chi+med
making medical devices safer

EPSRC Programme Grant EP/G059063/1

Public Paper no. 135

The Benefits of Formalising Design Guidelines: A Case Study on the Predictability of Drug Infusion Pumps

Paolo Masci, Rimvydas Rukšėnas, Patrick Oladimeji,
Abigail Cauchi, Andy Gimblett, Yunqiu Li,
Paul Curzon & Harold Thimbleby

Masci, P., Rukšėnas, R., Oladimeji, P., Cauchi, A., Gimblett, A., Li, Y., Curzon, P., & Thimbleby, H. (2015). The benefits of formalising design guidelines: A case study on the predictability of drug infusion pumps. *Innovations in Systems and Software Engineering, 11,* 73–93.

PP release date: 10 June 2013

file: WP135.pdf

UCL UCLIC Swansea University Prifysgol Abertawe FITLAB Queen Mary University of London CITY UNIVERSITY LONDON EPSRC Engineering and Physical Sciences Research Council

Noname manuscript No.
(will be inserted by the editor)

The benefits of formalising design guidelines: A case study on the predictability of drug infusion pumps

Paolo Masci · Rimvydas Rukšėnas · Patrick Oladimeji · Abigail Cauchi · Andy Gimblett · Yunqiu Li · Paul Curzon · Harold Thimbleby

Received: XXX / Accepted: YYY

Abstract A demonstration is presented of how automated reasoning tools can be used to check the *predictability of a user interface*. Predictability concerns the ability of a user to determine the outcomes of their actions reliably. It is especially important in situations such as a hospital ward where medical devices are assumed to be reliable devices by their expert users (clinicians) who are frequently interrupted and need to quickly and accurately continue a task. There are several forms of predictability. A definition is considered where information is only inferred from the current perceptible output of the system. In this definition, the user is not required to remember the history of actions that led to the current state. Higher-order logic is used to specify predictability, and the Symbolic Analysis Laboratory (SAL) is used to automatically verify predictability on real interactive number entry systems of two commercial drug infusion pumps — devices used in the healthcare domain to deliver fluids (e.g., medications, nutrients) into a patient's body in controlled amounts. Areas of unpredictability are precisely identified with the analysis. Verified solutions that make an unpredictable system predictable are presented through design modifications and verified user strategies that mitigate against the identified issues.

Keywords Predictability, Interactive System Design, Model Checking, Higher Order Logic, SAL

Paolo Masci, Rimvydas Rukšėnas, Paul Curzon
Queen Mary University of London
School of Electronic Engineering and Computer Science
E-mail: {paolo.masci,rimvydas,paul.curzon}@eecs.qmul.ac.uk

Patrick Oladimeji, Abigail Cauchi,
Yunqiu Li, Harold Thimbleby
Future Interaction Technology Lab
Swansea University, www.fitlab.eu
E-mail: {cspo,csabi,yunqiu.li,h.thimbleby}@swansea.ac.uk

1 Introduction and motivation

Infusion pumps are medical devices used to deliver drugs to patients at controlled rates. They are "programmed" by clinicians, and the process consists of interacting with buttons on the pump user interface for navigating through menus and entering values to set the infusion parameters. Infusion pumps are used in hospital wards, and increasingly in the patient's home. These devices have become a major concern because of several unexpected accidents due to use errors. For instance, a typical problem reported in drug adverse events with infusion pumps is that a clinician may enter a number ten times larger than intended [41]. Under-dosing is also a problem: if a patient receives too little of a drug, their recovery may be delayed or they may be in unnecessary pain.

Such errors, unfortunately, are not rare. In the UK, for instance, a recent bulletin from the UK government agency MHRA (the UK Medicines and Healthcare products Regulatory Agency), which is responsible for ensuring that medicines and medical devices work, reports that between the years 2005 and 2010 there were more than one thousand incidents involving infusion pumps in the UK alone (this is likely to be under-reported). A series of the errors were due to number entry [23]; examples include setting the wrong rate, confusing primary and secondary rates, and not confirming the set rate or the configuration.

