Verification of User Interface Software: The Example of Use-Related Safety Requirements and Programmable Medical Devices

Michael D. Harrison, Paolo Masci, José Creissac Campos, and Paul Curzon

***Abstract*—One part of demonstrating that a device is acceptably safe, often required by regulatory standards, is to show that it satisfies a set of requirements known to mitigate hazards. This paper is concerned with how to demonstrate that a user interface software design is compliant with *use-related* safety requirements. A methodology is presented based on the use of formal methods technologies to provide guidance to developers about addressing three key verification challenges: 1) how to validate a model, and show that it is a faithful representation of the device; 2) how to formalize requirements given in natural language, and demonstrate the benefits of the formalization process; and 3) how to prove requirements of a model using readily available formal verification tools. A model of a commercial device is used throughout the paper to demonstrate the methodology. A representative set of requirements are considered. They are based on US Food and Drug Administration (FDA) draft documentation for programmable medical devices, and on best practice in user interface design illustrated in relevant international standards. The methodology aims to demonstrate how to achieve the FDA's agenda of using formal methods to support the approval process for medical devices.**

***Index Terms*—Formal verification, human error, medical devices, model checking, modal action logic (MAL), performance, PVS, theorem proving.**

I. Introduction

DESIGN anomalies in user interface software are an important concern in safety-critical application domains, including aviation, power generation, and medicine. In these domains, manufacturers are mandated by regulators to ensure that their system or device has been developed using best practice, and that risks associated with the use of their products are minimal or "as low as reasonably practicable". This must be done before the system or device can be deployed in a safety critical context. Such a demonstration usually includes "proof" that the device satisfies a set of safety requirements designed to mitigate identified hazards (see [1]). The US Food and Drug Administration (FDA) has produced one such set of requirements in draft documentation [2] focusing on programmable medical devices, particularly infusion pumps.

This paper is concerned with how to demonstrate that a software design is compliant with *use-related* requirements. FDA guidelines propose that such a demonstration is "*highly dependent upon comprehensive software testing, inspections, analyses and other verification tasks performed at each stage of the software development cycle*" [3]. The data produced for such analysis is usually substantial and does not *prove* the absence of compliance breaches.

Formal techniques provide additional information. They are concise, precise, and exhaustive, and can be applied before a complete implementation is available. Using formal techniques for the analysis of a system design involves two main steps. The first step is to develop a formal (i.e., mathematical) model of the device that captures relevant characteristics and functionalities of the system. The second step is to use mechanized tools to perform a systematic analysis of the developed model, to check that the behaviors described in the model comply with relevant requirements. If the model correctly represents the real system, then requirements proved of the model apply also to the real system. Formal techniques provide information when the analysis of a requirement fails. This information, typically captured in a *counter-example*, identifies a precise scenario in which design aspects of the device can violate the constraints imposed by the requirements being analyzed. Developers can use counter-examples constructively, to explore the significance of the failure, and decide whether redesign is necessary or other factors or processes can be taken into account to mitigate the identified failure.

Three types of formal techniques are used in this work to analyze the user interface software of an interactive system: model checking, theorem proving, and simulation.

1) *Model checking* is used to validate models and analyze the interface mode behavior of the device against relevant safety requirements. This verification technology focuses

Manuscript received November 1, 2016; revised February 16, 2017 and April 27, 2017; accepted June 10, 2017. Date of publication July 26, 2017; date of current version November 13, 2017. This work was supported by the EPSRC research Grant EP/G059063/1: CHI+MED (Computer–Human Interaction for Medical Devices). The work of P. Masci and J. C. Campos was supported under Project NORTE-01-0145-FEDER-000016, financed by the North Portugal Regional Operational Programme (NORTE 2020), through the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF). This paper was recommended by Associate Editor M. L. Bolton. *(Corresponding author: Michael D. Harrison.)*

M. D. Harrison is with the Newcastle University, Newcastle upon Tyne NE1 7RU, U.K. and also with Queen Mary University of London, London E1 4NS, U.K. (e-mail: michael.harrison@newcastle.ac.uk).

P. Masci and J. C. Campos are with the Departamento de Informática, Universidade do Minho, Gualtar, 4710-057 Braga, Portugal and also with HASLab/INESC TEC, Portugal (e-mail: paolo.masci@inesctec.pt; jose.campos@di.uminho.pt).

P. Curzon is with the Queen Mary University of London, London E1 4NS, U.K. (e-mail: p.curzon@qmul.ac.uk).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/THMS.2017.2717910

on state transition systems. By exploring the state space of the transition system exhaustively, developers can check automatically whether a requirement is true of the model. The ingenuity in model checking is to build abstract models whose complexity can be handled by the analysis tools, and to formulate the requirements appropriately.

2) *Theorem proving* is used for the analysis of detailed models of the device that cannot be analyzed efficiently using model checking. The verification technology is based on natural deduction, and is concerned with resolving logic problems by mechanized application of inference rules. Arbitrarily complex models and requirements can be handled. However, proof of requirements is usually not fully automatic, and guidance from the analyst may be required.
3) *Simulation* is used throughout the analysis process to validate a developed model against the real system. The specific type of simulation technology adopted in the present work involves the use of *executable formal models* and *prototyping*. The detailed formal models guide the behavior of interactive prototypes whose features and characteristics closely resemble those of the real system. These prototypes can be used by domain experts to check whether the formal model mimics the behavior of the real device, and by human factors specialists to discuss use-related requirements with engineers and formal methods experts. Formal methods experts can demonstrate results obtained using model checking or theorem proving in a way that is accessible to the other team members. Finally, engineers can explore alternative design solutions at reduced cost. This process is fundamental in human-centered design processes carried out by a multidisciplinary team of developers [4].

To date, little or no guidance has been produced to demonstrate how to use formal methods technologies for the analysis of user interface software. As a result, formal verification of usability aspects have seen slow take-up in industry. We address this challenge, providing software developers with a demonstration of how formal techniques can be used to gain high assurance that user interface software design is compliant with given requirements, while also considering how software engineers, domain experts, and human factors experts can collaborate to ensure that the manner in which the requirements are satisfied will mitigate use-error.

A. Contribution

We present a methodology that can be used as guidance by developers when applying formal methods technologies to the analysis of user interface software against use-related requirements. The paper describes the steps taken to demonstrate that the requirements are satisfied. This includes demonstrating that

1) the model correctly describes the interactive behavior of the device;
2) requirements given in natural language are correctly translated into formal properties of the model;
3) the model satisfies the formal properties, and
4) the consequence of failure, if there is failure, is considered leading either to redesign of the device, modification of the property, or changes to the overall system to mitigate against the failure.

A case study based on a commercial infusion pump is used to illustrate the methodology.

We progress an agenda suggested by the FDA's use of formal techniques [5]. However, whether the analysis is feasible in the organizations that develop medical devices is an important consideration, and this challenge is part of the paper's discussion.

B. Organization

Section II sets the work described in the paper in context, discussing related work and scoping what is achieved. Section III describes the methodology adopted for the formal analysis of user interface software design. Section IV illustrates the user interface features of the medical device. Section V illustrates the two forms of the model that were developed. Section VI discusses how the models were validated against the actual device. Section VII describes the requirements and indicates how theorems representing the requirements were proved. Finally, Section VIII provides discussion and conclusions.

II. Related Work

Other work shares similar concerns with this paper about how to verify use-related requirements. Bolton *et al.* [6] use models of user tasks as a basis for generating properties for verification, while in [7] a set of patterns that embody usability principles are proposed. Bolton and Bass [8], [9] used SAL [10] to analyze related properties of the Baxter iPump. They translate formal representations of *normative* tasks into properties that can be checked on a model of the pump user interface. They also analyze the normative tasks systematically to identify potentially erroneous sequences. Example tasks include turning the pump ON and OFF, stopping the infusion, and entering a volume to be infused. Their papers include detailed discussion of the challenges faced when modeling and analyzing a realistic system model with a model checker. Mori *et al.* [11] and Fields [12] also represent tasks in a formal language and use a model checker to analyze tasks. Berstel *et al.* [13] use a formal notation to model and analyze WIMP style interfaces.

