Comput Math Organ Theory
DOI 10.1007/s10588-012-9138-6

SI: BRIMS 2011

Automatic validation and failure diagnosis of human-device interfaces using task analytic models and model checking

Matthew L. Bolton

Abstract When evaluating designs of human-device interfaces for safety critical systems, it is very important that they support the goal-directed tasks they were designed to facilitate. This paper describes a novel method that generates task-related temporal logic properties from task analytic models created early in the system design process. This allows analysts to use model checking (a means of performing exhaustive mathematical proofs) to automatically validate that formal models of human-device interfaces will let human operators successfully perform the necessary tasks with the system. This paper also presents an algorithm that uses the method to diagnose why a particular task is not valid for a given design. The application of both the method and algorithm are illustrated with a patient-controlled analgesia pump programming example. The method and algorithm are discussed and avenues for future work are described.

Keywords Formal methods · Model checking · Task analysis · Temporal logic · Validation · Human-automation interaction

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 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

M.L. Bolton (✉)
Department of Mechanical and Industrial Engineering, University of Illinois at Chicago, Chicago, IL 60607, USA
e-mail: mbolton@uic.edu

Published online: 21 August 2012

in formal methods, particularly formal verification via model checking, can help address this problem. This paper present a new method that utilizes formal methods, task analytic models, and model checking to allow analysts to automatically validate that HDIs allow human operators to fulfill the task goals they were intended to support. The paper also describes an algorithm capable of enabling the analyst to diagnose validation failures discovered during the methods initial application.

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 et al. 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 to determine if these criteria hold. If no violation is discovered, the model checker reports that the model verifies to true. If there is a violation, an execution trace is produced (a counterexample). 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. Conversely, a model checker can also be used to perform an existence proof, where the model's entire state space is searched to find a condition where a specification holds. If such a path is discovered, the model checker returns an execution trace (termed a witness) which illustrates a path through the model under which the specification property holds.

Model checking has traditionally been used to evaluate computer hardware and software (Clarke et al. 1999). However, some researchers have used model checking to investigate issues related to human-automation interaction (HAI). Model checkers have been used to investigate properties of human-device interfaces (HDIs) (Abowd et al. 1995; Campos and Harrison 1997, 2008; Paternò 1997), look for potential mode confusion and/or automation surprise (Joshi et al. 2003; Rushby 2002; Bredereke and Lankenau 2005), and evaluate system properties and human performance with modeled human task analytic behavior (Aït-Ameur and Baron 2006; Basnyat et al. 2007; Bolton et al. 2011; Fields 2001; Paternò and Santoro 2001) and/or human cognitive behavior (Cerone et al. 2005; Rukšenas et al. 2009; Basuki et al. 2009) (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 and thus find any that may be problematic. Of particular relevance to this paper is the research concerned with model checking 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 number 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 et al. 1995; Campos and Harrison 1997; Campos 2008; Paternò 1997). Campos and Harrison (1997) identified four categories of HDI properties that could be expressed in temporal logic:

– reachability: assertions about the ability of the interface to eventually reach a particular state;
– visibility: assertions that visual feedback will eventually result from an action;
– reliability: assertions that describe properties that support safe HAI; and
– 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 2006; Campos 2009; Feary 2007). Very simple task-related properties can be expressed with the aid of temporal logic patterns (Abowd et al. 1995; Campos 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

Analysts use task analytic behavior models such as OFM (Operator Function Model) (Mitchell and Miller 1986), EOFM (Enhanced Operator Function Model) (Bolton et al. 2011), Concur Task Trees (Paternò et al. 1997), User Action Notation (Hartson et al. 1990), and GOMS (Goals, Operators, Methods and Selection rules) (Kieras 2003) to describe the normative human behaviors required to control a system (Kirwan and Ainsworth 1992). These models represent the mental and physical activities operators use to achieve the goals the system was designed to support. The most 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 an activity completes. Modifiers on decompositions and/or between activities or actions control how many activities 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 the systems engineering process, 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 an analyst 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 to prove that a given HDI design is valid: that it supports the behavior captured in the task analytic models. Further, an algorithm is presented which shows how the method can be iteratively reapplied to diagnose problems when the HDI is shown to not be valid for a task. The paper describes the infrastructure that was developed to implement the method and diagnostic algorithm. It then illustrates how both the method and algorithm can be used to evaluate a HDI using a patient-controlled analgesia pump programming application. The results of this work are then discussed and directions for future work are explored.

2 Methods

The process shown in Fig. 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.

An analyst starts with a task analytic model (generated as part of an early task analysis) and a model of a HDI design. The analyst runs the task analytic behavior model through an automatic process which generates temporal logic properties. The

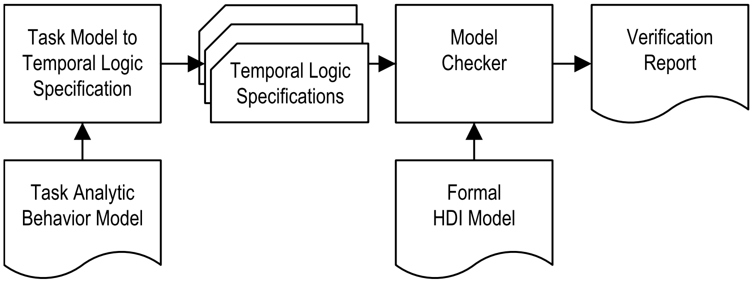

Task Model to Temporal Logic Specification
Temporal Logic Specifications
Model Checker
Verification Report
Task Analytic Behavior Model
Formal HDI Model

Fig. 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

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 were designed to support. This process produces a verification report which, if the task model goals are supported, will illustrate a path 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 as well as an algorithm that allows it to be iteratively applied to diagnose discovered validation failures.

2.1 Task analytic behavior modeling

Human task behavior is modeled using the EOFM (Bolton et al. 2011). 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 and Bass 2009, 2010a, 2010b; Bolton et al. 2011; Bolton 2010).

Tasks in EOFMs are hierarchical 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). Conditions are Boolean expressions written in terms of variables representing information from human-device interfaces, the operational environment, mission goals, other human operators, and/or any other external source. 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:

– *optor*: zero or more of the sub-acts must execute in any order;
– *or*: one or more of the sub-acts must execute in any order;
– *and*: all of the sub-acts must execute in any order;
– *ord*: all sub-acts must execute in the order they appear; and
– *xor*: exactly one sub-act must execute.

EOFM syntax is implemented in XML (eXtensible Markup Language) which preserves the hierarchal relationships between activity elements. It also has the added advantage of making EOFM models easy to edit and automatically parse.

EOFMs can be represented visually as discrete, tree-like graphs (shown later in Fig. 2) 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. 2011). As such, it has features which are more expressive than are needed for the high-level models required

Table 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

for this work. This includes the ability for every activity to 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 HDI modeling and model checking

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

2.3 Temporal logic specification

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 1) to assert properties about all paths through a model. In the work presented here, only the **G** and **F** operators are used in the construction of specifications.

2.4 Task model to temporal logic specification

Because LTL only allows specifications to reason about properties in all paths through a model, LTL model checkers (like SAL-SMC) do not allow for existence proofs. However, the equivalent of an existence proof can be obtained by having the generated property assert that the goals encompassed by the task model (t) will never happen (LTL pattern $\mathbf{G}\neg(t)$). When model checked against a HDI model, such a property will produce a counterexample (the equivalent of a witness in an existence proof) if the HDI model supports the task. If the task is not supported, the model checker will indicate that the property is true. Thus, existence is proved if a witness is produced and disproved if one is not.