Although use error is claimed as the primary causal factor of many of these incidents, careful enquiries usually suggest that actions carried out by clinicians typically have reasonable explanations, and the combination of multiple small failures is typically the cause of these incidents rather than unskilled behaviour or negligence [28,30]. In the US, for instance, the FDA has

analysed incidents due to infusion pumps, and there is evidence that these use errors are actually caused by the device design [12].

In order to contrast this negative trend, the FDA has launched the Infusion Pump Improvement Initiative with the aim to promote using verification techniques for analysis and development of infusion pumps. As part of this initiative, Kim et al [20] have demonstrated how a model-based development approach can be used to implement software for the control logic of a Patient Controlled Analgesia (PCA) pump prototype that is verified against the Generic PCA (GPCA) safety requirements provided by the FDA.

Our work presented in this paper complements the above in that the focus is on the verification of user interfaces of commercial infusion pumps. It takes a different approach with a focus on interaction design. That is, in this work we demonstrate how verification tools can be used to identify and solve problems related to predictability of a user interface [13]. Predictability is a design principle concerning the ability of a user to determine the outcomes of their actions on a device user interface reliably. This property is especially important for safety-critical subsystems of the device user interface (e.g., the number entry system) in situations such as a hospital ward where medical devices are assumed to be reliable devices by their expert users (clinicians) who are frequently interrupted and need to quickly and accurately continue a task.

In the next section, we motivate the choice of predictability by illustrating how it relates to high-level design principles presented in the ANSI/AAMI HF75:2009 human factors standard. This standard is used as guidance document by designers of user interfaces for medical devices and by medical device regulators such as the US Food and Drug Administration.

2 Predictability and the ANSI/AAMI HF75:2009 standard for medical devices

The ANSI/AAMI HF75:2009 standard has been developed by the Association for the Advancement of Medical Instruments (AAMI) in 2009 with the aim to create a reference document that covers general human factors engineering principles for the development of interactive medical devices. The principles covered in the standard have several links with the predictability property. In the following we provide excerpts from the standard and discuss these links.

> "Users can be forgetful, become distracted by other tasks, or be interrupted during device use. Therefore, designers should not depend on users to remember information needed to perform a task. It is far better to present to users the crucial information they need to perform the task correctly." — HF75:2009, Chapter 4, Section 6.4

The definition of predictability that we consider ensures that users can operate the device reliably and with confidence without remembering the history of past actions. There is empirical evidence that interruptions have a disruptive impact on people's performance and reliability [39]. If nurses are interrupted while setting up an infusion, they need to stop their task, turn their attention to the interrupting task, and then resume the infusion task. If the pump user interface does not show enough information to enable them to determine the exact device state, then they may fail to correctly resume the task. For example, a nurse might think they completed a step that they had not and so on resumption, they incorrectly skip that step. Laboratory evidence suggests that being confused about the step to resume on is a common problem on resumption [5]. Furthermore, if nurses are aware of this potential hazard, they may try to apply workarounds, such as resetting the device and starting over again. Such workarounds considerably slow down the number entry task which is a problem in itself, but may also result in new issues. For instance, as a side-effect such workaround actions may reset other parameters of the device, such as the unit of measurement, without the nurse realising.

> "People usually have a mental model or expectation of how a new device works. Usually, this expectation is based on previous experience using similar devices. Users might expect certain controls to function in a particular manner and are surprised if a control functions differently." — HF75:2009, Chapter 4, Section 8.1

Mental models are conceptual models developed by users to explain how things work [25]. Mental models developed by expert users *can* be highly accurate. However, even with such expertise, users frequently make errors and depart from these procedures, particularly when situations change or fail to meet expectations. This can occur in boundary situations that are rarely encountered so not built in to the conceptual model developed by users. Reliance on perceptible cues can help users understand the situation. Predictability ensures that those cues (whatever they are) are always there and are accurate.

