Patterns and templates for automated verification of user interface software design in PVS.

Michael D. Harrison, José Creissac Campos, Paolo Masci
Patterns and templates for automated verification of user interface software design in PVS.

Michael D. Harrison, José Creissac Campos, Paolo Masci

Abstract

This paper introduces a set of PVS theories for modelling and analysis of usability and safety aspects of user interface software design. Our PVS theories crystallise general usability issues 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 behaviour. These requirements have standard forms across different application domains, and can be instantiated as properties of specific devices. The modelling and analysis process in PVS is further facilitated by structuring the specification of the device according to a straightforward pattern. The verification process is supported by the PVS theorem prover, which provides support for the automated analysis of the modelled software behaviour. A concrete case study based on a commercial medical device is used to illustrate the approach. The device is modelled as a PVS theory and property templates are refined as theorems of the device specification. A detailed illustration is presented of the automated verification using the PVS theorem prover, demonstrating how failed proof attempts provide precise information about problematic user interface software features.
Bibliographical details

Title: Patterns and templates for automated verification of user interface software design in PVS.

Authors: Michael D. Harrison, José Creissac Campos, Paolo Masci

Pages: 13

NEWCASTLE UNIVERSITY

Abstract

This paper introduces a set of PVS theories for modelling and analysis of usability and safety aspects of user interface software design. Our PVS theories crystallise general usability issues 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 behaviour. These requirements have standard forms across different application domains, and can be instantiated as properties of specific devices. The modelling and analysis process in PVS is further facilitated by structuring the specification of the device according to a straightforward pattern. The verification process is supported by the PVS theorem prover, which provides support for the automated analysis of the modelled software behaviour. A concrete case study based on a commercial medical device is used to illustrate the approach. The device is modelled as a PVS theory and property templates are refined as theorems of the device specification. A detailed illustration is presented of the automated verification using the PVS theorem prover, demonstrating how failed proof attempts provide precise information about problematic user interface software features.

About the authors

Michael Harrison is emeritus professor and senior research investigator at Newcastle University. Jose Campos and Paolo Masci are at the Department of Informatics at University of Minho.

Suggested keywords

Automated Theorem Proving, User Interface Models, User Centred Requirements, Use Related Safety Requirements
Patterns and templates for automated verification of user interface software design in PVS.

Michael D. Harrison, José Creissac Campos, Paolo Masci

Abstract—This paper introduces a set of PVS theories for modelling and analysis of usability and safety aspects of user interface software design. Our PVS theories crystallise general usability issues 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 behaviour. These requirements have standard forms across different application domains, and can be instantiated as properties of specific devices. The modelling and analysis process in PVS is further facilitated by structuring the specification of the device according to a straightforward pattern. The verification process is supported by the PVS theorem prover, which provides support for the automated analysis of the modelled software behaviour. A concrete case study based on a commercial medical device is used to illustrate the approach. The device is modelled as a PVS theory and property templates are refined as theorems of the device specification. A detailed illustration is presented of the automated verification using the PVS theorem prover, demonstrating how failed proof attempts provide precise information about problematic user interface software features.

Index Terms—Automated Theorem Proving, User Interface Models, User Centred Requirements, Use Related Safety Requirements

I. INTRODUCTION

Demonstrating that safety requirements [1] are true of a design provides regulators with confidence that there are barriers that mitigate identified hazards. Many standards propose requirements that should be verified of a device or system to demonstrate safety. Some of these safety requirements are user-centred though domain specific, for example “The pump shall issue an alert if paused for more than 1 minutes”. Use related issues are an important reason for system or device failure and yet little work has been done on creating tool support for the analysis of use-related requirements.

Formal verification of usability aspects of user interface software has seen slow take-up because verification technology is perceived as difficult to use and to apply. In Drusinsky and others’ [2] review of formal tools and techniques, these barriers are presented over three dimensions: implementation cost, specification cost, and verification cost. The work described here aims to reduce these three costs in support of submissions to the regulatory process.

The paper is concerned with the use of modelling patterns to support the efficient structuring of executable formal specifications, and property templates for expressing formal theorems capturing general usability requirements. Our specific purpose is to enable the easy verification of user interface software design in programmable devices, of which infusion pumps are an example. Property templates describe general requirements that can be used to analyse essential usability aspects of the device, and if true can mitigate hazards that could arise through use error. The developed property templates are refined to the details of the particular device represented by the formal specification. The refined properties are then used within a theorem proving system to construct a proof demonstrating the conditions under which the property is true.

The theorem proving approach has been selected because an important feature of the analysis of many interactive devices concerns properties of number entry. In our specific case study based on a medical device, e.g., the domains used to specify volume to be infused, infusion rate and time are relatively large. Proof using model checking can result in analyses of very large models that can be intractable or very slow (see [3] for example).

Our approach to verification will be illustrated using an example based on an intravenous infusion device currently used in many hospitals. It shall be shown that proof of use centred safety requirements can be generated from the analysis of the property templates. This analysis enables the identification of inconsistencies that can result in poor understanding of the user interface and increase the risk of use errors. In a similar way the theorem development and improvement approach has been used to demonstrate that the devices satisfy a set of specific use-related requirements defined by the Food and Drug Administration (FDA) [4]. The proofs and their rationale can be used to provide a documented approach to demonstrate that requirements are satisfied to complement or replace test data.

Contribution. The present work extends and generalises a set of CTL (Computational Tree Logic) formulae identified in [5], providing the following new contributions:

- PVS theories for modelling user interface software behaviour;
- PVS theories defining property templates for the analysis of important usability aspects of user interface software design;
- A case study based on a commercial medical device in
use in hospitals across the EU and US.

**Organisation.** Section II presents the approach to automated analysis of user interface software based on modelling patterns and property templates. Section III introduces the medical device used for illustration and indicates how it is specified. Section IV presents the user interface model in PVS, instantiates the property templates to the details of the model, and illustrate relevant fragments of proofs. Section V discusses the results and considers related research. Section VII concludes the paper.

