Verification of interactive software for medical devices: PCA infusion pumps and FDA regulation as an example

Paolo Masci, Paul Curzon
Michael D. Harrison
Queen Mary University of London
United Kingdom
{paolo.masci,pc,mdh}@eecs.qmul.ac.uk

Anaheed Ayoub, Insup Lee
University of Pennsylvania
Philadelphia, PA, USA
{anaheed,lee}@seas.upenn.edu

Harold Thimbleby
Swansea University
United Kingdom
h.thimbleby@swansea.ac.uk

ABSTRACT

Medical device regulators such as the US Food and Drug Administration (FDA) aim to make sure that medical devices are reasonably safe before entering the market. To expedite the approval process and make it more uniform and rigorous, regulators are considering the development of reference models that encapsulate safety requirements against which software incorporated in to medical devices must be verified. Safety, insofar as it relates to interactive systems and its regulation, is generally a neglected topic, particularly in the context of medical systems. An example is presented here that illustrates how the interactive behaviour of a commercial Patient Controlled Analgesia (PCA) infusion pump can be verified against a reference model. Infusion pumps are medical devices used in healthcare to deliver drugs to patients, and PCA pumps are particular infusion pump devices that are often used to provide pain relief to patients on demand. The reference model encapsulates the Generic PCA safety requirements provided by the FDA, and the verification is performed using a refinement approach. The contribution of this work is that it demonstrates a concise and semantically unambiguous approach to representing what a regulator's requirements for a particular interactive device might be, in this case focusing on user-interface requirements. It provides an inspectable and repeatable process for demonstrating that the requirements are satisfied. It has the potential to replace the considerable documentation produced at the moment by a succinct document that can be subjected to careful and systematic analysis.

Author Keywords
Software Engineering Methods and Processes - Formal; Model-Based System Development.

ACM Classification Keywords
D.2.4. [Software/Program Verification];
D.2.2. [Design Tools and Techniques: User Interfaces]

INTRODUCTION

Many interactive systems are safety critical in the sense that user action may have consequences that will compromise safety (i.e., cause damage or harm). An important aspect of the engineering of interactive computer systems is to provide appropriate processes and evidence that systems are designed to satisfy the various requirements that have been established to reduce the risk of products causing such harm. This paper addresses the engineering problem by describing a formal technique that supports proof that user interface related safety requirements are satisfied in a specified interactive systems design. The medical domain is chosen to demonstrate the approach.

In many countries, medical equipment undergoes a degree of scrutiny prior to being placed on the market. This scrutiny is required by regulators to provide confidence that the device is safe and fit for purpose, and (through the statutory role of regulation) to manage potential litigation if defects are later discovered. Currently, device approval is not standardised across nations, and how the approval process should be carried out [18, 19] continues to be a matter for debate. Current approaches to medical device regulation typically combine premarket review and post-market surveillance. As part of the premarket review, manufacturers are required to "demonstrate the safety and effectiveness of a device" prior to introduction to market. Post-market surveillance involves the development of monitoring mechanisms to identify potential safety issues with deployed devices — in some cases these issues only become apparent after the device has been in service for several years with a large user population.

The level of scrutiny imposed by regulators in the premarket review depends on risks in device use. The US Food and Drug Administration (FDA), which is taken as a benchmark by many other countries, e.g., in Japan and China [10], has identified three main risk classes of medical devices. *Class I* medical devices are low risk devices subject only to "general controls" [23] for example relating to misbranding. An example of a Class I device is a syringe. *Class II* medical devices are medium risk devices that require general controls (the same as Class I devices) plus "special controls" such as verification of mandatory performance and safety requirements. External infusion pumps, used as examples in the paper, belong to this class. *Class III* medical devices are high risk devices that support or sustain human life and are of substantial importance in preventing impairment of human health. Examples include implantable devices such as pacemakers. These devices require general controls and "premarket approval". This involves submitting sufficient engineering and clinical evaluation evidence that the device can be safely deployed in its intended context. Most marketed medical devices are classified as Class II devices [24] because medical devices can be considered as Class II by regulators when manufacturers can demonstrate substantial equivalence to already legally marketed devices. A new device is substantially equivalent to a legally marketed device when it has the same intended use and either the same technological characteristics or different technological characteristics that do not raise new questions of safety and effectiveness. The manufacturer can satisfy the regulator by demonstrating that the new device is at least as safe and effective as the already marketed device. This review process is described in the *Premarket Notification document* [25], known as 510(k) clearance.

The FDA's Centre for Devices and Radiological Health (CDRH), which is responsible for medical device review, is fairly small and needs to review a large number of devices each year [29]. Recent figures suggest that over five thousand new devices require 510(k) clearance each year. 510(k) applications must be substantively reviewed within 90 calendar days of the date at which they were filed [27]. Device approval is entirely based on written documents and does not involve any direct evaluation of the product. The typical size of a 510(k) submission is tens of thousands of printed pages.

The FDA is currently reviewing the 510(k) clearance process because a number of unexpected incidents have involved cleared medical devices. There are a growing number of device recalls issued over the last few years for dangerous or defective products that could cause serious health consequences or death [3].

Several incidents have been due to external drug infusion pumps, most commonly a result of (i) "software errors" and (ii) "human factors errors" (e.g., use errors) [26].

To promote a more rigorous analysis of infusion pumps, the FDA is running a pilot project, the *Generic Infusion Pump* (GIP) [2], that aims to demonstrate how systematic and rigorous model-based analysis can be applied to software for infusion pumps. The idea is to specify safety requirements for broad classes of infusion pumps, such as Patient Controlled Analgesia and Insulin pumps. As part of the GIP project, Kim et al [9] have demonstrated how a model-based development approach can be used to implement software for the controller of a Patient Controlled Analgesia (PCA) pump verified against the Generic PCA (GPCA) safety requirements provided by the FDA. This paper, in contrast, focuses on the user interface module of a commercial infusion pump of the PCA family. It takes a different approach with a focus on user interface requirements, by creating a model from the software of an existing product and then verifying this model against the required safety requirements to verify the interactive behaviour of the already implemented software.

