò, 1997). Campos and Harrison (1997) identified four related categories of properties that could be expressed in temporal logic and thus formally verified for HDI models: (a) *reachability*: assertions about the ability of the interface to eventually reach a particular state; (b) *visibility*: assertions that visual feedback will eventually result from an action; and (c) *reliability*: assertions that describe properties that support safe HAI; and (d) *task-related*: assertions related to the ability of a human operator to achieve a particular goal. Analysts can use a model checker to verify that the system exhibits desired usability properties (reachability, visibility, or reliability), or they can validate that the system allows the human to accomplish goals derived from task analytic models (task related).

Expressing properties in temporal logic can be difficult. For this reason, a number of tools have been developed to assist analysts in developing, automatically generating, and/or automatically evaluating reachability, visibility, and reliability properties (Loer & Harrison, 2006; Campos & Harrison, 2009; Feary, 2007). Very simple task-related properties can be expressed with the aid of temporal logic patterns (Abowd et al., 1995; Campos & Harrison, 2008; Paternò, 1997). However this can easily become unmanageable as the complexity of the human task behavior being expressed increases.

1.3 Task analytic behavior models

Human factors engineers use task analytic methods to describe the normative human behaviors required to control a system (Kirwan & Ainsworth, 1992). The resulting task analytic models represent the mental and physical activities operators use to achieve the goals the system was designed to support. A common formulation structures tasks as a hierarchy, where goal-directed activities decompose into other activities and potentially (at the lowest level) atomic actions. In these models, strategic knowledge (condition logic) controls when activities can execute and specifies what must be true when the activity completes. Modifiers on decompositions and/or between activities or actions control how many can execute and what the temporal relationship is between them.

Such models can be used at many different stages in the design and/or analysis of human-automation interactive systems. At latter stages of design or analysis, task models may be very detailed, describing the specific sequences of actions human operators can use to normatively achieve goals with the system. However, at earlier stages of design/analysis, task analytic models may be much more abstract and only describe the basic constraints on high-level, goal-directed activities without representing atomic actions. It is these higher-level task models that are relevant to the formal verification of task-related properties since they represent the hierarchy of goals the operator is attempting to achieve and the basic temporal constraints on their completion.

1.4 Objectives

Even at early stages of design, task analytic behavior models may contain multiple levels of decomposition and a variety of temporal and cardinal restrictions between activities within a task model hierarchy. Therefore, it may be very difficult for human operators to manually translate such task models into task-related temporal logic properties for use in HDI design validation. This paper introduces a process for automatically generating task-related temporal logic properties from task analytic behavior models. A model checker can then be used to automatically check these properties against formal HDI models in order to prove that a given HDI design is valid: that it supports the behavior captured in the task analytic models. The paper describes the infrastructure that was developed to implement this. It then illustrates how this process can be used to evaluate a HDI using a pain medication pump programming application. The results of this work are then discussed and directions for future work are explored.

2. Methods

The process shown in Figure 2.1 was developed to allow analysts to automatically create task-related temporal logic specifications from task analytic behavior models and validate HDI designs using formal verification with model checking.

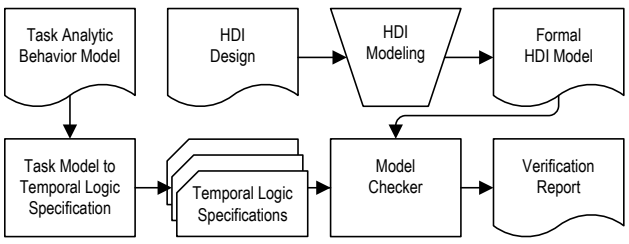

Figure 2.1. Process for automatically generating task-related temporal logic specifications from task analytic behavior models and using them to validate HDIs using formal verification with model checking.

An analysts starts with a task analytic model (generated as part of an early task analysis) and a target HDI design. The analyst uses the HDI design to create a formal model. The analyst runs the task analytic behavior model through an automatic process which generates temporal logic properties. The analyst can then use a model checker to automatically, formally, verify that the HDI model is valid: that it will allow for the fulfillment of the goals the task models are designed to support. This process produces a verification report which, if the task model goals are supported, will illustrate a paths through the model showing how the goals are achieved (a witness).

The remainder of this section discusses how a version of this process was implemented.

