Developing and Verifying User Interface Requirements for Infusion Pumps: A Refinement Approach

Rimvydas Rukšėnas, Paolo Masci & Paul Curzon

Rukšėnas, R., Masci, P., & Curzon, P. (2015). Developing and Verifying User Interface Requirements for Infusion Pumps: A Refinement Approach. Chapter 1 of L. Petre & E. Sekerinski (Eds), *From Action Systems to Distributed Systems: The Refinement Approach*, Chapman and Hall/CRC.

PP release date: 9 June 2015

Chapter 1

Developing and Verifying User Interface Requirements for Infusion Pumps: A Refinement Approach

Rimvydas Rukšėnas

Queen Mary University of London

Paolo Masci

Queen Mary University of London

Paul Curzon

Queen Mary University of London

1.1 Introduction

Demonstrating that interactive devices are acceptably safe is a significant and important element in their development in various domains. For example, interaction design errors in medical devices have an impact on patient safety and contribute to health-care costs. Because of this, medical device regulators require manufacturers to provide sufficient evidence that the risks associated with the device are "as low as reasonably practicable" as well as being fit for purpose before entering the market. This process is known as the premarket review process. For the majority of medical devices, the premarket approval relies on manufacturers demonstrating that the new device is as safe and effective as an already legally marketed device [18], or that it has been developed in accordance with recognised international standards [5].

In its current form, the premarket approval process involves the analysis of tens of thousands of printed pages [11] rather than a direct evaluation of the product. To reduce the amount of paperwork and enable the submission of more succinct and rigorous evidence, the use of formal methods is being promoted by the US Food and Drug Administration (FDA), the regulator for medical devices in the US. Their approach relies on *usage models* for the verification of software [10]. A usage model is a formal representation that describes the common characteristics and behaviour of software for broad classes of devices. The approach is based on the idea of developing usage models that satisfy core sets of safety requirements that are designed to mitigate typical hazards. This way, usage models can be used as a reference by manufacturers – if they are able to show that their product is compliant with the behaviours of the usage models, then regulators have evidence that the manufacturer's device meets minimum safety conditions. These models are developed manually starting from safety requirements, verifying the models against these requirements using model checking techniques.

Our approach shows how stepwise refinement and the Event-B/Rodin platform can be conveniently used to develop correct-by-construction usage models that are related to the interactive aspects of medical devices. It addresses two key points of the FDA's approach. The first is how to design safety requirements so that they are sufficiently precise to be effectively operationalised. The second is to provide, by operationalising requirements, the means for encompassing the range of input/output technologies that are likely to be encountered in interacting with the systems. Event-B is initially used here to express the high level requirements such as those proposed by the FDA. Stepwise refinement is then used both to make those high level requirements more precise and to demonstrate that the requirement can be cascaded into a hierarchy that encompasses potential input/output technologies.

To illustrate the approach, we focus specifically on infusion pumps. We take as a starting point a particular sample set of *user related* requirements specified by the FDA. The original FDA specifications are in natural language. We give abstract formal specifications of these requirements. We then show how they can be refined to more concrete versions. These versions can be verified against the formal specification of specific pump designs. Here we concentrate on a particular infusion pump design based on a commercially available pump.

This paper is based on and extends our earlier work [14] on user interface requirements for infusion pumps. In particular, it formally develops and verifies new requirements related to the safeguards against accidental tampering with infusion settings. Also, the specific pump design is modelled in more detail which reflects the actual device more truthfully.

1.1.1 Outline of the Approach

The proposed approach is based on three layers: requirements hierarchies, interface hierarchies and concrete interfaces, each described below.

The requirements hierarchies layer, which is directly relevant to regulators, concerns the development of user interface requirements. The regulator will be interested in the satisfaction of these requirements to assure them of the device's safety. A minimal set of such requirements, relevant to some usability aspect of device interfaces, is developed. The aim is that these requirements should be sufficiently abstract to encapsulate the behaviour of the largest class of possible devices. Refinements are then used to detail these requirements in a sequence of steps. It is also possible that refinement can lead to alternative interface requirements that also provide assurance of the safety of the device. These modified requirements would be developed as a contract between regulator and manufacturer. The development of the requirements hierarchy layer is discussed in Section 1.4.

The concrete interface layer focuses on the user interfaces of specific devices. This layer is most relevant to manufacturers as they demonstrate that the user interfaces of their devices satisfy the requirements developed in the requirements hierarchy layer. There are several possible approaches when trying to produce such a demonstration.

The first is to verify a specific interface against the safety requirements directly. How complicated this is will depend on the extent the requirements were operationalised. An example of this approach is discussed in Section 1.5.2.