> "One way that designers seek to simplify medical devices is to incorporate multiple operational modes. In principle, multiple operational modes are a sensible way to facilitate context-specific tasks

> and to limit user exposure to extraneous capabilities. However, problems can arise if the user does not realise the medical device is in the wrong mode." — HF75:2009, Chapter 4, Section 9.6

Predictable devices allow users to tell what mode the device is in. The limited physical dimension of devices and their ever-increasing number of functionalities push developers to overload the individual user interface elements. For instance, the *up* button on a pump user interface, which is typically used to increase the infusion rate on these devices, may also be used to provide other functionality in some device modes such as, confirming the infusion rate, undoing the last action, or recalling a rate from the device memory. However, even though the outcome of this mode-dependent functionality affects the interaction, its meaning in the current mode might not be readily visible to the user. This would make it difficult for the user to tell what exactly, in this case, the *up* button does in the current state of the system and hence how to plan future actions. It is also worth noting that skilled users, such as clinicians, tend to commit errors of the "strong but wrong" type [30]. That is, they are committed with misplaced confidence where, say, an attentional check might be omitted or mistimed, or where some aspect of the environment is misinterpreted. It is also worth noting that predictability is important in situations where undo operations are not possible. This is vital for infusion pumps at the point where an infusion is started, whatever the design. Once drug is pumped into a patient this can only be stopped not undone, and the effects of getting this wrong are safety critical.

3 Overview of the approach

The predictability analysis checks whether a device user interface *enables* users to tell what state the device is in simply by looking at its current persistent output and then predict the next state generated by interacting with the device (e.g., by clicking a button). This analysis targets critical subsystem of user interfaces, such as the number entry system.

This form of predictability has been formalised in[14, 13] through the Program-Interpretation-Effect (PIE) framework [14,15], which describes interactive systems in terms of sequence of commands issued by users (denominated *programs*), device states perceived by the users (denominated *effects*), and relations between command sequences and their effects on perceived device states (denominated *interpretations*). Following the notation of the PIE framework, a predictable system is defined as follows:

$$predictable(e) \triangleq \forall p, q, r \in \mathcal{P} : (I(p) = e = I(q)) \Rightarrow (I(pr) = I(qr))$$

where: $\mathcal{P}$ is the set of sequences of actions (key-presses in this case) that can be performed through the device user interface; $I : \mathcal{P} \rightarrow \mathcal{E}$ is the set of all possible computations performed by the device, where $\mathcal{E}$ is the set of observable states of the device.

A bisimulation-based approach is used in this work to specify and verify the above definition of predictability. Two models are specified: one model, that we call the *device model*, defines the interactive behaviour of the device; the other model, that we call the *prediction model*, defines the user's knowledge of the device. The prediction model used in the analysis encapsulates the following hypotheses on the user's knowledge: (1) the user makes decisions only on the basis of observable information provided by the device through its user interface; (2) the user has no memory of past device states or history of performed actions; (3) the user has a correct understanding of the functionalities of the device. Given the prediction and the device model, an equivalence relation is thus established on the observable device state. If the device model and the prediction model always *match*, that is the values of the corresponding variables in the observable state of the device and prediction models are equal in all the reachable states of the device, then the device is predictable. The Symbolic Analysis Laboratory SAL [24] is used in section 6 to perform this predictability analysis for the number entry systems of two commercial drug infusion pumps.

Because of the hypotheses imposed on the prediction model, that model and the device model share several behaviours. A procedure for building a prediction model from a device model is the following.

The initial prediction model is a simplified device model. The behaviour of this simplified model is obtained from the specification of the device behaviour by removing from it all conditions and transitions that are not observable from the device user interface. For instance, one of the analysed devices has an auxiliary memory for enabling undo at boundary cases. If the value stored in this auxiliary memory used by the device is not externalised on the device user interface, then the auxiliary memory is not included in the state of the prediction model, as well as any transition or condition based on the value stored in memory.