2.1 Enhanced operator function model

Human task behavior is modeled using the EOFM (Enhanced Operator Function Model) (Bolton, Siminiceanu, & Bass, n.d.). EOFM is well suited to this work because it is hierarchical, represents strategic knowledge, supports a number of different cardinal and temporal relationships between activities, has a formal semantics, and has a proven track record for use in formal verification (see Bolton & Bass, 2009, 2010a; Bolton et al., n.d.; Bolton & Bass, 2010b; Bolton, 2010).

EOFM extends the Operator Function Model (OFM) (Mitchell & Miller, 1986). EOFMs are hierarchical and heterarchical representations of goal-driven activities that decompose into lower level activities, and, finally, atomic actions. EOFMs express task knowledge by explicitly specifying the conditions under which human operator activities can execute (preconditions) and what must be true when they finish (completion conditions). Any activity can decompose into one or more activities or actions (sub-acts). A decomposition operator specifies the temporal relationships between and the cardinality of the decomposed sub-acts (when they can execute relative to each other and how many can execute). EOFM supports all of the following decomposition operators: (a) *optor* (zero or more of the sub-acts must execute in any order); (b) *or* (one or more of the sub-acts must execute in any order); (c) *and* (all of the sub-acts must execute in any order); (d) *ord* (all sub-acts must execute in the order they appear); and (e) *xor* (exactly one sub-act must execute).

EOFMs can be represented visually as discrete, tree-like graphs (see Figure 3.1) where activities are represented as 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 sub-acts. Conditions on activities are represented as shapes or arrows (annotated with the condition logic) connected to the activity that they constrain. A precondition is a yellow, downward-pointing triangle; and a completion condition is a magenta, upward-pointing triangle.

EOFM is primarily used to express the specific details of how a human operator performs normative, goal directed behavior (Bolton et al., n.d.). As such, it has features which are more expressive than are needed for the high-level models required for this work. This includes the requirement that every activity ultimately decompose into atomic actions; support for parallel and sequential modalities for each of the decomposition operators described above; and optional repeat conditions on each activity. These features are not utilized in the work presented here.

2.2 Symbolic analysis laboratory

Formal modeling of HDIs was performed using the notation of the Symbolic Analysis Laboratory (SAL) (De Moura, Owre, & Shankar, 2003). Formal verification was performed using SAL's symbolic model checker SAL-SMC (De Moura et al., 2003; Shankar, 2000).

2.3 Linear temporal logic

There are a number of different temporal logics used to specify properties for model checking analyses (Clarke et al., 1999; Emerson, 1990). The temporal logic supported by SAL-SMC is Linear Temporal Logic (LTL). Thus, LTL was used in this work.

LTL uses propositional variables, basic logic and Boolean operators ($\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow, =, \neq, <, >$, etc.), and temporal operators (Table 2.1) to assert properties about all paths through a model.

Table 2.1. Linear Temporal Logic Operators

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

Note. ϕ and ψ are two propositions about either a model state or path (a temporally ordered sequence of states) that can evaluate to true or false.

2.4 Generating LTL specifications from EOFM

Because LTL only allows specifications to reason about properties in all paths through a model, the generated property must assert that the goals encompassed by the task

model (t) will never happen (LTL pattern $\mathbf{G}\neg(t)$) in order to generate a counterexample/witness if the HDI model supports the task. If the task is not supported, the model checker will indicate that the property is true.

Every task structure in an EOFM instantiation is composed entirely of activities. Each activity in a task structure has at most three propositions that need to occur ordinally in the HDI: the precondition (x) should be true, then the conditions associated with the execution of the activities children (sub-activities; y) should be true, and then the completion condition (z) should be true. These temporally ordered relationships can be represented in LTL using temporal logic patterns. For example, $x \wedge \mathbf{F}(y)$ asserts that y eventually occurs after x. Similarly, $y \wedge \mathbf{F}(z)$ asserts that z eventually occurs after y. These can be combined in order to assert that z eventually occurs after y which eventually occurs after x by imbedding the second expression in the first: $x \wedge \mathbf{F}(y \wedge \mathbf{F}(z))$. In this way, LTL can be used to express sequences of ordinal conditions.