The second approach aims to simplify the process of demonstrating that a specific user interface adheres to the relevant set of user requirements. It facilitates the dialogue between regulators and manufacturers by means of an intermediate layer, the interface hierarchies. This layer essentially develops a refinement based hierarchy (classification) of user interfaces. The idea is that user requirements are verified once for most abstract classes of interfaces. More concrete classes of interfaces at the lower levels of this hierarchy are then guaranteed to satisfy the requirements by construction. Now, instead of directly verifying a specific interface against the requirements it suffices to demonstrate that it is an instance of some concrete class of user interfaces. This approach, briefly discussed in Section 1.5.3, correlates with the current FDA pre-market review process which involves providing evidence that a new device is 'substantially equivalent' to already approved and legally marketed medical devices.

1.2 Sample User Interface Requirements from FDA

The regulator's aim is to be assured that risks associated with the use of a device are as low as reasonably practicable. As previously discussed part of this assurance is achieved through a credible demonstration that safety requirements are true of the device. Before showing how this demonstration can be achieved in the proposed framework, we describe a subset of user related safety requirements developed by the FDA.

These requirements relate to two aspects of infusion pump designs: the usability of their data entry systems and the safeguards against inadvertent changes of or tampering with infusion settings. The subset considered is relevant to the volumetric infusion pump used by clinicians that forms the basis of the example contained in this paper. The safety requirements are taken from a larger set produced by the FDA [17]. This larger set is intended specifically for PCA (Patient Controlled Analgesic) pumps. As a result it has more emphasis on patient tampering than clinician errors, and therefore the overall focus is slightly different than is relevant to the volumetric infusion pump. The aim is to show how these independently determined properties can be framed in our framework.

The requirements from the FDA document, considered in the subsequent sections, are listed below:

R1 *The flow rate and vtbi (volume to be infused) for the pump shall be programmable.* This safety requirement aims to mitigate hazards due to incorrectly specified infusion parameters (e.g., flow rate is too high or low).

R2 *The vtbi settings shall cover the range from v_{min} to v_{max} ml.*

R3 *The user (clinician) shall be able to set the vtbi in j ml increments for volumes below x ml.*

R4 *The user (clinician) shall be able to set the vtbi in k ml increments for volumes above x ml.*

R5 *Clearing of the pump settings and resetting of the pump shall require confirmation.* This requirement aims to safeguard against clinicians changing infusion settings inadvertently.

R6 *To avoid accidental tampering of the infusion pump's settings such as flow rate/vtbi, at least two steps should be required to change the setting.*

1.3 Background

This section briefly discusses approaches to interface refinement and introduces the Event-B formalism used in our approach.

1.3.1 Interface refinement approaches

Several previous projects on formal refinement for user interface design had different foci to our work. For example, the main focus of Bowen and Reeves [2, 3] is on a description of the actions that the user can engage with and how these actions can be refined. The refinement process involves actions being replaced by more concrete actions in terms of more concrete structures. The refinement described by them is more akin to trace refinement. Although they argue that their interest is in ensuring that requirements are true of the more refined system, there is less concern with how the requirements are transformed through the levels of refinement. Duke and Harrison [7] are concerned with data refinement. They note that abstract representations of objects can be refined in two directions, into what is perceivable and into the architecture of the device. Darimont and van Lamsweerde [6] are concerned with requirements described in terms of the refinement of goals using the KAOS language. The interesting innovation in their proposal is that the formal refinement process may be achieved through a set of patterns.

The approach we take here has most in common with the work of Yeganefard and Butler [19] who demonstrate a similar refinement process, in this case for control systems, using Event-B. They describe an approach to requirements structuring to facilitate refinement-based formalisation. Their work considers control systems consisting of plants, controllers and operators. In these terms, our focus is narrower, encompassing phenomena shared between controller and the operator. Also, we emphasise the formalisation of high level requirements and their clarification through refinement, whereas Yeganefard and Butler focus attention on requirements structuring. The structure developed is then mapped to a formal model in the stepwise refinement process. Moreover, their work is yet to address non-functional requirements, considered here.

In our previous work [11, 9], we explored a different approach to formalisation and refinement of user-related requirements. First, an abstract logic model is created that encapsulates key notions and relationships presented in the textual description of the requirements. Second, a mapping relation is established between the abstract logic model and a concrete model of a device being verified. This mapping relation is used as a basis to instantiate requirements for the concrete device model. The concrete model is then mechanically verified against the instantiated requirements.

1.3.2 Event-B/Rodin framework

Event-B specifications are discrete models that consist of a state space and state transitions. A state includes constants and variables that describe the system. State transitions are specified as *events*. A specification of an event consists of two parts as seen in the example below. The first is a list of *guards*. Each guard is a predicate over the state variables and constants. All the guards are combined using logical conjunction which is implicit in the Event-B syntax. These guards together define the necessary conditions for the event to occur. The second part is a list of *actions* which describe how the state variables are modified as a result of event execution.

Specifications are structured in terms of *machines* and *contexts*. Machines specify the dynamic aspects of a system, whereas contexts specify its static aspects. A machine includes state variables and events. Invariant properties are expressed as machine invariants, i.e., predicates that must hold in all machine states. A context includes constants defined by a set of axioms. A machine may reference constants from the contexts it 'sees'.