II. AUTOMATED ANALYSIS OF USER INTERFACE SOFTWARE

A. A pattern for modelling user interface software design

The model of the interactive device specifies the state transformations that are possible, partly through user action and partly autonomously through the ongoing behaviour of the underlying process. An action’s effect on state may be visible, or otherwise perceivable. The effect of an action is often complicated because it depends on its mode. Mode is used to allow the same physical action to achieve different logical actions. One effect of the use of modes is to improve the utilisation of a physically constrained interaction medium. Mode may, for example, determine which types of value in an underlying process are changed by data entry. Modes can confuse users about an action’s effect and can be difficult to understand. Understanding can be compromised, for example, if the current mode is not clearly visible [6].

The state of a device is defined in terms of state attributes. Each state attribute has a type, for example if the attribute is the infusion rate then it is restricted to values that infusion rate can take. A set of modes are also defined as an attribute of the state. A further characteristic of these attributes is that they may be perceivable – device alarms, e.g., are modelled as state attributes, and they may be seen or heard by the user. Information about mode and perceivability of the state will be important in formulating use related properties of the device.

In summary the models to be considered have the following common characteristics.

- A set of actions \( a : A = S \rightarrow S \) where \( S \) is a set of states. Actions are partial functions. They are made total by including a value “undefined” (⊥). A function \( \text{per} \) takes an action and determines whether it is defined for a value in its domain \( \text{per} : A \rightarrow (S \rightarrow T) \) such that \( \text{per}(a)(s) = \text{true} \) if \( a(s) \neq \bot \).

- A state is a set of attributes. Functions of the form \( \text{filter} : S \rightarrow C \) where \( C \) is an attribute will often be used to extract an attribute of the state. Some elements of the state are part of the interface, representing the perceivable counterpart of other, non-perceivable, attributes. \( \text{filter}_p \) will often be used to describe the filter that extracts the corresponding perceivable attribute to the attribute extracted by \( \text{filter} \). Alternatively a predicate \( \text{vis}_\text{filter} : C \rightarrow T \) may be used to directly assert that a filtered attribute is visible.

- Mode is particular form of \( \text{filter} \), namely \( \text{mode} : S \rightarrow MS \) that extracts the modes of the model, where \( MS \) is an attribute that ranges over a set of modes. In the example, one set of modes relates to the types of variable being entered through number entry.

B. Property templates capturing usability requirements

Property templates are generic mathematical formulae designed to help developers to construct theorems appropriate to the analysis of user interface features. The aim is to use property templates to make user interfaces more predictable and easy to use. The particular set of templates introduced in the present work is now described, along with an illustration of the use-related concerns captured by the template.

The templates are formulated using three notions: actions, states, and modes. There are two types of templates: properties that relate states where a specific action has taken place, and properties that relate a state to any state that can be reached by any action from that state. The relation \( \text{transit} : S \times S \) relates states that can be reached by any action. For a particular model \( \text{transit} \) will be instantiated to connect states by actions supported by the system.

**Completeness.** This template checks that the software allows the user to reach significant states in one (or a few steps). For example, being able to reach “home” from any device screen in one step is a completeness property. The completeness template asserts that a user action will transform any state that satisfies a predicate \( \text{guard} : S \rightarrow T \) into another state that satisfies a predicate \( \text{goal} : S \rightarrow T \). The guard is introduced to make it possible to exclude states that may not be relevant.

\[
\forall s \in S : \text{guard}(s) \land \sim \text{goal}(s) \Rightarrow \exists a \in A \land \text{per}(a)(s) \land \text{goal}(a(s)) \tag{1}
\]

**Feedback.** When certain important actions are taken, a user needs to be aware of whether the resulting device status is appropriate or problematic [7]. Feedback breaks down into state feedback, requiring that a change in the state (usually specific attributes of the state rather than the whole state) is visible to the user, and action feedback, requiring that an action always has an effect that is visible to the user.

**State feedback**

\[
\forall s_1, s_2 \in S, \text{guard}(s_1) \land \text{guard}(s_2) \land \text{filter}(s_1) \neq \text{filter}(s_2) \Rightarrow \text{filter}_p(s_1) \neq \text{filter}_p(s_2) \tag{2}
\]

**Action feedback**

\[
\forall a \in S \rightarrow S, \forall s \in S : \text{per}(a)(s) \land \text{guard}(s) \land (\text{filter}(s) \neq \text{filter}(a(s))) \Rightarrow \text{filter}_p(s) \neq \text{filter}_p(a(s)) \tag{3}
\]

In the case of state feedback the guard may be used, for example, to ensure that the device or system remains in the same mode as a result of the state transition. Variants of the feedback properties will also be used that assume separate visible attributes are not specified in the model.
Instead a relevant predicate $\text{vis\_filter} : S \rightarrow T$ is linked to $\text{filter} : S \rightarrow A$. $\text{vis\_filter}(s)$ is true for $s \in S$ if $\text{filter}(s)$ is visible. Both these variants will be used in the illustration below. The choice is based on how the model is constructed.

**Consistency.** Users quickly develop a mental model that embodies their expectations of how to interact with a user interface. Because of this, the overall structure of a user interface should be consistent in its layout, screen structure, navigation, terminology, and control elements [7]. The consistency across states and across actions. The guard on the states is updated to consider the notion of mode becoming consistent across states and across actions. The guard on the states is updated to consider the notion of mode becoming

$$
\forall s, s' \in S : \text{guard}(s) \Rightarrow \exists \text{filter}\_\text{pre}(s, m), \text{filter}\_\text{post}(a(s), m)
$$

**Reversibility.** Users may perform incorrect actions, and the device needs to provide them with functions that allow them to recover by reversing the effect of the incorrect action. The reversibility template is formulated using a guard $\text{guard} : S \rightarrow T$, and a $\text{filter} : S \rightarrow FS$ which extracts a set of focus attributes of the state:

$$
\forall s \in S : \text{guard}(s) \Rightarrow \exists b : S \rightarrow S :
\text{filter}(a(b(s)) = \text{filter}(s)
$$