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 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 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 $x \wedge \mathbf{F}((y_1 \wedge \mathbf{F}(z)) \wedge (y_2 \wedge \mathbf{F}(z)))$.

Finally, if we want to use a model checker to generate a witness to illustrate how the last property can be performed (if it is valid), we can place it in the $\mathbf{G}\neg(t)$ pattern from above: $\mathbf{G}\neg(x \wedge \mathbf{F}((y_1 \wedge \mathbf{F}(z)) \wedge (y_2 \wedge \mathbf{F}(z))))$.

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 $\delta \in \{optor, or, and, xor, ord, \varepsilon\}$ which are as defined above except for ε which represents no decomposition operator (for an activity that does not decompose). Σ is an ordered set of the activity's children (sub-activities) such that for an integer $m \geq 0$, $\Sigma = (\sigma_1, \sigma_2, \ldots, \sigma_m)$. For a given activity a, let a_α, a_ω, a_δ, a_Σ, and a_m represent the activity's precondition, completion condition, decomposition operator, ordered set of children, and number of children respectively. Further, for an integer i, $1 \leq i \geq a_m$, 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 (1).

$$\mathrm{f}(a) = \mathbf{G}\neg\big(\mathrm{g}(a, \mathit{true})\big) \tag{1}$$

where

$$\mathrm{g}(a, \phi) = a_\alpha \wedge \mathbf{F}\left(\begin{cases} \bigvee_{i=1}^{a_m}(\mathrm{g}(a_{\sigma_i}, a_\omega \wedge \mathbf{F}(\phi))) & \text{if } a_\delta = or \\ \bigwedge_{i=1}^{a_m}(\mathrm{g}(a_{\sigma_i}, a_\omega \wedge \mathbf{F}(\phi))) & \text{if } a_\delta = and \\ \bigoplus_{i=1}^{a_m}(\mathrm{g}(a_{\sigma_i}, a_\omega \wedge \mathbf{F}(\phi))) & \text{if } a_\delta = xor \\ \mathrm{h}(a_{\sigma_1}, a_\omega \wedge \mathbf{F}(\phi)) & \text{if } a_\delta = ord \\ a_\omega \wedge \mathbf{F}(\phi) & \text{otherwise} \end{cases}\right) \tag{2}$$

and

$$\mathrm{h}(a_{\sigma_i}, \phi) = \begin{cases} \mathrm{g}(a_{\sigma_i}, \mathrm{g}(a_{\sigma_{i+1}}, \phi)) & \text{if } i < a_m \\ \mathrm{g}(a_{\sigma_i}, \phi) & \text{if } i = a_m \end{cases} \tag{3}$$

Note that in (2), $\bigoplus$ is a one-hot detector—a special type of exclusive or operator that is true only if exactly one of the a_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.

2.5 Problem diagnosis

For a given task model, the nature of the specification property generated by (1) is such that HDI models checked against it (using the method in Fig. 1) will not generate a witness if the HDI model does not support the task. While this will indicate that the HDI is not valid for the given task, it does not provide any information about why this is the case. However, the task model can be systematically modified and the analysis method (Fig. 1) repeatedly applied to assist an analyst in such a diagnostic. In this section we present an algorithm that an analyst can use to accomplish this.

The LTL property produced by (1) is only dependent on the temporal relationships between the preconditions and completion conditions contained in the task model's activities. As such, any problem associated with a failure to produce a witness will be due to the inability of the HDI model to satisfy one or more of the conditions in the context reinforced by the task structure. Thus, the Algorithm 1 effectively starts with an empty task structure and systematically reconstructs it until it isolates a particular configuration that cannot be satisfied by the HDI. The analyst can then examine the modified structure to determine why the original task's goals could not be validated.

Once a problem with the system is discovered using the algorithm,[1] the analyst can examine the last configuration of the modified task model to develop an understanding of the relationships between the conditions associated with the problem. The HDI model can then be modified to potentially address the unsatisfied conditions. Once the analyst thinks he or she has corrected the problem, he or she should attempt to verify the original specification (generated from the unmodified task structure). If that does not work, the analyst will want to restart the algorithm to reassess the model.

It should be noted that the process of creating and modifying a' throughout the procedure can be easily accomplished by commenting out and/or re-including elements of the EOFM's XML syntax. a' can be created (Lines 2–6) by commenting out a's conditions and sub-activities. Conditions can be added to a' by "un-commenting" them in the markup. Similarly, sub-activities can be added in by removing the commenting from them.

3 Application

To illustrate how the method (Fig. 1) can be used to validate a HDI, a patient-controlled analgesia (PCA) pump example is presented. A PCA pump is a medical device that allows patients to exert some control over the delivery of pain medication

[1]The analyst can stop the algorithm at any time during its execution if he or she feels enough information has been obtained.

Algorithm 1 Diagnoses the failure of a task model to validate

```
Input: a task structure with root activity $r$ and a HDI model where $\mathrm{f}(r)$, from (1), does not produce a
    witness when verified against the HDI model with a model checker
Output: diagnostics, preceded by the **diagnose** command, about what particular condition in the task
    structure that resulted in the validation failure
 1: $a \leftarrow r$
 2: $a' \leftarrow$ a new activity   {Create a duplicate of $a$ that has no sub-activities or conditions (lines 2–6)}
 3: $a'_\alpha \leftarrow \varepsilon$
 4: $a'_\omega \leftarrow \varepsilon$
 5: $a'_\delta \leftarrow a_\delta$
 6: $a'_\Sigma \leftarrow \emptyset$
 7: **replace** $a$ with $a'$ in the task structure with root activity $r$
 8: **if** $a_\alpha \neq \varepsilon$ **then**   {Use the method to evaluate the precondition of $a$}
 9:     $a'_\alpha \leftarrow a_\alpha$
10:     **check** $\mathrm{f}(r)$ against the HDI model using the model checker
11:     **if** a witness is not produced **then**
12:         **diagnose** that the preconditions for $a$ is never contextually satisfied by the HDI
13:         **stop execution**
14:     **end if**
15: **end if**
16: **if** $a_\omega \neq \varepsilon$ **then**   {Use the method to evaluate the completion condition of $a$}
17:     $a'_\omega \leftarrow a_\omega$
18:     **check** $\mathrm{f}(r)$ against the HDI model using the model checker
19:     **if** a witness is not produced **then**
20:         **diagnose** that the completion condition for $a$ is never contextually satisfied by the HDI
21:         **stop execution**
22:     **end if**
23: **end if**
24: **for** $i \leftarrow 1$ to $a_m$ **do**   {Use the method to evaluate each of the sub-activities of $a$}
25:     **add** $a_{\sigma_i}$ to $a'_\Sigma$ such that $a_{\sigma_i} = a'_{\sigma_i}$
26:     **check** $\mathrm{f}(r)$ against the HDI model using the model checker
27:     **if** a witness is not produced **then**   {Use the algorithm to investigate $a_{\sigma_i}$}
28:         **diagnose** that a condition somewhere in $a_{\sigma_i}$ is never contextually satisfied by the HDI
29:         $a \leftarrow a_{\sigma_i}$
30:         **goto** line 2
31:     **end if**
32: **end for**
```

Note. The above algorithmic pseudo code was constructed using the notation described by Brito (2009). Variable assignment is indicated with a $\leftarrow$ where the value or variable to the right of the arrow is assigned to the variable to the left of it. Comments are surrounded by { and }. ε is used to represent an empty precondition and completion condition. Thus, for an activity a, $a_\alpha = \varepsilon$ means that a has no precondition and $a_\omega \neq \varepsilon$ means that a has a completion condition. **stop execution** is used to indicate that the algorithm has reached a stopping point

(using a button attached to the device) based on a prescription programmed into it by a human operator. The presented analysis focuses on the HDI the human operator uses to program prescriptions into the pump.

