The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-319-66197-1_18

A Hazard Analysis Method for Systematic Identification of Safety Requirements for User Interface Software in Medical Devices

Paolo Masci[1], Yi Zhang[2], Paul Jones[2], José C. Campos[1]

[1] INESC TEC and Universidade do Minho, Portugal
{paolo.masci,jose.c.campos}@inesctec.pt
[2] US Food and Drug Administration, USA
{yi.zhang2,paul.jones}@fda.hhs.gov

Abstract. Formal methods technologies have the potential to verify the usability and safety of user interface (UI) software design in medical devices, enabling significant reductions in use errors and consequential safety incidents with such devices. This however depends on comprehensive and verifiable safety requirements to leverage these techniques for detecting and preventing flaws in UI software that can induce use errors. This paper presents a hazard analysis method that extends Leveson's System Theoretic Process Analysis (STPA) with a comprehensive set of causal factor categories, so as to provide developers with clear guidelines for systematic identification of use-related hazards associated with medical devices, their causes embedded in UI software design, and safety requirements for mitigating such hazards. The method is evaluated with a case study on the Gantry-2 radiation therapy system, which demonstrates that 1) as compared to standard STPA, our method allowed us to identify more UI software design issues likely to cause use-related hazards; and 2) the identified UI software design issues facilitated the definition of precise, verifiable safety requirements for UI software, which could be readily formalized in verification tools such as Prototype Verification System (PVS).

Keywords: Requirements identification/formalization; User interface software; Medical devices.

1 Introduction

Use errors with medical devices are a leading cause of device incidents reported in the healthcare domain (e.g., see [1,15]). Errors in user interface (UI) software in medical devices, including design flaws and implementation mistakes, can disrupt expected device-user interaction and induce use errors. For example, a diabetes management mobile app was recalled in the U.S. because its UI software could erroneously reset the recommended insulin bolus dosage when the user changes the smartphone's orientation, which might cause the user to inadvertently command and receive unsafe insulin therapies [9].

It is thus crucial to the safety of medical devices to systematically assess their UI software design and ensure that design issues likely inducing use errors

are appropriately addressed. Formal methods have the potential to enable such rigorous assessment. For example, our previous research shows that theorem proving can be used to detect latent flaws in data entry software [19]. In fact, the formal methods community has devoted substantial effort on methods and tools for analyzing various aspects of UI software design, including user tasks [3,26], cognitive errors [28,29], and general usability principles [4,6,10,25]. But little attention has been given to tools and methods that can support systematic identification and definition of use-related safety requirements for UI software design, even though the availability of such safety requirements has great impact to the applicability and effectiveness of formal methods in assessing UI software.

Current industrial practices rely on hazard analysis techniques to explore safety hazards associated with a system and formulate requirements to mitigate the identified hazards. However, existing hazard analysis techniques do not provide specialized assistance in analyzing UI software design. For example, Preliminary Hazard Analysis [5] and Root Cause Analysis [13] merely rely on group brainstorming to explore use hazards and their causes in safety-critical systems. Other more systematic methods such as Fault Tree Analysis [16], FMEA [30], and their variants focus on component failures or deviations in physical parameters (as in HAZOP [14]), rather than use hazards and their causes in UI software design. Whilst some variants of HAZOP can be used to analyze use errors (e.g., see [7,12]), their focus is mainly on assessing compatibility with given user tasks.

Leveson's System Theoretic Process Analysis (STPA) [17] is a relatively new hazard analysis method that can be utilized to identify use-related hazards in control systems, and system design issues contributing to these hazards. This method, however, offers limited guidance on formulating hypotheses on what aspects of UI software design could affect the usability and safety of a system under analysis. STPA requires developers to exercise their experience and expertise to find answers to the key question: *What UI software design features could induce the user to operate or interact with a system unsafely?*

In this paper, we extend the standard STPA analysis to provide developers with clear guidelines in finding the answer to this key question. This is done by extending standard STPA with a set of casual factor categories, which are tailored to guide developers in systematically examining UI software design against common use-related safety concerns and widely-accepted design principles. We also refine the standard STPA analysis process such that developers can utilize the casual factor categories to identify UI software design issues and define corresponding safety requirements for realistic medical device systems.

Contribution. This paper makes the following contributions: (i) a novel method that augments STPA to enable a more detailed analysis of UI software in medical devices in terms of safety and usability; (ii) a case study on an experimental medical device demonstrating the applicability and benefits of our method; and (iii) formalization in PVS [24] of natural language safety requirements produced in the case study, which results in the definition of a *safety reference model* [20] that encapsulates the semantics of the requirements against which UI software design must be verified.