The *contributions* of this paper are: (i) a verification approach, based on reverse engineering and model refinement that allows the verification of interactive software incorporated in user interfaces of medical devices (such as infusion pumps) against given safety requirements; (ii) a tool-neutral formalisation of selected GPCA safety requirements; (iii) an example based on a commercial PCA infusion pump, where PVS [14] (a higher-order logic based verification system) is used to verify a reverse engineered version of the software incorporated in the pump user interface against the GPCA safety requirements.

The structure of the paper is as follows. After discussing related research, the verification approach is presented. It combines reverse engineering and model refinement. A formalisation of the GPCA safety requirements provided by the FDA is developed in a form suitable for use with refinement approaches. The PVS verification system is then used to specify and verify the interactive behaviour of a commercial PCA pump against the formalised safety requirements. This leads to a discussion of the utility of the approach within the current regulatory and production cycles of medical devices. Finally conclusions are drawn.

RELATED WORK

The work described in this paper brings together a number of threads of previous research.

The verification of interactive systems against requirements has been performed both informally using techniques such as heuristic evaluation [13] and formally using model checking techniques [7]. Indeed the models that are behind the concrete models described in this paper were originally developed for model checking analyses using MAL and NuSMV. Other work checking properties of interactive behaviour includes the use of PVS to analyse predictability in number entry [11, 12]. The use of finite state machines to describe interactive systems has a very long history originating in the early 1980s. Notable in this history has been the work of Degani [6] and Thimbleby [21].

The original GPCA safety requirements [2] were also formalised in [9]. In that work, the authors verified the controller of the GPCA model provided by the FDA against the GPCA safety requirements. They used an approach based on model transformations and manually translated the safety requirements into properties of the controller of a formalisation of the GPCA model. Here, in contrast, (i) an approach is used based on refinement — this makes it possible to specify safety requirements independently of the model to be verified; and (ii) verification of the user-interface related GPCA safety requirements is carried out on the user interface of a commercial PCA pump, thus demonstrating that formal verification can be used effectively even if software has been already developed and the model (if there was any) is not available.

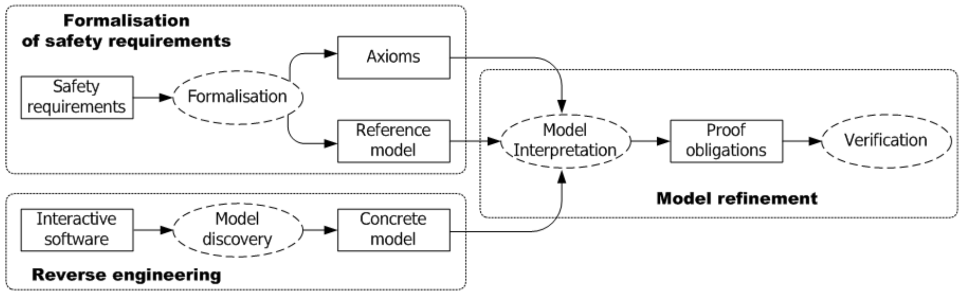

Figure 1. Approach based on reverse engineering and model refinement.

A refinement approach similar to that used in our work has been used in [16] for redeveloping core parts of a commercial air traffic information system. The functional requirements of the system were given as a VDM specification, and the original development produced a single large specification that was hard to comprehend and analyse. Event-B and the Rodin platform were used for creating an abstract specification that captured the core functionalities of the system. This abstract specification was then gradually refined to a distributed model that allowed reasoning about the consistency of the specification. This work shares with ours the use of a verification approach based on refinement for the verification of an already implemented system. However, in their work refinement is used just for reasoning about the consistency of the specification. In our work, in contrast, refinement is used to verify the interactive behaviour of an existing user interface against safety requirements independently provided by regulators.

Bowen and Reeves [5] presented a refinement approach for user interface design. Their work specifically targets the design of user interface layouts. Presentation Interaction Models (PIMs) are used to describe the user interface layout in terms of its component widgets. They use a mixture of formal and informal refinement for turning the initial specification into its implementation. Their work targets the verification of user interface layout. With this perspective, their approach could be used in a complementary way to ours, which is concerned with the verification of user interface behaviour.

THE VERIFICATION APPROACH

The verification approach combines reverse engineering and model refinement (see Figure 1). The verification process starts with two independent branches:

- *Formalisation of safety requirements*: An abstract representation of safety requirements is created that highlights relevant functionalities of the implemented software to be verified. This abstract representation is specified by means of properties (axioms) of a model, hereafter called the *reference model* of the user interface.

- *Reverse engineering*: An accurate model of the behaviour of interactive software incorporated in the device user interface is obtained through systematic exploration of the user interface functionalities and analysis of specification documents (e.g., user manuals). This accurate model will be referred to as the *concrete model* of the user interface.

Model refinement merges these two branches together. Model refinement relates the semantics of two models (the reference model and the concrete model in this case) and shows that the behaviour of the two models is constrained by the refinement relation. A technique called *model interpretation* is used in this work to perform refinement. Model interpretation transforms axioms of the reference model into proof obligations for the concrete model. Discharging these proof obligations makes it possible to demonstrate that the concrete model meets given safety requirements.

Formalisation of safety requirements. Safety requirements are typically provided within a document written in natural language. The approach translates the informal description into a specification by identifying key notions and relationships in the textual description. This identification process is based on heuristics [28], and can be performed either manually, or semi-automatically through natural language processing techniques. Here, a manual identification is performed as it is sufficient to demonstrate the approach. The aim of the formalisation is to create a model (the *reference model*) that will form the basis for the analysis of the implemented software. The reference model encapsulates the semantics of the requirements and can provide guidance when performing verification of the final implemented software, as a systematic comparison against the reference model helps to understand the correspondence between functionalities of the user interface behaviour and the safety requirements.