Intuitively, machine execution means that one of the events, with all guards being true, is chosen. The machine variables are modified as specified by the actions of that event. The basic syntactic form of an event is given below, other features of Event-B are introduced when needed.

Event $E \mathrel{\widehat{=}}$ **when** $G(v)$ **then** $T(v)$ **end**

Here v is a list of variables. $G(v)$ denotes the guards of E and $T(v)$ denotes the actions associated with E. The formal semantics of events is given using before-after predicates that encode the relation between the machine variables before and after an event occurrence. A detailed description of Event-B can be found in [1].

1.4 The requirement hierarchies

In this section, the informal requirements from Section 1.2 are first formalised in Event-B then made more precise through gradual refinement. The requirements **R1**–**R4** related to data entry interfaces are considered first.

1.4.1 Requirements for data entry

The informal requirements **R1** and **R2** provide a basis for the abstract specification of user requirements relevant to data entry. **R3** and **R4** are introduced in a later refinement.

1.4.1.1 Requirements R1 and R2

The requirement **R1** (*The vtbi/flow rate for the pump shall be programmable*) is expressed as the following machine in Event-B. This abstract description simply requires that a variable called *data* has the attribute that it is programmable. The requirement asserts that *data* commences with a value named *source* and describes the event *programmable* as changing the value to *target*. The possible values of *data* are given as the set *Numbers*. All three constants, *Numbers*, *source* and *target*, are defined in the context ReqParams1 below.

MACHINE Reqs1 **SEES** ReqParams1
VARIABLES *data* **INVARIANTS** $data \in Numbers$
EVENTS
Initialisation **begin** $data := source$ **end**
Event $programmable \mathrel{\widehat{=}}$ **begin** $data := target$ **end**
END

The invariant of Reqs1 simply gives typing of *data*. The initialisation event assigns the *source* value to it. Since the *programmable* event expresses an abstract requirement, its guard is assumed to be always true, and the **when** clause is omitted in the above specification.

The requirement **R2** (*The vtbi settings shall cover the range from v_{min} to v_{max} ml*) is specified in the context ReqParams1 which defines the corresponding constants *Min*, *Max*. It is assumed that *Max* exceeds *Min* and that *Min* is non-negative. The set constant (type) *Numbers* is assumed to be the interval $0 \mathinner{..} Max$. The context defines a number of other constants: *RefValues*, *source* and *target*. It is assumed that the *source* value belongs to the interval *Numbers* and it is assumed that *target* is a member of the set of reference values (*RefValues*) that covers the required range of settings. At this stage, no other assumptions are made as to what these values are.

CONTEXT ReqParams1
CONSTANTS *Min* *Max* *Numbers* *RefValues* *source* *target*
AXIOMS

$Min \geq 0 \qquad Max > Min \qquad Numbers = 0 \mathinner{..} Max$

$RefValues \subseteq Numbers \cap \{x | x \geq Min\}$

$source \in Numbers \qquad target \in RefValues$

END

Because the **R1** requirement is specified in a non-operational form it is necessary to refine the machine. Informally, machine refinement means verifying three constraints. The first concerns event refinement: a concrete event must refine the corresponding abstract one (new events must refine an implicit event that does nothing). The second constrains new events: they must ‘converge’ (i.e., not run forever on their own). The third states that the concrete machine must not deadlock before the machine it refines.

The following refinement of Reqs1 provides guidance about how **R1** can be implemented. The operational version of **R1** has a number of new characteristics. Two new variables are introduced: *entry* and *disp*. Whether a number is being entered is indicated by *entry*, whereas *disp* gives the displayed value of the number entered. The initial state requires that *data* and *disp* are both initialised to the *source* value and *entry* is false, indicating that entry of the target number has not commenced. The new requirement decomposes the event representing **R1** into three events. The first one (*choose*) is used to elect to enter the target value, while the second one models the modification of the display value (this is not necessarily the data value). The final event is triggered when the display and target values are equal. At this step the data value is set to be equal to the display value and entry becomes false. This operational requirement indicates more about the programming process but says little about how the value is entered.

MACHINE Reqs11 **REFINES** Reqs1 **SEES** ReqParams1
VARIABLES $data$ $disp$ $entry$ **INVARIANTS** $disp \in Numbers$ $entry \in BOOL$
EVENTS
Initialisation **begin** $data := source$ $disp := source$ $entry := FALSE$ **end**
Event $choose \,\widehat{=}$ **Status** anticipated
  when $entry = FALSE$ **then** $disp := data$ $entry := TRUE$ **end**
Event $modify \,\widehat{=}$ **Status** anticipated **when** $entry = TRUE$ **then** $disp :\in Numbers$
  end
Event $set \,\widehat{=}$ **refines** *programmable*
  when $disp = target$ $entry = TRUE$ **then** $data := disp$ $entry := FALSE$
    end
END

The machine Reqs11 specifies that *set* refines the abstract event *programmable* (intuitively, both events assign *target* to *data*). The other two events, *choose* and *modify*, are new. For the machine Reqs11 to refine Reqs1 these newly introduced events must ‘converge’ (i.e., they must not execute forever). The specification does not attempt to prove that. Rather than requiring their convergence, the specification assumes, as indicated by the keyword ‘anticipated’, that *choose* and *modify* will not run forever. If necessary, this assumption can be proven later.