Any given activity can have sub-activities (which themselves have sub-activities) all of which have their temporal relationships modified by a decomposition operator which either specifies a specific order (*ord*) or directly corresponds to a relationship expressible by a Boolean operator (*optor*, *or*, *and*, *xor*). Thus, the generated temporal logic property must enumerate all of the potential ordinal sequence of conditions associated with activity decomposition based on the decomposition operators. To illustrate this, assume that in order for y to be true, both y_1 and y_2 must have been true but in no particular order (an *and* relationship). In this situation the LTL expression from above becomes $x \wedge \mathbf{F}((y_1 \wedge \mathbf{F}(z)) \wedge (y_2 \wedge \mathbf{F}(z)))$.

This information can be used to develop a mathematical function to generate LTL properties from each root parent activity in an EOFM instance's task structures.

Let an activity be defined as a tuple $\langle \alpha, \omega, \delta, \Sigma \rangle$. α and ω are Boolean expressions representing the activity's precondition and completion condition respectively. δ represent the activities decomposition operator such that $\delta \in \{optor, or, and, xor, ord\}$ which are as defined above. Σ is an ordered set of the activity's children (sub-activities) such that for an integer $m \geq 1$, $\Sigma = (\sigma_1, \sigma_2, ..., \sigma_m)$. For a given activity a, let a_α, a_ω, a_δ, and a_Σ represent the activity's precondition, completion condition, decomposition operator, and ordered set of children respectively. Further, for an integer i, $1 \leq i \geq m$, let a_{σ_i} represent σ_i from a_Σ.

Thus for a given task structure with root parent a and LTL expression ϕ, the LTL specification property can be generated by

$$\mathrm{f}(a) = \mathbf{G}\neg(\mathrm{g}(a, true))$$

where

$$\mathrm{g}(a,\phi) = a_\alpha \wedge \mathbf{F}\left(\begin{cases} a_\omega \wedge \mathbf{F}(\phi) & \text{if } a_\delta = optor \\ \bigvee_{i=1}^{m} (\mathrm{g}(a_{\sigma_i}, a_\omega \wedge \mathbf{F}(\phi))) & \text{if } a_\delta = or \\ \bigwedge_{i=1}^{m} (\mathrm{g}(a_{\sigma_i}, a_\omega \wedge \mathbf{F}(\phi))) & \text{if } a_\delta = and \\ \bigoplus_{i=1}^{m} (\mathrm{g}(a_{\sigma_i}, a_\omega \wedge \mathbf{F}(\phi))) & \text{if } a_\delta = xor \\ \mathrm{h}(a_{\sigma_1}, a_\omega \wedge \mathbf{F}(\phi)) & \text{if } a_\delta = ord \end{cases}\right)$$

and

$$\mathrm{h}(a_{\sigma_i}, \phi) = \begin{cases} \mathrm{g}(a_{\sigma_i}, \mathrm{g}(a_{\sigma_{i+1}}, \phi)) & \text{if } i < m \\ \mathrm{g}(a_{\sigma_i}, \phi) & i = m \end{cases}$$

Note that in $\mathrm{g}(a,\phi)$, $\bigoplus$ is a one-hot detector – a special type of exclusive or operator that is true only if exactly one of the *m* expressions is true.

This formulation was implemented as a java program which would parse a file containing an EOFM instantiation and generate (print out) an LTL specification for each task structure it contained.

3. Application

To illustrate how this method (Figure 2.1) can be used to validate a HDI, an example is presented of a HDI for programming prescriptions into a Patient Controlled Analgesia (PCA) pump: a medical device that allows patients to control the delivery of pain medication based on a prescription programmed into it by a human operator.

3.1 Task analytic behavior modeling

In this example, it is assumed that a task analysis was performed early in the design process before an actual system has been developed. This revealed that the HDI for programming the PCA pump needed to allow the human operator to program in three types of prescriptions: (a) where the operator must specify only the parameters for patient controlled dosages (*PrescribedType = PCA*), (b) where the operator must specify parameters for patient controlled dosages and a continuous basal rate (*PrescribedType = BasalPCA*), and (c) where the operator must specify only the parameters for a continuous rate (*PrescribedType = Continuous*).

