Verification Templates for the Analysis of User Interface Software Design

Michael D. Harrison, School of Computing, Newcastle University, Newcastle upon Tyne, UK
Paolo Masci, HASLab / INESC TEC and Universidade do Minho, Braga, Portugal
José C. Campos, HASLab / INESC TEC and Universidade do Minho, Braga, Portugal

Abstract—The paper describes templates for model-based analysis of usability and safety aspects of user interface software design. The templates crystallize general usability principles commonly addressed in user-centred safety requirements, such as the ability to undo user actions, the visibility of operational modes, and the predictability of user interface behavior. These requirements have standard forms across different application domains, and can be instantiated as properties of specific devices. The modeling and analysis process is carried out using the Prototype Verification System (PVS), and is further facilitated by structuring the specification of the device using a format that is designed to be generic across interactive systems. A concrete case study based on a commercial infusion pump is used to illustrate the approach. A detailed presentation of the automated verification process using PVS shows how failed proof attempts provide precise information about problematic user interface software features.

Index Terms—Human-Computer Interaction, Model-based development, Formal specifications, Formal verification, Prototype Verification System (PVS).

1 Introduction

Demonstrating that a device design satisfies safety requirements is part of a process that provides regulators with confidence that there are barriers that mitigate identified hazards. Many standards propose requirements that should be verified to demonstrate safety (i.e., that the device does not harm people). Some of these safety requirements are use-centred and domain specific, for example [1] "The pump shall issue an alert if paused for more than t minutes".

User interface software issues are an important reason for system or device failure in application domains such as healthcare, avionics, and traffic control. Despite this, little work has been done to provide tool support for the analysis of use-related requirements. We therefore consider the question: *given an interactive device, can formal techniques be used to assess whether the design of the device satisfies requirements that concern its usability?* The aim is to provide an assessment method for the design of core user interface software components that is more concise and complete than existing techniques based on code inspection and testing.

Formal verification of user interface software has seen slow take-up because verification technology is perceived as difficult to use and to apply. A review of formal tools and techniques [2] places these barriers on three dimensions: implementation cost, specification cost and verification cost. The work described here aims to reduce these three costs by introducing a systematic process for early detection of user interface software issues at the design stage (see discussion in Section 2.1). The results of the analysis are of interest beyond software engineering and must also be meaningful to other disciplines, for example related to the domain or to human factors. In these cases clear demonstrations of the implications of the results are necessary so that considerations of the meaning of requirements and exceptions that arise through the analysis are possible. Furthermore the requirements should have the effect of improving the safety and usability of the interactive system for the *user*.

We introduce *property templates* to facilitate the introduction of formal methods technologies in the development life-cycle of core user interface components. These templates describe general user-centred requirements that can be adopted to analyze essential usability aspects of the device and, if true, can mitigate hazards that might arise through use error. The developed templates are instantiated to the details of the particular device represented by a formal specification of its design. The instantiated properties are then used within a formal verification system to analyze the conditions under which the device design satisfies the property.

Model development and verification will be illustrated using a medical device currently found in many hospitals across the EU and US. We will show that the analysis, based on templates, enables the identification of inconsistencies that can result in poor understanding of the user interface and increase the risk of use errors with the potential for harm to the patient.

An initial model of the specific medical device had already been developed. It had been developed using Modal Action Logic (MAL) [3], the language of the IVY [4] tool. IVY uses NuSMV [5], a model checker, as back-end for formal verification. This initial model was

Manuscript received mm, yyyy; revised mm, yyyy.

suitable for the analysis of properties of the modal behavior of the device, such as whether data entry modes were presented by the device without ambiguity [6]. The model has been extended to allow the analysis of the number entry system of the device. One effect of this extension was that the size of the model increased significantly, making the use of the NuSMV model checker time consuming and, for various properties related to number entry, infeasible (see also [7]). We therefore decided to change verification technology. Our option was the Prototype Verification System (PVS) [8], which is a theorem proving assistant. This technology differs from model-checking, which relies on automatic and exhaustive exploration of execution paths described in a model. Theorem proving builds on logic formulas and deduction methods, which better support the analysis of richer properties, and can better handle domains of variables with larger cardinality. The downside of using theorem proving is that the verification process is not fully automatic and temporal properties cannot be proved without adding constructs to the model. The theorem prover often needs guidance to complete the proof of complex properties. Overall, our experience of the available tools led to the conclusion that the benefits of using theorem proving outweighed the potential disadvantages arising from the lack of automation for two reasons.

- Translating the original MAL model into a PVS model was fairly straightforward thanks to the expressiveness of the PVS language. A manual translation of the MAL model into the language of a model checker, different from that supported by IVY, would have required an excessive amount of time and effort.
- A prototyping tool, PVSio-web [9], is available that makes it possible to generate an interactive simulation[1] of the device based on the developed PVS model. This simulation proved significant in validating hypotheses embedded in the model, as well as for discussing the analyzed properties and the results of the analysis with software engineers and end users of the device (i.e., nurses and medical device trainers).

Contribution. Three main contributions are offered.

1) A systematic process for the analysis of the design of core user interface software components is described. It can be performed either as part of a model-based development process, or retrospectively, by constructing a model of an existing design.
2) Property templates are described that capture general user-centred safety requirements related to user interface software design. These requirements are translated into logic formulas that can be checked using PVS or equivalent verification technology.
3) A case study is described based on a commercial medical device in use in many hospitals. The verification results can be used to complement and support test data necessary to demonstrate that the design of core user interface software components meets general user-centred safety requirements.

The full specification and the documentation of the illustrated case study may be found at our repository[2] and on Github[3].

Organization. Section 2 frames the work within the context of a software development process and relevant international standards. Section 3 provides background information about the PVS specification and verification language. Section 4 introduces the medical device used for illustration throughout the paper. Section 5 presents the PVS model of the selected medical device. Section 6 provides the main contribution, illustrating the property templates and demonstrating how the templates are instantiated to the details of the model. Relevant aspects of the verification process are presented and discussed. Section 7 presents a final discussion of the benefits of the method. Further related research beyond that described in Section 7 is presented in Section 8. Section 9 concludes the paper.

2 User interface software: designing for safety

The focus of the paper is *user interface software*. Of particular interest is the design of *core software components responsible for human-machine interaction*. These modules are safety-critical in the sense that latent anomalies in their design can lead to use error and potential harm. An example of such an anomaly recently involved a commercial medical device. A diabetes management mobile app erroneously resets the recommended insulin bolus dosage when the user changes the smartphone's orientation. This feature opens the possibility that the user inadvertently commands and receives unsafe insulin therapies [10].

The next sub-sections frame the contribution of the paper. Typical activities carried out within a software engineering process are first considered. The contribution is then related to three international standards that address usability and safety concerns in the context of medical devices. Note that, while medical standards are identified here, similar requirements may be identified in standards developed for other domains and therefore the contribution has a wider application. These standards define:

ISO 14971: the overall risk management process for medical devices;

ISO 62304: the life-cycle requirements for medical device software development;

1. http://www.pvsioweb.org/demos/AlarisGP
2. http://hcispecs.di.uminho.pt/m/5
3. http://github.com/haslab/hcispecs/archive/1.1.zip

ISO 62366-2: the characteristics of a usability engineering process suitable for minimizing use errors and use-associated risks in medical devices.

2.1 Software engineering process

A software engineering process typically includes the following main activities.

1) *Requirements*. System and software requirements are defined, including: functional capabilities of the system; safety, security, and human-factors specifications; criteria and conditions to assess compliance of a software product to its specification.
2) *Software design*. A detailed software specification is developed based on the given requirements.
3) *Verification*. This includes checking conformity of a design with the stated requirements.
4) *Validation*. This involves checking that the requirements correctly capture the intended characteristics and functionalities of the system to be developed.
5) *Coding / Implementation*. Software design documents are transformed into a concrete implementation for a target platform.
6) *Testing*. The software implementation is executed under known conditions and inputs, and its behavior is compared to expected outputs.
7) *Deployment and training*. The software is installed in the target environment, and the necessary training is provided to end users.
8) *Support and maintenance*. Activities are carried out to fix errors/faults, or to improve performance, usability, security and other quality-related aspects of the system.

These activities are not sequential. They can interleave and iterate in different ways depending on the particular software life-cycle adopted (e.g., waterfall, iterative, agile).

The paper contributes directly to activities 1–3.

- The property templates presented in Section 6 capture general use-related safety requirements that can prevent use error and facilitate recovery when a use error occurs (*Requirements*).
- The use of formal methods technologies such as PVS contributes to the systematic development of a software specification (*Software design*). Whilst this may increase the initial specification effort, experience shows that it will lead to reduced costs later in the development process [11].
- Formal verification of the property templates provides objective evidence that the software design meets general human factors design principles, as recommended in ISO 62366-2, Annex 1 (*Verification*).

When a model-based development approach is adopted, the benefits can also extend beyond activities 1–3.

- Models used within the approach can be imported into tools such as PVSio-web [9], and converted to realistic prototypes suitable for both design validation [12] and training of end users [13].
- The same analysis models can be used as a basis for code generation [14], thus reducing implementation cost.
- Verification results can inform test case generation [15], thus reducing the cost of testing.

2.2 ISO 14971

ISO 14971 describes five distinct activities that are required to implement a disciplined risk management process. (i) A hazard analysis is performed to identify all known and foreseeable hazards and their causes, where a hazard is defined as a potential source of physical injury or damage to people or the environment. (ii) Risk estimation is performed to assess the probability of occurrence and severity of harm of each hazard, the combination of which is defined as risk. (iii) Risk evaluation is conducted to decide if every identified risk is acceptable based on justifiable acceptability criteria. (iv) Control measures are designed and implemented to eliminate the risk, or to reduce it to an acceptable level, if a risk is considered to be unacceptable. (v) Verification and validation activities are conducted to ensure that the designed control measures are effective. These five activities iterate and interleave until the device's overall residual risk after mitigation is acceptable.

This paper contributes to the process described in ISO 14971 by defining:

- a set of property templates that can be used to generate requirements that mitigate known use-related hazards (Activity (iv) in the standard);
- a method for applying existing formal methods technologies to perform the verification of user interface software design against the property templates (Activity (v) in the standard).

2.3 ISO 62304

To demonstrate safety of software artefacts, ISO 62304 requires the definition and adoption of a rigorous development process. This requirement applies to any development strategy (waterfall, incremental, evolutionary, etc.), and follows from the observation that software testing alone is not sufficient to demonstrate that the software will operate safely. One of the key activities necessary to support such a rigorous process involves the definition of a set of safety requirements that can be verified based on objective criteria. The identified set of requirements can be used as a basis to argue about the safety of the system. The possibility of verifying the requirements gives developers the means to demonstrate that the device design is acceptably safe.

Our work contributes to the process described in ISO 62304 in two ways.

- Property templates define verifiable usability requirements of the user interface software design.

The templates are general in the sense that they are not limited to specific software implementations or architectures.
- Formal methods technologies provide developers with tools necessary to create objective evidence that a software design complies with given requirements.

2.4 ISO 62366-2

ISO 62366-2 defines the characteristics of a usability engineering process suitable for identifying use-related risks that might arise through poor user interface design. It is an iterative process that includes the following four main activities. (i) A conceptual user interface design is defined. (ii) Testable requirements are defined for user interface functions that are directly related to the safety of the medical device. (iii) A detailed user interface design is created. (iv) The user interface design is evaluated against the identified requirements. The standard exemplifies some common violations of user interface design heuristics. It states that developers need to take these into account when defining the requirements, as these violations could lead to use hazards. Examples include: complex controls or poor mapping of controls to actions; unclear medical device state; controversial modes, settings, measurements, or other information; insufficient visibility, audibility or tactility.

This paper is aligned with ISO 62366-2. It provides:
- property templates based on user interface design heuristics explicitly mentioned in the standard;
- a process for defining a detailed user interface design that can be verified, though the use of formal methods technologies, against the property templates.