Bowen and Reeves have produced some initial proposals for design patterns for modeling user interfaces [14], for example the *callback* pattern, representing the behavior of confirmation dialogs used to confirm user operations and the *binary choice* pattern, representing the behavior of input dialogs used to acquire data from the user. These patterns address use-related design issues. Their utility is orthogonal to our approach, as they focus on best modeling practice for user interface software models, rather than verification of given safety requirements.

Most previous research, relating to the formal verification of safety requirements has been devoted to the analysis of the control part of a system, rather than to the human-machine interface. For example, a set of FDA safety requirements was also formalized in [15] using the UPPAAL [16] model checker. Their analysis focuses on design aspects of the controller of the pump rather than user interface design and use-related requirements. Li *et al.* [17] have developed verification patterns that can be used for the analysis of safety interlock mechanisms in interoperable medical devices. Although they use the patterns to analyze use-related properties such as "*When the laser scalpel emits laser, the patient's trachea oxygen level must not exceed a threshold* Θ_{O_2}" the aim is the integration of a model checker in the actual implementation of the safety interlock as a runtime fault prevention mechanism, rather than the analysis of use-related aspects of the safety interlock. This and other similar research activities, e.g., [18]–[20], are not concerned with the analysis of use-related requirements. Proving requirements with similar characteristics to those described in this paper (though not explicitly use-related) has been the focus of a mature set of tools developed by Heitmeyer's team using SCR [21]. Their approach uses a tabular notation to describe requirements which makes the technique relatively acceptable to developers.

Combining simulation with model checking, as discussed in Section III when validating the model, has also been a focus in, for example, [22]–[24]. Recent work concerned with simulations of PVS specifications has been used to support the specific modeling process described in this paper with simulation [25].

While the present paper takes an existing device as its starting point, this must be seen in the context of work that uses formal specifications as part of the design process. Tools such as Event B [26] have been developed with such a goal in mind. With Event B, an initial model is first developed that specifies the device characteristics and incorporates the safety requirements. This model is gradually refined using details about how specific functionalities are implemented. Bowen and Reeves [27] have presented a similar refinement approach for user interface design. Their work specifically targets the design of user interface layouts. Presentation interaction models are used to describe the user interface layout in terms of its component widgets. A mixture of formal and informal refinement is used to transform the initial specification into its implementation.

III. Modeling and Analysis Approach

The process of proving regulatory requirements described in this paper involves a sequence of steps. These are as follows.

1) *Developing a model of the user interface:* This model captures the states of the device's user interface, the available user and internal actions, and how these change the device's state. These models are expressed as state machines. The concrete choice of modeling language depends on the type of formal user interface analysis to be carried out.
2) *Validating the model:* This is done using both model checking and simulation. The validation process involves generating witnesses for properties *p* that should be true of the device. This is done by creating invalid assertions in the form *always not p*, and using the model checker to find counter-examples for these invalid assertions. The generated counter-examples are witnesses for property *p*, in the sense that they identify sequences of actions that satisfy *p*. These sequences can be compared with logs from the actual device, to assess the plausibility of the identified actions and device behaviors. This technique is similar to that used in [28] for generating test cases. Simulation of the model provides an opportunity for analysts who are not formal methods experts to explore the behavior of the user interface interactively. This enables verification that the model reflects the design of the real system, or discussion of the significance of a given requirement. Whether model checking or simulation is used as validation methodology depends on the analyst (e.g., formal methods experts are likely to prefer model checking), and the model (e.g., complex models may exceed the capabilities of model checking tools, therefore, simulation is the only option). Concrete examples are discussed in Section VI.
3) *Formalizing the requirements:* This involves two stages. The first disambiguates the requirements so that they can be translated more easily into a device-specific property. This is described in more detail in [29]. These precise requirements are designed to be implementation independent, and can be used over a range of devices. The second stage involves refining each formalized requirement so that it is specifically about the device under analysis. Both stages are typically interactive and can involve discussion with both human factors specialists, checking the validity of the interpretation of the requirement, and regulator to check that the spirit of the original requirement is correctly captured by the developed property.
4) *Proving the formalized requirements:* The property representing the formalized requirement is proved of the model by model checking or theorem proving. In the example, all the formalized requirements were proved using both technologies except those that involve full number entry, which could only be proved efficiently using theorem proving.
5) *Iterating the process:* The steps of the analysis process may be iterated as the model is successively refined or may trigger the generation of arguments as to why the design, as modeled, satisfies or fails to satisfy the requirements.

Whilst the described process assumes a model-based view of development, the same methodology can also be applied retrospectively to existing devices. For already developed devices, the process of building the model involves reverse engineering the device implementation. A white box view of the system may be taken by reverse engineering the code, or a black box view of the system can be developed by modeling the system based on user manuals and experience of the device itself. In this paper, this retrospective application of the methodology has been used to perform the analysis of the example medical device.

IV. Medical Example

Intravenous infusion pumps are used in many contexts in hospitals, for example intensive care and oncology. They are designed to infuse prescribed doses of medication intravenously over specified periods of time. Use error is a particular concern in such devices. Nurses program them, often under pressure, from paper prescriptions provided by doctors or pharmacies. The format of prescriptions can vary in terms of presentation, legibility, units used, and whether the prescription focuses on

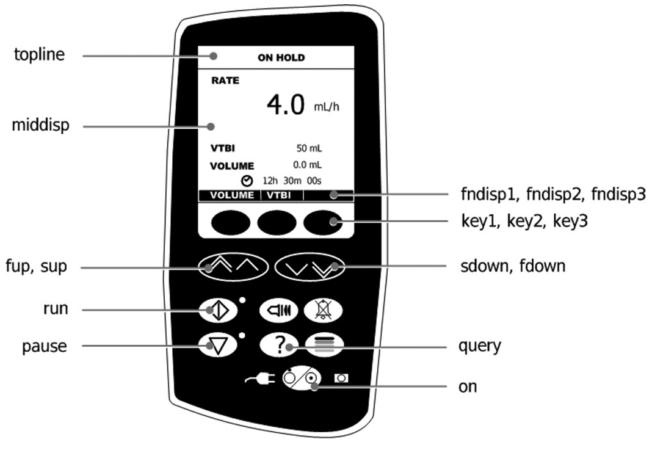


Fig. 1. Actions and attributes used to model the pump. Actions define functionalities provided by the device to enable user interaction (e.g., the *fup* key) and to handle device internal events (e.g., a function for handling alarms). Attributes define the characteristics of display elements used to present feedback to the user (e.g., the top-line display), as well as internal state variables necessary to represent the device behavior.

volume and rate, or volume and time—two alternative ways of describing a prescription. The chosen device (see Fig. 1) is an existing commercial product [30] that has characteristics that are common to many devices that control processes over time. The clinician user sets infusion pump parameters and monitors the infusion process using the device. The values and settings can be changed using a combination of function and chevron keys (see Fig. 1). Chevron keys are used to increase (using specified actions *fup* or *sup*), or decrease (using *fdown* or *sdown*), entered numbers incrementally. Holding the chevron key down accelerates the size of the increment or decrement.

Because the device is small it is reliant on modes to make effective use of the screen and available action keys. The mode structure first distinguishes whether the device is infusing (i.e., pumping medication into the patient) or holding (i.e., paused). Additional modes are offered by the device for changing therapy settings and pump configuration options. For example, data entry modes govern whether the chevron keys change infusion rate, vtbi, or time, or alternatively allow the user to move between options in a menu. Menus are available to select predefined settings in bag mode and in query mode. Bag mode allows the user to select from a set of infusion bag options, thereby setting vtbi to a predetermined value. Query mode, invoked by the query button, generates a menu of options configured by the manufacturer. Options include locking the infusion rate, disabling the locking of it, setting vtbi and time rather than vtbi and infusion rate, and changing the units of volume and infusion rate. The device allows movement between display modes via three function keys (*key1*, *key2*, and *key3*). Each function key has a display associated with it indicating its present function (*fndisp1*, *fndisp2*, and *fndisp3*).