Further, the analysis found that in all cases the operator should enter the fluid volume (*PrescribedFluidVolume*) of the medication reserve being used in the administration of treatment as the first parameter. It also revealed that a bolus dose (*PrescribedBolus*) should be the last parameter entered in all cases. All of the other parameters could be programmed into the pump in any order in between these two. However, the parameters differ between prescription types. A *PCA* prescription has a one hour limit (*Prescribed1HourLimit*) on the volume of administered medication, a medication dosage (*PrescribedPCADose*), and a minimum delay between dosages (*PrescribedDelay*). A *BasalPCA* prescription has all of the param-

eters of the former but also requires an additional basal rate (*PrescribedBasalRate*). A *Continuous* prescription requires only a one hour limit and a continuous medication delivery rate (*PrescribedContinuous*).

These restrictions were instantiated in an EOFM as three separate task models (Figure 3.1).

3.2 HDI Design and Formal Modeling

At a later stage in the development process, a HDI design is created for the PCA pump (Figure 3.2). This design contains a dynamic LCD display and eight buttons. The human operator uses the HDI to program a prescription. This involves the specification of the type of prescription being entered and all prescription parameters. The "Start" and "Stop" buttons start and stop the delivery of medication (stop must be pressed twice) at certain times during programming. The "On-Off" button is used to turn the device on (when pressed once) and off (when pressed twice). The LCD display is used to select prescription options (such as prescription type) and specify prescription values. When the operator must choose between two or more options: the interface message indicates what is being chosen and the initial or default option is displayed. The up button is used to scroll through the available options.

When a numerical value is required (such as the volume of a PCA dose), the value's name is listed in the interface message and the value is presented with the cursor under one of its digits. The programmer can move the position of the cursor by pressing the left and right buttons. He can press the up button to scroll through the different digit values available at that cursor position. The "Clear" button sets the displayed value to zero. The enter button is used to confirm values and treatment options.

This system was formally modeled in the language of SAL. The model is represented here as state transition diagrams (Figure 3.3). The HDI model was composed of variables representing the state of the interface as indicated by the LCD (*InterfaceState*; Figure 3.3(a)), the type of prescription selected (*Type*; Figure 3.3(b)), and the different prescription values in the pump (*FluidVolume*, *PCADose*, *Delay*, *1HourLimit*, *Bolus*, *BasalRate*, and *ContinuousRate*; all of which adhere to the behavior in Figure 3.3(c)). Note that in order to ensure that the HDI model is computationally tractable, all of the prescription values were modeled abstractly. They start assuming a value of *Incorrect*. When the human operator attempts to change the value by pressing the up button, the value can stay *Incorrect* or become *Correct*. Whenever a value is *Correct*, it will become *Incorrect* if the human operator presses up. The value always becomes *Incorrect* when the "Clear" button is pressed.