3 The Verification Technology

The theorem proving system used in this paper is the *Prototype Verification System (PVS)* [16]. It combines an expressive specification language based on higher-order logic with a theorem proving assistant. PVS has been used extensively in several application domains. It provides the usual basic types such as `bool`, `integer` and `real`. New types can be introduced either in a declarative form (these types are called *uninterpreted*), or through *type constructors*. Examples of type constructors, used in the case study, are function and record types. Function types are denoted `[D -> R]`, where `D` is the domain type and `R` is the range type. Predicates are functions with Boolean range type. Record types are defined by listing the field names and their types between square brackets and hash symbols. For instance, record `[# a1: A1, a2: A2 #]` has two fields (`a1` of type `A1`, and `a2` of type `A2`). The fields of a record type are accessed using the corresponding field names. Hence if a record `r` has the type defined above then `a2(r)` or `r`a2` can be used equivalently to access the value of field `a2` in record `r`. Predicate subtyping is a language mechanism used for restricting the domain of a type by using a predicate. An example of a subtype is `{ x:A | P(x) }`, which introduces a new type as the subset of those elements of type `A` that satisfy the predicate `P`. The notation `(P)` is an abbreviation of the subtype expression above. Predicate subtyping is useful for specifying partial functions. This will be used in the case study when defining actions that are only permitted given specific constraints.

A specification in PVS is expressed as a collection of *theories* which consist of declarations of names for types and constants, and expressions in terms of these names. Theories can be parametrized with types and constants, and can use declarations of other theories by importing them.

Properties of a PVS specification are expressed as named formulas declared using the keyword `THEOREM`. Structural induction will often be used to prove that a given property is an invariant of the system model. This process involves proving a property is true of all relevant *reachable* states when universal quantification is not possible as will be discussed in Section 6.

The *prelude* is a standard library automatically imported by PVS. It contains useful definitions and proved facts for types, including among others common base types such as Booleans (`bool`) and numbers (e.g., `nat`, `integer` and `real`), functions, sets, and lists. The *prelude* has been used explicitly in the proof of several of the properties developed from the templates.

The interactive theorem prover of PVS provides a collection of powerful primitive inference procedures that are applied interactively under user guidance within a sequent calculus framework. These include propositional and quantifier rules, induction, rewriting, simplification using decision procedures for equality and linear arithmetic, data and predicate abstraction. Additional information about the PVS theorem proving assistant will be given when necessary in Section 6, while presenting example proofs of property templates.

4 Case Study Overview

This section introduces the case study used throughout the paper as a reference. It will be used to define and illustrate the use of the property templates for automated analysis of user interface software.

4.1 A programmable infusion pump

The selected device is an infusion pump (see Figure 2). It is an existing device [17] used in many hospitals. Infusion pumps are devices used by clinicians to inject fluids (typically, medicines or nutrients) into patients. The typical architecture of an infusion pump includes the following main components (see Figure 1): a *user interface*, which allows operators to program infusion parameters and monitor the infusion process; a *controller*, representing the device components that drive the administration process; a *pump delivery mechanism*,

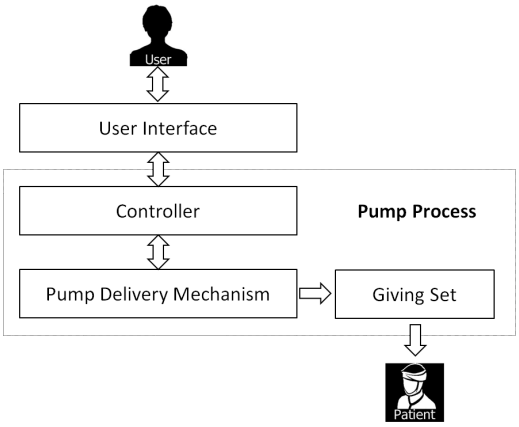

Fig. 1. Generic pump architecture (adapted from [22]).

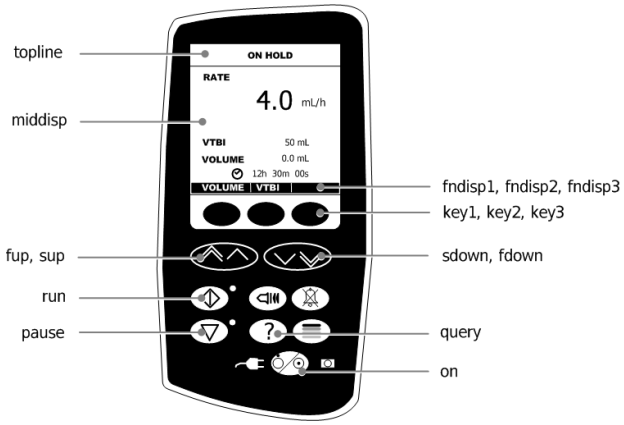

Fig. 2. The pump user interface and actions

representing the physical pump that injects the fluid in the patient; and *the giving set* representing the tube that connects a fluid reservoir to the patient.

Infusion pumps are "programmable" in the sense that infusion parameters and pump settings can be configured by clinicians. The characteristics of this case study are common to many devices that control the delivery of a therapy over time. Infusion pumps are used in several contexts within a hospital, including chemotherapy and intensive care. The clinician (usually a nurse) sets infusion pump parameters and connects the patient to the device using the "giving set" (i.e., a flexible clear plastic tube, one end connected to a bag with the fluid to be infused, the other end connected to the patient's veins through a needle) and then monitors the infusion process using the device. This type of device was chosen because it is susceptible to use error. In the United States, the US Food and Drug Administration (FDA), as reported in [18], received approximately 56,000 reports of adverse events relating to infusion pumps between 2005 and 2009 including at least 500 deaths. Many of the adverse events were use-related. 87 infusion pump recalls have resulted to address identified safety concerns, according to FDA data. Of these adverse event reports use error, due to anomalies in software design and user interface designs, has been a significant factor. Use error, as mentioned in ISO 62366-2, means for example: number entry errors; confusions over input modes (for example updating the wrong parameter value), transcription errors from prescription to device; failure to check that the value has been entered correctly.

Recent estimates over the whole spectrum of device types indicate that the number of deaths associated with preventable adverse events due to use error is over 400,000 per year in the US alone [19]. Mitigation of use errors has been indicated several times as one of the top priorities in device design for infusion pumps [20], ventilators [21], and other interactive medical devices.

4.2 The modes of the user interface

The user interface of the example device is characterized by a set of *entry modes* that determine the effect of user interactions. These modes are specific to the device brand (although a given manufacturer may have families of devices with similar mode structures). The entry modes are used by developers to make most effective use of the keys and displays available on the device front panel. For example, entry modes determine whether chevron keys alter infusion rate, VTBI (volume to be infused), time or move the cursor up or down the options or infusion bags menu. They also have an effect on the function keys (key_1, key_2 or key_3). The mode structure for the device considered in the case study is relatively complex. This will become clear later in the paper as properties of the device are identified and analyzed. A full list of the entry modes of the device and a brief description of each mode is in Table 1.

5 Modeling User Interface Software Design

Different types of models can be used to describe an interactive system (e.g., focusing on user interface layout or focusing on the interaction between user and system). The method to be described here structures models as a set of *actions* initiated by the user. The model specifies the effect that each action has on the state of the device. The device state is detailed as a set of *state attributes*. Each state attribute has a type, e.g., if the attribute is an infusion rate, as will be described in Section 5.2, then it is a number restricted to infusion rate values that the operator can input. Although user action is the focus of the analysis, the model also includes autonomous actions that describe the ongoing behavior of the underlying process controlled by the device (e.g., the amount of fluid infused by the device) insofar as they affect the user interface. It is not usually possible for realistic systems to get an understanding of the behavior of user actions without describing those autonomous actions that also modify the state of the device.

Entry Modes	Short Description
rmode	Infusion rate can be adjusted; this mode is only available when the pump is paused.
bagmode	VTBI can be selected from an infusion bags menu when the user is entering rate and vtbi.
tbagmode	VTBI can be selected from the infusion bags menu; this mode is accessed when entering vtbi over time and is only available when the pump is paused.
qmode	Query mode, in which a menu of options is made available to the user.
vtmode	VTBI can be adjusted, this also modifies time - calculating time using the infusion rate that has been entered.
vttmode	VTBI is being entered in vtbi over time mode that is accessed via the options menu.
ttmode	Time is being entered in vtbi over time mode that is accessed via the options menu.
infusemode	The home mode when the device is infusing. This mode can be recognised by checking the top part of the display, which shows "infusing". This mode also allows the rate to be changed unless the infusion rate has been locked.
nullmode	This mode describes a set of display only situations - for example where alarm displayed in top line, or options menu elicits an information display. In this mode no data entry is possible.

TABLE 1
The entry modes of the device

Some state attributes can be temporarily or permanently perceivable (usually visible, but could be audible, for example in the case of an alarm, or haptic). These attributes trigger the appropriate use of actions and indicate the effect when action has been taken. For example, the visibility of an attribute such as infusion rate can be used by the operator to decide whether the rate should be increased or decreased to meet the value prescribed, and what its effect on the patient should be.

The model of the interactive system also makes it clear what actions are permitted. The effect of an action can depend on modes. These modes, unless clearly signposted, can confuse users about an action's effect and can be difficult to understand [23].

5.1 A more rigorous description

A more rigorous definition of the elements of the model is now introduced. Throughout the paper, the following naming conventions will be used:

- A indicates the set of *actions*
- S indicates the set of *states* of the device
- **B** is the set { true, false }
- C is a set of *state attributes*;
- MS is a set of *modes*

The illustrated PVS models comply with the following structure.

- Actions are typically partial functions over states of the system $A = S \rightarrow S$. An action may be associated with a permission function per, which is a predicate that asserts whether an action is defined for a state in its domain $per : A \rightarrow (S \rightarrow \mathbf{B})$ such that $per(a)(s) = true$ if $a(s)$ is defined. An action could be, for example, number entry (pressing a key on a number pad), or a mode transition (for example pressing an ok key). Entry of a number may only be permitted if it falls within specific bounds.
- Functions of the form $filter : S \rightarrow C$ will often be used in the model to extract state attributes. The extracted attributes will sometimes be linked to corresponding perceivable elements also represented as attributes. The function p_filter will be used to describe the perceivable counterpart to the filtered value if available, and the predicate $vis\_filter : C \rightarrow \mathbf{B}$ may be used to assert whether a filtered attribute is perceivable. For example, $filter(s)$ could be a value of a variable in the underlying process being controlled by the interface, $p\_filter(s)$ could be a visible attribute that represents the variable and $vis\_filter(s)$ would be true if the underlying variable is visible.
- A function *mode*. This function is a particular form of $filter$ specifically designed to extract state attributes representing modes of the device. The function is in the form $mode : S \rightarrow MS$. Information about modes will be used in the description of some of the property templates to be described in the next sections.

The following sub-section exemplifies the use of the modeling approach for the selected case study.

5.2 PVS model

The PVS model is specified as a set of three PVS theories. The first theory contains common definitions of constants and types. The second theory (hereafter referred to as the *pump theory*) contains a specification of the underlying pump process (cf. Figure 1), which describes basic characteristics that are common across a variety of infusion pumps, syringe drivers and PCA (Patient Controlled Analgesia) pumps. These two theories have been designed to be reused in the modeling and analysis of other similar devices. Types are *parametrized* in these theories, e.g., the type identifying the range of infusion rates is a parameter that can be set when importing the theory. This allows maximum flexibility. These two models can be reused to represent the behavior of a family of infusion devices supporting different ranges of infusion rates. These parameters will come into play when analyzing requirements for the data entry system of the pump, e.g., in Section 6.4.3, as some of the properties hold true only under certain conditions on the range values.