1.4.1.2 Requirements R3 and R4

In the case of **R3** (*The user shall be able to set the VTBI in j ml increments for volumes below x ml*) and, similarly, **R4**, the requirements are expressed in a sufficiently concrete form to proceed directly to their operationalised versions. They are captured in the following context ReqParams11 which extends ReqParams1 by adding three relevant constants—*Threshold* (*x* in **R3** and **R4**), *j* and *k*—with three associated axioms:

CONTEXT ReqParams11 **EXTENDS** ReqParams1 **CONSTANTS** $Threshold$ j k

AXIOMS
  $Threshold \in Min + 1 \,..\, Max - 1$ $j < Threshold$ $k \leq Threshold$
  $RefValues \subseteq \{x \cdot x > 0 \wedge j * x \leq Threshold \mid j * x\} \cup \{x \cdot x > 0 \mid Threshold + k * x\}$
END

The fourth axiom restricts the reference set (*RefValues*) to the values obtained using the increments j and k. This context is used by Reqs111 which is the same machine as Reqs11 otherwise:

```
MACHINE  Reqs111  REFINES  Reqs11  SEES  ReqParams11  ....
```

The last step in the refinement of requirements has a more technical nature. It decomposes Reqs111 so that the assumptions about the user behaviour are removed from the requirements for the pump interfaces. In particular, one guard (*disp* = *target*) in the event *set* encompasses the notion of a target. Though the latter is relevant to the user behaviour, it would be meaningless to apply it to the pump interface. The decomposition introduces a machine that replaces the constant *target* by a variable that represents the display value 'passed' to the user. The details are omitted here, since this does not affect the actual data entry.

1.4.2 Safeguards against inadvertent changes or tampering

In this section, the requirements **R5** and **R6** are formally developed.

1.4.2.1 Requirement R5

The informal requirement **R5** (*Clearing of the pump settings and resetting of the pump shall require confirmation*) simply states that an attempt to clear the pump settings (such as vtbi and flow rate) cannot take its effect immediately but should result in a request for confirmation. The requirement is captured by the event *clear* in the following Event-B machine Reqs5. The variable *require* set to true represents a request for the confirmation of the clearing action. The event *clear* is enabled when the clearing action has not already been attempted (*require* = *FALSE*). In that case, it initiates a request to confirm the clearing action (*require* := *TRUE*). Note that this specification, faithful to the informal requirement, says nothing about the effect of the clearing action on the pump settings:

```
MACHINE  Reqs5
VARIABLES  require    INVARIANTS  require ∈ Bool
EVENTS
Initialisation  begin require := FALSE end
Event  clear ≙   when  require = FALSE  then  require := TRUE  end
END
```

Though it is not stipulated explicitly, the requirement **R5** also implies that there should be some kind of acknowledgement for the clearing request. By making this assumption explicit in the following refinement of the machine Reqs5, the requirement becomes more precise and rigorous. In the refinement Reqs51, the acknowledgement is modelled as a new event, *acknowledge*. Reqs51 also introduces a new variable, *ack*. When set to true, this variable represents an acknowledgement of the clearing action:

```
MACHINE Reqs51 REFINES Reqs5
VARIABLES require ack INVARIANTS ack ∈ BOOL
EVENTS
Event Initialisation extends Initialisation then ack := FALSE end
Event clear extends clear end
Event acknowledge ≙ Status anticipated
        when ack = FALSE require = TRUE then ack := TRUE end
END
```

The event *acknowledge* is enabled when there is an unacknowledged request for confirmation and sets *ack* to true. For the remaining events, the keyword **extends** indicates that the event in question incorporates (and refines) the corresponding event in Reqs5. In addition, the refined initialisation event sets *ack* to false.

The requirement as modelled by Reqs51 captures mode transitions in the pump behaviour. However, it puts no constraints on the changes to the pump settings associated with those transitions. The need to be more rigorous about that is addressed in the next refinement of Reqs51. Not to be bound by any specific interpretation what is included into the concept of pump settings, we assume that they are expressed as an abstract type (set), *Settings*. Constant *blank* from *Settings* represents the pump settings being cleared. These assumptions are modelled as the following context:

```
CONTEXT ReqParams5
SETS Settings CONSTANTS blank
AXIOMS
        blank ∈ Settings
END
```