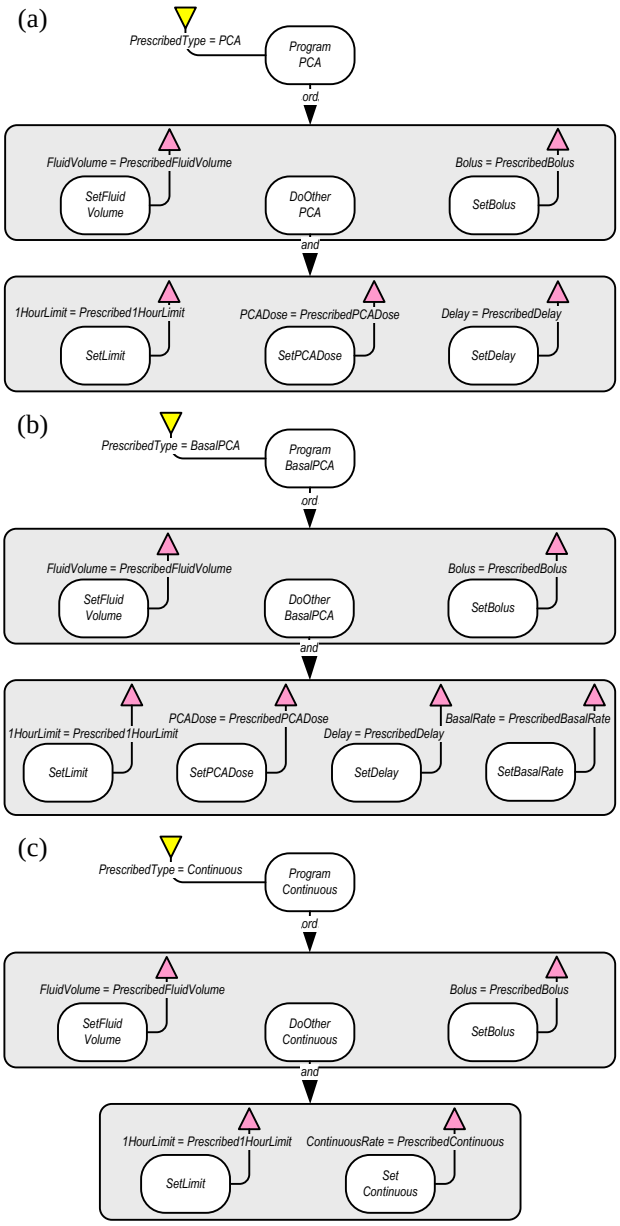

Figure 3.1. EOFM task analytic behavior models for programming the three different prescriptions into a PCA pump: (a) *PCA*, (b) *BasalPCA*, and (c) *Continuous*.

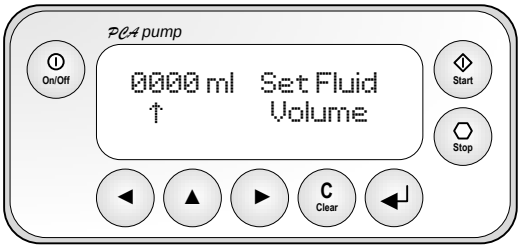

Figure 3.2. HDI design for a PCA pump.

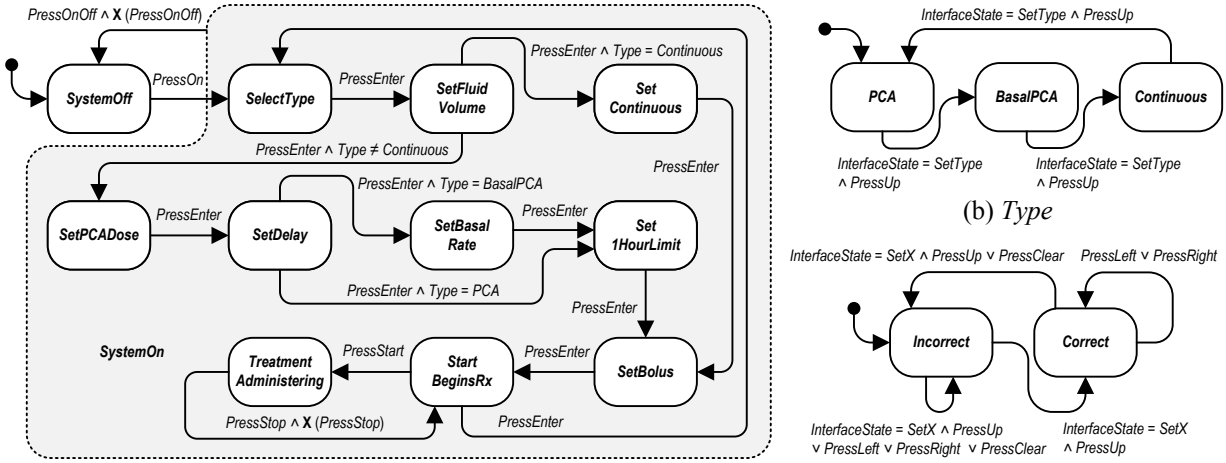

Figure 3.3. State transition model representation of the formal model of the PCA pump HDI. Rounded rectangles and boxes with dotted lines represent states. Arrows indicate guarded transitions between states. Transitions are labeled with transition logic. Note that variables in transition logic with the *Press* prefix indicate that a human has pressed a button on the HDI. (a) The state of LCD display. (b) The state of the type of prescription selected in the pump. (c) The behavior used to model the state of prescription value *X*, where *X* can be any values associated with a prescription.