**Visibility.** This property describes a relation between a state variable that is not necessarily visible to the user and a user interface value that is perceivable to the user. Examples of these properties are: the current operational mode is always unambiguously displayed; the same function key displays always appear that are specific to a given entry mode.

$$
\forall s_1, s_2 \in S : \text{transit}(s_1, s_2) \wedge \text{visible}(s_1) \Rightarrow \text{visible}(s_2)
$$

Predicate $\text{visible}$ has two alternative formulations that are appropriate in different circumstances. The first formulation is for boolean state variables (typically representing flags in software design), and uses predicates to assert that the visual representation is visible if an only if the state variable is set:

$$
\text{visible}(s) := \text{pred\_filter}(s) \leftrightarrow \text{pred\_filter\_p}(s)
$$

The second formulation uses a filter function to extract the actual value of the state variable, and compare it with the value of the corresponding visual representation. $\text{filter}(s)$ and $\text{filter\_p}(s)$ are the filters for the attribute and its perceivable counterpart. In this case, the state attribute is visible if and only if it is always the case that these two values are the same.

$$
\text{visible}(s) := \text{filter}(s) = \text{filter\_p}(s)
$$

**Universality.** Universality generalises the visibility property requiring that given two filters of the state: $\text{filter}_1$ and $\text{filter}_2$, there are predicates on the filters that are equivalently true. An example of a universality property 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.

$$
\forall s_1, s_2 \in S : \text{transit}(s_1, s_2) \wedge \text{universal}(s_1) \Rightarrow \text{universal}(s_2)
$$

Predicate $\text{universal}$ has two alternative formulations (as did the $\text{visible}$ predicate for the visibility template):

$$
\text{universal}(s) := \text{pred\_filter}_1(s) \leftrightarrow \text{pred\_filter}_2(s)
$$

or

$$
\text{universal}(s) := \text{filter}_1(s) = \text{filter}_2(s)
$$

### C. Tool support

The automated theorem prover used in this paper is the **Prototype Verification System (PVS)** [8]. It combines an expressive specification language based on higher-order logic with an interactive prover. PVS has been used extensively in several application domains. It is based on higher-order logic with the usual basic types such as boolean, 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 that are used in the illustrated model are function and record types. Function types are denoted $[D \rightarrow R]$, 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. Predicate subtyping is a language mechanism used for restricting the domain of a type by using a predicate. An example of a subtype is $\{x : A \mid P(x)\}$, 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 is used when defining actions that are only permitted given specific constraints.

Specifications in PVS are expressed as a collection of theories, which consist of declarations of names for types and constants, and expressions associated with those names. Theories can be parametrised with types and constants, and can use declarations of other theories by importing them.
Properties in PVS are expressed as theorems, which are formulae that have a name and are declared at the keyword theorem. To prove that a given property is an invariant of the system model, structural induction will be used extensively to obtain the reachable states that need to be considered in the proof.

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 theorems developed from the templates.

PVS has an automated theorem prover which provides a collection of powerful primitive inference procedures that are applied interactively under user guidance within a sequent calculus framework. The primitive inferences include propositional and quantifier rules, induction, rewriting, simplification using decision procedures for equality and linear arithmetic, data and predicate abstraction.

III. CASE STUDY: A PROGRAMMABLE INFUSION PUMP

The infusion pump (see Figure 1), which is an existing pump used in many hospitals, has characteristics that are common to many devices that control processes over time. The clinician (usually a nurse) sets infusion pump parameters and then monitors the infusion process using the device. Most infusion pumps have three basic states (their fundamental modes of operation): infusing, holding and off. In the infusing mode the Volume To Be Infused (VTBI) is pumped into the patient intravenously at a pre-determined infusion rate. If the VTBI is exhausted during infusion, the pump continues in what is called KVO (Keep Vein Open) mode and sets off an alarm. The KVO mode is designed to keep the patient’s vein open. When the pump is in holding state, values and settings can be changed. This is achieved through a combination of function keys and chevron buttons (for the device layout, see Figure 1). A subset of these features can also be changed when infusing. Chevron keys are used to enter numbers. These keys are used to increase or decrease entered numbers incrementally. Depending on current mode the chevron keys can be used to change infusion rate, volume to be infused and time. The chevron keys can also be used to move the device’s cursor between options in a menu. Menus are available to choose infusion bag sizes (that is in bag mode) and to choose between a selection of additional functions in query mode. Additional features can be invoked by pressing the query button to display a menu of set-up options. These options depend on how the device is configured by the manufacturer, and include the means of locking the infusion rate, or disabling the locking of it, or setting VTBI and time rather than VTBI and infusion rate. There is also the possibility of changing the units of volume and infusion rate. The device allows movement between display modes via three function keys (key1, key2 and key3). Each function key has a display associated with it, indicating its present function.

A. PVS MODEL OF THE USER INTERFACE SOFTWARE DESIGN

The illustrated model is specified as three theories. The first contains common definitions of constants and types. The second contains a specification of the underlying pump process, describing the details of the pump that are necessary to understand how the user of the device interacts with it. This specification describes basic characteristics that are common across a variety of infusion pumps, syringe drivers and PCAs (Patient Controlled Analgesia). These two theories have been reused in the modelling and analysis of other similar devices. The third theory describes features of the user interface software of the infusion pump. This theory is specific to the particular infusion product, though parts of it may be reused, for example in families of the same brand of pump. It describes whether attributes are perceivable, the modes of operation of the pump, and which basic attributes are visible at any state of the device. This theory specifies how the user sets, controls and views the operation of the device. The theories take the form of a set of transition functions that describe the actions

\[
\text{action: state} \rightarrow \text{state}
\]

which are permitted in particular modes or under certain conditions (sometimes, in all modes and under all conditions). For each action there is a predicate:

\[
\text{per\_action: state} \rightarrow \text{boolean}
\]