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.[2] This revealed that the HDI for

[2]The task analysis findings presented here are theoretical and are used only for the purpose of illustrating the method presented in this paper.

programming the PCA pump needed to allow the human operator to program in three types of prescriptions (encoded below as *PrescribedType*):

- One where the operator must specify the parameters associated with a treatment condition in which a patient must self administer all medication (*PrescribedType* = *PCA*);
- One where the operator must specify the parameters associated with a treatment condition under which the patient may self administer medication in addition to a continuous basal rate of medication that is delivered to the patient (*PrescribedType* = *BasalPCA*); and
- One where the operator must specify the parameters associated with a treatment condition in which only a continuous dosage is delivered at a steady rate (*PrescribedType* = *Continuous*) and the patient exerts no control over the administration of medication.

The analysis found that for all prescription types, the operator was required to enter the fluid volume (*PrescribedFluidVolume*) of the medication reserve being used in the administration of treatment as the first parameter. It also revealed that in all cases a bolus dose (*PrescribedBolus*), an initial supplementary dosage designed to raise the level of medication in the bloodstream to an effective level, should be the last parameter entered. 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 patient administrable medication dosage (*PrescribedPCADose*), and a minimum delay between dosages (*PrescribedDelay*). A *BasalPCA* prescription has all of the parameters 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 in which the human operator's goals (encoded into the preconditions and completion conditions) were to ensure the values programmed into the device (*FluidVolume*, *Bolus*, *PCADose*, *1HourLimit*, *Delay*, *BasalRate*, and *ContinuousRate*) match those that were prescribed (Fig. 2).

3.2 HDI design and formal modeling

At a later stage in the development process, a HDI design is created for the PCA pump (Fig. 3).[3] 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

[3]The presented device is based heavily on an actual PCA pump used in hospitals. Thus the presented HDI does not necessarily reflect the design philosophies of the author.

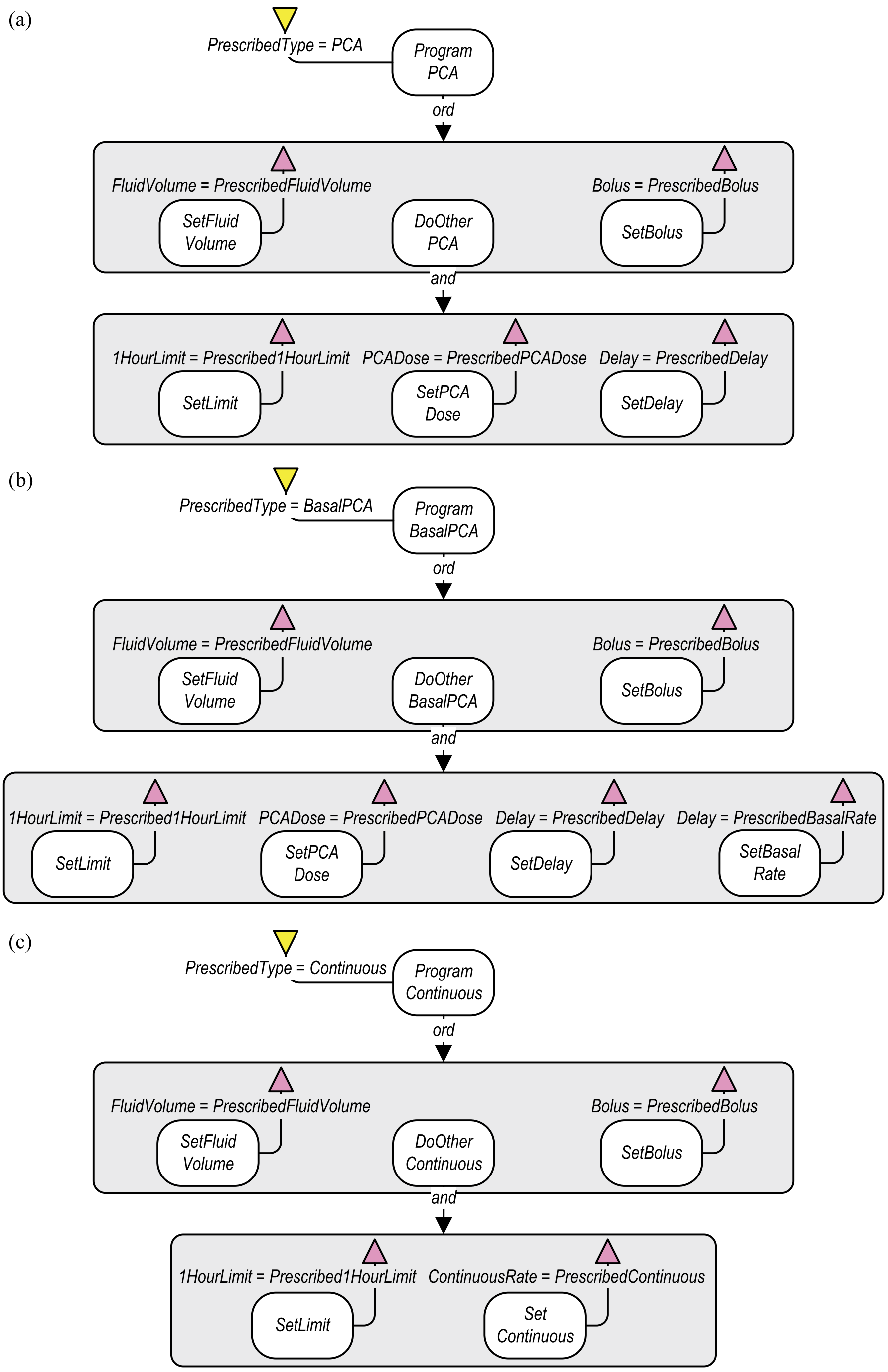


Fig. 2 EOFM task analytic behavior models for programming the three different prescriptions into a PCA pump: (**a**) *PCA*, (**b**) *BasalPCA*, and (**c**) *Continuous*

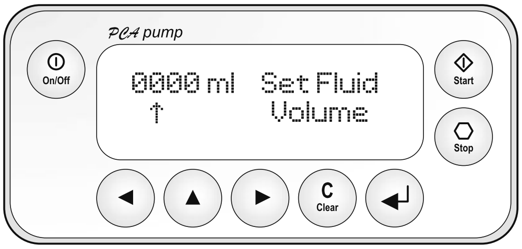


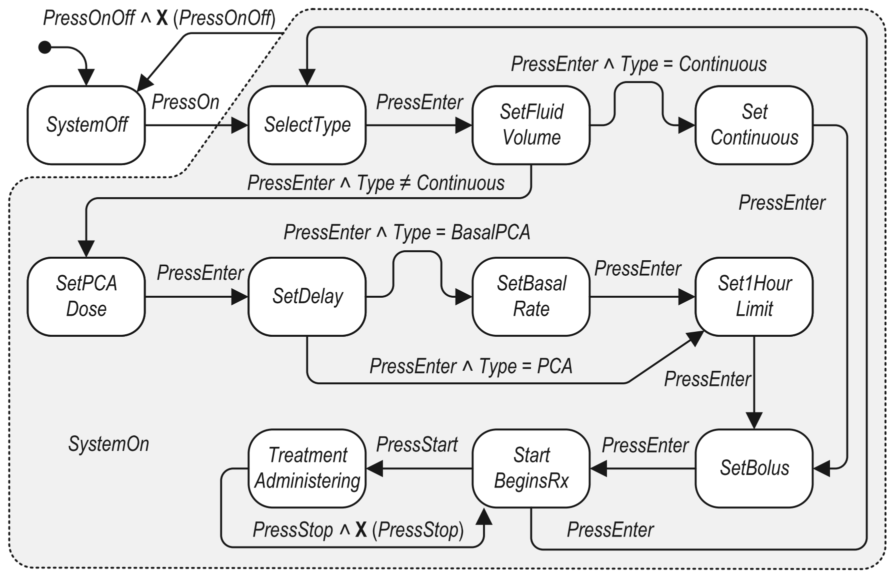
Fig. 3 HDI design for a PCA pump

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.