In the analysis of the requirements that follows, further details of the device and modes will be provided as necessary, particularly when the analysis of a requirement leads to interesting consequences.

V. Developing a Model of the Device

Neither a process of developing and refining models satisfying requirements nor producing models from program code was feasible in the present case. The device had already been developed, and the program code was not available to us. The model was therefore developed *by hand* using a combination of user manuals, simulations and the device itself.

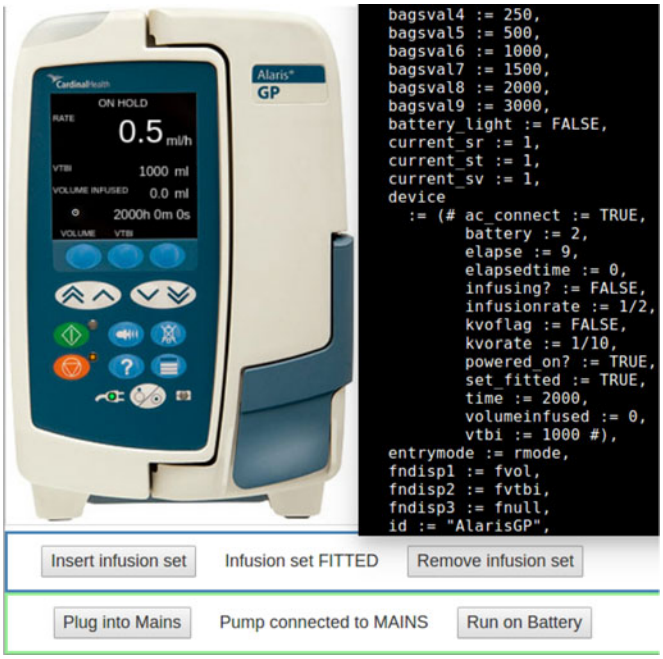
At the stage of the analysis described in this paper, a model of the particular pump had already been developed as part of a general analysis of usability properties of the device [31], and without the particular FDA requirements considered in the present paper in mind. It had been used to analyze properties of the interactive modes, such as whether device modes were presented by the device without ambiguity. This initial model was analyzed using the NuSMV model checking tool [32].

This model was further extended to facilitate analysis of the number entry system of the device as required to prove various FDA requirements. One effect of this extension was that the size of the model increased significantly, making the use of the NuSMV model checker infeasible. We, therefore, changed verification technology when analyzing the extended model. Our option was theorem proving, which is less automated than model checking, but can handle the analysis of more complex models (compare [8]). The specific theorem prover is PVS [33], which offers an expressive specification language that made it easy to translate and extend the initial model. NuSMV uses symbolic model checking. Alternative model checking technologies exist (e.g., bounded model checking), also available within the NuSMV toolset, that might have been able to analyze efficiently the extended model for the considered properties, possibly at the expense of completeness. Another key reason for the adoption of PVS was its ability to generate the prototypes that enabled validation of the model and broader discussion of the requirements.

A. Initial Model of the Device

The model of the device has two main elements: a generic “pump” component modeling the pumping mechanism controlled by the device and an “interface” component that is specific to the particular user interface of the device. The pump component has been reused in other models. For example, an infusion pump developed by another manufacturer has also been studied in detail in [31] which uses this component. The initial model focused on features of the device that involved interaction with the user. This model used the underlying pump process but focused on the effect of actions insofar as they change the basic modes of the device (infusing, paused, off), the interaction modes of the device and the information that is displayed.

To ease the modeling process, a first order notation oriented around actions was used to describe user actions that were provided by the infusion pump. The notation that was used [modal action logic (MAL)], and its mapping to SMV and analysis using NuSMV is supported by the IVY tool [7], [34]. MAL is a simple state transition language, easily translated from state transition diagrams or the SCR tabular format [21]. The notation is used because it is of a type that is more readily acceptable



```
bagsval4 := 250,
bagsval5 := 500,
bagsval6 := 1000,
bagsval7 := 1500,
bagsval8 := 2000,
bagsval9 := 3000,
battery_light := FALSE,
current_sr := 1,
current_st := 1,
current_sv := 1,
device
  := (# ac_connect := TRUE,
        battery := 2,
        elapse := 9,
        elapsedtime := 0,
        infusing? := FALSE,
        infusionrate := 1/2,
        kvoflag := FALSE,
        kvorate := 1/10,
        powered_on? := TRUE,
        set_fitted := TRUE,
        time := 2000,
        volumeinfused := 0,
        vtbi := 1000 #),
entrymode := rmode,
fndisp1 := fvol,
fndisp2 := fvtbi,
fndisp3 := fnull,
id := "AlarisGP",
```

Fig. 2. Simulating the PVS model of the pump.

by developers, making the actions and state transitions explicit in their description. The properties that translate the requirements are expressed in Temporal Logic. Both CTL and LTL logics (see [35]) are supported by IVY and NuSMV. LTL model checking is implemented by NuSMV in terms of CTL model checking, and is typically less efficient [36]. In practice, CTL is used unless the property can only be expressed in LTL. To facilitate tractability using model checking, *token values* were used for the pump variables vtbi, infusion rate, time, and volume infused. They were assumed to be integers in the range $[0 \ldots 7]$. These simplifications were considered to be valid when limiting the analysis to the modal behavior of the device.

The following MAL modal axiom describes the conditions in which *key1* (see Fig. 1) has the effect of confirming a device reset (Fig. 2 shows the pump display at this step).

$$
\begin{aligned}
&\mathit{topline} = \mathit{clearsetup} \rightarrow [\text{key1}]\\
&\qquad \mathit{topline'}{=}\mathit{holding}\ \ \&\ \ \mathit{middisp[drate]'}\ \ \&\\
&\qquad \mathit{middisp[dvtbi]'}\ \ \&\ \ \mathit{!middisp[dtime]'}\ \ \&\\
&\qquad \mathit{middisp[dvol]'}\ \ \&\ \ \mathit{!middisp[dbags]'}\ \ \&\\
&\qquad \mathit{!middisp[dquery]'}\ \ \&\ \ \mathit{!middisp[dkvorate]'}\ \ \&\\
&\qquad \mathit{fndisp1'}{=}\mathit{fvol}\ \ \&\ \ \mathit{fndisp2'}{=}\mathit{fvtbi}\ \ \&\\
&\qquad \mathit{fndisp3'}{=}\mathit{fnull}\ \ \&\ \ \mathit{entrymode'}{=}\mathit{rmode}\ \ \&\\
&\qquad \mathit{effect(device.reset)}\ \ \&\ \ \mathit{keep(bagscursor,\ rlock)}
\end{aligned}
$$

This axiom describes (after *[key1]*) the effect of action *key1*. The expression to the left of the action, namely ($\mathit{topline} = \mathit{clearsetup}$), states the condition under which the behavior described for the action is enabled. This specifies that when the top line of the display shows "clear setup," and the action is invoked, then the expression after *[key1]* describes the behavior. The rule describes changes to visible attributes *middisp*, *topline*, *fndisp1*, *fndisp2*, and *fndisp3*. The priming of an attribute (*topline*$'$, for example) indicates that the action changes the value of that attribute. The action *key1* also changes the mode of the device (*entrymode*) to allow entry of infusion rate (*rmode*). Finally, the rule describes how the action further invokes an action in the pump component (*device.reset*) that initializes all the pump variables. The action *reset* is accessed in the reusable pump component by using the pump$'$s identifier *device* in the MAL specification. The *keep(...)* expression specifies which attributes are not affected by the action and remain unchanged.

This example shows how the MAL model focuses on interface features and the modes of the device, describing concretely how actions change the display and modes of the device. It has a simple discrete model of time. An action *tick* increments time as the infusion process continues, or while the device is paused. In the latter case the value of time is used to determine how long the pause has been. This model, even without full number entry, requires substantial processing for analysis in NuSMV, around 90 min for the subset of properties that are tractable. The analysis was performed on an Apple MacBook Pro 2.9 GHz Intel core i5 with 8 GB of RAM. A detailed discussion of the challenges related to model checking realistic user interface models can be found in [8].