2 Background on STPA

STPA focuses on early identification of safety hazards in control systems and their causes in system design. It considers the system under analysis as a *control model*, which in its simplest form includes a control loop with an automated controller and a controlled process. The standard STPA process is carried out in three steps (see the left-hand side of Figure 1):

Step 1: Identify system boundaries and unsafe control actions. In this step, developers identify all system elements that could impact, directly or indirectly, system safety. Potential hazards associated with the system are explored in the form of *unsafe control actions* (UCAs) performed by system elements. To identify use-related hazards, a *human operator model* can be introduced to the system's control model to capture hypotheses about the operator's knowledge and understanding of how the system works. STPA guides developers to explore four types of UCAs potentially performed by system elements, including the human operator:

- A control action required for safety is not provided or not followed (e.g., an emergency stop button is not pressed when necessary).
- An unsafe control action is provided that leads to a hazard (e.g., the delivery of a therapy with an excessive dose of radiation).
- A potentially safe control action is provided at the wrong time or in the wrong order (e.g., a button for stopping the delivery of radiation is pressed too late).
- A potentially safe control action is stopped too soon or applied for too long (e.g., a button for decreasing the volume of an alarm is pressed for too long and, as a result, the alarm gets disabled).

Step 2: Identification of causal factors. At this step, developers examine the system's design to identify issues (also called *causal factors*) enabling UCAs. For UCAs from the human operator, STPA recommends to explore three categories of causal factors:

- **Feedback:** feedback provided to the operator is inadequate, missing, or delayed. An example is that the system does not provide an alarm on its UI to inform the operator of abnormal system conditions.
- **Mental Model:** the operator's perception or understanding of how the system works is inconsistent, incomplete, or incorrect. For example, using a non-standard color/symbol to display an alarm on the UI can be easily misunderstood by the operator.
- **External Information:** wrong or missing inputs or external information. This category explores issues related to information communicated to the operator by elements outside the control model. For example, a procedure described in the user manual is wrong, or a prescription used by the operator to setup the therapy contains incorrect information.

Step 3: Define safety requirements and constraints. Developers define testable safety requirements or constraints to address the identified casual factors, and in turn prevent or mitigate UCAs.

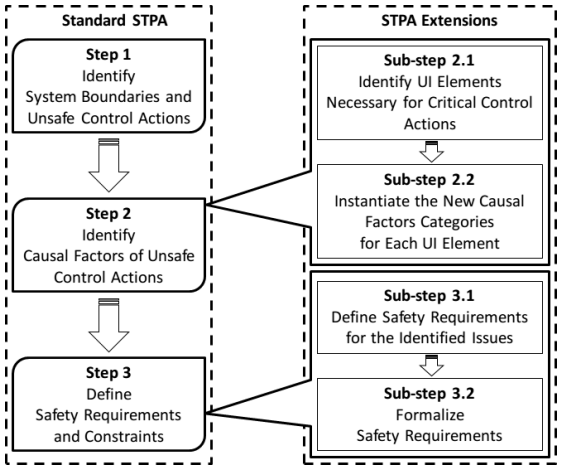

Fig. 1. Enhanced STPA process.

3 Enhanced STPA analysis

We enhanced the standard STPA analysis with the aim of providing developers with additional guidance in systematic identification of UI software issues that likely induce human operators to perform UCAs. The enhancement is realized by refining the standard STPA causal factor categories with 16 new categories that are specialized for identifying UI software issues in medical devices.

3.1 New causal factor categories

The new casual factor categories are derived from usability heuristics [23] and UI design guidelines defined in medical device usability standards (ANSI/AAMI HE75 and ISO 62366-1). For each category, we present a mnemonic name, a definition, and a rationale explaining its relevance to device usability and safety.

Categories refining STPA Feedback

- **F1: Consistency of feedback.** Feedback for control actions or events that are conceptually similar is not provided using the same modalities (e.g., visual, auditory, haptic) or the same UI elements. *Rationale: Inconsistent feedback leads to confusion and incorrect user actions.*
- **F2: Complexity of feedback:** Feedback for frequent actions or important events (e.g., system failure or patient-related emergencies) requires observing and understanding multiple information resources. *Rationale: In clinical settings, operators may become distracted by other tasks or interrupted during device use. Because of this, they could fail to monitor and analyze feedback from multiple information resources.*
- **F3: Availability of feedback:** Feedback reporting important information (e.g., therapy parameters) or events (e.g., change of device modes) is erroneous, not visible, partially visible, or is visible at the wrong time or for a

too short period of time. *Rationale: Operators could lose situation awareness during critical operations if relevant information is not reported on the UI.*
- **F4: Salience of feedback:** Feedback reporting important information or events is not prominent or easy to locate on the UI, or secondary information/events are erroneously more prominent than critical information/events. *Rationale: Feedback on the UI should promote selective attention on information that is important for correct decision-making.*