Reverse engineering. Reverse engineering makes it possible to create a detailed model (the *concrete model*) of a product (interactive software in this case) that can be used for analysis or re-engineering. Techniques that are suitable for reverse engineering a user interface include *model-discovery* [22] and *interaction walkthrough* [20]. Both techniques develop a *parallel system* from the concrete implementation by exploring implemented functionalities systematically. These approaches can be applied either by the development team (in this case compiled or source code would probably be the starting point), or by independent third parties (in this case the final product would probably be the starting point).

Model refinement. Refinement is a formal process that makes it possible to relate the semantics of two models and show that the behaviour of the two models are constrained by the refinement relation. This form of refinement is usually referred to as “structural” or “vertical” refinement, and the refinement relation is usually called the *glueing invari-*

ant. Safety requirements can be verified by this means. The reference model is used as the source model. The concrete model obtained through reverse engineering is used as the target model. The refinement relation connects the reference model to the concrete model. This makes it possible to translate axioms of the reference model (which encapsulate the semantics of given safety requirements) into safety properties for the concrete model. These safety properties of the concrete models are called *proof obligations*, and they are to be verified in order to demonstrate that the safety requirements are met.

The next sections illustrate the approach in more detail. First, a tool-neutral formalisation based on predicate logic is presented of the GPCA safety requirements relating to usability. The reverse engineering and refinement-based verification approach are then demonstrated using PVS [14]. The analysis is performed on the user interface of a commercial PCA infusion pump [4].

FORMALISATION OF THE GPCA REQUIREMENTS

The FDA has released a draft document [2] that includes safety requirements for PCA pumps.

Patients interact with the PCA pump with the help of a single button. This can be used by them to request an additional pre-specified amount of pain relief medication called a "bolus." The pump is pre-programmed by a clinician by specifying the infusion parameters within hard limits, usually preset by a technician, as appropriate for the class of treatment. PCA pumps, particularly epidural pumps, are small and can be carried conveniently by patients outside hospitals without clinical supervision.

The GPCA safety requirements are designed to mitigate identified hazards [2] for the software of PCA pump controllers. However, out of 97 GPCA requirements, we found that almost half of them can be actually related to user interface functionalities. These requirements have been designed by the FDA to be easily translated into a specification in either a programming language or formal language.

The first step in our formalisation process is to extract those terms that specify functionalities of the reference model that are relevant to the semantics of the safety requirement. The second step is to translate the semantics of each safety requirement into a logic formula. This translation step defines relational constraints that must be verified to meet the corresponding safety requirement. This systematic process leads to the creation of a model (the *reference model*) given as an abstract state machine — each extracted term defines a state transition function of the reference model, and each safety requirements is a property (axiom) of the reference model.

The formalisation of selected GPCA safety requirements provided by the FDA is now illustrated. In some cases the formalisation is based on our understanding of their semantics. A further process would be required in which the formalisation is validated with the FDA. In some cases there appears to be overlap or duplication between requirements. This process of formalisation and negotiation is valuable in bringing clarity to established safety requirements. Classical logic operators such as $\wedge$ (conjunction), $\vee$ (disjunction), $\Rightarrow$ (implication) are used to establish logic relations.

R1. *Clearing of the pump settings and resetting of the pump shall require confirmation.* (GPCA safety requirement 2.2.3 about user input [2])

This safety requirement is designed to mitigate some hazards that arise when clinicians or patients change infusion settings inadvertently. The terms that are extracted from the requirement are: **clearing settings**, **resetting pump**, and **require confirmation**. **R1** is ambiguous because the English "and" could mean a logical or, a logical and, or requiring the actions in a sequence. Further enquiries with the FDA confirmed the interpretation "or." The following formalisation reflects the concepts described in the requirement:

$$\boxed{(\mathit{clearing\_settings} \vee \mathit{resetting\_pump}) \Rightarrow \mathit{require\_confirmation}}$$

The formalisation also begs important questions that are not explicit in the requirement. For example, does resetting the pump imply changing the settings? Can settings be cleared if they have already been cleared? Can the pump be used if the settings are cleared? Analysis by the manufacturer during this process would enable them to probe other features of their design, which while not being crucial to safety would allow them to improve the design's usability.

R2. *To avoid accidental tampering of the infusion pump's settings such as flow rate/vtbi*[1]*, at least two steps should be required to change the setting.* (GPCA safety requirement 2.1.1 about user interface resistance to tampering and accidents [2])

This safety requirement is designed to mitigate hazards that result from accidental or intentional tampering with pump settings without authorisation. The formalisation of this requirement introduces the following terms: **changing settings** and **require two steps**:

$$\boxed{\mathit{changing\_settings} \Rightarrow \mathit{require\_two\_steps}}$$

Although there may be a relation between the "two steps" requirement for **R2** and the "confirmation" requirement mentioned in **R1**, this relation is not explicit in the GPCA safety requirements. To leave room for different interpretations, *require_two_steps* is introduced instead of reusing the term *require_confirmation* from requirement **R1**.