Iterative model refinement is used for eliminating mismatches due to oversimplification of device behaviours in the initial prediction model. This process makes explicit the implicit relations between observable state variables and hidden variables used by the device (e.g.,

the auxiliary memory used by the device is always clear when the display shows certain values). To refine the initial prediction model and still maintain the hypotheses on the user's knowledge, only conditions over the current observable device state can be introduced. New conditions are included in the prediction model until either the prediction model and the device model always match (in this case, the device is predictable), or until a situation is found where a refinement cannot be found that resolves the mismatch. In this second case, we say that the device is *not predictable*.

The simplicity of the procedure described above has the advantage that it can be easily implement in different languages and verification tools. A drawback is that several iterations may be required to refine the initial prediction model. This happens when the user interface has complex behaviours linked to variables that are not externalised on the device user interface. As such, the need of several iterations in itself is a symptom that the user interface design may need to be revised (even if predictability succeeds at the end) as the mental model that users would need to develop is likely to be too complex. It is worth recalling that the predictability analysis performed here is performed to help discover potential problems with interaction design in safety-critical parts of a user interface, such as the interactive number entry system of an infusion pump, rather that the whole user interface behaviour.

When predictability fails, the analysis provides a means to generate two kinds of recommendations: *verified design solutions* that allow to modify the device user interface behaviour and make it predictable; *verified user strategies* that can be easily applied by users to avoid the area of unpredictability in the design of the device user interface. The former is useful for device manufacturers in that it allow to evaluate the consequences of design alternatives that include different features. The latter is useful for user training, in that we can check whether a reasonably simple strategy exists (other than resetting the device) that allows one to circumvent the predictability issues evidenced in the analysis.

3.1 Contribution

We show via an example based on commercial devices how formal methods can help to check whether device designs are predictable. Specifically, an accepted definition of predictability is formalised, and a demonstration is presented of how it can be checked against the behaviour of the number entry system of two real infusion pumps, the Alaris GP [17] and the B-Braun Infusomat Space [4] (current models, 2011). The general style of these user interfaces is common to many such devices and so our approach is likely to apply more generally. The Symbolic Analyser Laboratory (SAL) [24] model checker is used to verify whether the predictability property holds for these device models. When predictability fails, consideration of failure traces from the model checking help discover solutions to the identified problems. These solutions are given either via design modifications or, if fixing the design itself is not feasible, through verified user strategies that if followed mitigate against the problem.

The contributions of this work are therefore: (1) an approach to verify predictability with automated verification tools; (2) a demonstration through an example that commercial devices can be very effectively analysed for their compliance to predictability; (3) a demonstration that problems can be precisely identified **and hence fixed** where this is deemed appropriate given the trade-offs involved.

This paper extends our previous work [22] as follows. We illustrate in detail the complete specification of the real number entry systems of the two considered infusion pumps. In particular, for the Alaris-GP, we include *press & hold* interactions where a key is pressed and held down for a certain amount of time and then released. We make clear the hypotheses behind the prediction model and present an approach for generating the prediction model from the device model. We present design recommendations for making the system predictable, and verified user strategies that mitigate against the lack of predictability of the two analysed devices.

3.2 Structure of the paper

In section 4, an overview of recent related work about formal analysis of interactive systems is provided. In section 5, a formalisation of the interactive number entry systems of two commercial drug infusion pumps is developed. In section 6, predictability is formalised in higher order logic, and a predictability analysis is performed for the developed specifications. In section 7, a discussion of why the property fails for both devices is provided, and possible solutions and recommendations to avoid predictability issues are illustrated. Conclusions are drawn in section 8.

4 Related work

In this section, we illustrate some recent work that directly relates to ours.

Campos and Harrison [9] used Modal Action Logic (MAL) [33] and the IVY tool [8] to analyse the interactive behaviour of the BBraun and Alaris pumps considered in this work. Their motivating problem is procurement of medical devices. They demonstrate the utility of performing a systematic comparison between device designs with model checking — subtle design variations can be systematically identified and evaluated. They use a layered approach to support model reuse: an 'inner' layer describes the temporal evolution of the infusion process; a 'middle' layer describes the mode structure and the information displayed by the pump; an 'outer' layer describes the normative tasks typical of the clinician's use of the device. The inner layer is common to all pumps, while the middle layer is device– and design-specific. The outer layer depends on the expected context of use of the device. A "battery" of interaction properties is then verified on the pump model for comparing different designs. Properties include mode clarity, consistency of actions, and appropriate feedback for critical actions. Their work complements ours in that they focus on device modes and other interaction properties, such as mode clarity and consistency of functions, while here we focus on predictability and analyse a detailed specification of the interaction design of the number entry system.