that asserts whether the action is permitted. A deterministic modelling approach is used that captures the behaviour of the user interface software design accurately. The states capture sufficient information so that each event/action has one possible effect. The state attributes of the pump are grouped together using a separate state attribute, device. This state attribute in a record type. The pump variables vtbi, infusionrate, volume and time are referenced as device(st)’vtbi and so on, where st represents the current state of the device. The actions of the device can be modified by the modes already mentioned. They describe, for example: how long chevron keys have been held down; whether the device is off, infusing or holding; the effect of chevron keys (in terms of the variable to be entered or which menu is being navigated). Modes are made visible using features of
the display. In the example (see Figure 1) there are attributes reserved to describe features of the display, for example \texttt{topline(st)} describes the information contained in that part of the display. Actions (key1, key2 and key3) are associated with function displays (fndisp1, fndisp2 and fndisp3). Booleans are associated with whether information is visible to the user (cf. \texttt{vis\_filter} in Section II-A). The two examples described below illustrate the combination of state attributes used to describe the interaction. These examples also provide an indication of the structure of the specification. The first specifies a permission associated with the \texttt{pause} action used to change the device mode from infusing to paused. The \texttt{pause} action is described using the following pause function.

\begin{verbatim}

\texttt{pause(st: \{st: state | per\_pause(st)'device\}): state =}
\texttt{ st WITH \{ device := pause(st'device),}
\texttt{ topline := holding,}
\texttt{ middisp := LAMBDA (x: imid_type): \ldots,}
\texttt{ fndisp1 := fvol,}
\texttt{ fndisp2 := fvtbi,}
\texttt{ fndisp3 := fnull,}
\texttt{ entrymode := rmode,}
\texttt{ pauselight := TRUE,}
\texttt{ runlight := FALSE \}}
\end{verbatim}

The pause function changes the mode of the \texttt{pump device} to \texttt{paused} and sets display features, for example: topline to “holding”, and function displays “volume” and “vtbi” that are visible to the user and indicate the effect of key1 and key2 (displayed above the keys using the display locations fndisp1 and fndisp2). The third key has no function display associated with it, hence \texttt{fnull}. A light also indicates that the device is paused. The mode of the device is \texttt{rmode} (the holding home mode) which allows infusion rate to be changed unless the rate has been locked.

The permission function has the following specification

\begin{verbatim}

\texttt{per\_pause(st: state): bool =}
\texttt{ per\_pause(st'device) AND no\_button\_down(st) AND}
\texttt{ ((topline(st) = infusing) OR}
\texttt{ (topline(st) = dispkvo) OR}
\texttt{ (topline(st) = dispvtbi) OR}
\texttt{ (topline(st) = volume) OR}
\texttt{ (topline(st) = locked))}
\end{verbatim}

which captures the following constraints on the pause action:

- The device is switched on and is infusing. These constraints are expressed using a permission function \texttt{per\_pause(st'device)} on the models of the process controlled by the pump.
- No other button has been pressed or is being pressed (\texttt{predict no\_button\_down(st)}).
- The top line of the display shows “infusing” or “KVO” or “vtbi” or “volume” or “locked”.

IV. MECHANISED ANALYSIS OF THE PROPERTY TEMPLATES IN PVS

This section illustrates how the templates of Section II-B are used to construct theorems over the PVS specification. The templates are instantiated with the state attributes used to describe the infusion device. A proof of the theorem generated is then attempted. The theorem is usually developed iteratively. Counter-examples are discovered while attempting the proof. This may be because the theorem is wrongly formulated or because the theory fails to represent the true properties of the device. Two situations are of interest however for the purposes of this paper. The first is that the device fails to satisfy the requirement with significant consequences. The second is when the failure is not important. Deciding which of these alternatives is the case involves discussion with domain and human factors experts. When the failure is not important the theorem is extended by changing the guard or in some cases qualifying the goal.

When developing theorems from templates it may be necessary to weaken requirements. For example, attributes that are visible before an action may differ from those visible after the action. It is important that any modification be appropriate to context while preserving safety requirements.

A. Completeness

Two “home” modes that are relevant for the example pump are \texttt{paused} and \texttt{infusing}. In both modes the chevron keys can be used to adjust the infusion rate, unless the infusion rate has previously been locked by the user. These are the normal default operating modes for the device.

The default mode, when the infusion device is paused, is signified by a top line that shows “holding” or “set rate”. In this mode the infusion rate can be changed if the rate has not been locked. The device finds this mode at start-up or after an alarm has occurred, when there has been no user activity for a period. When the device is infusing then the home mode is the one in which the top line shows “infusing”. This mode is taken when the \texttt{run} button is pressed if the device is ready to infuse. It is conjectured when formulating the property (perhaps through conversation with a human factors expert) that it will help the user if it is always possible to reach a familiar mode (the relevant “home” mode) in one step. This property is captured by the completeness template. The process of creating an appropriately tailored theorem involves specifying the \texttt{guard} and \texttt{goal} predicates. The template is instantiated to the illustrated model as follows when the device shows “holding” in the top line. As a first step, the general form of the property (see property 1 in Section II-B) can be re-expressed using PVS syntax as follows.

First, a predicate capturing an instantiation of the body of the universal quantifier is defined:

\begin{verbatim}

\texttt{comple\_to\_rmode(st: state): boolean =}
\texttt{ (guard\_rmode(st) AND NOT goal\_rmode(st)) ->}
\texttt{ ((per\_key1(st) => goal\_rmode(key1(st))) OR}
\texttt{ (per\_key3(st) => goal\_rmode(key3(st))))}
\end{verbatim}

The property template assumes that there is an action that will reach the goal state. In the instantiation it is asserted that this action is \texttt{key1} or \texttt{key3}. Otherwise the instantiation to the model reflects precisely the formulation of the template. The predicates for goal and guard properties are next specified in terms of the infusion device model. The goal predicate specifies the home mode in which infusion rate can be entered using the chevron keys.

\begin{verbatim}

\texttt{goal\_rmode(st: state): boolean =}
\texttt{ (entrymode(st) = rmode)}
\end{verbatim}

The guard predicate specifies that the device is paused.
guard_rmode(st: state): boolean = paused(st)

where

paused_state(st: state): boolean =
  (device(st)'powered_on? AND NOT device(st)'infusing?)