The refinement Reqs511 introduces new variable *data*. It is an abstract representation of the pump settings. The requirement **R5** is once again made more precise by distinguishing two possibilities associated with the acknowledgement event. The first one is that the clearing request is actually confirmed by the pump user. This is modelled by the event *confirm* which extends *acknowledge* by updating *data* to *blank* (the pump settings are cleared). The second possibility is that the user changes their mind and cancels the clearing request. This is modelled by the event *quit* which leaves *data* unchanged:

```
MACHINE Reqs511 REFINES Reqs51 SEES ReqParams5
VARIABLES require ack data INVARIANTS data ∈ Settings
EVENTS
Event Initialisation extends Initialisation then data :∈ Settings end
Event clear extends clear end
Event confirm extends acknowledge then data := blank end
Event quit extends acknowledge end
Event other ≙ Status anticipated
        when require = FALSE then data :∈ Settings end
END
```

A new event (*other*) is added to Reqs511 to avoid constraining changes to the pump settings when there is no request to clear them. In such states (*require* = *FALSE*), event *other* allows the pump settings to be updated in an arbitrary way (*data* :∈ *Settings*).

1.4.2.2 Requirement R6

Expressed in natural language, the requirement **R6** (*To avoid accidental tampering of the infusion pump's settings such as flow rate / vtbi, at least two steps should be required to change the setting*) is open to several interpretations. For example, the 'two step' condition can be interpreted as one change of the setting followed by another. However, in the context of safeguarding against accidental tampering, a more plausible interpretation of **R6** seems to be that any attempt to change a setting like vtbi should require a separate confirmation step. Such an interpretation relates **R6** to **R5**. Therefore, it would be plausible to view **R6** as a part of the requirements hierarchy formally developed for the requirement **R5**, since a two step (request - acknowledgement) structure has already been specified in the machine Reqs51. In particular, **R6** can be formally introduced as a refinement of Reqs51.

At the same time, there are several aspects in which **R6** is different from **R5**. Firstly, it applies to *any* change to a particular pump's setting, not just clearing of that setting. Secondly, **R6** refers to the changes to a particular setting as opposed to the simultaneous clearing of all settings as stated in **R5**. Though, due to the abstract nature, the type *Settings* in our specification can be interpreted as both, a particular pump's setting and all the settings combined, it is not obvious what advantages a formal linkage of **R6** and **R5** would provide.

Taking these considerations into account, the requirement **R6** is formalised in a separate development. Its starting point captures the 'two step' condition, modelled as the following Event-B machine:

```
MACHINE  Reqs6
VARIABLES  change    INVARIANTS  change ∈ Bool
EVENTS
Initialisation  begin change :∈ Bool end
Event  update ≙   when  change = TRUE  then  change := change  end
Event  acknowledge ≙   when  change = TRUE  then  change := FALSE  end
END
```

The variable *change* represents the mode of pump operation where changes to a particular pump's setting can be made. When the pump is in such a mode, the event *update* (first step) stands for all the possible updates to that setting. These updates leave the mode of pump operation unchanged (*change* := *change*). The event *acknowledge* (second step) stands for the confirmation of changes, which results in the pump exiting the change mode (*change* := *FALSE*).

Next, we look in more detail at how the two steps specified in Reqs6 relate to the actual changes to the pump's setting considered. The informal requirement **R6** suggests that any changes to the relevant setting are 'provisional'. In that respect, however, **R6** can be interpreted in two ways at least. The first interpretation is that the changes, before they are confirmed, do not affect the actual pump's setting. Instead, they are recorded in a temporary pump's parameter. Only the confirmation step updates the relevant setting with the new value from the temporary parameter. The second interpretation is that the changes are applied to the relevant setting immediately. However, they still have to be confirmed in the confirmation step. Otherwise, the setting is restored to its old value. Both interpretations are below formalised as refinements of Reqs6. Both refinements introduce a new variable, *param*, that represents the relevant pump's setting. We start with the first interpretation.

In addition to *param*, the refinement Reqs61 introduces another variable, *new*, that models provisional changes to *param*. This variable is initialised to the value of *param*:

```
MACHINE  Reqs61  REFINES Reqs6  SEES  ReqParams5
VARIABLES   change   param   new
INVARIANTS      param ∈ Settings    new ∈ Settings
EVENTS
Event  Initialisation   extends Initialisation   then   param :∈ Settings   new := param
        end
Event  update   extends update  then  new :∈ Settings  end
Event  confirm   extends acknowledge  then  param := new  end
Event  cancel   extends acknowledge  end
END
```

The event *update* extends the same event from Reqs6. It guarantees that any changes to the setting are provisional and temporary recorded in the variable *new*. Both events *confirm* and *cancel* refine the old event *acknowledge*. The confirmation of the changes to the setting is modelled by *confirm*. It extends *acknowledge* by updating *param* with the new value for this setting (*new*). Any changes are cancelled by the event *cancel*.