Categories refining STPA Mental Model

- **M1: Reversibility of control actions:** Functions for reversing the effect of control actions are not available. *Rationale: UI functions such as undo, stop, cancel, resume should be available to the operator to recover from errors.*
- **M2: Responsiveness to control actions:** The effect of critical control actions is not reported to the operator in a timely manner. *Rationale: If feedback for control actions is delayed, users could fail to realize whether the control actions on the UI have been successfully recognized by the system.*
- **M3: Consistency of controls:** The same UI controls produce different effects in conceptually similar situations, or UI controls that are conceptually similar require different interaction styles. *Rationale: Consistent UI controls facilitate the formation of accurate and complete mental models of how to interact with the system.*
- **M4: Complexity of controls:** Frequently used or critical UI controls require unreasonably complex manipulation or unreasonably long sequence of user actions. *Rationale: Users could fail to perform common or critical actions/tasks if too much time or effort is necessary to complete such actions/tasks with the available UI functions.*
- **M5: Predictability of controls:** The UI does not provide means to help the user anticipate the effects or the consequences of control actions. *Rationale: Predictable UI controls facilitate the formation of an accurate and complete mental model of how to interact with the system.*
- **M6: Forgiveness for erroneous control actions:** Safety interlocks are not available to prevent accidental activation of critical controls or block-/mitigate foreseeable use errors. *Rationale: Lack of built-in safeguards for known/common use errors (e.g., manipulation errors due to variability in human performance or lack of attention) could lead to catastrophic failures.*
- **M7: Availability of controls:** Control widgets necessary for safe operation are partially available or not available on the UI, or they are available at the wrong time or for a too short period of time. *Rationale: UI functions should be provided to allow operators to perform necessary control actions.*
- **M8: Affordance of controls:** The visual appearance (shape, label, etc.), relative position, and behavior of UI controls are not consistent with stereotypical knowledge about their function. *Rationale: Design features of UI widgets should promote correct understanding of the action-effect relation of the UI function associated with the widget.*
- **M9: Consistency with clinical workflows:** Workflows supported by the UI software are not consistent with best or actual clinical practice. *Ratio-*

nale: The UI software design should support and enhance existing clinical workflows rather than disrupting them.

Categories refining STPA External Information

- **E1: Availability of user manuals:** Information necessary to understand device feedback or operate the device are not available in the user manual. *Rational: User manuals should support the formation of a complete and correct mental model of UI software functions.*
- **E2: Consistency with user manuals:** Workflows described in the user manual are not consistent with the behavior of the device. *Rationale: The software development process should produce user manuals that are correct with respect to the UI software functions.*
- **E3: Complexity of user manuals:** Critical information in the user manual is mixed with non-critical information. *Rationale: Critical information (e.g., recovery procedures) should be readily accessible in the user manual.*

3.2 Process for using the new categories

Our method refines the standard STPA process to utilize the new casual factor categories in analyzing specific system designs. As illustrated in Figure 1, Step 2 of the standard STPA process is decomposed into two sub-steps: Sub-step 2.1 identifies UI elements (displays, controls, etc.) used to perform critical control actions; and Sub-step 2.2 instantiates the new casual factor categories for each identified UI element to explore issues that could induce such UCAs.

Step 3 of the standard STPA process is also decomposed into two sub-steps. Sub-step 3.1 defines a set of natural language safety requirements to address the identified UI software issues. In their simplest form, safety requirements can be formulated as the logical negation of the corresponding design issue. For example, consider the design issue "Changing the smartphone orientation resets the recommended insulin bolus dosage". This issue would have been detected using category *affordance of controls* — stereotypical knowledge associates smartphone orientation changes with view mode changes (landscape/portrait). A safety requirements can therefore be defined as: *"Changing the smartphone orientation shall not reset/change the recommended insulin bolus dosage"*.

Natural language requirements are formalized in Sub-step 3.2, where the method presented in [20] can be adopted for such formalization. The formalization method in [20] identifies key notions and relations in the textual description, and then provides an interpretation to operationalize these notions and relations for a specific design. This process further clarifies the correspondence between requirements and UI software functionalities, and creates a *safety reference model*. Demonstrating equivalence between a UI software design and the safety reference model constitutes evidence that the UI design complies with given requirements.

4 Case Study: The Gantry-2 System

We have evaluated the applicability and potential benefit of our method on the Gantry-2 system, an experimental radiation therapy device for advanced

cancer treatment. A team of researchers have applied the standard STPA to analyze the UI design of the Gantry-2 system, based on its preliminary design document [2]. We evaluated the UI software design in the Gantry-2 system and compared our results with [2]. To ensure fair comparison, our study was based on the same set of preliminary design documents, and the same control model and system boundaries, as those considered in [2]. Our study results (illustrated below) demonstrate that our method can help to identify not only all design issues reported in [2] but also new critical UI software issues that were not identified using the standard STPA.