R3. *The pump shall issue an alert if paused for more than t minutes.* (GPCA safety requirement 2.2.2 about user input [2])

This safety requirement is designed to mitigate situations where the infusion settings are inadvertently changed. The relevant terms are: **issue alert**, **paused more than t minutes**. The second term embeds two notions: the pump is paused, and the pump state has been the same for at least a given period of time. These terms keep the specification as general as possible, avoiding the necessity to be too specific about the meaning of "more than" — does it mean strictly greater

[1]VTBI stands for "volume to be infused" and is the limit of the total dose the pump will provide.

than (>) or greater than or equal (≥)? During interactive data entry this distinction may make a difference. The philosophy followed here is that the formalisation must not add more constraints in this phase than those described in the safety requirement. Given that the safety requirement is not specific about the meaning of "more than", the same should be true for the formalisation. The natural language description of the requirement need to be modified if "more than t minutes" needs to be more specific. The developed formalisation follows.

$$\boxed{\textit{paused\_more\_than\_t\_minutes} \Rightarrow \textit{issue\_alert}}$$

R4. *If the pump is in a state where user input is required, the pump shall issue periodic alerts/indications every t minutes until the required input is provided.* (GPCA safety requirement 2.2.1 about user input [2])

This safety requirement is designed to mitigate situations in which incomplete infusion parameters have been entered. The relevant terms are: **user input required**, **periodic alerts every t minutes**, **required input provided**. The terms extracted in this requirement can be phrased for reuse: the statement "*until the required input is provided*" can be rephrased into the equivalent statement *"if the required input is provided then the periodic alerts/indications are cancelled."* This rephrasing makes it possible to reuse **required input provided**, and leads to the identification of a new term **alert cancelled**. This reformulation does not change the semantics of the requirement — it is not adding additional constraints, and makes explicit an implicit relation between two concepts described in the same requirement. The requirement can be formalised as follows:

$$\boxed{\begin{array}{l}\textit{(user\_input\_required} \Rightarrow \textit{periodic\_alerts\_every\_t\_minutes)} \wedge \\ \textit{((periodic\_alerts\_every\_t\_minutes} \wedge \textit{required\_input\_provided)} \\ \quad\quad \Rightarrow \textit{alert\_cancelled)}\end{array}}$$

R5. *The flow rate for the pump shall be programmable.* (GPCA safety requirement 1.1.1 about flow rate infusion control [2])

This safety requirement is designed to mitigate hazards that result from incorrectly specified infusion parameters (e.g., flow rate too high or too low). To keep the formalisation general a single notion is introduced, **flow rate programmable** which allows different definitions of "programmable", e.g., a sequence of button clicks (this is the typical solution for the current generation of infusion pumps), wireless communication from the pharmacy, or voice activation controls (these functionalities are envisaged in future generations of infusion pumps). The specific definition will be given when mapping functionalities of a real device to the identified concept. This requirement is thus formalised as follows:

$$\boxed{\textit{flow\_rate\_programmable}}$$

The rest of the paper uses the formalisation of the GPCA safety requirements to verify the interactive behaviour of a commercial PCA pump [4]. The requirements are translated into a PVS [14] specification. The PVS theorem prover is then used to verify that a concrete user interface model maintains a defined refinement relation with the reference model.

VERIFICATION OF A COMMERCIAL PCA PUMP

The logic-based formalisation of the GPCA safety requirements is used to develop a PVS specification of the reference model. The specification is given as a set of properties (axioms) over the model. This development approach guarantees that the reference model meets the formalised requirements.

The PVS specification language builds on classical typed higher-order logic with the usual base types (e.g., `bool`, `nat`, `integer` and `real`), function type constructors `[A -> B]` (predicates are functions with range type `bool`), and abstract data types. The language supports *predicate subtyping*, which is a powerful mechanism to express complex consistency requirements.

PVS specifications are organised into modules called *theories* that describe types, axioms, definitions and theorems. Theories can be parametric in types and constants, and they can use definitions and theorems of other theories by importing them. PVS provides a pre-defined built-in prelude, and a number of loadable libraries of standard definitions and proved facts that can be used when developing new theories.

PVS has an automated theorem prover that can be used to interactively apply powerful inference procedures within a sequent calculus framework. The primitive inferences procedures include, among others, propositional and quantifier rules, induction, simplification using decision procedures for equality and linear arithmetic, data and predicate abstraction.

Formalisation of safety requirements

The extracted terms identified in the previous section define the transition functions (hereafter *function recognisers*) of the reference model. Function recognisers can be specified in PVS using uninterpreted predicates. For example the predicate `require_confirmation?` translates the recogniser *require_confirmation*. By this means GPCA safety requirements can be expressed as axioms of the reference model. As an illustration a complete example for safety requirement **R1** is given here. The specification and verification of the other requirements can be done similarly but are not included for lack of space. The reference model is specified in theory `reference_model_th` given in Listing 1.

Listing 1. The Reference Model

```
reference_model_th: THEORY BEGIN
 ui_state: TYPE
 st,st0,st1: VAR ui_state

 clearing_settings?(st): boolean
 resetting_pump?(st): boolean
 require_confirmation?(st): boolean

 %-- requirement R1
 R1(st): boolean =
  (clearing_settings?(st) OR resetting_pump?(st))
    => require_confirmation?(st)

 R1_Axiom: AXIOM
  (init?(st) => R1(st)) AND
   ((R1(st0) AND R1_trans(st0, st1)) => R1(st1))
END reference_model_th
```

The state of the reference model is defined with an uninterpreted type, `ui_state`. Two logical variables, `st0` and `st1`, identify the current state and the next state of the reference model respectively. A third logical variable, `st`, identifies a generic state of the reference model. When using theory interpretation, these logical variables of the reference model can be mapped to one or more states of the concrete model. Because of this, at this stage it is not necessary to specify how many distinct states are needed to express the requirement.

Safety requirements are specified as `axioms` of the reference model. Given that the safety properties are to be verified in a theorem prover, a sensible choice is to specify axioms in a form that supports structural induction. Structural induction proves a predicate *R* modelling a safety property as follows: (i) *R* must hold for the initial system state (base case); (ii) if *R* holds in a state *st0*, then *R* holds also for any state *st1* resulting from *st0* by applying any transition function (inductive step). This is illustrated in the specification of `R1_axiom` in Listing 1 for safety requirement **R1**. In the PVS specification, `init?` identifies the initial system state; `R1_trans` is the set of transitions; `R1` translates the logic-based formalisation of requirement **R1** into a predicate.