The second, perhaps less natural, interpretation of **R6** is formalised as the following refinement Reqs62. In addition to *param*, it introduces another variable, *old*. This variable stores the old setting and is initialised to *param*:

```
MACHINE  Reqs62  REFINES Reqs6  SEES  ReqParams5
VARIABLES   change   param   old
INVARIANTS      param ∈ Settings    old ∈ Settings
EVENTS
Event  Initialisation   extends Initialisation   then   param :∈ Settings   old := param
        end
Event  update   extends update  then  param :∈ Settings  end
Event  confirm   extends acknowledge  end
Event  cancel   extends acknowledge  then  param := old  end
END
```

According to this specification, the setting *param* is changed with each *update* event. However, the changes are disregarded by the event *cancel*, if this option is selected instead of *confirm*.

The difference between these formal interpretations of the requirement **R6** is quite subtle. If the changes to the setting are finished in a normal way by confirming or cancelling them, the two formal requirements make no difference. Only when the changes are abruptly interrupted (e.g., the pump is switched off before the confirmation step), Reqs61 and Reqs62 will result in different requirements for the pump design. It is up to the regulators of medical devices to decide which version of the requirement **R6** is preferable, or whether they both are equally acceptable. Formalising several alternative interpretations highlights the issue but also raises the possibility of exploring the consequences for safety of each choice formally.

1.5 Verification of concrete interfaces

Having produced an operational but abstract definition of the requirements, the next stage is to make sense of the requirement in terms of the particular device that the developer wishes to certify. The aim of this section is to show how an interface specification of a specific device can be shown to satisfy user related requirements. Ideally, such a specification would be provided by the device manufacturer. Alternatively, it can be reverse engineered by interactively exploring the actual device [16, 9].

To illustrate our approach, we consider the number entry module of the Alaris GP Volumetric Pump [4]. A specification of this module has been reverse engineered in PVS and SAL [12]. The specification given below is its direct translation to Event-B. The purpose of using this translation is to demonstrate two ways of verifying the relevant user requirements for the independently developed specifications of concrete interfaces.

1.5.1 Specification of the vtbi entry in Alaris

The Alaris pump uses a chevron based number entry interface. In this type of interface, the current data value is updated by pressing the ‘up’ (increase) and ‘down’ (decrease) chevron keys. The fast versions of these keys are used to speed up data entry. For example, a fast ‘up’ chevron increases the current value by a larger amount compared to a slow ‘up’ one.

In the PVS and SAL versions, the behaviour of the Alaris chevrons (slow and fast up/down keys) is captured using functions that specify how the current value is modified by pressing each chevron. In Event-B, the corresponding functions, *alaris_up*, *alaris_dn*, *alaris_UP* and *alaris_DN*, are defined in the following context. It extends RealDefinitions which provides an Event-B model for the real numbers supported by the Alaris pump. The definitions of *alaris_dn*, *alaris_UP* and *alaris_DN* (omitted here) are similar to that of *alaris_up*:

CONTEXT AlarisDefinitions **EXTENDS** RealDefinitions
CONSTANTS $trim \quad alaris\_up \quad alaris\_dn \quad alaris\_UP \quad alaris\_DN \quad init \quad ...$
AXIOMS

$$trim \in \mathbb{Z} \rightarrow real \quad alaris\_up \in real \rightarrow real \quad init \in real$$
$$\forall x \cdot (x < minAlaris \Rightarrow trim(x) = minAlaris) \wedge$$
$$(x > maxAlaris \Rightarrow trim(x) = maxAlaris) \wedge$$
$$(x \geq minAlaris \wedge x \leq maxAlaris \Rightarrow trim(x) = x)$$
$$\forall x \cdot x \in real \Rightarrow (x < r100 \Rightarrow alaris\_up(x) = trim((floor(x * 10) + r1)/10)) \wedge$$
$$(x \geq r100 \wedge x < r1000 \Rightarrow alaris\_up(x) = trim(x + r1)) \wedge$$
$$(x \geq r1000 \Rightarrow alaris\_up(x) = trim((floor(x/10) + r1) * 10)) \quad ...$$

END

The behaviour of the four chevrons when entering vtbi values is described by the events *up*, *dn*, *UP* and *DN*. E.g., *up* is specified below:

Event $up \mathrel{\widehat{=}}$ **Status** anticipated

when $topline = dispvtbi \quad entrymode = vtmode$ **then** $display := alaris\_up(display)$ **end**

Here the condition $topline = dispvtbi$ indicates that the pump is in the mode where the vtbi value can be changed, whereas $entrymode = vtmode$ says that the changes are performed by updating the vtbi value with the chevron keys. The *display* variable represents the displayed value of vtbi. This event does not change the actual vtbi setting represented by the variable *vtbi*. The specifications of the remaining chevrons are similar.