The third theory (referred to as the *interface theory*) describes features of the user interface software and

is particular to this specific device brand. It describes whether attributes are perceivable, how the entry modes are handled by the device, and how the user sets, controls and views the operation of the pump as specified in the pump theory. While this theory is specific to the particular infusion product, parts of it may be reused, for example, in families of the same brand of pump.

The specification of the model uses transition functions to describe actions. These functions take the following form in PVS (where `state` is a PVS record type listing the state attributes of the model):

```
action(st: state): state
```

The circumstances in which the actions are permitted must also be described. This specifies when the action is available to the user. Note that the availability of an action does not indicate that it is obvious to the user that the action is available. Rather, it indicates only that the function can be *activated* by the user. A family of action-indexed predicates of the form:

```
per_action(st: state): bool
```

assert whether actions are permitted. An autonomous function $tick$ describes the effect of updating key underlying state attributes associated with the pump process at discrete intervals. When permissions are used then actions will have the following form of signature in PVS:

```
action(st: (per_action)): state
```

which limits the domain of the function to the subset of states for which the action is permitted. A deterministic modeling approach is used to describe the effect of each transition function. That is, each event/action has one possible effect. The order in which functions are executed is not fixed in the model, i.e., functions can interleave in any order. Transition functions are atomic, i.e., the execution of a transition function is completed before another transition function is executed. This modeling approach is sufficient to represent the actual behavior of the user interface and does not compromise the veracity of the model for the analysis of the considered requirements.

The state attributes of the pump theory are encapsulated within a separate state attribute, `device`, which is itself a PVS record type. The state attributes for the pump process model (e.g., *vtbi*, *infusionrate*, *volume* and *time*) are therefore referenced as `device(st)`‘`vtbi` and so on, where *st* has type `state` and represents the current state of the device.

The developed PVS model of the user interface includes a description of the display elements that are presented on the device screen. Specifically, the display elements indicated in Figure 2 are specified. The attribute `topline(st)` describes the information contained in the top part of the display. Actions $key1$, $key2$ and $key3$ are associated with the function displays (*fndisp1*, *fndisp2* and *fndisp3*, respectively). An array of Booleans ($middisp$) indicates whether information is visible to the user, for example `middisp(drate) = true` means that the infusion rate is visible.

The example of the `pause` function presented in Listing 1 and discussed below illustrates the use of the state attributes to describe the interaction. The `pause` function is defined in the interface theory. It has the effect of pausing the infusion. The function updates various display elements.

```
1  pause(st: (per_pause)): state =
2   st WITH [
3    topline := holding,
4    middisp := LAMBDA (x: imid_type):
5      COND (x = drate) OR (x = dvol)  -> TRUE,
6           (x = dvtbi) OR (x = dtime) ->
7                          device(st)`vtbi /= 0,
8           ELSE -> FALSE ENDCOND,
9    fndisp1 := fvol,
10   fndisp2 := fvtbi,
11   fndisp3 := fnull,
12   entrymode  := rmode,
13   pauselight := TRUE,
14   runlight := FALSE,
15   device  := pause(st`device) ]
```

Listing 1. Transition function `pause`

The display element `topline` is set to display the information “holding” (line 3 in Listing 1), and the function `middisp` specifies that infusion rate and volume are made visible (line 5 in Listing 1), as are vtbi and time if the value of vtbi is not zero (lines 6-7 in Listing 1). The function key labels for $key1$ and $key2$ are set to “volume” and “vtbi” (lines 9-10 in Listing 1). The function key label for $key3$ is blank (line 11 in Listing 1). The entry mode of the device is set to `rmode` (line 12 in Listing 1). This allows the infusion rate to be changed when the pump is paused unless the rate has previously been locked. The pause light is set to on, and the run light switched off (lines 13-14 in Listing 1). Finally, a call to a function `pause`, defined in the pump theory, updates relevant state attributes of the pump process (line 15 in Listing 1). Hence, the `pause` function is overloaded, being defined in both the pump theory and the interface theory. Disambiguation is carried out as in object-oriented programming languages, by checking the type of the function argument.

To complete the specification of the `pause` function, it is necessary to indicate when the function is permitted. The domain of the pause function is specified to be restricted to those states for which `per_pause(st)` is true. This is indicated in the `pause` function using the subtyping notation `(per_pause)` to specify the type of the function argument. The permission function is defined as follows for the case study.

```
per_pause(st: state): bool =
 per_pause(st`device) AND no_button_down(st)
   AND ((topline(st) = infusing)
        OR (topline(st) = dispkvo)
        OR (topline(st) = dispvtbi)
        OR (topline(st) = volume)
        OR (topline(st) = locked))
```

This specifies that the pause function is permitted when:

- The device is switched on and is infusing. These constraints are found in the permission function `per_pause(st`'`device)` defined in the pump theory.
- No other button has been pressed or is being pressed (predicate `no_button_down(st)`).
- The top line display shows "infusing" or "KVO" or "vtbi" or "volume" or "locked".

This permission includes attributes that are visible to the user (e.g., the value of `topline`). This choice is driven by the fact that the developed specification is a description of the device behavior of the user interface, and not a translation of user interface software code. Display elements are therefore used in the model to describe when actions are permitted, as this makes the model more readily understandable without compromising the veracity of the model. The other actions in the PVS model are specified using the same approach.

6 Property Templates Capturing Usability Requirements

The property templates, used in this paper, are generic mathematical formulas designed to help developers to construct conjectures appropriate to the analysis of user interface features. The templates to be considered are: completeness, feedback, consistency, reversibility, visibility and universality. They are based on interaction design guidelines described, for example, by Nielsen [24], Dix and others [25], and Thimbleby [26]. Initial formalizations of some of these guidelines have previously been described in [4], [27] with a model checking context in mind. Our aim is to establish a set of template formalizations that can be translated easily into PVS theorems[4].

This section introduces the templates with illustrations of the use-related concerns captured by each of them. They are initially formulated in general terms using the concepts of actions, states, modes (introduced in Section 5), and a transition relation $transit : S \times S$ that relates states that can be reached by any action. Some templates will be special cases, syntactically, of others (e.g., the visibility and universality templates – Sections 6.5 and 6.6 – are special cases of the state consistency template described in Section 6.3). However, the goal is not to provide a syntactic classification of the formulations (as in for example, [28]) but to support analysts in expressing properties that capture relevant use-related requirements. The templates are designed to be useful, acting as triggers for the analyst.

In this section the templates are described and their instantiations are illustrated using the infusion pump case study introduced in Section 4. Instantiation involves tailoring the template to the characteristics and functionalities of the device under analysis, and formulating (based on the instantiated template) PVS theorems that can then be analyzed using PVS. Part of this process involves defining the relation $transit$ based on the actions supported by the device, and producing precise descriptions of the guards, goals and filters that are relevant to the PVS theory that models the interactive behavior of the device under consideration. This process will be described in more detail in the following sub-sections for each template. Each template will be motivated in part using ANSI/AAMI HE75:2009 [29] as a reference. The relevant sections of this document will be referred to in each template description.

4. Note that the word 'theorem' is used in this section to describe the syntactic element in the PVS theory that translates a property to be checked. It does not indicate that the property has been proved.

6.1 Completeness template

Accessing device features should require less than three interactions (Section 21.4.3: User interface structure [29]). Otherwise the user will consider the feature "buried" in the user interface. The completeness template is designed to address this type of concern, i.e., that the software allows the user to reach significant states in one (or a few) steps. For example, being able to reach the "home" screen (where all infusion parameters are presented) from any device screen in one step is a completeness property.

Completeness

$$\forall s \in S : guard(s) \wedge \sim goal(s) \\ \Rightarrow \exists a \in A : per(a)(s) \wedge goal(a(s)) \qquad (1)$$

The template asserts that there is always a user action (specified as a) that transforms a state satisfying a predicate $guard : S \rightarrow \mathbf{B}$ into a state satisfying predicate $goal : S \rightarrow \mathbf{B}$. For instance, the $goal$ could be true when the system is in some defined home screen. There may be situations where it is not possible to reach the home screen in one step or, for example, the property may only be relevant when the pump is paused. The guard used in the template makes it possible to exclude these cases, limiting the property to the situation of concern. It is envisaged that a final formulation of the template is always developed in discussion with human factors specialists and domain experts. This consultation will consider the implications of these exceptions and determine appropriate exclusions and definitions.

6.1.1 Instantiation of the completeness template

An instantiation of the completeness template is now illustrated by considering the possibility of reaching certain home screens when the pump is infusing or paused. These home screens allow the operator to watch relevant pump variables, and use the chevron keys to adjust the infusion rate (unless the infusion rate has previously been locked).

The relevant home screen, when the pump is paused, is signified to the user by a top line of "holding" or "set rate". The device shows this screen automatically at start-up, or after an alarm has occurred when there has been no user activity for a period. When the pump is

infusing, on the other hand, the home screen is signified by a top line display showing "infusing". In this screen, the clinician can update the infusion rate (unless the infusion rate is locked).

6.1.2 PVS translation of the completeness template

A PVS theorem for the paused pump is first considered. The first step in creating the PVS theorem involves specifying the *guard* and *goal*. The goal can be specified as a PVS predicate `goal_hosr`[5] indicating that the top line should be "holding" or "set rate".

```
goal_hosr(st: state): bool =
    topline(st) = holding OR topline(st) = setrate
```

The only constraint imposed by `guard` is that the pump is switched on and paused (i.e., not infusing).

```
simple_guard_hosr(st: state): bool =
device(st)`powered_on? AND NOT device(st)`infusing?
```

The completeness property `simple_complete_to_hosr` is then expressed using the PVS syntax as follows. The property aims to check that one of the user actions key_1 or key_3 will always reach the goal.

```
simple_complete_to_hosr(st: state): bool =
 (simple_guard_hosr(st) AND NOT goal_hosr(st))
  IMPLIES (per_key1(st) AND goal_hosr(key1(st))
       OR (per_key3(st) AND goal_hosr(key3(st))))
```

Finally, a PVS theorem is formulated that is suitable to prove the template. It needs to consider only *accessible* states, that is states that can be reached from the initial state of the device using the actions that the device supports. The PVS theorem (shown in Listing 2) is therefore formulated as a structural induction:

- The base step (lines 3-4 in Listing 2) proves that the property is true of the initial state (`init?(st)`) in which all the state attributes are initialized and the device is switched *off*.
- The induction step (lines 5-7 in Listing 2) assumes the property is true of a state (`pre`) and aims to prove that the property is also true for all states `post` related to `pre` by the transition relation `state_transitions_release`. This relation is true if `post` is related to `pre` by any of the permitted actions supported by the device. In this case actions related to the chevron keys are combined with a release action to specify that the chevron key is permitted only when no other key is held down. This reflects the actual behavior of the device. The full specification, referred to in Section 1, contains the definition of this relation.

The completeness theorem that follows from these deliberations is as follows:

5. Note that in this formulation of `goal_hosr` it is not required that the key parameters are also visible. This requirement could be added to the goal, though for illustration it makes the property more complex because *time* and *vtbi* are only actually visible if *vtbi* is non-zero.

```
1 simple_comp_pause: THEOREM
2  FORALL (pre, post: state):
3   (init?(pre) IMPLIES
4                  simple_complete_to_hosr(pre))
5    AND ((state_transitions_release(pre, post)
6          AND simple_complete_to_hosr(pre))
7          IMPLIES simple_complete_to_hosr(post))
```

Listing 2. Completeness theorem

6.1.3 PVS analysis of the completeness template

The PVS theorem cannot be proved in the form described in the previous section. PVS generates counter-examples indicating that the guard predicate `simple_guard_hosr` admits states for which the theorem is false. Refinement of the theorem, to exclude the conditions under which the property fails, may be appropriate. Exclusion can be justified if it can be demonstrated that these conditions are irrelevant for the considered property. The process necessary to refine the completeness theorem is now illustrated.