Description of the system. The Gantry-2 system is a radiation therapy device for treatment of tumors attached to mobile organs (e.g., lungs). These tumors require continuous scanning of the patient to ensure correct delivery of radiation to the tumor cells. The operator is responsible for setting up the system for the patient, starting the treatment, monitoring the patient and the device state. These operations are carried out using controls and displays on the Gantry-2 consoles. The specific treatment plan for the patient is loaded by the operator before starting the treatment. The treatment planning software uses the treatment plan to configure automated controllers that will manage the delivery of radiation to the patient. Feedback loops necessary to monitor the patient status and the overall treatment delivery process are realized through beamline sensors and cameras installed in the facility. Full details of STPA control models representing the Gantry-2 system can be found in [2].

Analysis results using the new categories. Due to space limit, we only discuss the analysis of the following UCA: *"Treatment start command is activated even when there is no patient to be treated"*. Identifying UI software issues inducing other UCAs can be carried out similarly.

For this UCA, our analysis identified seven critical issues that were not reported in [2]. To help readers better understand the identified issues and in turn the benefits of our method, we present each of these issues as: a description of the issue, UI elements, and causal factors categories used to identify the issue; a *scenario* describing hazardous situations where the issue potentially induces the operator to perform the UCA under analysis; and *safety requirements* for addressing the issue. Note that the decision of whether it is worth implementing a given safety requirement in the final system depends on the level of risk addressed by the requirement, which can be estimated using standard risk matrices.

- **Issue 1**: UI software does not provide feedback when patient is not ready.
  - UI elements: Display.
  - Causal factor: Availability of feedback.
  - Scenario: Operator erroneously starts the treatment because the alert indicating patient not ready is not available on the UI.
  - **Requirement R1.** *"Patient not ready" alerts shall be displayed on the UI.*
- **Issue 2**: UI software displays patient not ready alerts on the main operator console but not on the remote console.
  - UI elements: Display on the main and remote consoles.

  - Causal factors: Consistency of feedback / Availability of feedback.
  - Scenario: Operator erroneously starts the treatment because inconsistent alerts on two consoles cause incorrect understanding of the patient readiness status.
  - **Requirement R2.** *"Patient not ready" alerts shall be displayed at all consoles.*
- **Issue 3**: UI software fails to display patient not ready alerts in certain system modes when it should.
  - UI elements: Display.
  - Causal factor: Availability of feedback.
  - Scenario: Operator erroneously starts the treatment because the alert "Patient not ready" is not displayed in the "Experimental Mode" (e.g., the patient readiness status is always set as "Ready" in this system mode, which incorrectly disables the alert).
  - **Requirement R3.** *"Patient not ready" alerts shall be displayed when the patient is not ready, regardless of the system's operation mode.*
- **Issue 4**: UI software fails to block the operator's accidental press on the start command (e.g., during system maintenance).
  - UI elements: Start command.
  - Causal factor: Forgiveness for erroneous control actions.
  - Scenario: Operator erroneously starts the treatment because the UI software fails to block the operator's accidental press of the start command when the patient is not ready.
  - **Requirement R4.** *UI software shall not accept a start treatment command when the patient readiness status is not ready.*
- **Issue 5**: If the treatment activation sequence is interrupted (e.g., due to power loss), UI software always resumes the sequence from where it was interrupted, and does not provide the operator means to stop/abort/restart the treatment activation sequence.
  - UI elements: UI commands involved in the treatment activation sequence.
  - Causal factor: Reversibility of control actions.
  - Scenario: Operator erroneously starts the treatment because the treatment activation sequence cannot be canceled.
  - **Requirement R5.** *A control to cancel the treatment activation sequence shall be available on all user interface screens.*
- **Issue 6**: UI software fails to provide timely feedback when the operator presses the start command, which causes the user to press the command multiple times and each press is registered as a legitimate command to deliver the treatment.
  - UI elements: Start command; Displays providing feedback to operator.
  - Causal factor: Responsiveness to control actions.
  - Scenario: Operator erroneously starts the treatment multiple times because the system seems to be not responsive to the first start treatment command. Unbeknownst to the operator, the system registers all start commands and delivers the treatment multiple times, even when the patient is not ready.