In addition to the system behavior, the formal model also describes all of the possible types of prescriptions. Thus there is a *PrescribedType* which can assume all of the following values: *PCA*, *BasalPCA*, and *Continuous*. If *PrescribedType = PCA*, *PrescribedFluidVolume*, *PrescribedPCADose*, *PrescribedDelay*, *Prescribed1HourLimit*, and *PrescribedBolus* are all *Correct*. If *PrescribedType = BasalPCA*, *PrescribedFluidVolume*, *PrescribedPCADose*, *PrescribedDelay*, *Prescribed1HourLimit*, *PrescribedBolus*, and *BasalRate* are all *Correct*. If *PrescribedType = Continuous*, *PrescribedFluidVolume*, *Prescribed1HourLimit*, *PrescribedBolus*, and *ContinuousRate* are all *Correct*.

Finally, the system model is completed with a model that can issue any possible human action (button presses) at any given time: *PressOnOff*, *PressLeft*, *PressUp*, *PressRight*, *PressClear*, *PressEnter*, *PressStart*, and *PressStop*.

3.3 LTL property generation and formal verification

Using the process described above, the instantiated EOFM tasks (Figure 3.1) were converted into linear temporal logic properties. This resulted in the propositions in Figure 3.4.

When these were checked against the formal system model of the HDI design, properties 3.4(a) and (b) produced the expected counterexamples/witnesses illustrating how the associated tasks (3.1(a) and (b)) could successfully be performed. However, the property in 3.4(c) did not return a counterexample, indicating that the task in 3.1(c) could not be completed as specified.

(a) $$\mathbf{G}\neg\left(\begin{array}{l}PrescribedType = PCA\\ \wedge\mathbf{F}\left(\begin{array}{l}FluidVolume = PrescribedFluidVolume\\ \wedge\mathbf{F}\left(\begin{array}{l}\left(\begin{array}{l}1HourLimit = Prescribed1HourLimit\\ \wedge\mathbf{F}\left(Bolus = PrescribedBolus\right)\end{array}\right)\\ \wedge\left(\begin{array}{l}PCADose = PrescribedPCADose\\ \wedge\mathbf{F}\left(Bolus = PrescribedBolus\right)\end{array}\right)\\ \wedge\left(\begin{array}{l}Delay = PrescribedDelay\\ \wedge\mathbf{F}\left(Bolus = PrescribedBolus\right)\end{array}\right)\end{array}\right)\end{array}\right)\end{array}\right)$$

(b) $$\mathbf{G}\neg\left(\begin{array}{l}PrescribedType = PCA\\ \wedge\mathbf{F}\left(\begin{array}{l}FluidVolume = PrescribedFluidVolume\\ \wedge\mathbf{F}\left(\begin{array}{l}\left(\begin{array}{l}1HourLimit = Prescribed1HourLimit\\ \wedge\mathbf{F}\left(Bolus = PrescribedBolus\right)\end{array}\right)\\ \wedge\left(\begin{array}{l}PCADose = PrescribedPCADose\\ \wedge\mathbf{F}\left(Bolus = PrescribedBolus\right)\end{array}\right)\\ \wedge\left(\begin{array}{l}Delay = PrescribedDelay\\ \wedge\mathbf{F}\left(Bolus = PrescribedBolus\right)\end{array}\right)\\ \wedge\left(\begin{array}{l}BasalRate = PrescribedBasalRate\\ \wedge\mathbf{F}\left(Bolus = PrescribedBolus\right)\end{array}\right)\end{array}\right)\end{array}\right)\end{array}\right)$$

(c) $$\mathbf{G}\neg\left(\begin{array}{l}PrescribedType = PCA\\ \wedge\mathbf{F}\left(\begin{array}{l}FluidVolume = PrescribedFluidVolume\\ \wedge\mathbf{F}\left(\begin{array}{l}\left(\begin{array}{l}1HourLimit = Prescribed1HourLimit\\ \wedge\mathbf{F}\left(Bolus = PrescribedBolus\right)\end{array}\right)\\ \wedge\left(\begin{array}{l}ContinuousRate = PrescribedContinuous\\ \wedge\mathbf{F}\left(Bolus = PrescribedBolus\right)\end{array}\right)\end{array}\right)\end{array}\right)\end{array}\right)$$

Figure 3.4. LTL Properties generated from the instantiated EOFM model tasks from Figure 3.1.