The property is formulated using structural induction: the
reachable states are all those states that can be reached from
the initial state using the possible actions that are specified in
the model. The theorem specifies a conjunction of two clauses
which together achieve the required induction. The first clause
asserts that the property is true in the initial state after start-up.

The second clause assumes that the required property is true
of a state and then proves it is true for all states that can be
reached from this state by one of the actions that the device can
take. The predicate state_transitions_release relates states
pre to states post if post can be reached from pre by one of
the possible actions. The chevron keys only become effective
when they have been released (this is because holding down
the chevron key can accelerate the behaviour of the relevant
data entry action). An example transition involving chevron
keys then is post = release_sup(sup(pre)). Here post is
reached from pre by pressing the “single chevron up” and
then releasing it. The theorem using these definitions is as
follows.

completemplate_to_rmode: THEOREM
FORALL (pre, post: state):
  (init?(pre) AND post = on(pre) =>
  comple_to_rmode(post)) AND
  ((transitions_release(pre, post) AND
  comple_to_rmode(pre)) =>
  comple_to_rmode(post))

This version of the theorem could not be proved. A series
of proof attempts was required to refine the theorem. Failed
attempts generated counter-examples that were used to refine
the definition of guard_rmode until the state was sufficiently
constrained by the guard to prove the theorem. For example,
consider the following component of the proof of the theorem.

simple_completeness_to_rmode.2.6.1.5 :
{-1} {topline(pre!1) = options} AND
  (entrymode(pre!1) = qmode)
[-2] device(pre!1)'powered_on?
[-3] no_button_down(pre!1)
[-4] post!1 = key1(pre!1)
[-5] simple_completeness_rmode(pre!1)
[-----
[1] device(pre!1)'infusing?
[2] simple_completeness_rmode(post!1)

This represents a sub-step in the proof of the structural
induction involving the action key1. The sequents
[-1], ..., [-5] are assertions that are true: pre!1 and
post!1 are skolem constants obtained from pre and post
when eliminating the universal quantifier with the theorem
prover commands. Hence in this case it is asserted that:
simple_completeness_rmode(pre!1) is true, that is the prop-
erty is true for the state before the specified transition. post!1
represents the state reached by the model after the transition
is taken. The sequents [1] and [2] are goals. The theorem is
proved if either [1] or [2] is proved true. This sub-theorem as-
serts that the top line shows “options”, the function display for
key1 shows “ok” and the entry mode of the device is “qmode”.
The case to be proved assumes that completeness_rmode is
true of the device in the pre state and attempts to prove either
that it is true of the state of the device after key1 has been
pressed [2] or that the device is infusing [1]. PVS has not
been able to prove the sub-theorem automatically and presents
a further sub-theorem which it invites the analyst to attempt
to prove.

{-7} setvtempower?(optionsmenu(pre!1))
    (qcursor(pre!1)))
{-8} rlock(pre!1)
{-9} post!1 =
    pre!1
    WITH [topline := locked,
         middisp := LAMBDA (x:
           imid_type
           maxrate, maxinfuse,
           timeout, timeout,
           maxtime, bat_max,
           bat_min): FALSE,
           fndisp1 := fnull,
           fndisp2 := fnull,
           fndisp3 := fnull,
           entrymode := qmode]

The important elements of this further sub-theorem are as
follows. Sequents [-7] asserts that the state pre!1 is such that
the user has pressed key1 and has selected the menu entry “set
vtempower” when in query mode. Sequent [-8] asserts that
the infusion rate is locked. Sequent [-9] asserts that on pressing
key1 the top line shows “locked” and all function key displays
are blank, meaning that they are not enabled. In practice, a
fact that is not visible in the step represented by this theorem,
after one tick the device returns to the options display and
this display satisfies the completeness property. This particular
counter-example is therefore not a concern because the display
is temporary.

This case can therefore be safely excluded by modifying
guard_rmode. Several similar counter-examples result in the
following re-formulation.

guard_rmode(st: state): boolean =
  paused_state(st) AND
  (topline(st) /= locked AND
   NOT (topline(st) = dispvtempower AND
   (entrymode(st) = bagmode OR
    entrymode(st) = tbagmode)))

This new version excludes situations when entering vtempower
where actions keep the device in a mode in which vtempower
is entered. For example when selecting an infusion bag, the only
exit is to the “outer” VTBI mode where the value of vtempower
can be further modified using chevron keys. Counter-examples
can raise important issues from a usability point of view. For
example, when the device is paused, it is unlikely to be a
problem that the home mode cannot be reached in one step. It
could, for example, be reasonably argued that users are trained
in the use of the device and are familiar with the role of the
function keys.

Further completeness theorems have a similar format. When
the device is infusing, “home” is to return to the mode in which
the top line shows “infusing” or “kvo”. This is the situation
in which entrymode(st) = infusemode. The goal therefore requires that the device is infusing.

guard_infuse(st: state): boolean = entrymode(st) = infusemode

Only key3 is required here to make the step to the home mode. Through a similar series of attempts to the one described earlier the guard is modified.

guard_infuse(st: state) = infuse_state(st) AND topline(st) /= dispvtbi OR topline(st) /= bagmode

The guard contains similar constraints to the previous expression, dealing with the additional modes that are available when entering vtbi. Given this guard the theorem is proved true of the device.

B. Feedback

Any change to the settings of basic pump variables (infusion rate, VTBI and time) in an infusion pump should be visible to the clinician. Theorems to demonstrate this can be constructed from the feedback templates. Examples of feedback properties for an infusion device are the following.

- Any action, other than those resulting from chevron key presses, changes the mode, and the change of mode is visible.
- Any chevron key press always changes either a pump variable or cursor position in the menu, depending on the mode.
- If a pump variable (e.g., infusion rate) is changed then that change is visible.
- If the mode changes then the mode is visible.