- **Requirement R6.** *When a control action is performed by the operator, the UI should respond within x seconds to indicate whether the action has been recognized.*

– **Issue 7**: A system feature for automatic detection of patient readiness status is disabled when the system is powered on by the operator, and UI software forces the operator to navigate through multiple menus to enable it.
  - UI elements: Widgets for navigating menus and enabling/disabling detection of patient readiness status.
  - Causal factor: Complexity of controls.
  - Scenario: Operator erroneously starts the treatment when the patient is not ready because they fail to, or choose not to, enable the automatic patient detection feature.
  - **Requirement R7.** *Safety interlocks such as the automatic patient readiness detection shall always be enabled when the system is initialized.*
  - **Requirement R8.** *The status (enabled/disabled) of safety interlocks shall always be visible on the UI.*

5 Formalization of safety requirements in PVS

Safety requirements R1–R8 are testable in the sense that they prescribe specific properties/design features that the UI software needs to satisfy to ensure safe device-user interaction. Here we demonstrate how to formalize these requirements in the higher order specification language of PVS [24]. The formalization assumes a PVS model structured using the following general pattern [11,19,25]:

– The system state is a record of *state attributes*, each characterized by a unique name and a type. A special attribute `mode` identifies the current mode of operation of the system.
– The system behavior is defined in terms of *actions* initiated either by the operator or by an automated process. For example, action `start` models the operator's press on the start command.
– Additional aspects of the system such user manuals, workflows, and context of use are modeled using state attributes and system behaviors that capture facts about information resources (see [21,22]).

PVS syntax. A system is modeled as a *theory* and a set of logic expressions describing the system behavior. Requirements are expressed as *theorems*, which can be proved using the PVS theorem prover. PVS supports basic datatypes commonly seen in programming languages. New datatypes (e.g., record or enum types) can be defined using the keyword `TYPE`. The `IMPORTING` keyword allows to import definitions from other PVS theories. Functions are in the form of `f(x: T1): T2`, where `f` is the function name, `x` is an argument of type `T1`, and `T2` is the function return type. Arrays are defined as `[A -> B]`, where `A` is the array index type, and `B` is the datatype of the array elements.

5.1 Gantry-2 model in PVS

The specification fragment in Listing 1.1 presents a PVS model of the Gantry-2 system. Its characteristics are defined at a level of abstraction compatible with

the safety requirements discussed in Section 4. The model has been proved to satisfy all given safety requirements. Thus, it is also called a *safety reference model*, as it encapsulates the semantics of the safety requirements. A systematic comparison against the functionalities of the reference model can be used as a basis to verify a final UI software implementation [20].

```
1  gantry2: THEORY BEGIN IMPORTING gantry2_types
2    Console: TYPE = { main, remote }
3    Modes: TYPE = { off, on, ready, radiation, post }
4    UI: TYPE = [# viz: [Attr -> bool],
5                  feedback: [Act -> bool],
6                  alerts: [Alert -> bool] #]
7    Controller: TYPE = [# patient: PatientStatus,
8                          interlock: bool, mode: Modes #]
9    State: TYPE = [# console: [ Console -> UI],
10                    controller: Controller #]
11   %-- actions
12   per_stop(st:State): bool = (controller(st)`mode /= off)
13   stop(c:Console)(st:(per_stop)): State =
14     LET st = action_registered(stop,c)(st)
15      IN st WITH [ controller := controller(st)
16             WITH [ mode := ready ]]
17   %-- ...additional definitions of actions omitted
18 END gantry2
```

Listing 1.1. Fragment of the PVS model of the Gantry-2 system.

The PVS model in Listing 1.1 defines a record type `State` (lines 9–10) to track the system's operational status, which includes two state attributes for modeling relevant characteristics of the main and remote consoles and the beamline controller. Lines 4–6 in Listing 1.1 model UI consoles using three state attributes. Specifically, attribute `viz` indicates the visibility of a state attribute on the console. For example, `viz(mode) = true` means that state attribute `mode` is visible on the console. Attribute `feedback` indicates whether feedback is presented on the console for a given command. For example, `feedback(start) = true` indicates that feedback is presented for the start command. Lastly, attribute `alerts` indicates whether an alert is displayed on the console. This abstract representation of UI consoles is sufficient for the verification of safety requirements defined in Section 4. Similarly, the controller of the Gantry-2 system is represented using three state attributes (lines 7–8 in Listing 1.1): `patient`, which indicates the patient readiness status; `interlock`, which indicates whether a safety interlock is enabled to prevent erroneous treatment activation when the patient is not ready; and `mode`, which indicates the current operation mode of the controller.

The following user actions are defined in the PVS model: `start`, for starting a treatment; `stop`, for stopping a treatment; `on`, for powering on the system; and `off`, for shutting down the system. An automatic action `tick` models the advance of time in the system. Further, function `init` captures the initialization of the system. Lines 13-16 in Listing 1.1 demonstrate how user action `stop` is formally specified. Specifications for other actions can be defined similarly, but

are excluded from Listing 1.1 for brevity. The specification of action `stop` takes two parameters: `c` of type `Console` indicates to which UI console `stop` is applied; and `st` indicates the controller's current mode. Note that subtype `(per_stop)` applied to `st` indicates that `stop` can only be applied to system states where the controller's operation mode is not `off` (`(per_stop)` is a shorthand for predicate { `st: State | per_stop(st)` }). The `LET-IN` construct at lines 14–16 first registers the `stop` action (line 14), and then returns a new system state with the controller's operation mode set to `ready`.