The machine Alaris_vtbi1 includes these four chevron events and two events that model the acknowledgement (confirmation and cancellation) of the changes made to the vtbi value using the chevrons. The confirmation case is specified as follows:

Event $confirm \mathrel{\widehat{=}}$

when $topline = dispvtbi \quad entrymode = vtmode$ **then** $vtbi := display \quad topline := ptop(infstate) \quad entrymode := pentry(infstate)$ **end**

This event updates the vtbi setting with the entered value recorded by *display*. The mode of pump operation (*topline*) and the data entry mode (*entrymode*) go back to their previous values. These are given as the values of functions *ptop* and *pentry*, respectively. They depend on whether the pump is in the infusing state or not, which is modelled as the boolean variable *infstate*. The functions *ptop* and *pentry* are defined in the context AlarisDefinitions. The cancellation case is modelled similarly.

Now we illustrate two ways of verifying that the Alaris vtbi entry module satisfies the requirements formalised in Section 1.4. First, the requirement **R6** is verified directly for the machine Alaris_vtbi1.

1.5.2 Requirement R6

In this illustration, our first interpretation (machine Reqs61) is used for the informal requirement **R6**.

Since the structure of Alaris_vtbi1 is not that different from Reqs61, it is feasible to demonstrate the refinement relation between them directly. However, as a preparatory step, the abstract set *Settings* and constant *blank* used in Reqs61 must be instantiated to the concrete set *Numbers* and value *0*, respectively, used in Alaris_vtbi1. In Event-B, this is automatically done by applying the generic instantiation plugin to Reqs61 (we will use the same name for the instantiated machine).

To establish refinement between the instantiated machine Reqs61 and Alaris_vtbi1, one has to provide a 'glueing' invariant that relates the concrete variables in Alaris_vtbi1 and the abstract variables they replace in Reqs61. The abstract variables in question are *change*, *param* and *new*. As discussed earlier, *change* = *TRUE* models the mode where a relevant pump's setting can be changed. In Alaris_vtbi1, the corresponding mode for the vtbi entry is defined by the condition *topline* = *dispvtbi* ∧ *entrymode* = *vtmode*. Assuming this condition is true, the vtbi setting *vtbi* and its provisional value *display* are identified with their counterparts in Reqs61. The resulting glueing invariant allows one to prove the following refinement:

```
MACHINE  Alaris_vtbi1  REFINES Reqs61  SEES AlarisDefinitions    ...
INVARIANTS
          (change = TRUE) ⇔ (topline = dispvtbi ∧ entrymode = vtmode)
          (topline = dispvtbi ∧ entrymode = vtmode) ⇒ (vtbi = param ∧ display = new) ...
EVENTS
Event  up ≙  Status anticipated  refines update
        when      topline = dispvtbi   entrymode = vtmode   with   new′ : new′ = display′
             then   display := alaris_up(display)   end   ...
END
```

Similarly to *up*, the remaining chevron events (*dn*, *UP* and *DN*) in Alaris_vtbi1 refine the requirement event *update*. Finally, the acknowledgement events *confirm* and *cancel* refine their counterparts in Reqs61.

1.5.3 Requirements R1-R4

This section illustrates an alternative approach for demonstrating that the data entry systems in infusion pumps satisfy relevant safety requirements. A number of such systems are already used in infusion pumps [13] and there is future scope for many more. The approach presumes that a refinement-based hierarchy of user interfaces has been previously developed that is relevant for various modes of data entry in infusion pumps [14]. It is also assumed that the relevant requirements have been verified for the classes at the top of the hierarchy. If so, then the interface classes at the lower levels are guaranteed to preserve them by construction. To verify a specific interface against those requirements, it then suffices to show that the interface is an instance of some class in the hierarchy. This principle is demonstrated for the Alaris vtbi entry system.

We will show that the Alaris vtbi entry is an instance of the class of interfaces with four chevron keys. This class, represented by the machine Chevron_Entry11, has already been shown to satisfy the formalisation of the requirements **R1-R4** [14]. Thus, the demonstration that the Alaris vtbi entry interface is an instance of that class boils down to proving refinement between Chevron_Entry11 and Alaris_vtbi1. For such a proof, the generic parameters (such as *j*, *k* and *Threshold*) used by Chevron_Entry11 must be instantiated with the concrete values from the Alaris specification (context AlarisDefinitions) as, for example, shown below:

```
CONTEXT  ChevronAlarisParams  EXTENDS  ChevronDefinitions11  AlarisDefini-
         tions
AXIOMS
          Min = minAlaris      Max = maxAlaris
          j = r01      k = r1      Threshold = r100    ...
END
```

To specify the behaviour of four chevron keys, Chevron_Entry11 already includes the events *up*, revtdn, *UP* and *DN*. These must be refined by the corresponding events in Alaris_vtbi1. Finally, the invariants of Alaris_vtbi1 must include a glueing invariant that specifies the connection between the state spaces of both machines:

```
MACHINE  Alaris_vtbi1  REFINES  Chevron_Entry11  SEES  ChevronAlarisParams
         ...
INVARIANTS
          (entry = TRUE) ⇔ (topline = dispvtbi ∧ entrymode = vtmode)
          (vtbi = data ∧ display = disp)    ...
EVENTS
Event  up ≙  Status anticipated  refines up
         when  topline = dispvtbi   entrymode = vtmode   with   disp′ : disp′ = display′
               then  display := alaris_up(display)  end    ...
END
```

1.6 Conclusions

We have demonstrated how Event-B can be used to support manufacturers as they aim to demonstrate that the regulator's requirements are satisfied by their products. All the refinements described have been proven using the Rodin platform. The refinement hierarchies thus developed for requirements and user interfaces enable developers to trace the regulator requirements down to the specialised classes that match the physical characterisation of their device. Such an approach fits well with the FDA pre-market review process which involves providing evidence that a new device is 'substantially equivalent' to already approved and legally marketed medical devices.