State feedback. Feedback relating to infusion rate prompts a definition of filter that extracts the value of the infusion rate attribute (see property 2 in Section II-B). The property uses the version of state feedback that asserts that the value of the filtered attribute is visible rather than explicitly referring to a further visible attribute.

filterratefull(st: state): boolean = device(st)'infusionrate

The visibility or otherwise of this attribute is defined by a Boolean. The model contains Boolean function middisp that specifies whether key attributes are visible or not.

visratefull(st: state): boolean = middisp(st)(drate)

The feedback property then becomes:

(filterratefull(s1) /= filterratefull(s2)) => visratefull(s1) AND visratefull(s2)

The feedback theorem in this case is of the following form:

feedbackratefull: THEOREM
FORALL (pre, post: state):
(state_transitions(pre, post) AND (filterratefull(pre) /= filterratefull(post)))
=> (visratefull(pre) AND visratefull(post))

The theorem cannot be proved in this case because a side effect of entry of vtbi over time, using the options menu, is that it changes infusion rate. The infusion rate that is displayed is a temporary value (specified as newrate) that is used to update the infusion rate pump variable when exiting vtbi over time using ok. This may seem to be an irrelevant detail of the specification but the fact that it is temporary is important because if the machine is switched off then the modified value of infusion rate is lost, and the value of infusion rate on entry is preserved. Accordingly visratefull is modified.

visratefull(st: state): boolean =
middisp(st)(drate) OR middisp(st)(dnewrate)

The theorem is also slightly modified because after the change of state the value of the infusionrate attribute is always displayed, and therefore the property can be strengthened.

feedbackratefull: THEOREM
FORALL (pre, post: state):
(state_transitions(pre, post) AND (filterratefull(pre) /= filterratefull(post)))
=> (visratefull(pre) AND middisp(post)(drate))

The process involved in modifying the definition of visratefull is important because it leads the analyst to consider what the implications of such a change are on the visibility of the device. For example, if it is important that the infusion rate displayed in some situations is not the infusion rate that would be used by the pump process. This feedback property is one of several state based visibility properties that can be proved of the device.

Action feedback. Feedback related to actions is another important aspect of the usability of the infusion device. The ongoing infusion process is specified using the tick action. After each tick the time to infuse and the volume to be infused are reduced. When the device is paused, these values no longer change and the tick action assumes another role. In this case tick determines the delay since the last user action.

It is to be expected that the key pump variables show change as the process continues. The action feedback template (see property 3 in Section II-B) is instantiated identifying time, volume to be infused and infusion rate as relevant pump attributes.

A preliminary guard requires that the device is infusing. There are two modes in which the device can infuse: normal when there is still medication to infuse; keep vein open (KVO) mode when the medication is exhausted and the pump infuses to keep the vein open while alarming to alert the user. Each different mode of use is considered when identifying visibility properties. For the first case we assume the normal mode, that is:

guard_tick(st: state): boolean = 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 is also considered because of its importance to the clinician in the infusion calculation. Hence the relevant predicate is defined:
vis_tick(st: state): boolean =
middisp(st)(dvibi) AND
middisp(st)(drate) AND
middisp(st)(dtime)

Instantiating the template generates the following theorem.

feedback_tick: THEOREM
FORALL (st: state):
(per_tick(st) AND guard_tick(st) AND
guard_tick(tick(st))) => vis_tick(tick(st))

Numerous attempts at proving the theorem in the style
described in the previous section lead to a revised formulation
of the guard. This form of the guard acknowledges the
possibility that, if the device is not connected to mains power,
then a warning might occur and in this case the values of
the pump attributes will be concealed. It also recognises that
immediately after vtbi is first exhausted a display is generated
with top line of “vtbi done” that does not show the values of
the pump attributes.

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

Further properties can be proved for other situations, for
example when the top line shows “vtbi done” or when infusing
in “KVO” mode. For example, when the top line shows “vtbi
done” successive attempts to prove the theorem lead to the
following guard.

guard_tick_vtbidone(st: state): boolean =
(device(st)'infusionrate >> device(st)'vtbi) AND
NOT device(st)'kvoflag AND
device(st)'infusing? AND
device(st)'ac_connect AND
toline(st) = infusing

guard_tick_vtbidone(st: state): boolean =
toline(st) = vtbidone

This refinement of the guard leads to proof of the theorem.

Many other action feedback properties have been proved
for the model of this infusion device. For example, the use of
the switch action that changes the device’s power source
from running on battery to mains power and vice versa always
indicates the relevant status. The attributes (ac_light and
battery_light) are two Booleans that when true indicate that
the mains and battery lights respectively are on.

guard_light(st: state): boolean =
ac_light(st) /= battery_light(st)

The relevant action feedback theorem expressed as follows is true:

feedback_switch: THEOREM
FORALL (st: state):
per_switch(st) AND guard_light(st)
=> ((ac_light(st) /= ac_light(switch(st))) AND
(battery_light(st) /= battery_light(switch(st))))

C. Consistency

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
(see property 4 in Section II-B). Examples of these kinds of
consistency in the infusion 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 mode (entrymode) if that mode is
relevant to the change of pump variable. Note that in
some modes chevron keys are used to navigate the
cursor.

3) When a function key is associated with a soft display
of ok then the value of the relevant pump variable is
changed 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 and the pump
variable reverts to the value it had when it entered the
mode.