B. Detailed Model of the Device

The second model of the device, which includes additional details needed for the analysis of various FDA requirements, was developed by translating the MAL systematically into the PVS [33] theorem proving system, and then extending it with details related to the data entry system. PVS allows the analysis, in principle, of models and properties involving infinitely many states. The equivalent specification for the MAL fragment described in the previous section is

```
key1_case_clearsetup(st: (per_key1)):
  state = st WITH [ topline := holding,
                middisp := LAMBDA
                      (x: imid_type):
                      (x = drate) OR
                      (x = dvtbi) OR
                      (x = dvol),
                device := reset(device(st)),
                fndisp1 := fvol,
                fndisp2 := fvtbi,
                fndisp3 := fnull,
                entrymode := rmode ]
```

The PVS theory captures all the characteristics of the MAL model, including time, but also includes a full number entry model and other specific details. The PVS features that correspond to MAL elements can be seen in the specification. This function `key1_case_clearsetup` is invoked in the more general `key1` function when the condition `topline(st) = clearsetup` is true. The function has domain `(per_key1)`, where `per_key1` is a predicate that restricts the function domain to the set of states for which the action is accessible to the user. This information is used by PVS to ensure well-formedness of the model.

VI. Validating the Model Against the Real Device

Fidelity of the model to the implemented device was first demonstrated using the NuSMV [32] model checker by proving a range of properties. This validation of the model inevitably included state abstractions required to reduce the state space for the model checking analysis. The sequences generated as witnesses provided structural details such as a sequence of mode transitions, rather than the details of entry of a particular value for the infusion rate.

NuSMV accepts a finite state model (translated from the MAL model illustrated above) and analyzes it exhaustively to prove or disprove a property. An example of such a property is that, once relevant pump variables had been entered, infusion would lead to a state in which the volume infused was equal to the entered vtbi:

$$AG(device.infusionrate = 1 \ \& \ device.vtbi = 7 \\ \Rightarrow AG(device.volumeinfused \mathrel{!=} 7))$$

The property is expressed as a negation. It asserts that it is always the case, for all paths, that if infusion rate is set to 1 (a token value) and vtbi is set to 7, then a state cannot be reached in which volume infused is 7. This particular property is generic in the sense that it would be a desirable characteristic of any programmable infusion pump. It does not depend on the details of the device user interface, depending only on the generic pump model, but produces results that enable an analysis of the interface, making possible a comparison between alternative interfaces. As expected, the property fails when checked and produces a trace of steps (a *witness*) in which the infusion rate is set to 1 and vtbi is set to 7. It indicates that once this has happened, eventually the device is set to infuse, and then after more steps a state is reached where the volume that has been infused becomes 7. The trace can be compared with the actual device logs.

This model was used to validate a set of plausibility properties before translating it into the fuller PVS model. For each action specified in the first model, a function was described that transformed PVS descriptions of states, and for each permission describing when the action was enabled, a PVS predicate was produced. Texture was then added to provide a more detailed description of the user interface behavior, in particular related to the data entry system of the pump. A set of informal rules were used to achieve this translation. The correctness preserving properties of the translation were not however checked formally. A prototype was also produced automatically from the PVS model to obtain an interactive simulation with the "look and feel" of the actual device (see Fig. 2). The traces and simulations were indistinguishable from the behavior of the physical device. The only difference between the logs and traces obtained in the simulation and those of the real device was that the precise timings differed. The simulations were generated with the aim that they could be explored by regulator, manufacturer or for training purposes by the user. These simulations only allow, of course, an exploration of the paths that the user (for example, regulator) chooses to explore. Simulation can also be used to illustrate what the failure of a property means. Part of the argument that a failure is acceptable may then involve a demonstration of the features of the device that fail the requirement, showing that they do not present a risk. Validation of the PVS model was checked in this case not by proving the equivalence of the PVS and MAL models, though this is feasible, but by simulating the PVS model using PVSio-web [37]. Both MAL sequences and PVSio-web simulations were explored by hand with the help of expert users and the user manual of the real device.

The remainder of the present paper focuses on the PVS model and the theorems that were generated to prove use-related requirements. Relevant snippets of the developed models and theorems will be presented. The full MAL model with CTL properties, and the full PVS theory and theorems with proofs can be found in the specification repository.[1] All proofs can be reinvoked using the IVY and PVS tools.

VII. Formalizing and Proving Requirements

In this section, we demonstrate how the process of formalizing and proving use-related safety requirements can encourage a constructive dialog between the analyst and the other experts involved in the development of a user interface.

The developed device model was used to analyze two small but representative sets of requirements. The FDA requirements described in [5] are considered first. They are designed to mitigate use hazards that could lead to misprogramming of the infusion pump and, consequently, delayed therapy or even patient harm. A further set of requirements is then considered based on the *property templates* described in [38] and [39]. These additional requirements capture best practice in user interface design. Note that, whilst the process is demonstrated for a specific device, the applicability of the methodology is general, and can help design better user interfaces.

A. Formalization Process

To formalize the requirements given in natural language, it is necessary to consider their *precise* interpretations (see [29]). A way to develop a precise interpretation is to translate the requirements into logic properties. These properties are expressed using the PVS language, which combines a functional notation, similar to that used in programming languages, with logic connectives such as `AND` (conjunction), `OR` (disjunction), `IMPLIES` (implication). Precision is achieved by defining abstractions that can be more readily understood by different stakeholders. Also, this process makes it easier to construct properties that match the requirements as PVS theorems. The formalization must capture the essence of the requirements as understood both by the regulator, who developed them in the first place, and the human factors specialists, who can comment on the user aspects of the requirements and whether they are fulfilled by the specific properties of the device.

[1] http://hcispecs.di.uminho.pt/m/2

B. FDA Requirements

The FDA requirements are specific to the safety of infusion pumps. For each requirement, the following information is presented: the original formulation described in FDA documentation, the safety concerns addressed by the requirement, a formalization of the requirement, and a proof of compliance with the requirement for the example device.

R1: Clearing the pump settings and resetting of the pump shall require confirmation.

Safety concerns: This requirement is designed to check whether there is a barrier (in this case confirmation) to reduce the risk that infusion settings will be cleared or will revert to predefined settings inadvertently.

Formalization: To formalize this requirement, a logic property needs to be constructed that requires that relevant pump variables are not cleared until a `confirm_action` has occurred. The property needs to specify also that no other action (`no_confirm`) will change the specified pump variables. For the sake of clarity, the formalization is described for each pump variable separately. The requirement, for pump variable vtbi, can be expressed as follows:

```
ready_to_clear(vtbi)(st) IMPLIES
  (clear_setting(vtbi)(st) = x AND
    confirm_action(clear_setting(vtbi)
     (st)) = 0
     AND no_confirm(clear_setting(vtbi)
     (st)) = x)
```

In the above formula, `ready_to_clear(vtbi)` is a predicate that identifies the states in which the device is ready to clear the vtbi value. The action `clear_setting(vtbi)` specifies the first step, prior to possible confirmation, in which the settings are cleared. This action does not clear the setting itself but must precede the confirmation action.