5.2 Formalization of the requirements

Formalization of R1–R3. Safety requirements R1–R3 can be conveniently aggregated into a single requirement *"Patient not ready" alerts shall be presented on both the operator console and remote consoles in all operation modes*, which can be formalized as the PVS theorem `patient_alerts_th` below.

```
1 patient_alerts_th: THEOREM
2  FORALL (pre, post: State, c: Console):
3   %-- induction base
4   (init?(pre) IMPLIES pt_status_visible?(on(c)(pre))) AND
5   %-- induction step
6    ((pt_status_visible?(pre) AND trans(pre, post))
7        IMPLIES pt_status_visible?(post))
```

Theorem `patient_alerts_th` checks if every possible system state satisfies that the patient readiness status is actually visible, when it should be, on the display of both UI consoles. Predicate `pt_status_visible`, as elaborated below, holds true if the patient readiness status is visible (i.e., `viz(patient)` is true) on the display of both consoles when the patient is not ready and the system is on.

```
1 pt_status_visible?(st: State): bool =
2  (controller(st)`mode /= off AND
3     controller(st)`patient = NOT_READY) IMPLIES
4       (st`console(main)`viz(patient) AND
5           st`console(remote)`viz(patient))
```

Note that theorem `patient_alerts_th` uses *structural induction* to define all possible system states satisfying a property `p`, i.e., `p` should be satisfied at the system's initial state (*induction base*); then, if `p` holds for system state `pre`, it must hold for any successor state `post` reachable from `pre` as the result of executing system actions, as specified by `trans(pre, post)` (*induction step*).

Formalization of R4. Safety requirement R4 prohibits the start of treatment when the patient is not ready. Theorem `patient_not_ready_th` listed below formalizes this requirement, asserting that user action `start` shall not be permitted (i.e., `per_start(st)` returns false) when the patient status is `NOT_READY`.

```
1 patient_not_ready_th: THEOREM FORALL (st: State):
2   controller(st)`patient = NOT_READY IMPLIES
3     NOT per_start(st)
```

Formalization of R5. This requirement allows the operator to cancel the treatment activation sequence. Its formalization, as listed below as theorem `cancel_activation_th`, asserts that the command is always available when the device is turned on (`controller(st)‘mode /= off`), and the system shall go back to mode ‘ready’ as result of pressing button `stop`.

```
cancel_activation_th: THEOREM
 FORALL (st: State, c: Console):
   controller(st)`mode /= off IMPLIES
     controller(stop(c)(st))`mode = ready
```

Formalization of R6. This requirement mandates timely feedback after the operator performs a control action. Theorem `acknowledge_start_th` listed below formalizes this requirement for user action `start`. It is defined based on attribute `feedback(start)`, which is true when feedback is presented on the UI. The attribute is checked immediately after registering user action `start`. This formalization is sufficient for the demonstrative purposes of this paper. An additional condition is also included to ensure that feedback is not displayed persistently (in this case after one `tick`) on the console after pressing `start`. Formalizing R6 for other user actions can be done similarly.

```
acknowledge_start_th: THEOREM
 FORALL (pre, post: State, c: Console):
  (per_start(pre) AND post = start(c)(pre)) IMPLIES
   (console(post)(c)`feedback(start) AND
     NOT console(tick(post))(c)`feedback(start))
```

Formalization of R7. Theorem `interlocks_active_th` formalizes safety requirement R7, which ensures that `interlock` is true (i.e., safety interlocks are enabled) after the system is powered on (modeled by action `on`).

```
interlocks_active_th: THEOREM FORALL (st: State):
 init?(st) AND per_on(st) IMPLIES
  (FORALL (c: Console): controller(on(c)(st))`interlock)
```

Formalization of R8. Formalization of safety requirement R8 involves defining a predicate `interlock_visible?` that checks whether the status of interlocks (attribute `interlock`) is visible on both UI consoles. Structural induction can then be used to build a theorem, like that used in formalization of R1–R3, to assert that `interlock_visible?` holds true for all reachable system states.

```
interlock_visible?(st: State): bool =
 controller(st)`mode /= off IMPLIES
  (FORALL (c: Console): st`console(c)`viz(interlock))
```

We have proved that the PVS model in Listing 1.1 satisfies all PVS theorems discussed in this section. The proof was carried out using the predefined PVS proof strategy `grind`, which performs automatic instantiation, rewriting, and expansion of definitions. The full PVS theory and proof can be downloaded from `https://goo.gl/7ftTlv`.