Consider the sub-goal of the completeness theorem shown in Listing 3. Sequents [-1], ..., [-5] are assertions that are true, and can be interpreted as the hypotheses under which the sub-step is being analyzed. Sequents [1] and [2] are goals. If either goal is true then the sub-goal of the theorem is successfully verified.

```
1 simple_comp_pause.2.6.1.5:
2 [-1] ((topline(pre!1) = options) AND
3       (fndisp1(pre!1) = fok) AND
4       (entrymode(pre!1) = qmode))
5 [-2]  device(pre!1)`powered_on?
6 [-3]  no_button_down(pre!1)
7 [-4]  post!1 = key1(pre!1)
8 [-5]  simple_complete_to_hosr(pre!1)
9 |-------
10 [1]   device(pre!1)`infusing?
11 [2]   simple_complete_to_hosr(post!1)
```

Listing 3. Sub-goal of the completeness theorem

The symbolic constants `pre!1` and `post!1` are called *skolem* constants and are obtained from `pre` and `post` when removing the universal quantifier over states. Hence, `pre!1` represents the state before taking the transition, and `post!1` represents the state reached by the model after the transition is taken. This specific sub-step in the proof involves the action key_1 (this can be seen by inspecting sequent [-4]). The top line for the state before the transition shows "options" (line 2 in Listing 3), the function display for key_1 shows "ok" (line 3 in Listing 3), and the entry mode of the device is $qmode$ (line 4 in Listing 3).

The case to be proved starts from the hypothesis that `simple_complete_to_hosr` is true for `pre` (sequent [-5]), and attempts to prove that either the device is infusing (sequent [1]), *or* that the property is true for `post` (sequent [2]). The value of `post` can be seen in sequent [-4], i.e., `post` is obtained from `pre` by pressing key_1.

In attempting the automatic proof of this sub-goal, PVS stops and presents an unprovable sub-theorem (see

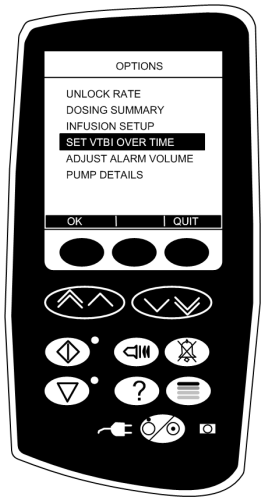

Fig. 3. Selecting vtbi over time when infusion rate is locked

Listing 4) which can be regarded as a counter-example. The important elements in Listing 4 are as follows. Sequent [-7] asserts that the user has selected the menu entry "set vtbi over time" (see Figure 3 for an illustration of the situation). The entry mode is *qmode* (line 10 in Listing 4). Sequent [-8] asserts that the infusion rate is locked. Sequent [-9] asserts that, after pressing key_1, the transition leads to a new state $post!1$ in which the top line shows "locked" (line 5 in Listing 4) and all function key displays are blank (lines 7-9 in Listing 4), meaning that they are not enabled.

```
1  {-7} setvtbiovertime?(optionsmenu(pre!1)
2                        (qcursor(pre!1)))
3  {-8} rlock(pre!1)
4  {-9} post!1 = pre!1 WITH [
5          topline := locked,
6          middisp := LAMBDA (x: imid_type): FALSE,
7          fndisp1 := fnull,
8          fndisp2 := fnull,
9          fndisp3 := fnull,
10         entrymode := qmode ]
11  ... % more sequents omitted
```

Listing 4. Counter-example for the completeness theorem

This particular counter-example is not a concern because this device state is temporary. In fact, the PVS model (reflecting the behavior of the device) specifies that the device automatically returns to the previous state. This behavior is described in the `tick` function, which models automatic actions taken by the device. It can therefore be safely excluded by modifying `simple_guard_hosr`.

```
guard_hosr(st: state): bool =
 device(st)`powered_on? AND
   NOT device(st)`infusing? AND
   (topline(st) /= locked AND
    NOT (topline(st) = dispvtbi
         AND (entrymode(st) = bagmode
              OR entrymode(st) = tbagmode)))
```

The new guard excludes situations when entering *vtbi* where actions keep the device in an entry mode in which *vtbi* is entered. For example when selecting an infusion bag, the only exit is to the "outer" mode where the *vtbi* value can be further modified using chevron keys.

Further completeness theorems have a similar format. For example, in the case that the pump is infusing, "home" is to return to the entry mode in which the top line shows "infusing" or "kvo". The initial guard simply indicates that the pump is switched on and infusing.

```
guard_infuse(st: state): bool =
 device(st)`powered_on? AND device(st)`infusing?
```

The goal in this case is that the device displays a top line of "infusing" or "kvo" (the latter is the display when vtbi has been exhausted and the pump is continuing the infusion to keep the vein open).

```
goal_infuse(st: state): bool =
 topline(st) = infusing OR topline(st) = dispkvo
```

In this case action key_3 should be sufficient to enable users to move to the home screen in one step. Through a similar series of attempts and analysis of counter-examples, it can be found that this initial formulation of the guard is not strong enough, and again it is necessary to prove the theorem under the hypothesis that the top line is not showing locked and the device is not in *bagmode* while infusing. The final formulation of the guard that makes it possible to prove the theorem is as follows:

```
guard_infuse(st: state): bool =
 device(st)`powered_on? AND
 device(st)`infusing? AND ((topline(st) /= locked)
  AND NOT ((topline(st) = dispvtbi) AND
           (entrymode(st) = bagmode)))
```

6.2 Feedback template

Information presented in the user interface should allow the clinician to understand the effect of important actions (Section 5.2: Types of use errors [29]). The feedback template addresses this concern by describing properties that demonstrate that state changes are perceivable. Feedback may be considered in two contexts. The first is *state feedback*, which requires that any change in the state (usually specific attributes of the state rather than the whole state) is perceivable to the user.

State feedback

$$\forall s_1, s_2 \in S,\ guard(s_1) \wedge guard(s_2) \wedge filter(s_1) \neq filter(s_2) \Rightarrow visible\_change(s_1, s_2) \quad (2)$$

More specifically, *action feedback* requires that a specified action always has an effect that is perceivable to the user. The expression $visible\_change$ is a place marker that describes how the perceivable change is represented in the model. Two choices can be used to instantiate the expression. The choices differ in how perceivability of attributes is specified in the model.

Action feedback

$$\forall a \in S \to S,\ \forall s \in S : per(a)(s) \wedge guard(s) \wedge (filter(s) \neq filter(a(s))) \Rightarrow visible\_change(s, a(s)) \quad (3)$$

The first case of $visible\_change$ assumes that there are perceivable attributes ($p\_filter(s)$) that represent the values $filter(s)$ (for example, numerals that represent the numbers). In this case:

$$visible\_change(s1, s2) := (p\_filter(s1) \neq p\_filter(s2))$$

In the second case, the specification simply indicates that the attribute extracted by the filter is perceivable, that is $vis\_filter(s)$ is true for $s \in S$ if $filter(s)$ is perceivable. In this situation,

$$visible\_change(s1, s2) := (vis\_filter(s1) \wedge vis\_filter(s2))$$

6.2.1 Instantiation of the feedback templates

An instantiation of the feedback templates can be used to check whether means are provided by the user interface of the pump to allow a clinician to monitor the infusion process. This involves making relevant changes of basic pump variables (i.e., infusion rate, VTBI and time) visible to the clinician. Hence, example instantiations of the feedback templates are the following.

- If a pump variable (e.g., infusion rate) is changed, then that change is visible in the user interface.
- If the entry mode changes, then that change is made visible in the user interface.

6.2.2 PVS translation of the state feedback template

When considering feedback related to infusion rate, the relevant filter is:

```
filter_rate(st: state): irates =
  device(st)`infusionrate
```

where `irates` is a subtype of reals defining the range of rate values supported by the device. The visibility or otherwise of this attribute is defined by a Boolean. As was noted in Section 5.2 the developed model contains a Boolean function *middisp* that specifies whether this and other key attributes are visible or not. This attribute can be used to define a predicate `vis_filter_rate` representing the filter used in the state feedback template:

```
vis_filter_rate(st: state): bool =
  middisp(st)(drate)
```

The feedback property then becomes:

```
state_feedback_simple(pre, post: state): bool =
 (filter_rate(pre) /= filter_rate(post))
  IMPLIES (vis_filter_rate(pre)
           AND vis_filter_rate(post))
```

This property is not true for all states. A structural induction is required. The feedback theorem then is of the following form:

```
feedback_rate_theorem: THEOREM
 FORALL (pre, post: state):
   state_transitions(pre, post)
   AND guard_vis_rate(pre) AND guard_vis_rate(post)
   AND state_feedback_simple(pre, post)
```

The transition relation `state_transitions` includes all actions – in this case, it is not necessary to include the additional constraint that the chevron key has been released, as is done in Section 6.1.2 for the completeness template. Note also that `guard_vis_rate` has been included which in this initial form of the theorem is trivially true. The guard is further refined in the next sub-section.

6.2.3 PVS analysis of the state feedback template

The initial attempt to prove `feedback_rate_theorem` generates a counter-example indicating that when the top line shows "vtbi over time" (sequent [-6] in Listing 5) a temporary attribute *newrate* is visible (sequent [-8] in Listing 5) and not the actual infusion rate. The reason for this is that the actual infusion rate value will be updated only after the clinician confirms the value by pressing the ok button.

```
[-1]  device(pre!1)`powered_on?
[-2]  nob?(which_press(pre!1))
[-3]  pressed(pre!1) = 5
[-4]  fok?(fndisp1(pre!1))
[-5]  middisp(pre!1)(dnewvtbi)
[-6]  vtbitime?(topline(pre!1))
[-7]  ttmode?(entrymode(pre!1))
[-8]  middisp(pre!1)(dnewrate)
[-9]  middisp(pre!1)(dnewtime)
[-10] newtime(pre!1) = 0
 ... % more sequents omitted
```

Listing 5. Counter-example for the feedback theorem

The fact that the displayed value of the infusion rate is temporary before the confirmation action is important because if the machine is switched off then the modified value of infusion rate is lost. This subtle corner case needs to be discussed with domain experts to make sure that it is unlikely to lead to dangerous use errors. The team might recommend, for example, that when switching off the software automatically saves the modified value and on restart prompts the user. The current design asks the user whether the pump variables are to be cleared or restored to the values they held at close down. This request could include information about any temporary variables in cases such as this one. Prototypes based on the formal model (using PVSio-web [9]) can be used to present the issues to the designers, domain and human factors experts.

The following guard can be used to ease the exploration of additional corner cases. The proof process involves excluding the highlighted case and continuing the theorem:

```
guard_rate(st: state): irates =
   NOT (topline(st) = vtbitime)
```

With the above guard, the proof of the theorem can be completed successfully, indicating that the case with "vtbi over time" is the only corner case.

6.2.4 PVS translation of the action feedback template

It is to be expected that key user interface variables show change when the infusion process is running. The action feedback template is instantiated for the *tick* action of the PVS model, which represents progress of the ongoing infusion process – after each *tick*, the time to infuse and the volume to be infused are reduced. When the pump is paused, these two state attributes no longer change, and the *tick* action updates the time delay since the last user action.

A preliminary guard used to formulate the property requires that the pump must be infusing, and VTBI has not been exhausted (otherwise the pump infuses to keep the vein open while alarming to alert the user). Hence the following guard is employed.

```
guard_tick(st: state): bool =
  device(st)`infusing? AND NOT device(st)`kvoflag
```

The pump attributes that are the primary focus are the ones that change, namely vtbi and time. Although infusion rate should remain unchanged, this state attribute is also considered here because of its importance to the clinician in the infusion calculation. Hence the relevant predicate is defined as follows:

```
vis_tick(st: state): bool =
  middisp(st)(dvtbi) AND middisp(st)(drate) AND
    middisp(st)(dtime)
```

The PVS theorem is as follows, where the `tick` action is only defined for specific states as constrained by the permission `per_tick`.

```
action_feedback_tick: THEOREM
 FORALL (st: state):
  (per_tick(st) AND guard_tick(st) AND
   guard_tick(tick(st))) IMPLIES vis_tick(tick(st))
```

The proof is attempted in this case for all possible states (as opposed to all accessible states). If the proof fails, depending on the type of counter-examples returned by the theorem prover, either the guard is refined, or the theorem is reformulated as a structural induction, or genuine design defects are identified that require redesigning device functions. Refinement may also involve more than one of these possibilities.