Interpretation for the specific pump: The process of further refining the terms in the abstract property is valuable in reaching agreement about how the requirement applies to the specific pump. It leads to consideration as to what, for example, should be the state of the device when it is "ready to clear" and how the state of the device should be communicated to the user. Further, it leads to consideration of what the confirmation should be and how it should be signaled to the user. The actions captured by `no_confirm` are also a matter of concern, as they define what actions should be permitted to abandon the clearing of the setting. For example, in the example device, vtbi can be cleared when the device is turned on and is in the holding state. The action of clearing the pump variable is only possible if its value is nonzero. Relevant state attributes for expressing whether the pump is infusing and turned on are `infusing?` and `powered_on`, respectively. Predicate `ready_to_clear` can therefore be expressed as

```
ready_to_clear(vtbi)(st) =
    NOT device(st) `infusing? AND
     device(st)`powered_on? AND
      device(st)`vtbi = x AND x /= 0
```

where `x` is a parameter representing a generic, but given, vtbi value. Normally, a specification of `ready_to_clear` is necessary that includes visual attributes of the user interface. For example, in this case the device should be ready to clear unless the top line of the display indicates that it is infusing. It was felt unnecessary however to include this in the formulation—other requirements were considered in the analysis that demonstrate the visibility within the user interface of operating modes. The required clearing of the pump variables is achieved by switching the pump off and then switching it back on again. When the settings are cleared in this way, as the device is switched back on, a request is made for confirmation. This confirmation can be given by the user through action `key1`. The request for confirmation is indicated by a top line display of "clear setup."

Proving the requirement: The requirement can be proved in PVS with no human intervention by using the predefined proof strategy `grind` (which performs quantifier elimination, expansion of definitions, and propositional simplification) to prove each subgoal.

R2: The pump shall issue an alert if paused for more than t minutes.

Safety concerns: This requirement aims to ensure that the user is alerted if the device is left unattended.

Formalization: The requirement can be formalized using a predicate `user_input_strictly_overdue`, which indicates whether the device has been paused without activity for a specified period, and a predicate `alert`, which describes an appropriate alert produced by the device:

```
user_input_strictly_overdue(st)
    IMPLIES alert(st)
```

Interpretation for the specific pump: The generic formulation can be refined to capture specific scenarios relevant for the device, for example if the device is left unattended when paused. In this case, predicate `user_input_strictly_overdue` can be expressed in more detail as

```
user_input_strictly_overdue(st) =
    paused(st) AND elapsed(st) > timeout
```

where `paused` and `elapsed` will have specific meanings for the particular infusion pump. In this case, the pump is paused when the device is powered on and not infusing:

```
pause(st) = device(st)`powered_on? AND
  NOT device(st)`infusing?
```

Attribute `elapse` specifies the time since the device was last used when in holding mode. This attribute is incremented each `tick` action (which simulates the evolution of time in the developed model) when the device is paused.

Proving the requirement: This requirement can be proved for all reachable states through structural induction. That is, the PVS theorem contains the following two parts: the first part proves that the formalized requirement (denoted as `R2assertion` in the theorem) is true of the initial device state and the second part proves that, given a generic state `pre` for which the requirement is true, it is always the case that the requirement is also true for any state `post` reached from `pre` by any available action:

```
R2: THEOREM
   FORALL (pre, post: state):
     (init?(pre) IMPLIES R2assertion(pre))
       AND
       (R2assertion(pre) AND
         state_transitions(pre, post)
         IMPLIES
           R2assertion(post))
```

In the theorem, predicate `state_transitions` is used as a means to relate states `st1` to `st2` if `st2` can be reached by any available action from `st1`. This requirement can be proved in PVS by splitting the theorem into subgoals based on the available actions, and then using `grind` to complete the proof of each subgoal.

R3: 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.

Safety concerns: This requirement aims to mitigate situations where the clinician has entered incomplete infusion parameters. This might occur, for example, if the clinician is interrupted while programming the device.

Formalization: The formalized requirement includes the following elements: `alert`, a predicate describing an appropriate alert produced by the device; `ready`, a predicate identifying the device state before the alert was issued; and `user_confirm`, an action that clears the alert:

```
user_input_strictly_overdue(st) IMPLIES
   (alert(st) AND ready(user_confirm(st)))
```

Interpretation for the specific pump: Providing an interpretation for these abstract terms challenges the designer and the human factors specialist to consider the features of the design that need to be considered in the requirement. For example, `user_input_strictly_overdue` should be true when the device has been paused without activity for a specified period (compare requirement R2). The formalization also challenges the analysis team to consider how this device state is communicated to the user, and which key is used as confirmation key. In the specific pump, alert is indicated by an appropriate top line display (`attention`) which is characterized also by an audible alarm. Action `user_confirm` is defined as `key3`. Note that the confirm does nothing if `key3` is not enabled. Finally, predicate `ready` needs to be defined. It checks that the device returns to a *pause* state and the elapsed time is set to a value less than the time out.

Proving the requirement: This requirement can be proved in PVS with minimal human intervention, using a structural induction similar to the previous example.

R4: The flow rate for the pump shall be programmable.

Safety concerns: This requirement aims to ensure that the clinician can program the pump with the flow rate values indicated in the prescription provided by the pharmacy.

Formalization: This requirement is more challenging to make precise. It requires a description of what is meant by "programmable" in this context. The requirement is designed to ensure that any flow rate value indicated in a prescription can be programmed in the pump. A reasonable interpretation is that there is always an available action (when in a relevant mode) that will change the flow rate programmed in the pump so that it is closer to the target rate. The requirement can be reformulated therefore as follows: "*If the device is ready to enter the flow rate, then there is always an action that will take the flow rate closer to the expected rate, and eventually the intended rate will be reached*." This reformulation can be translated into the following logic property:

```
entry_ready(rate)(st) AND
  (rate(st) > e IMPLIES
    rate(st) - e >= rate(a1(st)) - e) AND
    (rate(st) < e IMPLIES
      e - rate(st) >= e - rate(a2(st)))
```

where `e` is a target value for the flow rate, `a1` is an action that reduces the rate, and `a2` is an action that increases the rate.

Interpretation for the specific pump: This requirement, as expressed, raises questions about the nature of the actions `a1` and `a2`, how the target rate is made visible to the user, and how the device indicates the progress that is being made to reaching it. For the specific pump, the device is ready to accept a rate value (`rate_entry_ready`) when: the device is switched on, infusion rate is not locked, and the top line display shows "holding" or "infusing." The two actions that provide the expected programmability are single chevron up (*sup*) and single chevron down (*sdown*). Note that the requirement is not concerned with how efficient the programmability is. Rather it aims to check that there exists a sequence of actions for entering a given value. A separate requirement needs to be defined to address efficiency of programming.

Proving requirement R4: The PVS prover completes the proof of this theorem unaided, using `grind`.

R5: To avoid accidental tampering of the infusion pump's settings such as flow rate/vtbi, at least two steps should be required to change the setting.

Safety concerns: This requirement is designed to mitigate hazards resulting from accidental tampering of pump settings, as a result for example of a single erroneous button click.

Formalization: This requirement is useful to further illustrate the role of abstraction as a means of communication with different stakeholders. Whilst this requirement shares similarities with requirement R1, it is interesting to note the different safety mechanism indicated in the requirement: R1 requires a "*confirmation action*"; R5 requires "*at least two steps.*" This is done intentionally, because a confirmation action can be performed automatically by the user without further thought, e.g., after a timeout. In R5, the requirement is that two *user actions* are performed—a safety mechanism based on timeouts can be easily defeated in the case of accidental key presses. To ease the formalization, the requirement can be reformulated as follows: "*For any value* `x` *of a given pump setting, if data entry is ready* *(*`vtbi_entry_ready`*) for that pump setting (in this example* `vtbi`*), then the pump setting cannot be updated to* `x` *in a single step*." This property needs to be proved for all possible actions (`state_transitions_xkey1`) with the exception of *key1* which does update the pump setting:

```
(vtbi_entry_ready(pre) AND
   vtbi_value(pre) = x AND
    newvtbi(pre) /= x AND
     state_transitions_xkey1(pre, post))
        IMPLIES vtbi_value(post) = x
```

Interpretation for the specific pump: The example pump is ready to accept a vtbi value when the pump is turned on and in a relevant data entry mode. Specifically, the top line display needs to be either `dispvtbi` (i.e., when entering vtbi and rate) or `vtbitime` (i.e., when entering vtbi and time).

Proving the requirement: PVS is able to prove the requirement unaided for vtbi and time, using `grind`. For infusion rate, on the other hand, the requirement *fails*. Based on the counterexample provided by PVS, it can be shown that the requirement can be satisfied for the infusion rate only if it is assumed that the clinician always locks the rate before starting the infusion. Whilst there is always a reminder to lock the rate when starting the infusion, the assumption needs to be validated against clinical practice adopted in the hospital within which the pump is to be used.