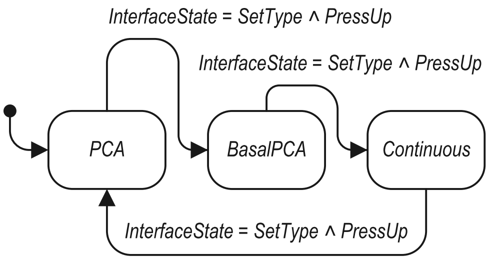
When a numerical value is required (such as the volume of a PCA dose), the value's name is listed in the interface message and the value is presented with the cursor under one of its digits. The programmer can move the position of the cursor by pressing the left and right buttons. He or she can press 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. However, for presentation purposes, the model is represented here as state transition diagrams (Fig. 4). The HDI model accepts a number of boolean input variables representing human actions, each corresponding to a button press. All are named with a *Press* prefix. Each is true when the associated action is being performed and false otherwise. The HDI model is composed of variables representing the state of the interface as indicated by the LCD (*InterfaceState*; Fig. 4(a)), the type of prescription selected (*Type*; Fig. 4(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 Fig. 4(c)). Note that to ensure that the HDI model is computationally tractable, all of the prescription values were modeled abstractly. A value initializes to *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.

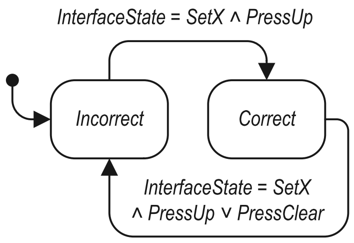
In addition to the system behavior, the formal model describes all of the possible types of prescriptions. Thus there is a *PrescribedType* which can assume each of the following values: *PCA*, *BasalPCA*, and *Continuous*. If *PrescribedType* = *PCA*: *PrescribedFluidVolume*, *PrescribedPCADose*, *PrescribedDelay*,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



(a) *InterfaceMessage*


InterfaceState = SetType ∧ PressUp
InterfaceState = SetType ∧ PressUp
PCA
BasalPCA
Continuous
InterfaceState = SetType ∧ PressUp

(b) *Type*



(c) *ValueX*

Fig. 4 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

Finally, the system model is completed with a model that can issue (making true) any possible human action (button press) 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 from Fig. 2(a), (b), and (c) were converted into linear temporal logic properties (4), (5), and (6) respectively.

$$\mathbf{G}\neg\left(\begin{array}{l}\mathit{PrescribedMode} = \mathit{PCA}\\ \wedge\mathbf{F}\left(\begin{array}{l}\mathit{FluidVolume} = \mathit{PrescribedFluidVolume}\\ \wedge\mathbf{F}\left(\begin{array}{l}\left(\begin{array}{l}\mathit{1HourLimit} = \mathit{Prescribed1HourLimit}\\ \wedge\mathbf{F}\,(\mathit{Bolus} = \mathit{PrescribedBolus})\end{array}\right)\\ \wedge\left(\begin{array}{l}\mathit{PCADose} = \mathit{PrescribedPCADose}\\ \wedge\mathbf{F}\,(\mathit{Bolus} = \mathit{PrescribedBolus})\end{array}\right)\\ \wedge\left(\begin{array}{l}\mathit{Delay} = \mathit{PrescribedDelay}\\ \wedge\mathbf{F}\,(\mathit{Bolus} = \mathit{PrescribedBolus})\end{array}\right)\end{array}\right)\end{array}\right)\end{array}\right) \quad (4)$$

$$\mathbf{G}\neg\left(\begin{array}{l}\mathit{PrescribedType} = \mathit{BasalPCA}\\ \wedge\mathbf{F}\left(\begin{array}{l}\mathit{FluidVolume} = \mathit{PrescribedFluidVolume}\\ \wedge\mathbf{F}\left(\begin{array}{l}\left(\begin{array}{l}\mathit{1HourLimit} = \mathit{Prescribed1HourLimit}\\ \wedge\mathbf{F}\,(\mathit{Bolus} = \mathit{PrescribedBolus})\end{array}\right)\\ \wedge\left(\begin{array}{l}\mathit{PCADose} = \mathit{PrescribedPCADose}\\ \wedge\mathbf{F}\,(\mathit{Bolus} = \mathit{PrescribedBolus})\end{array}\right)\\ \wedge\left(\begin{array}{l}\mathit{Delay} = \mathit{PrescribedDelay}\\ \wedge\mathbf{F}\,(\mathit{Bolus} = \mathit{PrescribedBolus})\end{array}\right)\\ \wedge\left(\begin{array}{l}\mathit{BasalRate} = \mathit{PrescribedBasalRate}\\ \wedge\mathbf{F}\,(\mathit{Bolus} = \mathit{PrescribedBolus})\end{array}\right)\end{array}\right)\end{array}\right)\end{array}\right) \quad (5)$$

$$\mathbf{G}\neg\left(\begin{array}{l}\mathit{PrescribedType} = \mathit{Continuous}\\ \wedge\mathbf{F}\left(\begin{array}{l}\mathit{FluidVolume} = \mathit{PrescribedFluidVolume}\\ \wedge\mathbf{F}\left(\begin{array}{l}\left(\begin{array}{l}\mathit{1HourLimit} = \mathit{Prescribed1HourLimit}\\ \wedge\mathbf{F}\,(\mathit{Bolus} = \mathit{PrescribedBolus})\end{array}\right)\\ \wedge\left(\begin{array}{l}\mathit{ContinuousRate} = \mathit{PrescribedContinuous}\\ \wedge\mathbf{F}\,(\mathit{Bolus} = \mathit{PrescribedBolus})\end{array}\right)\end{array}\right)\end{array}\right)\end{array}\right) \quad (6)$$

Analyses were performed using SAL-SMC running on Cygwin on a laptop computer with a 2.5 GHz Intel Core 2 Duo processor and 4 gigabytes of RAM.[4] When checked against the formal system model of the HDI design, the properties (4) and (5) produced the expected counterexamples/witnesses illustrating how the associated tasks (Figs. 2(a) and (b)) could successfully be performed. However, (6) did not return a counterexample, indicating that the task in Fig. 2(c) could not be completed as specified. Thus, the HDI design is valid for the tasks described in Figs. 2(a) and (b). However, it is not for the task in Fig. 2(c). The entire model checking process took 32 seconds (for all three properties combined). The model had a reported 53,248,244 states.

3.4 Problem diagnosis

Algorithm 1 was applied to diagnose the discovered failure:

- The process was started with $r = \mathit{ProgramContinuous}$.
- At line 1, a was set to r making $a = \mathit{ProgramContinuous}$.

[4]Machine specifications are reported because they influence reported verification time statistics.