6.2.5 PVS analysis of the action feedback template

The initial formulation of the theorem fails. A revised guard is developed based on the counter-examples returned by PVS. When the device is not connected to mains power, that is flag *ac_connected* is false, then a warning may appear momentarily in the top line that conceals the displayed values of the infusion rate. Furthermore, this display does not show the pump attributes immediately after vtbi is first exhausted. In that case a display is generated with top line of "vtbi done" and a function key display "cancel" is associated with key 3.

```
guard_tick(st: state): bool =
  device(st)`infusing? AND NOT device(st)`kvoflag
  AND device(st)`ac_connect
  AND topline(st) /= vtbidone
```

The refined property can be proved successfully. The exception introduced, however, indicates that when the pump is not connected to the mains, feedback attributes may be concealed when the error message is displayed. This could be a cause for concern. Running infusion pumps on battery by mistake is a common problem in hospitals. For this specific infusion pump, however, it could also be argued that because an alarm is shown to the user when the pump runs on battery the potential hazard will be avoided. It could also be argued in this context that the user will be aware of the situation and therefore recognize the fact that the pump variables may be obscured from view. Again, developers need to discuss these arguments with human factors specialists and domain experts to be assured that the constraints imposed by the guard are reasonable.

Further action feedback properties have been proved, e.g., related to the use of the *switch* action that changes the device's power source. This action toggles flag `ac_connected` in the model. The feedback property requires that the value of `ac_connected` is always reflected in the mains and battery lights. The attributes (`ac_light` and `battery_light`) are two Booleans that specify whether the mains and battery lights are on. This property shows that changing from mains to battery and vice-versa is indicated through the feedback of the two status lights. It does not show that the status lights have a consistent effect (this will be proved as a visibility property in Section 6.5).

```
guard_switch(st: state): bool = per_switch(st)
```

The relevant *action feedback* theorem expressed as follows can be proved automatically in PVS:

```
action_feedback_switch: THEOREM
 FORALL (st: state):
  guard_switch(st) IMPLIES
   (battery_light(st) /= battery_light(switch(st))
    AND (ac_light(st) /= ac_light(switch(st))))
```

6.3 Consistency template

Users quickly develop a mental model that helps them interact with a user interface. To encourage the development of an accurate and complete mental model, a user interface should be consistent in its layout, screen structure, navigation, terminology, and control elements (Section 21.4.13: Consistency [29]). The *action consistency* and *state consistency* templates address these concerns. Action consistency is defined to require that the action consistently changes state attributes, for example irrespective of what the mode is. State consistency requires

that all states reachable within the device have a common property, e.g., the function display "quit" is always associated with the same function key.

The *action consistency* template is formulated as a property of either a single action, or of a group of actions (we will refer to them as Act) which may exhibit similar behaviors. A relation $\varphi : C \times C$ connects a filtered state, before an action occurs (captured by $pre\_filter : S \times MS \rightarrow C$), with a filtered state after the action (captured by $post\_filter : S \times MS \rightarrow C$).

Action Consistency

$$\forall a \in Act, s \in S, m \in MS : \\ guard(s, m) \wedge \\ pre\_filter(s, m) \ \varphi \ post\_filter(a(s), m) \quad (4)$$

Note that both the filters *may* depend on the mode. As the template is expressed, the filters are both assumed to be dependent on mode. In practice this will not always be true – particularly if the effect of action is to change mode. In the example to be used as illustration, for the actions that are considered, the mode *does* change as a result of the action but the filter that is used as $post\_filter$ depends on the mode before the action is taken. It may be appropriate in other cases to relax the mode constraint, either completely or on one of the filters. The relation φ is in the set $\{=, \neq, <, \leq, >, \geq\}$. The guard on states is also sometimes extended within this template to be sensitive to mode ($guard : S \times MS \rightarrow \mathbf{B}$).

The *state consistency* template more generally requires that for all accessible states, possibly constrained by a guard, an $invariant : S \rightarrow \mathbf{B}$ is always satisfied.

State Consistency

$$\forall s \in S : guard(s) \Rightarrow invariant(s) \quad (5)$$

6.3.1 Instantiation of the consistency templates

A consistency property may require that in a given mode (defined in the guard), specific attributes (defined in the filters) are never changed, or alternatively always changed or may demonstrate that all state changes satisfy consistent properties. Examples of consistency properties for the illustrative device are as follows.

(1) Actions designated as function keys always change the entry mode.
(2) A chevron key will always change the pump variable relevant to the entry mode ($entrymode$) if that mode is relevant to entry of that type of pump variable. Note that in some modes chevron keys are used to navigate the cursor. Different properties will apply in these cases.
(3) When a function key is associated with a soft display of ok then the value of the relevant pump variable is changed, when that action is taken, to the value set within the entry mode.
(4) When a function key shows a soft display of $quit$ then the value set in the mode is discarded, when that action is taken, and the pump variable remains the value it had when it entered the mode.
(5) The same function keys are always associated with the same soft key displays.

6.3.2 PVS translation of the consistency templates

An example instantiation of *state consistency* is that $quit$ never appears as the function key display for key_1 or key_2. This is easy to express using the state consistency template. The invariant in this case is

```
f3quit_invariant(st: state): bool =
  fndisp1(st) /= fquit AND fndisp2(st) /= fquit
```

The theorem uses the invariant property within a structural induction as follows:

```
f3quit_consistent_theorem: THEOREM
FORALL (pre, post: state):
    (init?(pre)  => f3quit_invariant(pre)) AND
     (state_transitions(pre, post) AND
    f3quit_invariant(pre) IMPLIES
                f3quit_invariant(post))
```

An example instantiation of *action consistency* template relates to the chevron keys that increment or decrement values, or navigate menus up or down, depending on entry mode. We formulate a requirement that these actions shall never change entry mode. The pre_filter and $post\_filter$ both extract the entry mode, and are of the form:

```
filter_entrymode(st: state): emodes = entrymode(st)
```

In this case it is not necessary to use mode as a parameter as the filter definition is the same in all modes. A transition function `state_transitions_chevrons` is defined that relates a `pre` state to a `post` state by a chevron action (*sup*, *sdown*, *fup*, *fdown*). The relation φ in this case is equality. The theorem that instantiates the consistency template is:

```
consistency_entrymode_theorem_chevronkeys: THEOREM
   FORALL (pre, post: state):
     (state_transitions_chevrons(pre, post))
       IMPLIES (filter_entrymode(pre) =
                          filter_entrymode(post))
```

6.3.3 PVS analysis of the consistency templates

We consider the ok example (3) shown in Section 6.3.1 in more detail. The guard requires that key_1 is permitted and that the function display shows ok. The guard is parametrized by entry mode and excludes the case of entry mode being $vttmode$ (i.e., vtbi is being entered in vtbi over time mode, see Table 1). The failure of the theorem in the case of $vttmode$ is because the system supports a sequence of actions. Updates are not made until **both** $vtbi$ and $time$ have been entered. At the interim stage defined by $vttmode$ the process is not complete and should make a smooth transition to $ttmode$ (i.e., when time is being entered). The example infusion device allows entry of **both** $vtbi$ and $rate$ **and** $vtbi$ and $time$. However the mechanism for entry in each case

is different. The differences between the two are sufficiently significant that they would lead to a discussion with relevant parties about whether the inconsistency is a problem.

```
guard_em_ok(em: emodes, st: state): bool =
  per_key1(st) AND fndisp1(st) = fok
  AND entrymode(st) = em
  AND entrymode(st) /= vttmode
```

Other counter-examples include when the *ok* function key display does not appear. This happens in the cases of entry modes when infusion rate is being updated (that is *rmode* and *infusemode*). It also occurs in the bag modes (*bagmode* and *tbagmode*) when the temporary value of vtbi is updated with the bag specified by the selected menu item before returning to the vtbi entry modes (*vtmode* and *vttmode*). The pump variable vtbi is not updated using *ok* until exiting *vtmode*. In the case of *vttmode*, transition is made to a mode in which time is updated and the pump variable is not updated at that stage (hence the exception). The "real" value filter extracts the actual attribute that is updated and used in the pump process for each of these modes.

The filters used in establishing the consistency both depend on the entry mode *before* the action is taken. The "temporary" value before key_1 is pressed is defined in this case as:

```
temp_mode_filter(em: emodes, st: state): real =
 COND
  em = rmode -> device(st)`infusionrate,
  em = infusemode -> device(st)`infusionrate,
  em = vtmode -> newvtbi(st),
  em = vttmode -> newvtbi(st),
 (em = bagmode OR em = tbagmode) ->
    COND bagscursor(st) = 0 -> 0,
         bagscursor(st) = 1 -> 50,
         bagscursor(st) = 2 -> 100,
         bagscursor(st) = 3 -> 200,
         bagscursor(st) = 4 -> 250,
         bagscursor(st) = 5 -> 500,
         bagscursor(st) = 6 -> 1000,
         bagscursor(st) = 7 -> 1500,
         bagscursor(st) = 8 -> 2000,
         ELSE -> 3000 ENDCOND,
  em = ttmode -> newtime(st),
  ELSE -> device(st)`infusionrate
 ENDCOND
```

and the "real" filter specifies the actual pump parameters that are updated when the *ok* action is taken. Again the filtered value depends on the entry mode before the action.

```
real_mode_filter(em: emodes, st: state): real =
 COND
  em = rmode -> device(st)`infusionrate,
  em = infusemode -> device(st)`infusionrate,
  em = vtmode -> device(st)`vtbi,
  em = vttmode -> device(st)`vtbi,
  em = bagmode -> newvtbi(st),
  em = tbagmode -> newvtbi(st),
  em = ttmode -> device(st)`time,
  ELSE -> device(st)`infusionrate
 ENDCOND
```

The instantiation of the consistency template leads to the following theorem:

```
consistency_ok_em: THEOREM
 FORALL (em: emodes, st: state):
  guard_em_ok(em, st) IMPLIES
   temp_mode_filter(em, st) =
    real_mode_filter(em, key1(st))
```

Further consistency properties can be proved subject to relevant constraints applied through specified guards.

- When the function display shows $quit$ then key_3 takes the top line to show "holding".
- When top line is volume and the infusion pump is not infusing then key_2 always changes volume infused to zero and changes the entry mode to $rmode$.

6.4 Reversibility template

Users may perform incorrect actions, and the device should provide appropriate reversing functions that allow users to easily stop, modify, and restart the automated processes in the case of problems or abnormal situations (Section 20.2.4: User understanding of the automation [29]). An example of such a function is an "undo" function in an editor, or in the case of a number entry action, an "increment value" action to reverse the effect of a "decrement value". The reversibility template is formulated for a group of actions $Act \subset S \to S$ using $guard : S \to \mathbf{B}$, and a $filter : S \to C$ relevant to the entry mode. For each $a \in Act$, there corresponds a $b \in Act$ such that:

Reversibility

$$\forall s \in S : guard(s) \Rightarrow filter(b(a(s)) = filter(s)) \quad (6)$$

Note that this property could be formulated to be sensitive to mode. We chose to deal with each mode separately to ease formulation and verification of the property.

6.4.1 Instantiation of the reversibility template

Reversibility can be used to ensure that data entry with chevron keys allows the clinician to undo a value change with a single reversing action in the example infusion pump. We consider one example to illustrate the process, namely that "single chevron up" can be used to reverse the effect of "single chevron down".