Reverse engineering

A reproduction of the layout of the particular PCA pump [4] considered in this work is shown in Figure 2. A detailed model of the interactive behaviour of a commercial volumetric infusion pump that shares many of the characteristics of this device is described in [7]. The model has been obtained by reverse engineering the pump using interaction walkthrough [20]. The model is specified using a "layered" approach to enable specification reuse. The inner "pump" layer abstracts the behaviour of the controller for devices of the same class. The middle "device" layer is specific to the device being modelled and describes its user interface behaviour. The outer "activity" layer (which is not needed here) describes intended user activities. The original model is given in Modal Action Logic (MAL) [17] and is available in the Minho repository (**`http://hcispecs.di.uminho.pt`**).

This MAL specification has been translated into PVS higher order logic. The use of PVS is motivated by the fact that (i) PVS has a mechanism (*theory interpretations* [15]) for performing model refinement, and (ii) PVS has an expressive specification language which allowed us to create a specification that can be mapped easily to the original MAL model. Other tools that support refinement like Rodin [1] have a less rich specification language, and translating the original MAL specification would have been less straightforward. A fragment of the MAL specification is sufficient to illustrate the verification approach. Relevant excerpts of the developed PVS model that translate those fragments are now illustrated.

The state of the inner "pump" layer is specified in theory `pump_th` with a new PVS record type, `pump` as shown in Listing 2. The type has four fields: `powered_on?` is a Boolean field modelling whether the pump is turned on; `infusing?` is a Boolean field modelling whether an infusion is running; `infusionrate` is a bounded non-negative real number modelling the actual rate pumped by the device;

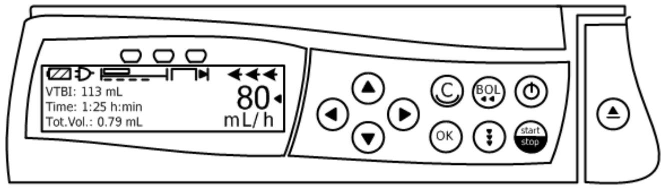

Figure 2. Reproduction of the layout of the commercial PCA pump user interface analysed in this work.

and `time` is a bounded non-negative real number modelling the time left to complete the programmed infusion.

Listing 2. The 'pump' layer

```
pump: TYPE =
[# powered_on? : boolean,
   infusing?   : boolean,
   infusionrate: {x: nonneg_real | x < maxrate},
   time        : {x: nonneg_real | x < maxtime} #]
```

The state of the user interface is specified in theory `concrete_model_th` with a new PVS record type, `concrete_state` given in Listing 3. The type includes the following fields: `device` of type `pump` holds the state of the inner "pump" layer; the enumerated type `displaymode` identifies the current display mode of the user interface; `disprate` models the rate displayed by the user interface and has type `irates`, a record with two fields — `val`, a real number, and `vis`, a Boolean that models whether the rate is visible on the user interface; `entrymode`, an enumerated type that models two generic user interface modes — interactive data entry mode (an interactive mode where the user enters the infusion parameters) or in confirm mode (where the interface requires input from the user); two fields `prevdm` and `prevem` that store the previous display mode and entry mode (this information is used for the functionalities of buttons like `clear` that need to restore the previous display and entry modes); `timeout`, a field holding the timer counter used in various user interface modes; `btnpressed`, an enumerated field that stores information about the button being clicked — this is used to support modelling of key *press & hold* interactions.

Listing 3. The 'device' layer

```
concrete_state: TYPE =
[# device      : pump,
   displaymode: dispmode,
   disprate    : irates,
   entrymode   : emode,
   prevdm      : dispmode,
   prevem      : emode,
   timeout     : nonneg_real,
   btnpressed : btnID #]
```

Actions are specified as transition functions over states of the user interface (`concrete_state`). An example of an action is `on`. This describes the behaviour of the *on* button that toggles the pump on and off. The behaviour of *on* and *clear* is now illustrated as they are relevant to requirement **R1**.

Button *on* can be used to reset the pump settings. When the device is powered off, interactions with the *on* button involve button clicks (a button click turns on the pump). When the device is powered on, pressing and holding down the button for more than 3 seconds turns the pump off. When the *on* button is initially pressed, the user interface enters a confirmation mode and a countdown elapses while the button is continuously held down. A way to model this functionality is to split the specification of transition function `on` into two parts: one for *key pressed* actions (`on_pressed`, as shown in Listing 4), and the other one for *key released* actions (`on_released`, as shown in Listing 5).

A detailed illustration of function `on_pressed` given in Listing 4 is now provided. The argument, `p`, of the function is a device state. The type of `p` is a subtype of `concrete_state` obtained through predicate `per_press_on`. The predicate translates a MAL "permission", that is the condition under which the action can be applied. The use of permissions is useful to enforce legal sequences. In this case, for instance, the permission states that the press action for button *on* can be performed only if another button is not already pressed. This reflects the actual behaviour of the pump user interface, which disables pressing multiple-keys simultaneously. The return type of `on_pressed` is `concrete_state`. The body of the function is thus defined. The button overloads the on and off functionalities. Therefore a distinction is made for the two cases. If the device is powered on (the condition is encoded in the Boolean field `powered_on?` of the device layer `device(p)`) and the user interface is not showing a power off confirmation message (condition `NOT msg_turningoff?` in Listing 4) then the user interface enters a confirmation mode where the power off message is shown. A timer is also started that expires in 3 seconds. The user interface stays in this confirmation mode as long as the *on* button is pressed and held down continuously (condition `msg_turningoff?(..) AND timeout(p)-1>0` in Listing 4). When the timer expires, the pump powers off.