6 Related work

STPA extensions dedicated to improving the analysis of use hazards were proposed in [31]. These extensions refine the human controller model in standard STPA using concepts from applied psychology: a detection/interpretation component is introduced to capture the operator's ability of observing and understanding feedback correctly and in a timely manner; and an action identification component is included to model the operator's ability to identify actions suitable to manipulate controls provided on the UI. Whereas this refined model enables developers to identify human cognitive errors that lead to UCAs (e.g., the operator performs an unsafe action because of incorrect interpretation of system feedback), it does not help to explore design issues in the UI software that likely induce such human cognitive errors.

Leveson has introduced an STPA extension designed to validate the assumptions made by developers when defining a control model for the system under analysis [18]. The validation is done using indicators suitable for measuring changes made by developers during the definition of the system's control model. This extension is orthogonal to our method in that it improves the fidelity of control models necessary for the STPA analysis of a system.

Dokas et.a. [8] extended the STPA control model to facilitate the analysis of catastrophic failures. The intuition is that detecting early warning signs of catastrophic failures and studying how they propagate in the control model can help improve the overall safety of the system. This extension is suitable to identify management-level causes of use hazards in complex socio-technical systems, rather than UI software design issues.

Procter and Hatcliff used standard STPA to analyze interoperable medical devices and identify safety requirements for mitigating design issues in these systems [27]. They enriched STPA with additional guidelines to support the identification of UCAs and their casual factors in interoperable medical systems, and created AADL extensions for documenting the analysis results and subsequently defined safety requirements. Their focus is on interoperability issues, rather UI software design issues.

7 Conclusion

We have presented an enhanced STPA analysis method to support i) systematic identification of UI software design issues that could induce use errors with medical devices, and ii) definition of safety requirements to address such design issues. The benefits of our method has been demonstrated in a case study on an experimental medical system. Our method facilitated the identification of subtle UI software design issues that are difficult to detect with the standard STPA. It also helped us to define testable safety requirements that can be readily formalized to address the identified issues. This might potentially improve the applicability of formal methods in evaluating UI software design in safety-critical systems such as medical devices. Future research includes investigating ways to

mechanize the instantiation of the casual factor categories to a specific design, and detailed pragmatic guidelines for formalizing safety requirements identified by our method.

Acknowledgments. Sandy Weininger (FDA), Scott Thiel (Navigant Consulting, Inc.), Michelle Jump (Stryker), Stefania Gnesi (ISTI/CNR) and the CHI+MED team (`www.chi-med.ac.uk`) provided useful feedback and inputs. Paolo Masci's work is supported by the North Portugal Regional Operational Programme (NORTE 2020) under the PORTUGAL 2020 Partnership Agreement, and by the European Regional Development Fund (ERDF) within Project "NORTE-01-0145-FEDER-000016".

References

1. Association for the Advancement of Medical Instrumentation: Infusing patients safely: Priority issues from the AAMI/FDA Infusion Device Summit. AAMI (2010)
2. Blandine, A.: System Theoretic Hazard Analysis applied to the risk review of complex systems. Ph.D. thesis, MIT (2012)
3. Bolton, M.L., Bass, E.J.: A method for the formal verification of human-interactive systems. In: Proc of the Human Factors and Ergonomics Society Annual Meeting. vol. 53(12), pp. 764–768. Sage Publications (2009), http://dx.doi.org/10.1177/154193120905301201
4. Bowen, J., Reeves, S.: A simplified Z semantics for presentation interaction models. In: FM 2014. pp. 148–162. Springer International Publishing (2014), http://dx.doi.org/10.1007/978-3-319-06410-9_11
5. C. Ericson: Hazard analysis techniques for system safety. John Wiley & Sons (2015), https://doi.org/10.1002/0471739421.ch1
6. Campos, J.C., Harrison, M.D.: Interaction engineering using the IVY tool. In: EICS'09. pp. 35–44. ACM (2009), https://doi.org/10.1145/1570433.1570442
7. Chudleigh, M., Clare, J.N.: The benefits of SUSI: Safety analysis of user system interaction. In: SAFECOMP '93. pp. 123–132 (1993), https://doi.org/10.1007/978-1-4471-2061-2_13
8. Dokas, I., Feehan, J., Imran, S.: EWaSAP: An early warning sign identification approach based on a systemic hazard analysis. Safety Science 58, 11–26 (2013), https://doi.org/10.1016/j.ssci.2013.03.013
9. Food and Drug Administration (FDA): Class 2 Device Recall ACCUCHEK Connect Diabetes Management App (2015), https://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfRES/res.cfm?id=134687
10. Harrison, M.D., Campos, J.C., Masci, P.: Reusing models and properties in the analysis of similar interactive devices. Innovations in Systems and Software Engineering 11(2), 95–111 (2015), https://doi.org/10.1007/s11334-013-0201-3
11. Harrison, M.D., Masci, P., Campos, J.C., Curzon, P.: Demonstrating that medical devices satisfy user related safety requirements. In: 4th International Symposium on Foundations of Healthcare Information Engineering and Systems (2014)
12. Hussey, A.: HAZOP Analysis of Formal Models of Safety-Critical Interactive Systems. In: SAFECOMP'00. pp. 371–381. Springer Berlin Heidelberg (2000), http://dx.doi.org/10.1007/3-540-40891-6_32
13. Ishikawa, K., Lu, D.J.: What is total quality control? The Japanese way. Prentice Hall business classics, Prentice-Hall (1985)
14. Kletz, T.A.: Hazop and Hazan: Identifying and Assessing Process Industry Hazards. Disaster Prevention and Management: An International Journal 10(1), 30–31 (2001), http://dx.doi.org/10.1108/dpm.2001.10.1.30.4