6.4.2 PVS translation of the reversibility template

A guard needs to be specified to construct the PVS theorem for infusion rate entry requiring that the device is ready to enter the infusion rate (`rate_entry_ready`). Additionally, the guard should require that the relevant action, and its reverse action, are enabled. The guard can be specified as follows in PVS:

```
guard_supsdown_rate(st: state): bool =
 rate_entry_ready(st) AND per_sdown(st)
  AND per_sup(release_sdown(sdown(st)))
```

where rate_entry_ready takes into account the fact that rate values can be entered only when device is switched on and the infusion rate is not locked (line 2 in Listing 6), and that the entry mode actually allows the clinician to enter rate (lines 3-4 in Listing 6).

```
1 rate_entry_ready(st: state): bool =
2  switchedon?(st) AND NOT rlock(st)
3  AND (entrymode(st) = rmode
4       OR entrymode(st) = infusemode)
```

Listing 6. Guard for the reversibility theorem

6.4.3 PVS analysis of the reversibility template

The reversibility theorem is proved by first considering pairs of chevron keys, and a specific parameter (vtbi, rate or time). Proving each theorem based on the template initially results in failure. The failures indicate anomalies at certain values or ranges of values. These are compensated by augmenting the guard as will be illustrated. For example, let us check whether *sup* can be used to undo *sdown*, in the case of infusion rate:

```
supsdown_rate: THEOREM FORALL (st: state):
 guard_supsdown_rate(st) IMPLIES
  filter_rate(release_sup(sup(
    release_sdown(sdown(st))))) = filter_rate(st)
```

This part of the reversibility theorem fails. The first counter-example that reveals an issue is as follows:

```
[-13] device(st!1)`infusionrate < 100
[-14] (ceiling(10 * device(st!1)`infusionrate)
                        - 1) / 10 > maxrate
[-15] holding?(holding)
  |-------
 [1]  device(st!1)`infusing?
 [2]  device(st!1)`vtbi = 0
 [3]  rlock(st!1)
 [4]  maxrate = 0
```

This counter-example suggests that it is necessary to constrain the parameterized constant *maxrate* (sequent [4]) – the theory is parametrized to allow the analysis of different designs (as discussed in Section 5.2). Experimentation with the device under analysis and other similar devices indicates that a reasonable hypothesis is to set the maximum rate to be at least 1000. This constraint is added to the existing guard.

As a result of this change, a further counter-example is revealed:

```
 [-15] device(st!1)`infusionrate < 100
 [-16] maxrate > 1000
  |-------
 [1]   device(st!1)`infusing?
 [2]   (device(st!1)`vtbi = 0)
 [3]   rlock(st!1)
 [4]   (ceiling(10 * device(st!1)`infusionrate)
                         - 1) / 10 > maxrate
 [5]   (ceiling(10 * device(st!1)`infusionrate)
                               - 1) / 10 < 0
 [6]   (ceiling(10 * device(st!1)`infusionrate)
                               - 1) / 10 = 0
 [7]   floor(ceiling(10 *
               device(st!1)`infusionrate)) / 10 =
                     device(st!1)`infusionrate
```

This particular branch of the proof is focusing on the case infusionrate < 100 (see sequent [-15]). The goal to prove in this branch is in sequent [7]:

```
floor(ceiling(10 *
           device(st!1)`infusionrate)) / 10 =
                     device(st!1)`infusionrate
```

Sequents [4] – [6] suggest that additional conditions need to be introduced to take into account the decimal accuracy supported by the device. In fact, in the specific range considered in the branch, the device allows only one decimal place.

A further attempt to prove the theorem with this additional constraint returns a further counter-example. Indeed in proving all the reversibility theorems relating to chevron keys several anomalies can be identified.

- Applying double chevron up to 99 and then applying double chevron down produces 90.
- Applying double chevron down to 100 and then applying double chevron up produces 91.
- Applying single chevron up to 99.9 and then applying single chevron down produces 99.
- Applying single chevron down to 100 and then applying single chevron up produces 99.9.

The team of experts would likely argue that these anomalies are unacceptable in that they increase the likelihood of failure when attempting to recover from data entry error. In this analysis our aim was to scope the problem, recognizing the cases where these anomalies occur. Therefore in the lowest range case, being used here as an example, the following constraint was used.

```
maxrate > 1000 AND v <100 AND v >= small_step/10
  AND floor(v*10) = v*10 AND ceil_rate(v*10) = v*10
```

where v is a shorthand for device(st)`infusionrate.

It is clear that this theorem has highlighted issues about the way the number entry behaves that could affect the usability of the device. After producing this analysis, we noticed that new releases of the device firmware have fixed these corner cases. The chevron keys now have the required reversing effect and therefore the more general theorem can be proved for the new firmware.

6.5 Visibility template

Visual or auditory cuing should be used to draw the user's attention to important information necessary for correct decision-making (Section 25.3.3: Design guidance related to cognitive capabilities and limitations [29]). The visibility template is designed to help the analyst identify situations where users must be made aware of relevant status information about the system. It does this by describing a relation between relevant state attributes (which may not necessarily be visible to the user) and user interface elements that are perceivable. This template complements the feedback templates described in Section 6.2, which deal with *awareness of changes*.

Visibility

$$\forall\, s1, s2 \in S : transit(s1, s2) \wedge guard(s1) \wedge visible(s1) \Rightarrow visible(s2) \quad (7)$$

The predicate $visible$ relates the filtered attribute(s) ($filter$) to the relevant visible attributes ($p\_filter(s)$):

$$visible(s) \coloneqq (filter(s) = p\_filter(s))$$

6.5.1 Instantiation of the visibility template

An example instantiation of the visibility template for infusion pumps is a property that requires that the status of the pump process is always unambiguously mirrored in the user interface. This includes, for example, displaying the power status (on battery, or connected to mains) and lighting up the "paused" light and "run" light according to the status of the pump process.

6.5.2 PVS translation of the visibility template

The property to be proved in this case is:

```
visible_run(st: state): bool =
  run_filter(st) = run_p_filter(st)
```

`run_filter` and `run_p_filter` are defined as:

```
run_filter(st: state): bool =
device(st)`powered_on? AND device(st)`infusing?
```

and

```
run_p_filter(st: state): bool =
  runlight(st) AND NOT pauselight(st)
```

and the PVS theorem is:

```
visible_run_theorem: THEOREM
 FORALL (pre, post: state):
  (init?(pre) => visible_run(pre))
   AND ((state_transitions(pre, post)
   AND visible_run(pre)) IMPLIES visible_run(post))
```

The guard has been omitted in the formulation because no constraints are required to complete the proof of this theorem. Further visibility properties can be formulated to demonstrate the visual distinctness of entry modes. The following property provides the link between entry mode $rmode$ and the top line display.

```
visible_rmode(st: state): bool =
  guard_vis_rmode(st) IMPLIES
    rmode_filter(st) = rmode_p_filter(st)
```

Filters and guards are defined as follows:

```
rmode_filter(st: state): bool =
  entrymode(st) = rmode
```

```
rmode_p_filter(st: state): bool =
  topline(st) = holding OR topline(st) = setrate
```

```
guard_vis_rmode(st: state): bool =
  topline(st) /= locked
```

The case that is excluded by the guard is when the infusion rate is locked and the user presses a chevron key – the top line shows a temporary screen where the top line is "locked".

6.6 Universality template

Specific guidelines are concerned with the design of user interface elements such as *soft keys* (e.g., on-screen labels and soft keys should be consistent between data screens), *knobs* (e.g., rotating the knob in a particular direction should change a value in a particular way), and *touchscreen user interfaces* (e.g., whenever possible, touch targets should be placed in the same location on every screen) (Section 21.4.11: Special interactive mechanisms [29]).

Universality captures these concerns. It is designed to be useful when the analyst requires that focused state attributes always have defined values. Universality is a particular example of consistency designed to encourage the analyst to consider these circumstances. Universality differs from visibility because it is concerned with the relation between perceivable attributes or between other state attributes, for example mode and internal state attributes. Visibility on the other hand always relates a state attribute to a perceivable state attribute. The formulation of the universality template is as follows:

Universality

$$\forall\, s1, s2 \in S : transit(s1, s2) \wedge guard(s1) \wedge universal(s1) \Rightarrow universal(s2) \quad (8)$$

where

$$universal(s) \coloneqq (filter1(s) = filter2(s))$$

6.6.1 Instantiation of the universality template

A universality property for the example is that a particular top line display is always associated with the same function key displays. In this case two sets of display attributes are related. For example, we may want to prove that whenever the top line shows "volume", the function displays for the three action keys are "blank", "clear" and "quit" respectively, see Figure 4.

6.6.2 PVS translation of the universality template

The universality property discussed in the previous subsection involves instantiations such as the following:

```
pred_filter_volume_keys(st: state): bool =
 (topline(st) = volume)
```

```
pred_filter_keys_volume(st: state): bool =
  fndisp1(st) = fnull AND fndisp2(st) = fclear
   AND fndisp3(st) = fquit
```

Hence, the universality property becomes:

```
universality_volume_keys(st: state): bool =
    pred_filter_volume_keys(st) =
     pred_filter_keys_volume(st))
```

This property can be proved in PVS using structural induction.

Fig. 4. The same function keys are always shown when top line is "volume".

7 Discussion

The techniques described in the paper have been applied to medical devices but are clearly applicable to a broader range of systems. Preliminary studies have explored aspects of a nuclear power interface [30] and the flight control unit of a large commercial aircraft [31]. It is clear that there are many real systems whose user interfaces can be analyzed in this way. Important features, not yet explored but scheduled for future work, relate to interfaces that depend on access to large scale databases and networked data. Modeling techniques are required to provide suitable abstractions for these extensions as well as property templates that will help developers consider appropriate usability concerns. This is relevant to the current example in relation to new developments defined to verify and control the entry of prescriptions for particular medications (in the case of the device used in the case study this is referred to as "Guardrails"). This is a topic also under development as part of the analysis of a newly designed pill dispenser device and analysis of the Integrated Clinical Environment (ICE) [32].

Model checking and theorem proving can both perform analyses of the type described in the paper but at different levels of detail. With the model checking technology that we used, the technology was considered to be easier to understand, but it was often necessary to simplify models, making them more abstract, to make the process tractable. Model checking is algorithmic which means that when the property is true it is not necessary to understand how to prove it is true. When it fails, counter-examples are produced that can be used to correct the model or to change the model or to understand why the property fails. Using the model checking tools, performance deteriorated rapidly as the model grew and then became infeasible to use in an interactive style. In the example used in this paper, performance of the verification tool was an important consideration because an analysis of the full number entry system was required as part of the process. For these reasons, and because the alternative technology was familiar to us, theorem proving was used as the basis for analysis.

Four issues were important in the analysis and are topics of current and future research.

- The model accurately reflected the fielded system (see Section 7.1).
- The process of producing theorems that reflected use-related requirements was capable of mechanization (see Section 7.2).
- Tools were available that would ease the modeling and proof process (see Section 7.3).
- The approach was relevant to actual or proposed certification requirements (see Section 7.4).

We briefly address these issues in this section, mentioning relevant research, and describing current plans for further research.

7.1 Modeling and specification

Our approach assumes that a model is constructed from an existing system. This model was developed by hand using the user manual of the device, a simulation[6] and the device itself. Tools also exist for generating models from program code using transformation rules that guarantee correctness (see [33] and [34]) but program code was not available to us. States of the earlier MAL model [6] on which the PVS specification was based had been analyzed by examining the traces of actions produced by the model checker, as counter-examples, with actual sequences generated by the device itself.

When the template properties were proved of the theory, then any counter-examples discovered, as proof of the theorems were attempted, were compared with the actual device. A prototype was produced automatically, as a further process of validation, from the model to compare the "look and feel" of the actual device with the prototype, see [35] for details. The simulations were indistinguishable from the behavior of the physical device. The only difference between the simulation and the real device was that precise timings differed. This difference, however, is not relevant for the considered use-related properties. The simulations were generated with the aim that the developed device models could be explored by regulator or manufacturer (this allows them to gain confidence that the model correctly represents the actual device behavior). It is of course the case that they only allow an exploration of the paths that the regulator chooses to explore. The same simulations in our case are also used to illustrate what the failure of a property means. Part of the argument to the regulator that this 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.