- In lines 2–6, a' was set to a new activity (referred to as *ProgramContinuous*$'$ henceforth) that was a copy of *ProgramContinuous* without its precondition, completion condition, or sub-activities.
- At line 7, *ProgramContinuous* was replaced in r by *ProgramContinuous*$'$.
- Because (at line 8) *ProgramContinuous* had a precondition, *ProgramContinuous*$'$ was given the precondition of *ProgramContinuous* at line 9, creating the task structure shown in Fig. 5(a).
- At line 10, f(r) was computed using (1) to produce (7) that was checked against the HDI model, producing a witness.[5]

$$\mathbf{G}\neg\left(\mathit{PrescribedType} = \mathit{Continuous}\right) \tag{7}$$

- Because (at line 11) a witness had been produced and because (at line 16) *ProgramContinuous* did not have a completion condition, execution moved to line 24 where i was set to 1.
- At line 25, a_{σ_1} (activity *SetFluidVolume*) was added to the sub-activities of *ProgramContinuous*$'$ creating the task structure in Fig. 5(b).
- At line 26, f(r) was computed using (1) to produce (8) which was checked against the HDI model, producing a witness.

$$\mathbf{G}\neg\begin{pmatrix}\mathit{PrescribedType} = \mathit{Continuous}\\ \wedge\mathbf{F}\,(\mathit{FluidVolume} = \mathit{PrescribedFluidVolume})\end{pmatrix} \tag{8}$$

- Because (at line 27) a witness had been produced, execution moved to line 24 where i was set to 2.
- At line 25, a_{σ_2} (activity *DoOtherContinuous*) was added to the sub-activities of *ProgramContinuous*$'$ creating the task structure in Fig. 5(c).
- At line 26, f(r) was computed using (1) to produce (9) which was checked against the HDI, but no witness was produced.

$$\mathbf{G}\neg\begin{pmatrix}\mathit{PrescribedType} = \mathit{Continuous}\\ \wedge\mathbf{F}\begin{pmatrix}\mathit{FluidVolume} = \mathit{PrescribedFluidVolume}\\ \wedge\mathbf{F}\begin{pmatrix}\mathit{1HourLimit} = \mathit{Prescribed1HourLimit}\\ \wedge\mathit{ContinuousRate} = \mathit{PrescribedContinuous}\end{pmatrix}\end{pmatrix}\end{pmatrix} \tag{9}$$

- Because (at line 27) a witness was not produced, the algorithm indicated (at line 28) that there was a problem with *ProgramContinuous*.
- At line 29, a was set to *DoOtherContinuous*.
- At line 30, the algorithm jumped to line 2.
- In lines 2–6, a' was set to a new activity (referred to as *DoOtherContinuous*$'$ henceforth) that was a copy of *DoOtherContinuous* without its precondition, completion condition, or sub-activities.
- At line 7, *DoOtherContinuous* was replaced in r by *DoOtherContinuous*$'$.
- Because (at line 8) *DoOtherContinuous* had no precondition and (at line 16) no completion condition, execution moved to line 24 where i was set to 1.
- At line 25, a_{σ_1} (activity *SetLimit*) was added to the sub-activities of *ProgramContinuous*$'$ creating the task structure in Fig. 5(d).

[5] All model checking verifications discussed in this list were performed in 24 seconds or less.

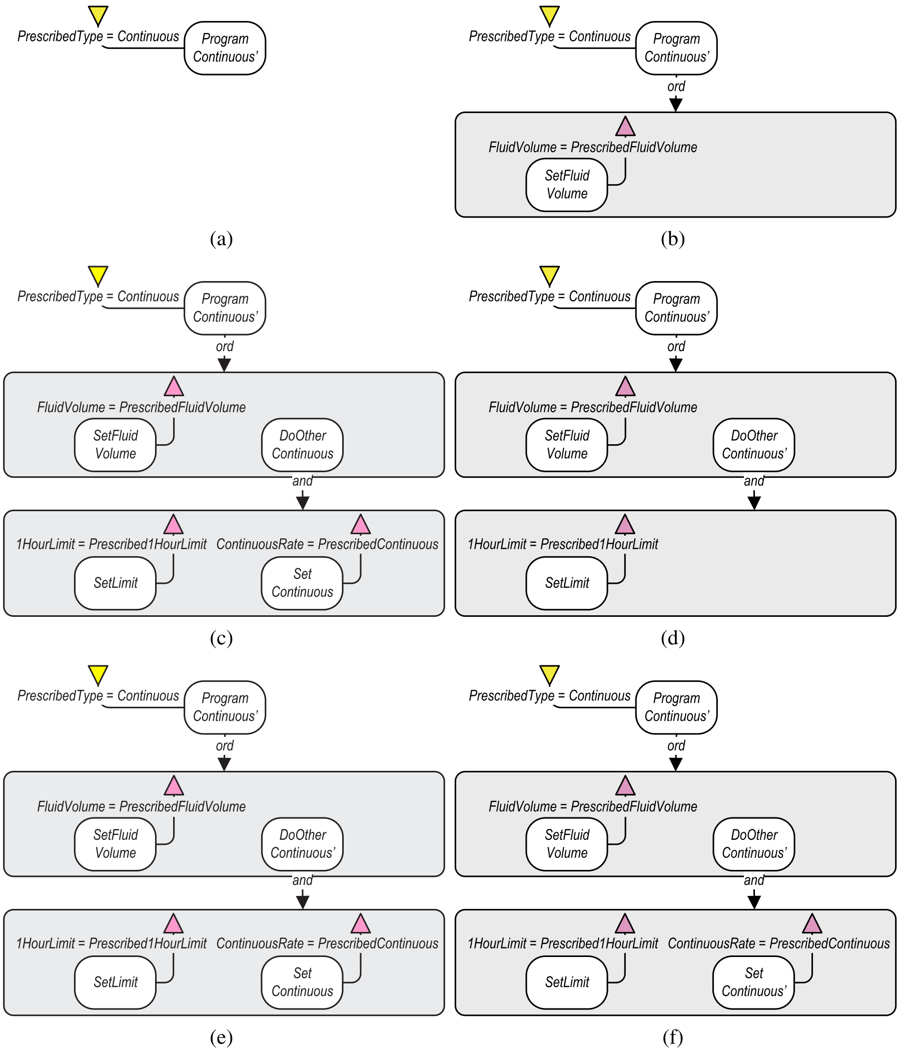

PrescribedType = Continuous
Program Continuous'
ord
FluidVolume = PrescribedFluidVolume
SetFluid Volume
DoOther Continuous
DoOther Continuous'
and
1HourLimit = Prescribed1HourLimit
ContinuousRate = PrescribedContinuous
SetLimit
Set Continuous
Set Continuous'
(a)
(b)
(c)
(d)
(e)
(f)

Fig. 5 Visualizations of the task models iteratively generated ((**a**)–(**f**)) by the diagnosis algorithm (1) when it was used to evaluate the task model from Fig. 2(c)

– At line 26, f(r) was computed using (1) to produce (10) which was checked against the HDI model, producing a witness.

$$\mathbf{G}\neg\left(\begin{array}{l} PrescribedType = Continuous \\ \wedge\mathbf{F}\left(\begin{array}{l} FluidVolume = PrescribedFluidVolume \\ \wedge\mathbf{F}\,(1HourLimit = Prescribed1HourLimit)\end{array}\right)\end{array}\right) \tag{10}$$

– Because (at line 27) a witness was produced, execution moved to line 24 where i was set to 2.

– At line 25, a_{σ_2} (activity *SetContinuous*) was added to the sub-activities of *DoOther-Continuous′* creating the task structure in Fig. 5(e).