15. Leape, L.L., Berwick, D.M.: Five years after to err is human: what have we learned? JAMA 293(19) (2005), https://doi.org/10.1001/jama.293.19.2384
16. Lee, W.S., Grosh, D.L., Tillman, F.A., Lie, C.H.: Fault tree analysis, methods, and applications: A review. IEEE Transactions on Reliability 34(3), 194–203 (1985), https://doi.org/10.1109/TR.1985.5222114
17. Leveson, N.: Engineering a safer world. MIT Press (2011)
18. Leveson, N.: A systems approach to risk management through leading safety indicators. Reliability Engineering & System Safety 136, 17–34 (2015), https://doi.org/10.1016/j.ress.2014.10.008
19. Masci, P., Zhang, Y., Jones, P., Curzon, P., Thimbleby, H.: Formal verification of medical device user interfaces using PVS. In: ETAPS/FASE14. pp. 200–214. Springer Berlin Heidelberg (2014), http://dx.doi.org/10.1007/978-3-642-54804-8_14
20. Masci, P., Ayoub, A., Curzon, P., Harrison, M.D., Lee, I., Thimbleby, H.: Verification of Interactive Software for Medical Devices: PCA Infusion Pumps and FDA Regulation As an Example. In: EICS'13. pp. 81–90. ACM (2013), http://doi.acm.org/10.1145/2494603.2480302
21. Masci, P., Curzon, P., Furniss, D., Blandford, A.: Using PVS to support the analysis of distributed cognition systems. Innovation in Systems and Software Engineering 11(2), 113–130 (2015), http://dx.doi.org/10.1007/s11334-013-0202-2
22. Masci, P., Furniss, D., Curzon, P., Harrison, M.D., Blandford, A.: Supporting field investigators with PVS: a case study in the healthcare domain. In: SERENE 2012. pp. 150–164. Springer (2012), https://doi.org/10.1007/978-3-642-33176-3_11
23. Nielsen, J.: Usability engineering. Morgan Kaufmann (1993), https://doi.org/10.1016/B978-0-08-052029-2.50001-2
24. Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. In: International Conference on Automated Deduction. pp. 748–752. Springer (1992), https://doi.org/10.1007/3-540-55602-8_217
25. P. Masci and R. Rukšėnas and P. Oladimeji and A. Cauchi and A. Gimblett and Y. Li and P. Curzon and H. Thimbleby: The benefits of formalising design guidelines: a case study on the predictability of drug infusion pumps. Innovations in Systems and Software Engineering 11(2), 73–93 (2015), http://dx.doi.org/10.1007/s11334-013-0200-4
26. Paternò, F.: ConcurTaskTrees: an engineered notation for task models. The handbook of task analysis for human-computer interaction pp. 483–503 (2004), https://doi.org/10.1007/978-0-387-35175-9_58
27. Procter, S., Hatcliff, J.: An architecturally-integrated, systems-based hazard analysis for medical applications. In: MEMOCODE 2014. pp. 124–133. IEEE (2014), https://doi.org/10.1109/MEMCOD.2014.6961850
28. Rukšėnas, R., Curzon, P., Back, J., Blandford, A.: Formal modelling of cognitive interpretation. In: DSV-IS 2006. pp. 123–136. Springer (2006), https://doi.org/10.1007/978-3-540-69554-7_10
29. Rushby, J.: Using model checking to help discover mode confusions and other automation surprises. Reliability Engineering & System Safety 75(2), 167–177 (2002), https://doi.org/10.1016/S0951-8320(01)00092-8
30. Stamatis, D.: Failure Mode And Effect Analysis. ASQ Quality Press (2003)
31. Thornberry, C.: Extending the Human-Controller Methodology in Systems-Theoretic Process Analysis (STPA). Ph.D. thesis, MIT (2014)