6. http://cs.swan.ac.uk/~cspo/simulations/medical/infusionpump/agp/ downloaded 8/4/17

The described device is typical of a range of medical devices indicating that the modeling approach scales to devices of this kind. The techniques described can therefore deal with this scale but further work is required to analyze devices and requirements that relate to networked devices as well as more complex data structures. An important limitation of a theorem proving approach is the inability to express, simply, temporal properties. This means that reachability and liveness properties are not simple to analyze with a theorem prover. For this reason it makes sense to consider the complementary use of model checking and theorem proving tools. Further properties, for example relating to time, and to multiple viewpoints within a collaborative configuration of device, are currently being considered.

7.2 Mechanised analysis of the property templates in PVS

Proofs of the properties were developed using a pragmatic approach. In many cases universal quantification of states was first attempted, before moving to a structural induction. The process of proof therefore uses the following heuristics.

1) The PVS theorem is formulated and the proof attempted for all possible states.
2) If the proof fails then either the PVS theorem is refined to exclude irrelevant cases or the theorem is reformulated as a structural induction.
3) If the structural induction fails then the PVS theorem may also be refined to exclude irrelevant cases.

Proof therefore is a process of refinement that takes account of possible exceptions. Counter-examples are discovered while the proof is attempted. This may be because the property is wrongly formulated (as illustrated in some of the cases above), or because the theory fails to represent the behavior of the device accurately, or because the device fails to satisfy the requirement. It is therefore always necessary to consider carefully the nature of the failure identified in the proof. The PVSio-web tool [9] can be used to present and discuss the counter-example with domain experts and human factors specialists. When the failure can be compensated, the theorem is extended by changing the guard or in some cases qualifying the goal. If the device fails to satisfy the requirement with significant consequences this may indicate the need for redesign. However it is also possible that the failure is not significant. For example it may be considered highly unlikely to be an issue in practice or it may be the case that the broader system defends against the discovered weakness. This may be achieved by requiring that an operating procedure must be followed to avoid the circumstance. In this sense, the failure can be compensated. Deciding what to do when a property is not true needs to be carefully evaluated. In the case study under consideration, this was usually not a difficult process. However in some circumstances the appropriate step may be difficult to establish and require further analysis from a human factors perspective, for example through user studies.

The representation of counter-examples is less clear than is the case with model checking tools such as the IVY tool. Typically it requires some understanding of the details of PVS to make full use of them. Further research and development is required to make the nature of the counter-example more transparent to the broader range of analysts. This is discussed again briefly in the next sub-section. At this stage the analysis of counter-examples is a manual process and requires some knowledge of the proof system. As mentioned in the conclusion, an important next step in this research is to produce tools to support this process.

7.3 Modeling and proof tools

Tools are being developed, by the authors and others, that are designed to be accessible to developers who are not specifically expert using formal techniques. These developments include the following.

- Specification templates or patterns are being designed to ease the construction of formal specifications, see for example Bowen and Reeves [36] who focus on specifications of interactive behavior.
- Patterns have been designed to ease the process of generating properties as originally described by Dwyer [37]. The templates used in this paper are based on this work and our earlier work using model checking techniques [27].
- Tools are being developed for presenting proofs and counter-examples to aid comprehension, for example earlier work upon which this paper is based [38].
- Strategies have been devised and mechanized to support proof, for example early work in the context of SCR [39].
- Approaches designed to ease the development of models of interactive systems have been developed, for example Degani [40] uses a statechart based approach to modeling devices as well as the user's model of the device and Berstel and others [41] describe a framework for describing widget level interface behavior.

Tailoring these tools to the particular requirements described in this paper is part of our ongoing research. Further templates would include, for example, those that relate to timing and error recovery. Tools are being developed to facilitate the construction of conjectures from the templates, using an analogous approach to that provided by the IVY tool [27] tailored to PVS conjectures. Presenting counter-examples in a simple format and supporting general tactics for proof are each more complicated. The tools and techniques described in the paper and their developments will only be valuable if they can be used readily by developers, more specifically those whose task it is to produce the documentation and evidence that a design is such that risks associated with it are as low as reasonably practicable. Experience of using

similar techniques with the IVY tool is described in [42]. This work involved a team of developers producing a risk analysis for a dialysis machine. Requirements were developed collaboratively and formulated by a member of the team who was familiar with the IVY tool and its modeling notations. It was possible to formulate properties, based on requirements, attempt to prove them, and make modifications if necessary within a risk meeting without seriously disrupting the flow of the meeting. Where more serious issues were found these were analyzed outside the context of the meeting within an hour. It is envisaged that a similar process is feasible using the technologies described in this paper but further evaluations are envisaged to assess the usability of these tools and their developments.

7.4 Relevance to medical devices and certification

Certification authorities typically require that risk control measures are included as requirements (see ISO 62304 [43] for example), and that the identified control measures are verified, and the verification documented. Verification typically means that some form of systematic testing has taken place. The document explaining the verification should document a trace: from hazardous situation to user interface behavior; from the user interface behavior to the software feature causing the problem; from the software cause to the risk control measure, and to the verification of the risk control measure. Examples of use errors identified in the usability standard ISO 62366 [29] for medical devices include ergonomic concerns (confusing buttons, cracking catheter connectors) but also include the software issues relevant to the present paper: over-reliance on the alarm system; user enters incorrect sequence; user takes a short cut and omits important steps, defeating software interlock. ISO 62366 argues that causes of use error include ambiguous or unclear medical device state or controversial modes or mappings. Here we have demonstrated that formal techniques may be used to verify these risk control measures. The process of proving regulatory requirements has been discussed in more detail in [44], [45]. This process is typically interactive and in principle involves discussion with both human factors specialists, who are engaged in checking the validity of the interpretation of the user-related requirement, and regulator to check that the property captures the spirit of the original requirement. Templates can provide a source for the requirements that form the software control measures. This approach can be a key component of the broader safety analysis process increasing the confidence provided by software testing and trials, see [42].

8 Related work

Property templates have been studied extensively in engineering practices. Most of the effort, however, has been devoted to the control part of a system, rather than the human-machine interface. For example, such analysis in relation to complex systems has been discussed in [46] where verification patterns are introduced that can be used for the analysis of safety interlock mechanisms in interoperable medical devices. An example of such a pattern as "*When the laser scalpel emits laser, the patient's trachea oxygen level must not exceed a threshold* Θ_{O_2}". Although, superficially, this property appears to be use-related the aim of their patterns is to facilitate the introduction of a model checker in the actual implementation of the safety interlock, rather than defining property templates for the analysis of use-related aspects of the safety interlock. Other similar work, e.g., [47]–[49], also introduce mechanisms similar to templates for the verification of safety interlocks, but the focus of them is again on translating verified design models into a concrete implementation – in [47], for example, the automatic translation of hybrid automata models of a safety interlock into a concrete implementation.

Proving requirements similar to the properties produced from our templates of this paper has been the focus of the work of Atlee and Gannon [50] as well as Heitmeyer's team using SCR [51]. The latter approach uses a tabular notation to describe requirements which makes the technique relatively acceptable to developers. Combining simulation with model checking has also been a focus in, for example, [52]. Braderman and others [53] focused on the role of scenarios in model checking real-time properties of systems and Mori and others [54] used task representations based on a LOTOS like language to analyze user tasks by means of simulation. Recent work concerned with simulations of PVS specifications provides valuable support to this complementarity [35]. Had the specification been developed as part of a design process then a tool such as Event B [55] might have been used. Singh et al demonstrate a refinement process from tabular expressions using Event B [56]. In such an approach 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. Several GUI testing tools have been developed for Android Apps, Java, and Windows Applications (see [57] for a recent overview). For example, van der Merwe et al [58] use the Java PathFinder model checker as a basis to discover design errors in the user interface of Android Apps. These tools focus on implementation errors such as unhandled exceptions, concurrency errors, and null pointer dereferencing rather than properties that would relate to use errors.

Bowen and Reeves [36] focus on design patterns for creating user interface models, rather than verification of use-related requirements. An example pattern is the *call-back* pattern, representing the behavior of confirmation dialogs used to confirm user operations. They are also concerned with generating properties or obligations for the proof of interactive systems, with a particular focus on medical systems [59], [60]. Further relevant work by

them focuses on modeling user manuals [61].

9 Conclusions

We have argued that the formulation of use-related requirements has the effect of improving usability and use-related safety of interactive systems. The requirements that have been described are related to the usability heuristics often used in the informal analysis of interactive systems [62]. The model and theorems based on the templates can be found in the repositories referred to in Section 1. The model, as discussed, involves two main theories describing 38 pump actions and 18 user interface actions. The analysis involved 138 theorems based on the templates. The theory files amount to approximately 4000 lines including comments, and the theorem files approximately 5000 lines. The run time for each proof is indicated in the template files. Times range from 1 hour 19 minutes to less than 1 second. The PVS system was installed on an Apple Macbook Pro with a 2.9 GHz Intel Core i5.

The paper addresses two questions: how to support the analyst in the process of identification and analysis of properties of a user interface software design to improve the clarity of the user interface; and how to structure the specification of the design and formulate the properties so that it is feasible to establish the practice as part of the development of user interfaces.

While PVS is conceptually rich, the proposed style of specification based, as it is, on state transitions is amenable to development. The property templates aim to provide clear guides to the developer as they consider and then prove properties of the specification. The process of developing the properties from the templates is valuable in recognising areas where the properties fail, and this triggers further consideration of the design of the interface.

The examples illustrate the process, demonstrating how the development of theorems becomes a systematic process. The steps involved in the analysis process are made clear. A further step in this process will be to provide specialized tool support so that templates can be offered to the analyst with the means to define the guards, goals and filters that are relevant to the device under consideration. The illustrated example is realistic and the proofs demonstrate the feasibility of the approach for a relatively large specification.

Acknowledgement

This work has been funded by the EPSRC research grant EP/G059063/1: CHI+MED (Computer–Human Interaction for Medical Devices). We are grateful to Harold Thimbleby's team at Swansea University, part of the CHI+MED project, and especially Patrick Oladimeji who developed the infusion pump simulation that helped us develop the models. We also thank the anonymous reviewers for valuable feedback. José C. Campos and Paolo Masci were funded by project NORTE-01-0145-FEDER-000016, financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF).

References

[1] D. Arney, R. Jetley, P. Jones, I. Lee, O. Sokolsky, A. Ray, and Y. Zhang, "Generic infusion pump hazard analysis and safety requirements," University of Pennsylvania, Tech. Rep. MS-CIS-08-31, February 2009.

[2] D. Drusinsky, J. Michael, and M.-T. Shing, "A visual tradeoff space for formal verification and validation techniques," *Systems Journal, IEEE*, vol. 2, no. 4, pp. 513–519, Dec 2008.

[3] M. Ryan, J. Fiadeiro, and T. Maibaum, "Sharing actions and attributes in modal action logic," in *Theoretical Aspects of Computer Software*, ser. Lecture Notes in Computer Science. Springer-Verlag, 1991, vol. 526, pp. 569–593.

[4] J. C. Campos and M. D. Harrison, "Interaction engineering using the IVY tool," in *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, G. Calvary, T. Graham, and P. Gray, Eds. ACM Press, 2009, pp. 35–44.

[5] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An Open Source Tool for Symbolic Model Checking," in *Computer-Aided Verification (CAV '02)*, ser. Lecture Notes in Computer Science, K. G. Larsen and E. Brinksma, Eds. Springer-Verlag, 2002, vol. 2404.