We envisage showing that a device satisfies a full set of requirements by developing specification fragments. Each fragment would address one or more requirements and would provably demonstrate that the requirements in question are satisfied. It then remains an open question as to how one proves that these components are consistent with each other and how they might fit into a larger specification. This is future work. It would explore work on composition [15] and product lines [8] in Event-B being carried out at Southampton. The advantage of using Event-B is that the approach is tool supported. It is feasible that standard refinement processes such as these can be made easier for developers to use.

Acknowledgments

This work is supported by EPSRC as part of CHI+MED (Computer-Human Interaction for Medical Devices, EPSRC research grant [EP/G059063/1]). We are grateful to Michael Harrison who collaborated in various aspects of the work described.

Bibliography

[1] J-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[2] J. Bowen and S. Reeves. Refinement for user interface designs. *Formal Aspects of Computing*, 21:589–612, 2009.

[3] 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*, EICS '13, pages 91–100. ACM, 2013.

[4] Cardinal Health Inc. Alaris GP volumetric pump: directions for use. Technical report, Cardinal Health, 1180 Rolle, Switzerland, 2006.

[5] Council of the European Communities. Council directive 93/42/EEC of 14 June 1993 concerning medical devices. `http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CONSLEG:1993L0042:20071011:EN:PDF`, 2007.

[6] R. Darimont and A. van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. In *Proceedings 4th ACM Symposium on the Foundations of Software Engineering (FSE'03)*, pages 179–190. ACM Press, 1996.

[7] D. J. Duke and M. D. Harrison. Mapping user requirements to implementations. *Software Engineering Journal*, 10(1):13–20, 1995.

[8] A. Gondal, M. Poppleton, and C. Snook. Feature composition - towards product lines of Event-B models. In *1st International Workshop on Model-Driven Product Line Engineering (MDPLE'09)*. CTIT Workshop Proceedings, 2009.

[9] M. D. Harrison, P. Masci, J. C. Campos, and P. Curzon. Demonstrating that medical devices satisfy user related safety requirements. In *4th International Symposium on Foundations of Healthcare Information Engineering and Systems (FHIES2014)*, 2014.

[10] R. Jetley, S. Purushothaman Iyer, and P. L. Jones. A formal methods approach to medical device review. *Computer*, 39(4):61–67, 2006.

[11] P. Masci, A. Ayoub, P. Curzon, M. D. Harrison, I. Lee, and H. Thimbleby. Verification of interactive software for medical devices: PCA infusion pumps

and FDA regulation as an example. In *EICS 2013, Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, pages 81–90. ACM New York, NY, USA, 2013.

[12] P. Masci, R. Rukšėnas, P. Oladimeji, A. Cauchi, A. Gimblett, Y. Li, P. Curzon, and H. Thimbleby. On formalising interactive number entry on infusion pumps. *Electronic Communications of the EASST*, 45, 2011.

[13] P. Oladimeji, H. Thimbleby, and A. Cox. Number entry and their effects on error detection. In P. Campos et al., editors, *Interact 2011*, number 6949 in Lecture Notes in Computer Science, pages 178–185. Springer Verlag, 2011.

[14] R. Rukšėnas, P. Masci, M. D. Harrison, and P. Curzon. Developing and verifying user interface requirements for infusion pumps: A refinement approach. *Electronic Communications of the EASST*, 69, 2013.

[15] R. Silva and M. Butler. Supporting reuse mechanisms for developments in event-b: Composition. Technical report, University of Southampton, 2009.

[16] H. Thimbleby. Interaction walkthrough: evaluation of safety critical interactive systems. In G. Doherty and A. Blandford, editors, *Interactive Systems: Design, Specification and Verification*, number 4323 in Lecture Notes in Computer Science, pages 52–66. Springer Verlag, 2007.

[17] Safety requirements for the generic PCA pump. `http://rtg.cis.upenn.edu/gip-docs/Safety_Requirements_GPCA.doc`. Accessed: 04.04.2013.

[18] US Food and Drug Administration. Guidance for the content of premarket submissions for software contained in medical devices, May 2005.

[19] S. Yeganefard and M. Butler. Structuring functional requirements of control systems to facilitate refinement-based formalisation. In *Proceedings of the 11th International Workshop on Automated Verification of Critical Systems (AVoCS 2011)*, volume 46. Electronic Communications of the EASST, 2011.