Three examples illustrate the use of the consistency tem-
plate. The first is that chevron keys never change entry mode.
The pre_filter and post_filter both extract the entry mode,
and are of the form.

filterem(st: state):emodes = entrymode(st)

This property relates directly to mode and therefore mode is
not used as a parameter as specified in Property 4. The set of
actions determined by state_transitions_chevrons consists of
all the chevron actions. The consistent relation in this case
is equality and the theorem that instantiates the consistency
template is:

chevconsistent: THEOREM
FORALL (pre, post: state):
(state_transitions_chevrons(pre, post))
=> (filterem(pre) = filterem(post))

On the other hand all the other actions, except tick, always
change the entry mode. So here consistent is inequality.

nochevfeed: THEOREM
FORALL (pre, post: state):
(state_transitions_nochevrons(pre, post) AND
guardem(pre)) => (filterem(pre) /= filterem(post))

Two sets of actions are involved here defined by two
versions of the state transition predicate. The first assumes
that the only actions are chevron actions while the second
assumes that only non-chevron actions are involved. A guard
is required in the case of nochevfeed because when infusion
rate is locked, and the user has selected the option “set vtbi
over time” in query mode then the effect of the action is to
leave the entry mode unchanged (see Figure 2). The required
guard is:

guardem(st: state): boolean =
NOT topline(st) = options AND rlock(st) AND
optionsmenu(st)(qcursor(st)) = setvtbiovertime

Other examples of consistency include the property of action
key1 when associated with a function display that shows ok.
When key1 is associated with ok a consistent effect is that the
action changes the relevant pump variable to the value that has
been updated within the entry mode. Similarly, when key3 is
associated with quit the action leaves the mode and reinstates
the value of the pump variable to the value that it was before
it was modified in that mode.

We consider the ok example in more detail. The guard
assumes that key1 is permitted and the function display shows
A further constraint that was established through attempting proof was to exclude `vttmode` which is used transiently during the definition of vtbi over time.

The filters used in establishing the consistency depend on the entry mode. So the “temporary” value before `key1` is pressed is defined 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 ->
    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
```

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

- when the function display shows `ok` then `key1` takes the top line to holding
- when function display shows `quit` then `key3` takes the top line to holding
- when top line is volume and not infusing then `key2` always changes volume infused to zero and changes the entry mode to `rmode`.

### D. Goal Reversibility

Reversibility (see property 5 in Section II-B) ensures that an action can be undone with a single reversing action. To construct the theorem to deal with the case of infusion rate entry, a guard is specified:

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

This guard requires that the device is ready to enter the infusion rate (`rate_entry_ready`). This can be done either when infusing (entrymode = `infusemode`) or when paused (entrymode = `rmode`), but only when the infusion rate is not locked.