- At line 26, f(r) was computed using (1) to produce (11) which was checked against the HDI model, but no witness was produced.

$$\mathbf{G}\neg\left(\begin{array}{l} \mathit{PrescribedType} = \mathit{Continuous} \\ \wedge\mathbf{F}\left(\begin{array}{l} \mathit{FluidVolume} = \mathit{PrescribedFluidVolume} \\ \wedge\mathbf{F}\left(\begin{array}{l} \mathit{1HourLimit} = \mathit{Prescribed1HourLimit} \\ \wedge\mathit{ContinuousRate} = \mathit{PrescribedContinuous} \end{array}\right) \end{array}\right) \end{array}\right) \tag{11}$$

- Because (at line 27) a witness was not produced, the algorithm indicated (at line 28) that there was a problem with *SetContinuous*.
- At line 29, a was set to *SetContinuous*.
- At line 30, the algorithm jumped to line 2.
- In lines 2–6, a' was set to a new activity (referred to as *SetContinuous*$'$ henceforth) that was a copy of *SetContinuous* without its precondition, completion condition, or sub-activities.
- At line 7, *SetContinuous* was replaced in r by *SetContinuous*$'$.
- Because (at line 8) *SetContinuous* had no precondition but (at line 16) did have a completion condition, execution moved to line 17.
- At line 17, *SetContinuous*$'$ was given the completion condition of *SetContinuous* creating the task structure shown in Fig. 5(f).
- At line 18, f(r) was computed using (1) to produce (12) which was checked against the HDI model, but no witness was produced.

$$\mathbf{G}\neg\left(\begin{array}{l} \mathit{PrescribedType} = \mathit{Continuous} \\ \wedge\mathbf{F}\left(\begin{array}{l} \mathit{FluidVolume} = \mathit{PrescribedFluidVolume} \\ \wedge\mathbf{F}\left(\begin{array}{l} \mathit{1HourLimit} = \mathit{Prescribed1HourLimit} \\ \wedge\mathit{ContinuousRate} = \mathit{PrescribedContinuous} \end{array}\right) \end{array}\right) \end{array}\right) \tag{12}$$

- Because (at line 19) a witness was not produced, the algorithm indicated (at line 20) that the completion condition on *SetContinuous* would never be contextually satisfied by the HDI.
- At line 21, execution of the algorithm stopped.

By examining the last modified task structure produced by the algorithm (Fig. 5(f)) and its corresponding temporal logic specification (12), we can see that when the human operator is attempting to program a continuous prescription, the HDI never allows the human operator to program both a one hour limit and a continuous rate after entering a fluid volume.

3.5 Redesign

The HDI model was examined to determine why the discovered problem was occurring. This revealed that (in Fig. 4(a)) when the human operator is attempting to program in a *Continuous* prescription type, that the interface does not allow him or her to set a one hour limit, something required for a continuous prescription according to the task in Fig. 2(c). Thus, the HDI can be redesigned so that after the human operator sets the continuous rate, he or she sets the one hour limit, and then enters the bolus.

Making this change in the formal model results in (4), (5), and (6) producing the expected witnesses after 14 seconds of model checking.

4 Discussion and conclusions

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. These can then be used to validate that formal models of HDI designs support the associated tasks using the process from Fig. 1. Further, the work has shown that this method can be used iteratively while modifying the original task behavior model as part of an algorithm to diagnose discovered validation failures. An implementation of this process was presented that uses SAL and EOFM. Both the method and diagnostic algorithm were illustrated with a PCA pump programming HDI example that showed how shortcomings in a design could be discovered using these approaches.

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 2008; Paternò 1997). Despite this, there are several limitations of the presented method which future work should address. Additionally, for the method to be useful for designers, it should be adapted to the emerging human system integration design methods. Both of these are discussed below.

4.1 Method improvements

4.1.1 Automating the diagnostic algorithm

As presented here, the diagnostic algorithm must be performed manually. However, it is possible that it could be completely automated. Future work will investigate this possibility.

4.1.2 Task model condition logic

EOFM requires that strategic knowledge in preconditions and completion conditions be specified using variables in Boolean expressions (Bolton et al. 2011). 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 any HDI models they wish to evaluate. 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 to avoid such problems.

4.1.3 Other task-related properties

The task-related properties generated in this work fall into a subcategory of HDI specification properties called weak task completeness (Abowd et al. 1995; Paternò

1997). Such specifications assert that there is at least one action sequence from initial interface states that will eventually achieve a specific task's goals. However, there are other task-related specification properties. Strong task completeness asserts that, for any given possible sequences of actions from a specific initial interface state, there is a set of additional actions that will eventually achieve task goals (Abowd et al. 1995). Task connectedness properties assert that from any interface state, there is at least one action sequence that will achieve task goals (Abowd et al. 1995). Finally, undo/reversibility specification properties assert that the effects of a specific action or task can eventually be undone with at least one sequence of actions (Abowd et al. 1995; Campos 2008; Paternò 1997). These other task-related properties can provide additional useful insights into how a HDI supports task goals. Future work should determine if the method presented here could be adapted to generate them.

4.1.4 Additional diagnostics and task exploration

The diagnostic algorithm introduced here allows an analyst to isolate a set of conditions associated with a HDI model's failure to validate. However, an analyst may wish to perform additional diagnostics to better understand a problem. For example, if a problem is isolated to the sub-activities of an activity that contains an *and* decomposition, it may not be clear which of the sub-activities are causing the problem. In such a situation, the analyst may wish to consider every possible subset of sub-activities to determine which combination is associated with the problem being evaluated. Future work should investigate what types of additional diagnostic analyses would be most useful to analysts.

The method 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. However, an analyst may want to determine when a specific execution of the task (satisfaction of task goals) can occur. In such a situation, an analyst may be able to modify the task structure to reflect the behavior of interest. Future work should investigate how this might be done systematically, possibly through the use of automated test case generation methods (Hamon et al. 2005).

4.1.5 Scalability

The model presented in the paper was relatively simple. However, the method will likely not scale well to more complex applications. This is because all model checking analyses (not just those that make use of SAL) are impacted by the state explosion problem: as the complexity of the modeled system increases, the memory and time required to store the combinatorially expanding state space can easily exceed the available resources. An example of this problem that directly relates to the application presented here is discussed by Bolton and Bass (2010a).

In practice, the state explosion problem can be addressed in a number of different ways. Firstly, symbolic model checking (which is used in this work) allows the state space of the formal system model to be transformed into a much more efficient representation before model checking is started (Burch 1992). Abstraction techniques, where system details that are not relevant to the analysis goals are abstracted away in the system model, can also be used. In this work, all numerical values were represented abstractly to keep the model tractable. However, there are a number of different abstraction techniques that can be employed. An overview of these was documented by Mansouri-Samani et al. (2007). Other more automated techniques allow select portions of the state space to be searched without compromising the accuracy of the verification. These can include partial order reduction (Holzmann and Peled 1994) and symmetry (Clarke et al. 1996), which reduces the model state space by exploiting redundancies in the formal model; compositional verification (Cobleigh et al. 2003), which divides specification properties into properties about composable parts of the larger system such that, if each part of the system verifies to true separately, the entire system will satisfy the undivided specifications; and counterexample-guided abstraction refinement (Clarke et al. 2003), where an abstraction of the system model is automatically created and intelligently de-abstracted over multiple model checking runs until a genuine problem is discovered or the specification property is proven true.

While these techniques do not solve the state explosion problem, they do allow for useful analyses to be performed. Future advances in formal methods and model checking technology will only improve this situation.