[6] M. Harrison, J. Campos, and P. Masci, "Reusing models and properties in the analysis of similar interactive devices," *Innovations in Systems and Software Engineering*, vol. 11, no. 2, pp. 95–111, June 2015.

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

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

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

[10] Food and Drug Administration (FDA), "Class 2 Device Recall ACCUCHEK Connect Diabetes Management App," 2015. [Online]. Available: https://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfRES/res.cfm?id=134687

[11] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff, "How Amazon web services use formal methods," *Communications of the ACM*, vol. 58, no. 4, pp. 66–73, April 2015.

[12] P. Masci, P. Mallozzi, F. Luca, D. Angelis, G. Di Marzo, and P. Curzon, "Using PVSio-web and SAPERE for rapid prototyping of user interfaces in integrated clinical environments," in *3rd Workshop on Verification and Assurance (Verisure 2015), at CAV-2015, San Francisco, CA, USA*, 2015.

[13] P. Masci, P. Oladimeji, P. Curzon, and H. Thimbleby, "Tool demo: Using PVSio-web to demonstrate software issues in medical user interfaces," in *Software Engineering in Health Care: 4th International Symposium, FHIES 2014, and 6th International Workshop, SEHC 2014, Washington, DC, USA, July 17-18, 2014, Revised Selected Papers*. Cham: Springer International Publishing, 2017, pp. 214–221. [Online]. Available: https://doi.org/10.1007/978-3-319-63194-3_14

[14] G. Mauro, H. Thimbleby, A. Domenici, and C. Bernardeschi, "Extending a user interface prototyping tool with automatic MISRA C code generation," in *Proceedings of the Third Workshop on Formal Integrated Development Environment, F-IDE@FM 2016, Limassol, Cyprus, November 8, 2016.*, 2016, pp. 53–66. [Online]. Available: https://doi.org/10.4204/EPTCS.240.4

[15] P. Masci, Y. Zhang, P. Jones, P. Curzon, and H. W. Thimbleby, "Formal verification of medical device user interfaces using PVS," in *ETAPS/FASE2014, 17th International Conference on Fundamental Approaches to Software Engineering*. Berlin, Heidelberg: Springer-Verlag, 2014.

[16] N. Shankar, S. Owre, J. M. Rushby, and D. Stringer-Calvert, "PVS System Guide, PVS Language Reference, PVS Prover Guide, PVS Prelude Library, Abstract Datatypes in PVS, and Theory Interpretations in PVS," Computer Science Laboratory, SRI International, Menlo Park, CA, 1999, available at http://pvs.csl.sri.com/documentation.shtml.

[17] Cardinal Health Inc, "Alaris GP volumetric pump: directions for use," Cardinal Health, 1180 Rolle, Switzerland, Tech. Rep., 2006.

[18] US Food and Drug Administration, "Infusion pump improvement initiative," Center for Devices and Radiological Health, Tech. Rep., April 2010. [Online]. Available: http://www.fda.gov/MedicalDevices

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

[20] AAMI, "Infusing patients safely: priority issues from the AAMI/FDA infusion device summit," *AAMI/FDA Infusion Device Summit Proceedings*, pp. 5–6, 2010.

[21] ——, "AAMI/FDA summit on ventilation technology," *AAMI/FDA Infusion Device Summit Proceedings*, 2015.

[22] Y. Zhang, P. Jones, and R. Jetley, "A hazard analysis for a generic insulin infusion pump," *Journal of diabetes science and technology*, vol. 4, no. 2, p. 263, 2010.

[23] J. Rushby, "Using model checking to help discover mode confusions and other automation surprises," *Reliability Engineering and System Safety*, vol. 75, no. 2, pp. 167–177, Feb. 2002.

[24] J. Nielsen and R. Molich, "Heuristic evaluation of user interfaces," in *ACM CHI Proceedings CHI '90: Empowering People*, J. Chew and J. Whiteside, Eds., 1990, pp. 249–256.

[25] A. Dix, J. Finlay, G. Abowd, and R. Beale, *Human Computer Interaction (2nd edition)*. Prentice Hall Europe, 1998.

[26] H. Thimbleby, "Safer user interfaces: A case study in improving number entry," *IEEE Transactions on Software Engineering*, vol. 41, no. 7, pp. 711–729, 2015.

[27] J. C. Campos and M. D. Harrison, "Systematic analysis of control panel interfaces using formal tools," in *Interactive systems: Design, Specification and Verification, DSVIS '08*, ser. Lecture Notes in Computer Science, N. Graham and P. Palanque, Eds., no. 5136. Springer-Verlag, 2008, pp. 72–85.

[28] Z. Manna and A. Pnueli, *The temporal logic of reactive and concurrent systems-specification*. New York: Springer-Verlag, 1992.

[29] AAMI, "Medical devices - application of usability engineering to medical devices," Association for the Advancement of Medical Instrumentation, 4301 N Fairfax Drive, Suite 301, Arlington VA 22203-1633, Tech. Rep. ANSI AMI IEC 62366:2007, 2010.

[30] M. D. Harrison, P. M. Masci, J. C. Campos, and P. Curzon, "The specification and analysis of use properties of a nuclear control system," in *The Handbook of Formal Methods in Human-Computer Interaction*, B. Weyers, J. Bowen, A. Dix, and P. Palanque, Eds. Cham: Springer International Publishing, 2017, pp. 379–403. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-51838-1_14

[31] C. Fayollas, C. Martinie, P. Palanque, P. Masci, M. D. Harrison, J. C. Campos, and S. R. e Silva, "Evaluation of formal IDEs for human-machine interface design and analysis: the case of CIRCUS and PVSio-web," *EPTCS: arXiv preprint arXiv:1701.08465*, 2017.

[32] J. Hatcliff, E. Vasserman, S. Weininger, and J. Goldman, "An overview of regulatory and trust issues for the integrated clinical environment," *Proceedings of HCMDSS*, vol. 2011, pp. 23–34, 2011.

[33] G. J. Holzmann, "Trends in software verification," in *FME 2003: Formal Methods*, ser. Lecture Notes in Computer Science, K. Araki, S. Gnesi, and D. Mandrioli, Eds. Springer-Verlag, 2003, vol. 2805, pp. 40–50.

[34] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng, "Bandera: extracting finite-state models from java source code," in *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, 2000, pp. 439–448.

[35] 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, F. Bitsch, J. Guiochet, and M. Kaâniche, Eds. Springer-Verlag, 2013, vol. 8153, pp. 228–240.

[36] J. Bowen and S. Reeves, "Design patterns for models of interactive systems," in *Software Engineering Conference (ASWEC), 2015 24th Australasian*. IEEE, 2015, pp. 223–232.

[37] M. Dwyer, G. Avrunin, and J. Corbett, "Patterns in property specifications for finite-state verification," in *21st International Conference on Software Engineering, Los Angeles, California*, May 1999, pp. 411–420.

[38] K. Loer and M. Harrison, "An integrated framework for the analysis of dependable interactive systems (IFADIS): its tool support and evaluation," *Automated Software Engineering*, vol. 13, no. 4, pp. 469–496, 2006.

[39] M. Archer, "TAME: Using PVS strategies for special-purpose theorem proving," *Annals of Mathematics and Artificial Intelligence*, vol. 29, pp. 139–181, 2000.

[40] A. Degani, *Taming HAL: designing interfaces beyond 2001*. Palgrave, Macmillan, 2003.

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

[42] M. Harrison, M. Drinnan, J. C. Campos, P. Masci, L. Freitas, C. di Maria, and M. Whitaker, "Safety analysis of software components of a dialysis machine using model checking," in *Proceedings of Formal Aspects of Component Software (FACS 2017)*, ser. Lecture Notes in Computer Science, J. Proença and M. Lumpe, Eds., no. 10487. Springer-Verlag, 2017, pp. 137–154.

[43] BSI, "Medical device software - software life cycle processes," British Standards Institution, CENELEC, Avenue Marnix 17, B-1000 Brussels, Tech. Rep. BS EN 62304:2006, 2008.

[44] P. Masci, A. Ayoub, P. Curzon, M. Harrison, I. Lee, O. Sokolsky, and H. Thimbleby, "Verification of interactive software for medical devices: PCA infusion pumps and FDA regulation as an example," in *Proceedings ACM Symposium Engineering Interactive Systems (EICS 2013)*. ACM Press, 2013, pp. 81–90.

[45] M. D. Harrison, P. Masci, J. C. Campos, and P. Curzon, "Verification of user interface software: the example of use-related safety requirements and programmable medical devices," *IEEE Transactions on Human Machine Systems*, vol. 47, no. 6, pp. 834–846, 2017.

[46] 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," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, no. 3, pp. 642–652, 2014.

[47] 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," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 26, no. 10, pp. 2630–2642, 2015.

[48] A. L. King, S. Procter, D. Andresen, J. Hatcliff, S. Warren, W. Spees, R. Jetley, P. Raoul, P. Jones, and S. Weininger, "An open test bed for medical device integration and coordination." in *ICSE Companion*, 2009, pp. 141–151.

[49] B. Larson, J. Hatcliff, S. Procter, and P. Chalin, "Requirements specification for apps in medical application platforms," in *Proceedings of the 4th International Workshop on Software Engineering in Health Care*. IEEE Press, 2012, pp. 26–32.

[50] J. M. Atlee and J. Gannon, "State-based model checking of event-driven system requirements," *IEEE Transactions on Software Engineering*, vol. 19, no. 1, pp. 24–40, 1993.

[51] C. Heitmeyer, J. Kirby, B. Labaw, and R. Bharadwaj, "SCR: A toolset for specifying and analyzing software requirements," in *Computer Aided Verification*. Springer, 1998, pp. 526–531.

[52] G. Gelman, K. Feigh, and J. Rushby, "Example of a complementary use of model checking and agent-based simulation," in *Systems, Man, and Cybernetics (SMC), 2013 IEEE International Conference on*, Oct 2013, pp. 900–905.

[53] V. Braberman, N. Kicillof, and A. Olivero, "A scenario-matching approach to the description and model checking of real-time properties," *IEEE Transactions on Software Engineering*, vol. 31, no. 12, pp. 1028–1041, 2005.

[54] G. Mori, F. Paternò, and C. Santoro, "CTTE: Support for developing and analyzing task models for interactive system design," *IEEE Transactions on Software Engineering*, vol. 28, no. 8, pp. 797–813, 2002.

[55] J.-R. Abrial, *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.
[56] N. K. Singh, M. Lawford, T. S. E. Maibaum, and A. Wassyng, "Use of tabular expressions for refinement automation," in *Model and Data Engineering: 7th International Conference, MEDI 2017, Barcelona, Spain, October 4–6, 2017, Proceedings*, Y. Ouhammou, M. Ivanovic, A. Abelló, and L. Bellatreche, Eds. Cham: Springer International Publishing, 2017, pp. 167–182. [Online]. Available: https://doi.org/10.1007/978-3-319-66854-3_13
[57] V. Lelli, A. Blouin, B. Baudry, and F. Coulon, "On model-based testing advanced guis," in *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, April 2015, pp. 1–10.
[58] H. van der Merwe, B. van der Merwe, and W. Visser, "Verifying android applications using java pathfinder," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 6, pp. 1–5, 2012.
[59] J. Bowen and S. Reeves, "Modelling safety properties of interactive medical systems," in *Proceedings of the 5th ACM SIGCHI symposium on Engineering interactive computing systems*. ACM, 2013, pp. 91–100.
[60] ——, "Generating obligations, assertions and tests from UI models," *Proceedings of the ACM on Human-Computer Interaction*, vol. 1, no. 1, p. 5, 2017.
[61] ——, "Modelling user manuals of modal medical devices and learning from the experience," in *Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems*. ACM, 2012, pp. 121–130.
[62] J. Nielsen, "Heuristic Evaluation," in *Usability Inspection Methods*, J. Nielsen and R. Mack, Eds. John Wiley & Sons, Inc., 1994, ch. 2.