Listing 4. Specification of the *on* button pressed action

```
on_pressed(p: (per_press_on))
 : concrete_state =
IF device(p)`powered_on?
 THEN
   COND
  NOT msg_turningoff?(displaymode(p))
  -> display_poweroff_confirmation(p, 3),
  msg_turningoff?(displaymode(p))
    AND timeout(p) - 1 > 0
  -> decrement_poweroff_timer(p),
  msg_turningoff?(displaymode(p))
   AND timeout(p) - 1 <= 0
  -> power_down(p)
   ENDCOND
 ELSE power_up(p) ENDIF
```

The specification of function `on_released` is given in a similar way. The *on* button can be released only after being pressed (subtype `(per_release_on)` for the argument `p`). The body of the function specifies that the action restores the previous user interface state if the power off countdown was started. Otherwise, the function just resets the action permissions (function `restore_permission`).

Listing 5. Specification of the *on* button released action

```
on_released(p: (per_release_on))
  : concrete_state =
IF device(p)`powered_on?
  AND msg_turningoff?(displaymode(p))
 THEN restore_prevmode(p)
 ELSE restore_permission(p) ENDIF
```

Button *clear* can be used to clear settings. Interactions with the *clear* button involve button clicks while the pump is turned on. The cases given in Listing 6 describe the behaviour of the button: (i) if the user interface is in interactive data-entry mode, a *clear* button click zeroes the value on the display and moves the pump into a confirmation mode; (ii) if the user interface is alarming, a *clear* button click clears the alarm; (iii) if the user interface is displaying the main menu and the device is not infusing, then the *clear* button makes it possible to clear the infusion parameters; (iv) if the device is not infusing then the *clear* button makes it possible to navigate in reverse order the bootstrap sequence of the user interface screens (main screen, main menu, use last therapy, and new patient). When in confirmation mode, if the user interface is left idle for predefined time periods then the user interface enters an alert mode. An excerpt of the PVS specification for click actions of button *clear* (`clear_clicked`) is given in Listing 6. A detailed description is omitted from this paper due to lack of space.

Listing 6. Specification of the *clear* button

```
clear_clicked(p: (per_click_clear))
  : concrete_state =
IF disprate?(displaymode(p))
    AND entrymode(p) = dataentry
 THEN clear_display(p,cm_timeout)
ELSIF mainmenu?(displaymode(p))
       AND entrymode(p) = nullemode THEN COND
  device(p)`infusing? = FALSE
   -> display_last_therapy(p, tm_timeout),
  device(p)`infusing? = TRUE
   -> display_mainscreen(p,0) ENDCOND
ELSIF %.. additional conditions omitted
```

Model refinement

Model refinement can be implemented in PVS through theory interpretations [15]. It is a way to give semantics to uninterpreted terms by mapping them to concrete expressions. In this case the reference model is interpreted using the concrete model. Given that the reference model is an abstract state machine specified in terms of function recognisers, the interpretation consists of mapping functionalities of the concrete model into those function recognisers. With this approach, axioms of the reference model are turned into proof obligations for the concrete model. Discharging these proof obligations makes it possible to show that the behaviour of the concrete model is *consistent* with the behaviour of the reference model. The reference model is verified against the safety requirements by construction — safety requirements are properties (axioms) of the model. Therefore the verification of proof obligations corresponds to demonstrating that the concrete model meets the same safety requirements.

Theory interpretation is now applied to refine function recognisers used for the axiom (`R1_axiom`) that specifies safety requirement **R1**. The syntax for specifying a theory interpretation in PVS is that of a PVS theory importing clause (keyword `IMPORTING` followed by the theory name) with actual parameters (a list of substitutions provided within double curly brackets).

For `R1_axiom`, an interpretation must be provided for the following types and constants (Listing 7): the reference model state (`ui_state`) is interpreted with a pair of concrete user interface states in this case (the current and the next device state); the uninterpreted constant `init?` is interpreted with the predicate `concrete_init?` that identifies the initial state of the concrete model; `R1_trans` is interpreted by enumerating all actions (transition functions) of the user interface:

Listing 7. Theory interpretation for requirement R1 (part 1 of 4)

```
IMPORTING reference_model_th {{
ui_state := [concrete_state, concrete_state],
init? := LAMBDA (st: [concrete_state,
                      concrete_state]):
      concrete_init?(st`1),
R1_trans := LAMBDA (st, st_prime:
                     [concrete_state,
                      concrete_state]):
      (per_click_clear(st`1) AND
        st_prime`1 = clear_clicked(st`1)) OR %...
```

The interpretations of the function recognisers "*clearing settings*," "*resetting pump*" and "*require confirmation*" in terms of the concrete model are as follows.

Clearing settings. The PCA pump under analysis makes it possible to clear settings with the *clear* button in the following situations (Listing 8): (i) when the user is entering infusion parameters (i.e., the outer layer of the concrete model is in `dataentry` mode), irrespective of whether or not the device is infusing; (ii) from the main menu, when the pump is not infusing (i.e., the outer layer of the concrete model is in `mainmenu` mode and the `infusing?` field of the inner layer is *false*). This mapping relation is specified in terms of the theory interpretation parameters:

Listing 8. Theory interpretation for requirement R1 (part 2 of 4)

```
clearing_settings?
 := LAMBDA (st: [concrete_state, concrete_state]):
     (dataentry?(entrymode(st`1))
      OR (mainmenu?(displaymode(st`1))
       AND NOT infusing?(device(st`1))))
        AND per_click_clear(st`1)
         AND st`2 = clear_clicked(st`1),
```

Resetting pump. The pump under analysis can be reset either by clicking the *clear* key or by pressing and holding down the *on* key for more than three seconds (Listing 9). The following situations can be identified: (i) the *clear* key can be used to reset the infusion parameters to their default settings from the main menu when the pump is not infusing; (ii) the *on* key can be clicked and held down in any situation for turning off the pump and hence reset the infusion status and parameters to their default values. This mapping relation can be specified as follows in the theory interpretation parameters:

Listing 9. Theory interpretation for requirement R1 (part 3 of 4)

```
resetting_pump?
 := LAMBDA (st: [concrete_state, concrete_state]):
     (per_click_clear(st`1)
      AND st`2 = clear_clicked(st`1)
       AND mainmenu?(displaymode(st`1))
        AND NOT infusing?(device(st`1)))
      OR (per_press_on(st`1)
           AND st`2 = on_pressed(st`1)),
```

Require confirmation. The PCA pump under analysis has predefined confirmation screens. In the developed model, they correspond to states where field `entrymode` is `confirmmode` shown in Listing 10. The mapping relation that completes the specification of the theory interpretation parameters is therefore the following:

Listing 10. Theory interpretation for requirement R1 (part 4 of 4)

```
require_confirmation?
 := LAMBDA (st: [concrete_state, concrete_state]):
     entrymode(st`2) = confirmmode }}
```

The theory interpretation given in Listing 7, 8, 9, and 10 generates the proof obligation named type check condition (TCC) shown in Listing 11. The TCC must be discharged to show that **R1** is satisfied.