Thimbleby and Gimblett [38] discuss the causes of loss of predictability in interactive user interfaces. In particular, they argue that one of the main contributing factors of unpredictability is that manufacturers use ad hoc number entry methods — apparently identical user interfaces may therefore have completely different behaviours. They remark how the problem is particularly evident when considering the way syntax errors are handled by devices. In order to mitigate the problem, they developed an approach for implementing dependable interactive number entry. Their approach, denominated correct-by-construction user interface, is based on finite state machines. They use regular expressions for specifying the features of the interface, and then a compiler for generating the finite state machine. They demonstrate how the approach can be applied for implementing an interface that follows the rules defined by the Institute for Safe Medication Practices (ISMP) [1] for writing numbers in a safer way (e.g., write 0.5 instead of .5, as the latter may be easily misread as 5). This work shares with ours the concern that dependable interactive data entry interfaces should be predictable, and they aim to tackle the problem at the root, by developing tools that allow developers to be clear about their design decisions.

Rushby's work on mode confusion [31,6] relates to our work. He used model checking approaches for comparing plausible mental models developed by users and the actual implementation of the system. He argues that any strong divergence between mental models and device models is a potential cause of "automation surprises", i.e., situations where the automated system behaves in a way that is different from that expected by the operator. He proposed a constructive method for deriving mental models from the specification of the interactive systems [32], and he applied the approach to the analysis of an MD-88 autopilot system. He generates the specification of a mental model by simplifying the interactive system specification through rules reflecting psychological processes, such as frequential simplification [19]. Starting from a simple description of the dynamics of the aircraft, when a significant divergence is found between the abstract model and a pilot's mental model, he refines the model until either the divergence is discharged or a credible anomalous scenario is found.

Degani and Heymann [11] describe a systematic approach for evaluating whether a device interface provides the necessary information. They argue that this is a necessary precondition for enabling operators to perform specified tasks correctly and reliably. They perform a systematic comparison between the behaviour of device user interfaces and mental models of operators. Such descriptions are both simplified versions of the device's behaviour, and they aim to verify that they are correct with respect to the specification of the device when operators perform normative tasks. They use an approach based on state-machines, where the verification consists in checking that the parallel and synchronised execution of the interface and mental models consistently match. They show that they are able to identify situations where operators are unaware that certain events can take place. They illustrate the approach by analysing a model of the autopilot system of a flight control system.

Our work draws ideas from Rushby's and Degani and Heymann's work. In particular, we embrace the idea that a systematic comparison between mental models and actual device specifications is an effective way of checking properties of interest on interactive system. In our work, we systematically compare a *prediction model* with the actual interface specification. The prediction model is essentially a mental model of an idealised expert user that knows all functionalities of the device, but makes decisions only on the basis of the persistent observable state of the device (e.g., what is shown on the device displays). The argument about using an idealised expert user is that it allows us to perform a conservative analysis — if the idealised expert user is not able to predict the next observable state, neither a real user could. When the verification fails, the model

checker shows a counterexample that provides precise insights about why predictability failed.

Bolton and Bass [7] used SAL [24] to verify a model of the Baxter iPump. They verify whether some basic *normative* tasks (i.e., sequences of actions described in written documents, such as user manuals) are properly supported by the device. Examples of properties include: turning on and off the pump, stopping the infusion, and entering a volume to be infused. They developed a graphical modelling language, denominated Enhanced Operator Functional Model (EOFM), for specifying tasks in such a way that non-experts of formal methods could be able to inspect the specifications and the traces generated by the model checker. Their paper mainly focuses on the technical lessons learnt when modelling an interactive drug infusion pump. They illustrate that they had to reduce the model in order to tackle state space explosion, and argue that the verification approach they have used needs to be revised if a real system is to be verified. In our work, rather than considering a simplified model of the pumps, we focus on a detailed model of the interactive number entry system. Although we don't explicitly model tasks and work environment, some aspects of them are captured implicitly by the interaction design principle we consider. The verification approach we used that systematically compares a prediction model and the specification of the interactive number entry system allowed us to consider the full range of values, without the need of specific simplifications.