C. Requirements From Property Templates

The second route to generating use-centred safety requirements is to adopt a set of property templates based on usability design principles. The templates are designed to help developers to construct requirements appropriate to the analysis of user interface features. They can be instantiated to the particular details of the device, and provide indications of how to develop user interfaces that are easier to use and promote more transparency of the effect of actions.

Three property templates will be used to illustrate the approach: "feedback," "consistency," and "reversibility." Further property templates and detailed examples can be found in [39].

D. Feedback Template

When certain important actions are taken, a user needs to be aware of whether the resulting device status is appropriate or problematic [40]. Feedback can be considered conveniently as *action feedback*, requiring that an action always has an effect that is visible to the user, and *state feedback*, requiring that a change in the state (usually specific attributes of the state rather than the whole state) is visible to the user. Two example requirements are now illustrated that are based on this template.

R6: Whenever a pump variable is being entered, the variable should be clearly identified and its current value visible to the user.

The instantiation of the general form of action feedback (as described in [39]) requires that entry of the relevant variable is enabled (i.e., the device is in the appropriate data entry mode), and that the variable relevant to the mode is visible. This requirement can be formalized as follows:

```
entry_ready(entrymode)(st) IMPLIES
   visible_variable(mode)(st)
```

Proving the requirement: The requirement can be proved automatically in PVS for all reachable states through structural induction (compare requirement R2).

R7: The current mode should be clearly identified, and changes in mode should have perceivable feedback.

This requirement is an example of state feedback. It requires that, in any situation, if the mode changes then the mode is visible. For the considered pump, top line is the indicator of the entry mode. The formulation can be further refined therefore as follows: "*When the entry mode changes then the top line changes.*" The requirement expressed as logic formula is

```
entrymode(pre) /= entrymode(post) IMPLIES
   topline(pre) /= topline(post)
```

Proving the requirement: The proof of this requirement illustrates the type of human intervention that is necessary to complete an interactive proof attempt. The theorem prover fails to prove the theorem, and generates a first counter-example at the point of failure when the device is in *bag mode*, which allows standard bag volumes to be assigned to vtbi. The counter-example shows that

1) the top line indicates "volume to be infused,"
2) action *key1* can be used to exit the mode, and
3) the new mode, after performing action *key1*, has a top line which also indicates "volume to be infused", but in this mode chevron keys change the value of vtbi through up and down adjustments rather than by navigating the infusion bag menu.

The fact that the prover identifies this counter-example leads the analyst, along with domain and human factors experts, to consider whether this ambiguity is likely to be an issue. Further discussion might suggest that the format of the display *as a whole* between the two modes is significantly different, and therefore enough to prevent mode confusion. This case can be therefore excluded by introducing the following *guard* in the theorem:

```
entrymode(pre) /= bagmode
```

When the proof is attempted again with the guard, the theorem prover throws up another similar situation for another data entry mode, where a top line "vtbi over time" is displayed in two different modes. As in the previous case, this counter-example can be considered a false positive because the overall format of the two displays is significantly different. An additional guard is therefore added to the theorem to exclude this new situation:

```
entrymode(pre) /= ttmode
```

This further refinement of the theorem is sufficient to complete the proof of the requirement.

E. Consistency Template

Users quickly develop a mental model that embodies their expectations of how to interact with a user interface. Because of this, the overall structure of a user interface should be consistent in its layout, screen structure, navigation, terminology, and control elements [40]. The consistency template is formulated as a property of a group of actions, or it may be the same action under different modes, requiring that all actions in the group have similar effects on specific state attributes. The following requirement for example can be constructed based on this template.

R8: When entering numbers, a given action will always confirm the value entered.

This requirement aims to check that the same key can be used across different data entry modes to confirm user input. A general formulation of the requirement includes the following elements: a predicate `guard_em_ok` restricts the set of relevant entry modes where a given confirmation action `confirm` is available; predicate `temp_em_filter` extracts the state attributes that have been changed "temporarily" because of the mode; and predicate `real_em_filter` extracts the state attributes that are changed as a result of exiting from the mode. All predicates have device data entry mode `em` as parameter, as the precise definition of the predicates depends on the data entry mode in the general case.

```
guard_em_ok(em, st) IMPLIES
    temp_em_filter(em, st) =
      real_em_filter(em, confirm(st))
```

For the example pump, *key1* is typically associated with the *ok* function during data entry. The verification effort therefore aims to ensure that, whenever *key1* is enabled and associated with function display *ok*, the confirmation action behaves in a similar way in all data entry modes.

Proving the requirement: When attempting the proof in PVS, a counter-example is found by the theorem prover. The use of the confirmation key is not consistent in data entry mode "vtbi over time." This particular data entry mode involves setting vtbi and time. The theorem fails because when vtbi has been entered, and *key1* is pressed with the *ok* function display, the pump value of vtbi is *not* changed. It is only changed after the next step when time has been changed. The failure, identified while proving the theorem, *may require further scrutiny* to demonstrate that the system's safety is not affected. The device in fact assumes that vtbi and rate are the standard mechanisms for setting up the infusion rate. There may therefore be issues when a prescription is received that presents vtbi and time.

F. Reversibility Template

Users may perform incorrect actions, and the device needs to provide them with functions that allow them to recover by reversing the effect of the incorrect action. In [41], it has been shown that lack of compliance to this requirement could lead to data entry errors. An example requirement based on this template is as follows.

R9: Any data entry action should be reversible.

To facilitate the formalization process, this requirement can be reformulated as follows: "*For any particular action* `a1`*, there is a reversing action* `a2` *which returns the device to its original state.*" This reformulation can be easily translated into the following generic formula:

```
data_entry_ready(st) IMPLIES
  pump_variable(a2(a1(st))) =
    pump_variable(st)
```

For the example pump, the formula needs to consider all the chevron keys and entry of infusion rate, time and vtbi. For the sake of clarity the example is given only for the case of entering infusion rate and the single chevron up (*sup*) key with inverse single chevron down (*sdown*) key. The example pump is designed so that the user can accelerate the size of the increment by holding the chevron key down. This possibility is reflected in the specification but is simplified here for illustration.

```
click_sdown(st: state): state =
          release_sdown(sdown(st))
  click_sup(st: state): state =
          release_sup(sup(st))
```

The formulation for the other chevron keys and data entry modes is identical. The generic version of the requirement is therefore instantiated as follows:

```
rate_entry_ready(st) IMPLIES
 infusion_rate(click_sdown(click_sup(st)))
     = infusion_rate(st)
```

Proving the requirement: This final proof example further shows how feedback from the theorem prover can be used constructively to refine the understanding of the interaction design of the device. Proof of the theorem fails, and the theorem prover returns counter-examples indicating anomalies for certain specific values. An example counter-example identified by the theorem prover occurs at 99.9: pressing *sup* and then *sdown* produces 99 (instead of 99.9). This happens because of "step boundaries" where the size of the increment or decrement changes. Up to, but not including, 100, both the single chevron up key and the single chevron down key make a step of 0.1. From 100 upward (up to 1000), the increment is 1. Therefore, to prove that *sup* can be reversed by *sdown*, the theorem must take account of these step boundaries. The additional conditions necessary to prove the theorem for the range 0 to 100 for the *sup* and *sdown* keys are illustrated here. The first condition ensures that the analysis is performed within a range of values where the same increment step (in this case, 0.1) is always maintained:

```
infusion_rate(device(st)) >= 0 AND
    infusion_rate(device(st)) + 0.1 < 100
```

A second condition is necessary to further restrict the domain of values considered in the proof to those handled by the device: numbers below 100 can have only one decimal place:

```
infusion_rate(st) =
 (floor(10 * infusion_rate(st))/10) AND
   infusion_rate(st) =
    (ceil_rate(10 * infusion_rate(st))/10)
```

where `floor` returns the largest integer less than or equal to the specified number, and `ceil` returns the smallest integer greater than or equal to the given number. Wrapping all these constraints together, for all step boundaries, and for all chevron keys, the property to be proved for requirement R9 becomes much more complex. Proof of this requirement, with all its qualifications, provides limited assurance that number entry actions are easily reversible by users. Clearly the process was valuable in understanding the characteristics of the number entry system, and identify *precisely* in the design space where the property fails. Formulating and proving the theorem raised practical questions about whether the data entry system of the device is acceptable, or whether it is likely to lead to use error. It should be noted that later releases of the firmware for this device have fixed these issues with step boundaries.