Thus, the HDI design is valid for the tasks described in 3.1(a) and (b). However, it must be redesigned or modified in order to be valid for (c). An examination of Figure 3.3(a) shows that when the human operator is attempting to program in a *Continuous* prescription type, that the interface does not allow him to set a one hour limit – something re-

quired by 3.1(c). Thus, the HDI can be redesigned so that after the human operator sets the continuous rate, he sets the one hour limit, and then enters the bolus. Making this change in the formal model results in all of the properties in 3.4 producing the expected counterexamples/witnesses.

4. Discussion

Given how important it is for HDIs of safety critical systems to enable the tasks the system was designed to support, the work presented here represents a significant contribution in that it helps automate the process of validating HDI designs. This work has shown that it is possible to generate temporal logic properties from task analytic models created early in the system design process and use them to validated that formal models of HDI designs support the associated tasks using a model checker. An implementation of this process was presented which uses SAL and EOFM. The process was illustrated with a PCA pump programming HDI example which showed how shortcomings in a design could be discovered using the presented method.

This process has a distinct advantage over previous work where properties had to be created manually from scratch or through the application of temporal logic patterns (Abowd et al., 1995; Campos & Harrison, 2008; Paternò, 1997). Despite this, there are several limitations of the presented method which future work should address.

4.1 Human-device interface modeling

A distinct disadvantage of this method is that it requires the analyst to create a formal model of an HDI design, something that is not standard practice and is likely unfamiliar to most HDI designers. This problem could be rectified through the use of HDI design tools such as ADEPT (Feary, 2007) and Dwyer et al.'s formalization of HDIs defined in Visual Basic and Java Swing (Dwyer, Carr, & Hines, 1997; Dwyer, Robby, Tkachuk, & Visser, 2004). Such tools are capable of creating formal models of HDIs without the human operator needing to be an expert on formal modeling. Future work should investigate how the method presented here might be made to work with these types of tools. Future work should also investigate how formal models compatible with this method might be generated from other HDI design tools currently used by human factors engineers.

4.2 Task model condition logic

Another limitation of this work comes from EOFM's requirement that strategic knowledge in preconditions and completion conditions must be specified using variables in Boolean expressions (Bolton et al., n.d.). While this allows for precise definition of the conditions which indicate when tasks can be performed and what goals they achieve, it is unlikely that these variables will be defined the same in the early design process (when the task analytic models are created) as they will be in HDI design models. Thus, in actual practice, analysts will need to redefine task model preconditions and completion conditions so that they accurately reflect the variable names present in the HDI models. Such a process may be time consuming and prone to analyst error. Future work should investigate how design tools can be used to integrate task analytic model and HDI design development in order to avoid such problems.

4.3 Lack of diagnosis

The analysis presented here will indicate that a task is not supported by an HDI if a counterexample/witness is not returned. However, the method does nothing to tell the analyst why the task was not supported. While the problem in the presented HDI was easily diagnosed by examining the presented figures, this may not be the case with other applications. By allowing the analyst to generate discrete, checkable temporal logic properties for each subtask (activity) in a model, the method would give analysts a means of potentially diagnosing why a task was not supported. Future work should investigate the feasibility of this.

4.4 Multiple means of satisfying goals and sub-goals

Finally, the process presented here is only capable of producing a single path (counterexample/witness) through a HDI model, illustrating one way a given task can be accomplished. Thus, if a task behavior model encompasses multiple options for achieving the task's associated goals, the method presented here will not provide insights on the feasibility of any but one of them. This is a distinct limitation because analysts may wish to understand all of the different ways a HDI supports a task rather than just one. For example, an interface that facilitates multiple means of performing a task based on different environmental constraints may be more desirable than one that only support the task under very limited environmental conditions. Future work will investigate how the method presented here can be extended to provide insights into the different ways that an HDI can support the modeled tasks.

5. Acknowledgement

The author would like to thank Michael Feary for his feedback during the preparation of this manuscript. This work was supported in part by NASA cooperative agreement NNX08AX12A, the NASA Aviation Safety program, and the FAA-NASA Nextgen flight research project. The content is solely the responsibility of the author and does not necessarily represent the official views of NASA or the FAA.

6. References

Abowd, G. D., Wang, H., & Monk, A. F. (1995). A formal technique for automated dialogue development. In *Proceedings of the 1st Conference on Designing Interactive Systems* (pp. 219–226). New York: ACM.

Bolton, M. L. (2010). *Using task analytic behavior modeling, erroneous human behavior generation, and formal methods to evaluate the role of human-automation interaction in system failure*. Unpublished doctoral dissertation, University of Virginia, Charlottesville.