Listing 11. Proof obligation generated from R1

```
% Mapped-axiom TCC generated
IMP_reference_model_th_R1_Axiom_TCC1: OBLIGATION
FORALL (st, st0, st1:
        [concrete_state, concrete_state]):
 (concrete_init?(st`1) => R1(st)) AND
   ((R1(st0) AND
      ((per_click_clear(st0`1)
         AND st1`1 = clear_clicked(st0`1))
        OR %... ))) => R1(st1));
```

This proof obligation is simply the interpretation of axiom `R1_Axiom` and can be proved with the PVS theorem prover. In the following we illustrate how we discharged the above proof obligation in PVS. The verification was almost automatic, and a similar verification approach has also been used to discharge proof obligations generated for other axioms.

Proving the axiom. The proof of the induction base was automatically discharged by PVS in seconds with `grind`, a predefined decision procedure of PVS that repeatedly applies definition expansion, propositional simplification, and type-appropriate decision procedures. The verification of the inductive step was discharged with a small amount of manual intervention. More in details, a direct application of `grind` was initially not successful because expansions and substitutions automatically performed by the strategy were leading to unreachable device states (e.g., device infusion when turned off). This is not a weakness of the theorem prover or a mistake in the specification, but a lack of conditions in the permissions – the subtyping constraints imposed in the permission of the transition functions were not accurate enough. Manual case-splitting allowed us to diagnose the combination of cases leading to the unreachable states. A small number of additional subtyping conditions were thus included in the permission of transition functions to avoid these combination of cases. After the modification, `grind` automatically proved the inductive step.

DISCUSSION

The review process of medical devices is a dialogue between manufacturers and regulators. The verification approach illustrated here aims to make this dialogue precise and inspectable, as manufacturers can specify in a precise way what a requirement means with respect to the behaviour of their device user interface — the "interpretation" phase within our approach. Additionally, manufacturers can also provide a mathematical proof that the behaviour of the device user interface is always compliant to the provided interpretation.

The possibility to provide different interpretations is a flexibility that both regulators and manufacturers are willing to have, because they need to take into account different trade-offs in different systems, and want to leave space for innovation. The validity of the interpretation is negotiated during the review process.

The presented approach relies on reverse engineering. As such, the produced model may contain mistakes, and therefore have a behaviour that is slightly different from the original device. The impact of mistakes in the reverse engineering process can be mitigated through validation of failure traces on the original device – this way false positives can be identified and the model adjusted. When the verification of a safety requirement succeeds, a successful trace should be generated and validated against the original device — this way false negatives can also be identified.

Within production and regulatory cycles, the presented approach can be used in several ways.

(1.) It can be used to generate supporting evidence for human factors claims, for example, as part of a safety case [8]. For example a claim that a clinician will not be able to enter an incorrect infusion rate may be supported in part by providing evidence that the device requires confirmation whenever any number entry is completed.
(2.) It can be used to develop user interface reference models that characterise the key features of broad classes of devices.
(3.) It can be used to identify the source of user interface problems (e.g., during forensic engineering investigations) by checking the device systematically against the reference model and encapsulated requirements.
(4.) It can be used to understand how to *fix* design problems, e.g., when a device is recalled or fixed in the field. Analysing these requirements and the match to the reference model is formative in that failure of a requirement will lead to suggestions about how the design could be improved.

Before the techniques described here can be used as part of a regulatory process, the community, both regulators and manufacturers, must be convinced that the process of learning these skills and applying them is justified. At one level this process can only be achieved through successive generations of the process to demonstrate both the savings in terms of documentation and the clarity and inspectability of the result. Further work also needs to be done to automate these processes, rendering the requirements and reference models as libraries and automating the means of refining the reference models to the concrete models and proving the resulting refined requirements.

CONCLUSIONS

There are a number of obstacles to developing systems from models. The classical model based design process lacks flexibility, failing to address the iterative and experimental approach that is usually the practice of developers, particularly in relation to the interactive system. In practice a process of refinement closer to reverse engineering, as described in this paper, enables a developer to assess the extent to which requirements are satisfied in a design. Reference models, that describe the generic classes of systems, are useful because, typically, there are many products that are designed to satisfy a similar set of requirements and to support a similar set of activities.

This paper describes and illustrates a method of assessing the extent to which a design satisfies requirements. The method is of broad applicability but is clearly relevant to the case of interactive systems where a particular class of requirements must be satisfied. This paper demonstrates that verification approaches based on model refinement are applicable to this situation by using a specific class of medical devices that are subject to regulatory control. Safety requirements, as specified in FDA draft regulatory material, have been translated into a logic-based formal representation using function recognisers. These function recognisers are used as the basis for a reference model generic to a class of systems, in this case PCA pumps. It describes how, using a process akin to interaction walkthrough, a concrete model can be constructed using PVS and that the function recognisers can be linked to this concrete model using a refinement relation. Finally, it has been demonstrated by illustration that these formalised requirements can be re-expressed as theorems and proved of the PVS concrete model.