VIII. Discussion and Conclusions

Demonstrating that the design of a medical device is compliant to relevant safety and usability requirements is a serious problem. The techniques described in this paper are designed to address this issue. It is estimated that there were 56 000 adverse event reports relating to infusion pumps between 2005 and 2009 in the United States including at least 500 deaths [42]. This has resulted in 87 infusion pump recalls to address identified safety concerns, according to FDA data. Of these adverse event reports, use error has been a significant factor. The documentation provided by manufacturers to regulators as a part of a safety argument is usually substantial. The scale of the argument inevitably makes it difficult for regulators to comprehend them and to be confident that the evidence provided is of satisfactory quality. The use of formal techniques has advantages.

1) It is precise and concise, potentially avoiding the documentation explosion generated by a typical deposition.
2) Tools like model checkers and theorem provers enable mechanical and exhaustive verification.
3) The use of simulation techniques combined with formal modeling can clearly demonstrate how potential problems are addressed that can be of value to regulators, human factors and domain specialists.

There are well-known obstacles to the immediate take-up of these facilities. They are not routinely part of a typical developer's suite of tools. They are not routinely used in product development. However there are signs of interest in these techniques. For example, the FDA has developed generic PCA models [43] using Simulink. It is not however a feasible option to expect regulators to construct models after the fact. An ideal option would be that manufacturers produce models as part of their design process demonstrating that a submitted product adheres to safety requirements. The regulators would then use tools to validate the models provided as part of the developer's submission.

We have used model checking combined with simulation to support the process of validation of models by generating traces to be validated on the device. However, manufacturers have access to source code, and even if they do not develop their devices using models, they can create faithful models systematically. A further important issue, not addressed here, is how to validate that the considered safety requirements correctly address the usability and safety of the device for the context in which the device is to be used. This problem is orthogonal to the techniques presented in this paper. Safety aspects can be addressed with a human factors emphasis using a hazard analysis such as the one presented in [44].

We have illustrated how formalizing the requirements provides benefits in addition to the ability to prove them. It has led to much more detailed thinking about the precise nature of the requirements, both in general and for a specific device, than was possible in the informal natural language version. The pragmatic and informal combination of model checking and theorem proving provided powerful tools for analysis. By using each flexibly for requirements they were suited to, rather than ideologically favoring one for all requirements, or trying to combine them into a single tool applying both, it was possible to prove the requirements with relatively low effort. One potential drawback of this approach is the need to master the two verification techniques. Indeed, both verification methods currently require significant skills for analysis. However, we have observed recurrent patterns in the structure of the formal models of devices from different manufacturers, and in the strategies needed to complete verification of several types of safety requirements. Therefore, there are clear opportunities to create automated proof strategies that can be used to reduce the analysis effort.

References

[1] N. G. Leveson, *Engineering a Safer World: Systems Thinking Applied to Safety (Engineering Systems)*. Cambridge, MA, USA: MIT Press, 2011.

[2] D. Arney *et al.*, "Generic infusion pump hazard analysis and safety requirements," Univ. Pennsylvania, Philadelphia, PA, USA, Tech. Rep. MS-CIS-08-31, Feb. 2009.

[3] US Food and Drug Administration, "General principles of software validation; Final guidance for industry and FDA staff," Center for Devices and Radiological Health, Tech. Rep., Jan. 2002. [Online]. Available: http://www.fda.gov/medicaldevices/deviceregulationandguidance

[4] Y. Rogers, H. Sharp, and J. Preece, *Interaction Design: Beyond Human Computer Interaction*, 3rd ed. Hoboken, NJ, USA: Wiley, 2011.

[5] R. Jetley, S. P. Iyer, and P. Jones, "A formal methods approach to medical device review," *Computer*, vol. 39, no. 4, pp. 61–67, 2006.

[6] M. Bolton, N. Jiménez, M. van Paassen, and M. Trujillo, "Automatically generating specification properties from task models for the verification of human-automation interaction," *IEEE Trans. Human Mach. Syst.*, vol. 44, no. 5, pp. 561–575, 2014.

[7] J. C. Campos and M. D. Harrison, "Interaction engineering using the IVY tool," in *Proc. ACM SIGCHI Symp. Eng. Interactive Comput. Syst.*, G. Calvary, T. Graham, and P. Gray, Eds. ACM Press, 2009, pp. 35–44.

[8] M. L. Bolton and E. J. Bass, "Formally verifying human-automation interaction as part of a system model: limitations and tradeoffs," *Innovations Syst. Softw. Eng.*, vol. 6, no. 3, pp. 219–231, 2010.

[9] M. L. Bolton and E. J. Bass, "Generating erroneous human behavior from strategic knowledge in task models and evaluating its impact on system safety with model checking," *IEEE Trans. Syst., Man, Cybern., Syst.*, vol. 43, no. 6, pp. 1314–1327, 2013.

[10] L. de Moura, "SAL: Tutorial," SRI International, Computer Science Laboratory, Menlo Park, CA, Tech. Rep., 2004.

[11] G. Mori, F. Paternò, and C. Santoro, "CTTE: Support for developing and analyzing task models for interactive system design," *IEEE Trans. Softw. Eng.*, vol. 28, no. 8, pp. 797–813, Aug. 2002.

[12] R. E. Fields, "Analysis of erroneous actions in the design of critical systems," Ph.D. dissertation, Dept. Comput. Sci., Univ. York, York, U.K., 2001.

[13] J. Berstel, S. Reghizzi, G. Rouseel, and P. Pietro, "A scalable formal method for the design and automatic checking of user interfaces," *ACM Trans. Softw. Eng. Methodology*, vol. 14, no. 2, pp. 124–167, 2005.

[14] J. Bowen and S. Reeves, "Design patterns for models of interactive systems," in *Proc. 2015 24th Australasian Softw. Eng. Conf.*, 2015, pp. 223–232.

[15] B. Kim *et al.*, "Safety-assured development of the GPCA infusion pump software," in *Proceedings of the 9th ACM International Conference on Embedded Software.* New York, NY, USA: ACM, 2011, pp. 155–164. [Online]. Available: http://doi.acm.org/10.1145/2038642.2038667

[16] G. Behrmann, A. David, and K. Larsen, "A tutorial on UPPAAL," in *Formal methods for the design of real-time systems* (ser. Lecture Notes in Computer Science). M. Bernardo and F. Corradini, Eds. New York, NY, USA: Springer-Verlag, 2004, no. 3185, pp. 200–236.

[17] T. Li, F. Tan, Q. Wang, L. Bu, J. Cao, and X. Liu, "From offline toward real time: A hybrid systems model checking and CPS codesign approach for medical device plug-and-play collaborations," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 3, pp. 642–652, Mar. 2014.

[18] F. Tan, Y. Wang, Q. Wang, L. Bu, and N. Suri, "A lease based hybrid design pattern for proper-temporal-embedding of wireless CPS interlocking," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 10, pp. 2630–2642, Oct. 2015.

[19] A. L. King *et al.*, "An open test bed for medical device integration and coordination," in *Proc. Int. Conf. Softw. Eng. Companion*, 2009, pp. 141–151.

[20] B. Larson, J. Hatcliff, S. Procter, and P. Chalin, "Requirements specification for apps in medical application platforms," in *Proc. 4th Int. Workshop Softw. Eng. Health Care*, 2012, pp. 26–32.