Bolton, M. L., & Bass, E. J. (2009). A method for the formal verification of human interactive systems. In *Proceedings of the 53rd Annual Meeting of the Human Factors and Ergonomics Society* (pp. 764–768). Santa Monica: HFES.

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

Bolton, M. L., & Bass, E. J. (2010b). Using task analytic models to visualize model checker counterexamples. In *Proceedings of the 2010 IEEE International Conference on Systems, Man, and Cybernetics* (pp. 2069–2074). Piscataway: IEEE.

Bolton, M. L., Siminiceanu, R. I., & Bass, E. J. (n.d.). A systematic approach to model checking human-automation interaction using task-analytic models. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*. (In Press)

Campos, J. C., & Harrison, M. (1997). Formally verifying interactive systems: A review. In *Proceedings of the Fouth International Eurographics Workshop on Design, Specification, and Verification of Interactive Systems* (pp. 109–124). Berlin: Springer.

Campos, J. C., & Harrison, M. D. (2008). Systematic analysis of control panel interfaces using formal tools. In *Proceedings of the 15th International Workshop on the Design, Verification and Specification of Interactive Systems* (pp. 72–85). Berlin: Springer.

Campos, J. C., & Harrison, M. D. (2009). Interaction engineering using the ivy tool. In *Proceedings of the 1st ACM SIGCHI Symposium on Engineering Interactive Computing Systems* (pp. 35–44). New York: ACM.

Clarke, E. M., Grumberg, O., & Peled, D. A. (1999). *Model checking*. Cambridge: MIT Press.

De Moura, L., Owre, S., & Shankar, N. (2003). *The SAL language manual* (Tech. Rep. No. CSL-01-01). Menlo Park: Computer Science Laboratory, SRI International.

Dwyer, M. B., Carr, V., & Hines, L. (1997). Model checking graphical user interfaces using abstractions. In *Proceedings of the Sixth European Software Engineering Conference* (pp. 244–261). New York: Springer.

Dwyer, M. B., Robby, Tkachuk, O., & Visser, W. (2004). Analyzing interaction orderings with model checking. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering* (pp. 154–163). Los Alamitos: IEEE Computer Society.

Emerson, E. A. (1990). Temporal and modal logic. In J. van Leeuwen, A. R. Meyer, M. Nivat, M. Paterson, & D. Perrin (Eds.), *Handbook of theoretical computer science* (pp. 995–1072). Cambridge: MIT Press.

Feary, M. (2007). Automatic detection of interaction vulnerabilities in an executable specification. In *Proceedings of the 7th International Conference on Engineering Psychology and Cognitive Ergonomics* (pp. 487–496). Berlin: Springer.

Kirwan, B., & Ainsworth, L. K. (1992). *A guide to task analysis*. London: Taylor and Francis.

Loer, K., & Harrison, M. D. (2006). An integrated framework for the analysis of dependable interactive systems (IFADIS): Its tool support and evaluation. *Automated Software Engineering*, *13*(4), 469–496.

Mitchell, C. M., & Miller, R. A. (1986). A discrete control model of operator function: A methodology for information display design. *IEEE Transactions on Systems Man Cybernetics Part A: Systems and Humans*, *16*(3), 343–357.

Parnas, D. L. (1969). On the use of transition diagrams in the design of a user interface for an interactive computer system. In *Proceedings of the 24th National ACM Conference* (pp. 379–385). New York: ACM.

Paternò, F. (1997). Formal reasoning about dialogue properties with automatic support. *Interacting with Computers*, *9*(2), 173–196.

Shankar, N. (2000). Symbolic analysis of transition systems. In *Proceedings of the international workshop on abstract state machines, theory and applications* (pp. 287–302). London: Springer.

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

Author Biographies

MATTHEW BOLTON received the B.S. in computer science in 2003, the M.S. in systems engineering in 2006, and the Ph.D. in systems engineering in 2010 from the University of Virginia, Charlotteville, USA. He is a Senior Research Associate for the San José State University Research Foundation at the NASA Ames Research Center. His primary research focuss is on the development of tools and techniques for using formal methods in the modeling, validation, verification, and design of safety-critical, human-automation interactive systems.