The contribution of this work is that it demonstrates a concise and semantically unambiguous approach to representing what a regulator's requirements for a particular device might be. It provides an inspectable and repeatable process for demonstrating that the requirements are satisfied. It has the potential to replace the considerable documentation produced at the moment by a succinct document that can be subjected to careful and systematic analysis. The challenge of this work is to facilitate these techniques so that their use as part of the everyday development of interactive software can be clearly justified. Formal approaches raise and address subtle consequences of a design that require a systematic approach; in other words, a formal approach like the one we outline is necessary. Using a formal development process rigorously enough in principle for regulatory use relies on advanced formal skill levels that are unlikely to be familiar to industrial developers. It is therefore a matter of priority to develop good formally-based tools that work effectively in real environments.

Acknowledgements
CHI+MED (EPSRC EP/G059063/1), NSF CNS-1035715, and NSF CNS-1042829.

REFERENCES

1. Event-B and the Rodin Platform. `http://www.event-b.org/`.
2. The Generic Patient Controlled Analgesia Pump Hazard Analysis and Safety Requirements. `http://rtg.cis.upenn.edu/gip.php3`.
3. *Medical Devices and the Public's Health: The FDA 510(k) Clearance Process at 35 Years*. Institute of Medicine, 2011.
4. B-Braun Melsungen AG. Perfusor Space and Accessory: Instruction for Use.
5. Bowen, J., and Reeves, S. Refinement for user interface designs. *Formal Aspects of Computing 21*, 6 (2009).
6. Degani, A. *Taming HAL: Designing Interfaces Beyond 2001*. Palgrave Macmillan, 2004.
7. Harrison, M., Campos, J., and Masci, P. Reusing models and properties in the analysis of similar interactive devices. *Innovations in Systems and Software Engineering* (2013), 1–17.
8. Kelly, T. *Arguing safety – a systematic approach to managing safety cases*. PhD thesis, Department of Computer Science, University of York, 1998.
9. Kim, B., Ayoub, A., Sokolsky, O., Lee, I., Jones, P., Zhang, Y., and Jetley, R. Safety-assured development of the GPCA infusion pump software. In *Proceedings of the ninth ACM international conference on Embedded software*, EMSOFT '11, ACM (New York, NY, USA, 2011), 155–164.
10. Liu, G., Fukuda, T., Lee, C., Chen, V., Zheng, Q., and Kamae, I. Evidence-Based Decision-Making on Medical Technologies in China, Japan, and Singapore. *Value in Health 12, Supplement 3*, 0 (2009), S12–S17.
11. Masci, P., Rukšėnas, R., Oladimeji, P., Cauchi, A., Gimblett, A., Li, Y., Curzon, P., and Thimbleby, H. On formalising interactive number entry on infusion pumps. *ECEASST 45* (2011).
12. Masci, P., Rukšėnas, R., Oladimeji, P., Cauchi, A., Gimblett, A., Li, Y., Curzon, P., and Thimbleby, H. The benefits of formalising design guidelines: a case study on the predictability of drug infusion pumps. *Innovations in Systems and Software Engineering* (2013), 1–21.
13. Nielsen, J., and Molich, R. Heuristic evaluation of user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '90, ACM (New York, NY, USA, 1990), 249–256.
14. Owre, S., Rajan, S., Rushby, J., Shankar, N., and Srivas, M. PVS: Combining Specification, Proof Checking, and Model Checking. In *Computer-Aided Verification, CAV '96*, no. 1102 in Lecture Notes in Computer Science, Springer-Verlag (1996), 411–414.
15. Owre, S., and Shankar, N. Theory Interpretations in PVS. Tech. Rep. SRI-CSL-01-01, Computer Science Laboratory, SRI International, Menlo Park, CA, 2001.
16. Rezazadeh, A., Evans, N., and Butler, M. Redevelopment of an Industrial Case Study Using Event-B and Rodin. In *BCS-FACS Meeting - Formal Methods In Industry* (December 2007).
17. Ryan, M., Fiadeiro, J., and Maibaum, T. Sharing actions and attributes in modal action logic. In *Proceedings of the International Conference on Theoretical Aspects of Computer Software*, TACS '91, Springer-Verlag (London, UK, UK, 1991), 569–593.
18. Schoonmaker, M. The U.S. Approval Process for Medical Devices: Legislative Issues and Comparison with the Drug Model, March 2005. CSR Report for Congress.
19. Sorrel, S. Medical device development: U.S. and EU differences. *Applied Clinical Trials Online* (August 2006).
20. Thimbleby, H. Interaction walkthrough: evaluation of safety critical interactive systems. In *Proceedings of the 13th international conference on Interactive systems: Design, specification, and verification*, DSVIS'06, Springer-Verlag (Berlin, Heidelberg, 2007), 52–66.
21. Thimbleby, H. *Press On: Principles of Interaction Programming*. Mit Press, 2010.
22. Thimbleby, H., and Gimblett, A. User interface model discovery: Towards a generic approach. In *Proceedings ACM SIGCHI Symposium on Engineering Interactive Computing Systems — EICS 2010*, G. Doherty, J. Nichols, and M. D. Harrison, Eds., ACM (2010), 145–154.
23. US Food and Drug Administration. General Controls for Medical Devices, 2009.
24. US Food and Drug Administration. Learn if a Medical Device Has Been Cleared by FDA for Marketing, 2009.
25. US Food and Drug Administration. Premarket Notification (510k), 2009.
26. US Food and Drug Administration. Total Product Life Cycle: Infusion Pump - Premarket Notification [510(k)] Submissions - Draft Guidance, April 2010.
27. US Food and Drug Administration. FDA and Industry Actions on Premarket Approval Applications (PMAs): Effect on FDA Review Clock and Goals, October 2012.
28. Vadera, S. and Meziane, F. From English to formal specifications. *The Computer Journal 37*, 9 (1994).
29. Zuckerman, D., Brown, P., and Nissen, S. Medical device recalls and the FDA approval process. *Archives of Internal Medicine 171*, 11 (2011), 1006–1011.