4.2 Support for human systems integration

In a report for the National Academies, Pew and Mavor (2007) make a number of recommendations for how to improve human systems integration (HSI) in the systems engineering and development process. Among their policy recommendations is the suggestion that large organizations with system acquisition programs should establish methods to verify and validate HSI requirements. The report also emphasizes the need to include human design considerations early and throughout the system design process. This point is also advocated by Booher and Minninger (2003), who list the use of human-centered design (where human factors considerations are included is systems specifications and iteratively evaluated over the system life cycle) as the second most important factor in the success of HSI programs.

The method presented here requires that task analyses be conducted early in the design process to establish requirements on human task behavior. Further, formal verification is used to validated that designs satisfy these requirements in later iterations of design. Thus, the presented method fits nicely within the recommendations of Pew and Mavor (2007) and Booher and Minninger (2003). However, for the method to be used in such a context, it will need to be made to be compatible with other design and analysis infrastructures. The discussion below explores how the method could be integrated with other HSI design tools and practices.

4.2.1 Human-device interface design and modeling

The presented method requires the analyst to create a formal model of the HDI design he or she wishes to validate, something that is not standard practice in HSI work and is likely unfamiliar to most HDI designers and analysts. 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 et al. 1997, 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 in HSI.

4.2.2 Task and cognitive models used at different stages of design and analysis

In the approach presented here, task analytic behavior models are used early in the design process to represent the constraints between human operator goals. However, task analytic behavior models can be used at different stages of HSI design or analysis. Models constructed before the design of a new HDI (like those used in this work) can be used to inform design (Limbourg and Vanderdonckt 2003). Models constructed for a given design will be much more detailed, often modeling atomic human actions at the bottom of the goals hierarchy. These can be used to do all of the following: construct intelligent tutoring systems (Chu 1995), evaluate usability (Lecerof and Paternò 1998), or model human behavior in simulations (Göknur 2004). They can also be paired with a cognitive architecture to perform human performance analyses of HDI designs as has been done with the GOMS family of models (John and Kieras 1996; Kieras 2003; Amant et al. 2005). Similarly, analyses that use cognitive modeling often employ task knowledge. Such models have proven useful in evaluating human performance with HDIs (see, for example, Ritter et al. 2006, 2007; Mueller et al. 2011) and the human operators role in the failure of complex systems (Byrne and Kirlik 2005).

Detailed models have also been used with model checking to evaluate system safety properties (Bolton and Bass 2010a; Bolton et al. 2012, 2011; Palanque et al. 1996; Aït-Ameur et al. 2003; Aït-Ameur and Baron 2006; Basnyat et al. 2007, 2008; Fields 2001; Cerone et al. 2005; Rukšenas et al. 2009; Basuki et al. 2009). However, such analyses represent human behavior differently than is done in the presented method: cognitive and/or task behavior models are incorporated into the larger formal model so that they can be evaluated as part of the system.

It should be possible for task models developed early in the design process to be adapted into more detailed models for use in latter analyses.

Further, while the implementation of the method presented here is dependent on EOFM, it should be adaptable to other task analytic behavior modeling techniques more commonly associated with the analyses discussed above. Future work should investigate this.

4.2.3 Team and organizational behavior

The method presented here assumes that the device being evaluated is for a single operator. However, HSI projects may involve interactions and/or collaboration between multiple human operators (operating independently or as part of an organization) and multiple automated systems. While formal methods are very good at evaluating concurrently executing automated systems and processes (Clarke et al. 1999; Wing

1990), there have been few instances were formal methods have been used to evaluate systems involving multiple human operators. Paternò et al. (1998) and Bass et al. (2011) have investigated how human collaboration can be represented using unique task structures in formal task analytic behavior models. Alternatively, Jonker et al. (2007) have investigated how organizational protocols could be modeled and evaluated in a formal context. Future work should determine how the method presented in this paper could be adapted to exploit these advances.

Acknowledgements The author would like to thank Dorrit Billman and Michael Feary for the feedback they provided during the preparation of this manuscript. The work documented here was completed while the author was a Senior Research Associate for the San José State University Research Foundation at the NASA Ames Research Center. 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.

References

Abowd GD, Wang H, Monk AF (1995) A formal technique for automated dialogue development. In: Proceedings of the 1st conference on designing interactive systems. ACM Press, New York, pp 219–226

Aït-Ameur Y, Baron M (2006) Formal and experimental validation approaches in HCI systems design based on a shared event B model. Int J Softw Tools Technol Transf 8(6):547–563

Aït-Ameur Y, Baron M, Girard P (2003) Formal validation of HCI user tasks. In: Proceedings of the international conference on software engineering research and practice. CSREA Press, Las Vegas, pp 732–738

Amant R, Freed A, Ritter F (2005) Specifying act-r models of user interaction with a goms language. Cogn Syst Res 6(1):71–88

Basnyat S, Palanque P, Schupp B, Wright P (2007) Formal socio-technical barrier modelling for safety-critical interactive systems design. Saf Sci 45(5):545–565

Basnyat S, Palanque PA, Bernhaupt R, Poupart E (2008) Formal modelling of incidents and accidents as a means for enriching training material for satellite control operations. In: Proceedings of the joint ESREL 2008 and 17th SRA-Europe conference. Taylor and Francis, London, CD–ROM

Bass EJ, Bolton ML, Feigh K, Griffith D, Gunter E, Mansky W, Rushby J (2011) 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. IEEE, Piscataway, pp 1817–1824

Basuki TA, Cerone A, Griesmayer A, Schlatte R (2009) Model-checking user behaviour using interacting components. Form Asp Comput 21(6):571–588

Bolton ML (2010) Using task analytic behavior modeling, erroneous human behavior generation, and formal methods to evaluate the role of human-automation interaction in system failure. PhD thesis, University of Virginia, Charlottesville

Bolton ML, Bass EJ (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. HFES, Santa Monica, pp 764–768

Bolton ML, Bass EJ (2010a) Formally verifying human-automation interaction as part of a system model: limitations and tradeoffs. Innov Syst Softw Eng 6(3):219–231

Bolton ML, Bass EJ (2010b) Using task analytic models to visualize model checker counterexamples. In: Proceedings of the 2010 IEEE international conference on systems, man, and cybernetics. IEEE, Piscataway, pp 2069–2074

Bolton ML, Bass EJ, Siminiceanu RI (2012) Using formal verification to evaluate human-automation interaction in safety critical systems, a review. IEEE Trans Syst Man Cybern, Part A, Syst Hum (accepted)

Bolton ML, Siminiceanu RI, Bass EJ (2011) A systematic approach to model checking human-automation interaction using task-analytic models. IEEE Trans Syst Man Cybern, Part A, Syst Hum 41(5):961–976

Booher H, Minninger J (2003) Human systems integration in army systems acquisition. In: Booher HR (ed) Handbook of human systems integration. Wiley, Hoboken, pp 663–698

Bredereke J, Lankenau A (2005) Safety-relevant mode confusions–modelling and reducing them. Reliab Eng Syst Saf 88(3):229–245

Brito R (2009) The algorithms bundle. http://carroll.aset.psu.edu/pub/CTAN/macros/latex/contrib/algorithms/algorithms.pdf

Burch JR, Clarke EM, Dill DL, Hwang J, McMillan KL (1992) Symbolic model checking: 10^{20} states and beyond. Inf Comput 98(2):142–171

Byrne M, Kirlik A (2005) Using computational cognitive modeling to diagnose possible sources of aviation error. Int J Aviat Psychol 15(2):135–155

Campos JC, Harrison M (1997) Formally verifying interactive systems: a review. In: Proceedings of the fourth international Eurographics workshop on the design, specification, and verification of interactive systems. Springer, Berlin, pp 109–124

Campos JC, Harrison MD (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. Springer, Berlin, pp 72–85