```
rate_entry_ready(st: state): boolean =
  switchedon?(st) AND NOT rlock(st) AND
  ((entrymode(st) = rmode AND
    (topline(st) = holding OR
      topline(st) = setrate)) OR
   (entrymode(st) = infusemode AND
    topline(st) = infusing))
```

It also requires that the relevant action, and its reverse action, are enabled. In the process of proving the theorem exceptions to the required property were discovered at specific boundaries for the infusion rate. Examples of these boundaries are as follows.
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.
Further guards must therefore be imposed. For example, one of the three cases for \textit{sup} followed by \textit{sdow} is:
\begin{verbatim}
low_range_rate(v: irates): boolean =
  (maxrate > 1000) AND v < 100 AND 
  (v >= small_step/10) AND 
  (floor(v*10) = v*10) AND 
  (ceil_rate(v*10) = v*10)
\end{verbatim}

An additional constraint ensures that the value of the infusion rate is expressed in terms of at most one decimal place: \(\{\text{floor}(v*10) = v*10\} AND \text{ceil}(v*10) = v*10\). This constraint describes the behaviour of the numbers as supported by the modelled device. The proposed theorem that follows the reversibility template is:
\begin{verbatim}
low_supsdown: THEOREM
FORALL (st: state):
guard_supsdown_rate(st) AND 
low_range_rate(filter_rate(st)) => filter_rate(release_sup(sup(release_sdown(sdown(st))))) = filter_rate(st)
\end{verbatim}
A final assumption is made about the maximum value of the infusion rate: maxrate > 1000. This constraint again describes the device. Proving this theorem highlighted issues in relation to the way that number entry behaves for the device. These issues affect its usability. Later releases of the firmware for this particular device have fixed these inconsistencies.

E. Visibility

The general form of the first version of the visibility property (see property 6 in Section II-B) is:
\begin{verbatim}
guard(st) => (pred_filter(st) <=> pred_p_filter(st))
\end{verbatim}
For example, either the battery light or the AC light are always on when the device is switched on. Which of the two lights is on depends on whether the power is switched on. Similarly, the “paused” light and “run” light accurately reflect the paused and infusing modes of the device (see Figure 1). The following property captures the situation when the run light is on.
\begin{verbatim}
pred_onandinfusing(st: state): bool =
  (device(st)'powered_on? AND device(st)'infusing?)
pred_p_onandinfusing(st: state): bool =
  (runlight(st) AND (NOT pauselight(st)))
\end{verbatim}
In this case the property is generally true, guard(st) = true. Further visibility properties demonstrate the visual distinctness of entry modes. For example:
\begin{verbatim}
pred_rmode(st: state): bool = entrymode(st) = rmode
pred_p_rmode(st: state):bool =
  (topline(st) = holding OR 
  topline(st) = setrate
\end{verbatim}

F. Universality

Universality (see property 7 in Section II-B) captures, more generally, properties of equivalence between state attributes. The focus here is not necessarily on the visibility of state attributes. For example, it may specify an invariant relationship between display states: the top line shows some value if and only if the function displays are in a particular configuration. A universality property of this kind is as follows:
\begin{verbatim}
pred_filter_volume(st: state): boolean =
  (topline(st) = volume)
pred_filter_vol_fn_displays(st: state): boolean =
  fndisp1(st) = fnull AND 
  fndisp2(st) = fclear AND 
  fndisp3(st) = fquit
\end{verbatim}
The universality property becomes:
\begin{verbatim}
univ_vol(st: state): boolean =
  pred_filter_volume(st) <=>
  pred_filter_vol_fn_displays(st)
\end{verbatim}
It specifies that when the top line shows volume then function key 1 is always blank, key 2 is always clear and key 3 is always quit (see Figure 3).
Other instances of the universality template are:
- function key 3 is the only key that can show quit
- function key 1 is the only key that can show ok

V. Discussion

Both model checking and theorem proving can be used to prove that property templates and requirements are true of a formal specification. While model checking is considered to be easier to understand, it is 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. Model checking performance deteriorates rapidly as the model grows and becomes infeasible to use in an interactive style. In the considered example based on a real system, 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 theorem proving was used as the basis for analysis. Further considerations that were important in the analysis were ensuring:
- The model accurately reflected the fielded system (Section V-A)
- The approach bore relevance to actual or proposed certification requirements (Section V-B)
- Tools were available that would ease the modelling and proof process (Section V-C)

A. Modeling and specification

The approach taken in the paper assumes that a model is constructed from an existing system. This is not usually considered feasible in the context of regulation because of the cost of creating the model and the expertise involved. To demonstrate that the approach works for a realistic scale we have taken an existing design. The model was developed by hand using the user manual of the device, a simulation of the device, and the device itself. An alternative approach would have been to generate the model from the code using a set of transformation rules that guarantee correctness discussed in [9]. Given the state of current tools and the size of the program, this would have been impossible. While some tools exist to achieve this, for example [10], this would have been an infeasible process, given the size of the program, its style of construction and our ability to access code without the required relationship with developers. However, for small examples it is a valuable and effective means of analysing requirements, and has been demonstrated for subsets of existing devices, for example the number entry components of an infusion pump, see [11]. The fidelity of the model in relation to the simulation, the model and the device itself was checked by proving a range of properties. The PVS model was based on a prior model of substantial parts of the device that had been analysed using a model checking approach [12]. This model was translated systematically into PVS as a basis for the new model. In the earlier model, states of the modelled device were examined by comparing the traces of actions produced by the model checker, as counter-examples for these properties, with actual sequences generated by the device itself.

Properties were constructed of the full model that were designed to check that the theory behaved as expected. When the template properties were proved of the theory then any counter-examples discovered as the theorems were refined were compared with the actual device. As a further process of verification a prototype was produced automatically from the model to compare the “look and feel” of the actual device with the prototype, see [13] for details. The traces and simulations were indistinguishable from the behaviour 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 behaviour). 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.

B. Relevance to medical devices and certification

Certification authorities typically require that risk control measures are included as requirements (see 62304 [14] for example), and that the identified control measure 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 behaviour; from the user interface behaviour 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 62366 [7] 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. 62366 argues that causes of use error include ambiguous or unclear medical device state or controversial modes or mappings. It has been argued elsewhere 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 [4]. 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. Part of the implicit argument of the present paper is that the templates can provide a source for the requirements that form the software control measures.
C. Modelling and proof tools

Progress is being made to make the tools described in the paper accessible to developers who are not specifically expert using formal techniques. These developments include the following.

- Specification templates or patterns have been developed designed to ease the construction of specifications [15].
- Patterns have also been used to ease the process of generating properties as originally described by Dwyer [16]. The patterns used here are based on our earlier work using model checking techniques [5].
- Tools have been developed for presenting proofs and counter-examples to aid comprehension [17].
- Strategies have been devised and mechanised to support proof [18].

Tailoring these tools to the particular requirements described in this paper is part of our ongoing research. It is relatively straightforward to support the construction of theorems from the templates, using an analogous approach to that provided by the IVY tool [5] tailored to PVS theorems. Presenting counter-examples in a simple format and supporting general tactics for proof is more complicated. These tools and techniques 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. Evaluations will be required to assess the usability of such tools.

VI. Related work

Design patterns and property templates have been extensively studied in engineering practices. Most of the effort, however, has been devoted to creating patterns and templates for the control part of a system, rather than for the human-machine interface.

Gamma et al. [19] established a comprehensive set of standard design patterns for software components of a system. An example pattern is the abstract factory, which facilitates the creation of families of software objects (e.g., windows of a user interface). Another example is the adapter pattern, which converts the interface of software components to enable integration of otherwise incompatible software components. These patterns are a de-facto standard in the software engineering community and they are widely adopted in engineering practices to solve common problems related to the realisation of software components. Konrad and Cheng [20] discuss design patterns for cyber-physical elements of embedded systems. An example pattern is the actuator-sensor pattern, providing a standard interface for sensors and actuators connected to the control unit of an embedded system. Similarly, Sorouri et al [21] present a design pattern for representing the control logic of an embedded system. Lavagno et al [22] introduced Models of computation (MoC) as design patterns for representing interactions between distributed system components. Recently, Steiner and Rushby [23] have demonstrated how these MoC can be used in model-based development of cyber-physical systems, to represent in a uniform way different time synchronisation services executed within the system. These activities and other similar work are concerned with design patterns for the control part of a system, as opposed to the human-machine interface – e.g., problems like how to correctly design the behaviour of data entry software in human-machine interfaces are out of scope.

Various research activities introduced design patterns for the analysis of complex systems. For example, in [24], verification patterns are introduced that can be used for the analysis of safety interlock mechanisms in interoperable medical devices. Although they use the patterns to analyse use-related properties such as “When the laser scalpel emits laser, the patient’s trachea oxygen level must not exceed a threshold \( \Theta_2 \)”, 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., [25]–[27], also introduce design patterns for the verification of safety interlocks, but the focus of the patterns is again on translating verified design models into a concrete implementation – in [25], for example, the design patterns for the automatic translation of hybrid automata models of a safety interlock into a concrete implementation.

Proving requirements similar to the properties produced from the templates of this paper has been the focus of a mature set of tools that have been developed by Heitmeyer’s team using SCR [28]. Their 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, [29]. Recent work concerned with simulations of PVS specifications provides valuable support to this complementarity [13]. Had the specification been developed as part of a design process then a tool such as Event B [30] might have been used. 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.

Bowen and Reeves have introduced four design patterns for user interfaces [31]: the callback pattern, representing the behaviour of confirmation dialogues used to confirm user operations; the binary choice pattern, representing the behaviour of input dialogs used to acquire data from the user; the iterator pattern, representing the behaviour of parametric user interface widgets that share the same behaviour but have a different value parameter, such as the numeric entry keys 0–9; and the update pattern, for representing the behaviour of a numeric display.

VII. 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 [32]. The paper addresses two questions: how to mechanise 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 that is relatively straightforward to perform. A further step in this process will be to provide specialised tool support so that templates can be offered to 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 given relatively large specification.

REFERENCES