[21] C. Heitmeyer, J. Kirby, and B. Labaw, "Applying the SRC requirements method to a weapons control panel: An experience report," in *Proc. 2nd Workshop Formal Methods Softw. Practice*, 1998, pp. 92–102.

[22] G. Gelman, K. Feigh, and J. Rushby, "Example of a complementary use of model checking and agent-based simulation," in *Proc. 2013 IEEE Int. Conf. Syst., Man, Cybern.*, Oct. 2013, pp. 900–905.

[23] M. van Paassen, M. L. Bolton, and N. Jiménez, "Checking formal verification models for human-automation interaction," in *Proc. 2014 IEEE Int. Conf. Syst., Man Cybern.*, 2014, pp. 3709–3714.

[24] D. Billman, C. Fayollas, M. Feary, C. Martinie, and P. Palanque, "Complementary tools and techniques for supporting fitness-for-purpose of interactive critical systems," in *Proc. Int. Conf. Human-Centred Softw. Eng.*, 2016, pp. 181–202.

[25] P. Masci, A. Ayoub, P. Curzon, I. Lee, O. Sokolsky, and H. Thimbleby, "Model-based development of the generic PCA infusion pump user interface prototype in PVS," in *Computer Safety, Reliability, and Security* (ser. Lecture Notes in Computer Science), vol. 8153, F. Bitsch, J. Guiochet, and M. Kaâniche, Eds. New York, NY, USA: Springer-Verlag, 2013, pp. 228–240.

[26] J.-R. Abrial, *Modeling in Event-B: System and Software Engineering*. Cambridge, U.K.: Cambridge Univ. Press, 2010.

[27] J. Bowen and S. Reeves, "Refinement for user interface designs," *Formal Aspects Comput.*, vol. 21, pp. 589–612, 2009.

[28] G. Hamon, L. De Moura, and J. Rushby, "Generating efficient test sets with a model checker," in *Proc. Proc. 2nd Int. Conf. Softw. Eng. Formal Methods*, 2004, pp. 261–270.

[29] P. Masci *et al.*, "Verification of interactive software for medical devices: PCA infusion pumps and FDA regulation as an example," in *Proc. ACM Symp. Eng. Interactive Syst.*, 2013, pp. 81–90.

[30] Cardinal Health Inc., "Alaris GP volumetric pump: directions for use," Cardinal Health Rolle, Switzerland, Tech. Rep. 1000DF00546, 2006.

[31] M. Harrison, J. Campos, and P. Masci, "Reusing models and properties in the analysis of similar interactive devices," *Innovations Syst. Softw. Eng.*, vol. 11, no. 2, pp. 95–111, Jun. 2015.

[32] A. Cimatti *et al.*, "NuSMV 2: An open source tool for symbolic model checking," in *Computer-Aided Verification* (ser. Lecture Notes in Computer Science), vol. 2404, K. G. Larsen and E. Brinksma, Eds. New York, NY, USA: Springer-Verlag, 2002.

[33] S. Owre, J. Rushby, and N. Shankar, "PVS: A prototype verification system," in *11th International Conference on Automated Deduction* (ser. Lecture Notes in Artificial Intelligence), vol. 607, D. Kapur, Ed. New York, NY, USA: Springer-Verlag, 1992, pp. 748–752.

[34] J. C. Campos, M. Sousa, M. C. B. Alves, and M. D. Harrison, "Formal verification of a space system's user interface with the IVY workbench," *IEEE Trans. Human Mach. Syst.*, vol. 46, no. 2, pp. 303–316, Apr. 2016.

[35] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.

[36] E. M. Clarke, O. Grumberg, and K. Hamaguchi, "Another look at LTL model checking," *Formal Methods Syst. Design*, vol. 10, no. 1, pp. 47–71, 1997. [Online]. Available: http://dx.doi.org/10.1023/A:1008615614281

[37] P. Masci, P. Oladimeji, Y. Zhang, P. Jones, P. Curzon, and H. Thimbleby, *PVSio-web 2.0: Joining PVS to HCI*. Cham, Switzerland: Springer, 2015, pp. 470–478. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-21690-4_30

[38] J. C. Campos and M. D. Harrison, "Systematic analysis of control panel interfaces using formal tools," in *Interactive Systems: Design, Specification and Verification* (ser. Lecture Notes in Computer Science), N. Graham and P. Palanque, Eds. New York, NY, USA: Springer-Verlag, 2008, no. 5136, pp. 72–85.

[39] M. Harrison, J. Campos, and P. Masci, "Patterns and templates for automated verification of user interface software design in PVS." , School Comput. Sci., Newcastle Univ., Newcastle upon Tyne, U.K., Tech. Rep. TR-1485, 2015.

[40] AAMI, "Medical devices - Application of usability engineering to medical devices," Assoc. Adv. Med. Instrum., Arlington, VA, USA, Tech. Rep. ANSI AMI IEC 62366:2007, 2010.

[41] H. Thimbleby, "Safer user interfaces: A case study in improving number entry," *IEEE Trans. Softw. Eng.*, vol. 41, no. 7, pp. 711–729, Jul. 2015.

[42] J. T. James, "A new, evidence-based estimate of patient harms associated with hospital care," *J. Patient Safety*, vol. 9, no. 3, pp. 122–128, 2013.

[43] A. Murugesan, M. W. Whalen, S. Rayadurgam, and M. P. E. Heimdahl, "Compositional verification of a medical device system," in *Proc. ACM High Integrity Lang. Technol.*. ACM Press, 2013, pp. 51–64.

[44] P. Masci, Y. Zhang, P. Jones, H. Thimbleby, and P. Curzon, "A generic user interface architecture for analyzing use hazards in infusion pump software," in *5th Workshop on Medical Cyber-Physical Systems* (ser. OpenAccess Series in Informatics (OASIcs)), vol. 36, V. Turau, M. Kwiatkowska, R. Mangharam, and C. Weyer, Eds. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014, pp. 1–14. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2014/4518

Michael D. Harrison received the B.A. degree in mathematics/Computation from University of York, England, UK, in 1973; and D.Phil. degree in mathematics/computation from University of Oxford, England, UK, in 1978. He is an Emeritus Professor and Senior Research Investigator at Newcastle University, Newcastle upon Tyne, U.K. and a Research Fellow at Queen Mary University of London, London, U.K. (funded to work on the analysis of medical devices). He is a Visiting Researcher at the University of Minho, Portugal. His research focuses on the systematic analysis of the functional behavior of interactive systems using a combination of model checking and automated theorem proving techniques.

Paolo Masci received the M.Sc. degree in computer engineering and Ph.D. degree in information engineering from University of Pisa, Italy, in 2003 and 2008, respectively. He is a Senior Researcher at HASLab/INESC TEC, and a Visiting Researcher at the US Food and Drug Administration (FDA), Silver Spring, MD, USA. In the past, he has been a Visiting Researcher at Stanford Research Institute (SRI), NASA Langley, and University of Pennsylvania. His research focuses on the development of tools and methods for the analysis of human–machine interfaces in medical cyber-physical systems.

José Creissac Campos received the Lic. degree in informatics and systems engineering, M.Sc. degree in informatics both from University of Minho, Braga, Portugal, in 1991, 1993, respectively, and D.Phil. degree in computer science from University of York, England, UK, in 2001. He is an Assistant Professor in the Department of Informatics, University of Minho, Portugal, and a Senior Researcher at HASLab/INESC TEC, Braga, Portugal. His research focuses on the application of software engineering techniques and tools to the modeling and analysis of interactive systems, aiming at bringing closer software engineering and human-computer interaction (HCI).

Paul Curzon received the B.A., M.A. degrees in computer science from University of Cambridge, England, UK, in 1986, 1989, respectively, and Ph.D. degree in computer science from University of Cambridge, England, UK, in 1990. He is a Professor of Computer Science in the School of Electronic Engineering and Computer Science, Queen Mary University of London, London, U.K. His research focuses on the application of interactive theorem proving to interaction design and human–computer interaction. He has a particular interest in modeling and verification approaches to detect design flaws that lead to systematic human error.