Campos JC, Harrison MD (2009) Interaction engineering using the ivy tool. In: Proceedings of the 1st ACM SIGCHI symposium on engineering interactive computing systems. ACM Press, New York, pp 35–44

Cerone A, PA Lindsay, Connelly S (2005) Formal analysis of human-computer interaction using model-checking. In: Proceedings of the 3rd IEEE international conference on software engineering and formal methods. IEEE Computer Society, Los Alamitos, pp 352–362

Chu RW, Mitchell CM, Jones PM (1995) Using the operator function model and OFMspert as the basis for an intelligent tutoring system: towards a tutor/aid paradigm for operators of supervisory control systems. IEEE Trans Syst Man Cybern, Part A, Syst Hum 25(7):1054–1075

Clarke E, Grumberg O, Jha S, Lu Y, Veith H (2003) Counterexample-guided abstraction refinement for symbolic model checking. J ACM 50(5):752–794

Clarke EM, Enders R, Filkorn T, Jha S (1996) Exploiting symmetry in temporal logic model checking. Form Methods Syst Des 9(1):77–104

Clarke EM, Grumberg O, Peled DA (1999) Model checking. MIT Press, Cambridge

Cobleigh J, Giannakopoulou D, Păsăreanu C (2003) In: Proceedings of the 9th international conference on tools and algorithms for the construction and analysis of systems. Springer, Berlin, pp. 331–346

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

Dwyer MB, Carr V, Hines L (1997) Model checking graphical user interfaces using abstractions. In: Proceedings of the sixth European software engineering conference. Springer, New York, pp 244–261

Dwyer MB, Tkachuk O, Robby, Visser W (2004) Analyzing interaction orderings with model checking. In: Proceedings of the 19th IEEE international conference on automated software engineering. IEEE Computer Society, Los Alamitos, pp 154–163

Emerson EA (1990) Temporal and modal logic. In: van Leeuwen J, Meyer AR, Nivat M, Paterson M, Perrin D (eds) Handbook of theoretical computer science. MIT Press, Cambridge, Chap 16, pp 995–1072

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. Springer, Berlin, pp 487–496

Fields RE (2001) Analysis of erroneous actions in the design of critical systems. PhD thesis, University of York, York

Göknur S, Bolton ML, Bass EJ (2004) Adding a motor control component to the operator function model expert system to investigate air traffic management concepts using simulation. In: Proceedings of the IEEE international conference and systems, man, and cybernetics. IEEE, Piscataway, pp 886–892

Hamon G, De Moura L, Rushby J (2005) Automated test generation with SAL. Tech. rep., Menlo Park. http://www.csl.sri.com/users/rushby/papers/salatg.pdf

Hartson HR, Siochi AC, Hix D (1990) The UAN: a user-oriented representation for direct manipulation interface designs. ACM Trans Inf Syst 8(3):181–203

Holzmann G, Peled D (1994) An improvement in formal verification. In: Proceedings of the 7th international conference on formal description techniques. Chapman and Hall, London, pp 197–211

John BE, Kieras DE (1996) Using GOMS for user interface design and evaluation: which technique? ACM Trans Comput-Hum Interact 3(4):287–319

Jonker CM, Schut MC, Treur J, Yolum P (2007) Analysis of meeting protocols by formalisation, simulation, and verification. Comput Math Organ Theory 13(3):283–314

Joshi A, Miller SP, Heimdahl MP (2003) Mode confusion analysis of a flight guidance system using formal methods. In: Proceedings of the 22nd digital avionics systems conference. IEEE, Piscataway, pp 2.D.1-1–2.D.1-12

Kieras D (2003) Goms models for task analysis. Lawrence Erlbaum Associates, Mahwah, pp 83–116

Kirwan B, Ainsworth LK (1992) A guide to task analysis. Taylor and Francis, London

Lecerof A, Paternò F (1998) Automatic support for usability evaluation. IEEE Trans Softw Eng 24(10):863–888

Limbourg Q, Vanderdonckt J (2003) Comparing task models for user interface design. In: Diaper D, Stanton N (eds) The handbook of task analysis for human-computer interaction. Lawrence Erlbaum Associates, Mahwah, pp 135–154

Loer K, Harrison MD (2006) An integrated framework for the analysis of dependable interactive systems (IFADIS): its tool support and evaluation. Autom Softw Eng 13(4):469–496

Mansouri-Samani M, Pasareanu CS, Penix JJ, Mehlitz PC, O'Malley O, Visser WC, Brat GP, Markosian LZ, Pressburger TT (2007) Program model checking: a practitioner's guide. Tech. rep., Intelligent Systems Division, NASA Ames Research Center, Moffett Field

Mitchell CM, Miller RA (1986) A discrete control model of operator function: a methodology for information display design. IEEE Trans Syst Man Cybern, Part A, Syst Hum 16(3):343–357

Mueller S, Simpkins B, Anno G, Fallon C, Price O, McClellan G (2011) Adapting the task-taxon-task methodology to model the impact of chemical protective gear. Comput Math Organ Theory 17:251–271

Palanque PA, Bastide R, Senges V (1996) Validating interactive system design through the verification of formal task and system models. In: Proceedings of the IFIP TC2/WG2.7 working conference on engineering for human-computer interaction. Chapman and Hall, London, pp 189–212

Parnas DL (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. ACM Press, New York, pp 379–385

Paternò F (1997) Formal reasoning about dialogue properties with automatic support. Interact Comput 9(2):173–196

Paternò F, Santoro C (2001) 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. Springer, Berlin, pp 135–150

Paternò F, Mancini C, Meniconi S (1997) Concurtasktrees: a diagrammatic notation for specifying task models. In: Proceedings of the IFIP TC13 international conference on human-computer interaction. Chapman and Hall, London, pp 362–369

Paternò F, Santoro C, Tahmassebi S (1998) Formal model for cooperative tasks: concepts and an application for en-route air traffic control. In: Proceedings of the 5th international conference on the design, specification, and verification of interactive systems. Springer, Vienna, pp 71–86

Pew R, Mavor A (2007) Human-system integration in the system development process: a new look. National Academies Press, Washington

Ritter F, Kukreja U, Amant R (2007) Including a model of visual processing with a cognitive architecture to model a simple teleoperation task. J Cogn Eng Decis Mak 1(2):121

Ritter FE, Van Rooy D, Amant RS, Simpson K (2006) Providing user models direct access to interfaces: an exploratory study of a simple interface with implications for HRI and HCI. IEEE Trans Syst Man Cybern, Part A, Syst Hum 36(3):592–601

Rukšenas R, Back J, Curzon P, Blandford A (2009) Verification-guided modelling of salience and cognitive load. Form Asp Comput 21(6):541–569

Rushby J (2002) Using model checking to help discover mode confusions and other automation surprises. Reliab Eng Syst Saf 75(2):167–177

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

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

Matthew L. Bolton is an Assistant Professor in the Department of Mechanical and Industrial Engineering at the University of Illinois at Chicago. He received the B.S. degree in computer science, the M.S. degree

in systems engineering, and the Ph.D. degree in systems engineering from the University of Virginia, Charlottesville, in 2003, 2006, and 2010, respectively. He has over 20 peer reviewed publications on a variety of subjects including spatial awareness, psychophysics, and human behavior modeling. His primary research interest 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. He is an officer in the Human Factors and Ergonomics Society's Human Performance Modeling Technical Group, the Co-chair of the IEEE Systems Man and Cybernetics Human-Computer Interaction Technical Group, as well as the founder and administrator of Formal Methods for Human Factors Engineering (http://www.fmhfe.com/).