Kim et al [20] applied a model-based engineering approach for generating software codes for a prototype infusion pump from verified specifications. Their work has been carried out within the Generic Infusion Pump (GIP) [3] project, whose aim is to develop a set of generic safety requirements for the software codes executed by programmable drug infusion pumps. In their work, they start from a formal specification given as timed automata, then verify safety requirements and, if the requirements are successfully verified, they automatically translate the timed automata into C code. They demonstrate the approach by generating software codes for a prototype infusion pump. Our work complements this work in that we focus on predictability of the interaction design, and our analysis aims also to verify whether appropriate user strategies can be defined for mitigating deficiencies in existing pumps designs.

5 Formal specification of the interactive number entry systems of two infusion pumps

A detailed specification of the behaviour of the interactive number entry systems of two real medical devices is now developed. The considered devices are the B-Braun Infusomat Space [4] and the Alaris GP [17] infusion pumps. These devices were chosen as typical examples of commercial infusion pumps, though where different design decisions have been taken in the design of their number entry systems. Anecdotal evidence suggests similar issues are likely to arise if other makes and models were analysed.

The actual values displayed by the devices are modelled, as well as the actual action-effect relation of interactions through the buttons on the device user interfaces. The developed models are state machines. Information relevant to the display of the device user interface is modelled as part of the device state, and functionalities of the device user interface are then specified using transition functions over device states. The higher-order logic specification language of SAL [24] is used. It is based on typed higher-order logic, and includes, among other types, function, tuple, and record type constructors for the definition of new types. The function type with domain type `D` and range type `R` is denoted `[D -> R]`. Relevant features of the SAL specification language are illustrated further as needed when presenting the developed specifications.

The specifications were obtained by reverse-engineering the behaviour of the real devices using interaction walkthrough [36]. We thus reverse engineered the specifications used below from the user documentation together with careful manual exploration of the actual devices, following an iterative methodology until we had accurate specifications. This approach is potentially error-prone. Our results apply to the specifications as reverse engineered and clearly may not actually apply to the real devices if there are errors. Nevertheless, this is acceptable for the aim of this paper, which is to demonstrate how verification tools can be used to verify interaction properties on real devices. In principle, formal specifications could have been derived from the actual software codes of the devices if these were available, perhaps as provided by the manufacturers. Thus if adopted by manufacturers this would not be a limitation in the use of the approach described. In particular, we are showing that the analysis and the general approach can help discover potential issues in the details of the interaction design of devices, and can help identify how to fix them. The predictability issues identified by the analysis performed in section 6 can be reproduced on the real devices.

In the following, for each of the two considered drug infusion pumps, a description of the behaviour of the interactive number entry system is provided, followed by a detailed illustration of the developed higher-order logic specifications. In section 6 the predictability of the

Fig. 1: B-Braun Infusomat Space programmable infusion device.

behaviour described by the developed specifications is then analysed.

5.1 B-Braun Infusomat Space

The B-Braun Infusomat Space's number entry system is an example of "5-keys" user interfaces [10]: four arrow keys and a confirmation button (see figure 1). The up and down keys increase or decrease the current number by 10^{cursor} respectively, where *cursor* denotes the cursor position. The left and right arrow keys are used to change the cursor position. The left key increases the cursor position and the right key decreases the cursor position. In the developed model, the convention is used that the cursor is (at position) 0 when it selects the unit value on the number displayed. The cursor position ranges from -2 (i.e., cursor on the thousandths digit) to 4 (i.e., cursor on the ten-thousands digit). The cursor position is manually selected by the user. This reflects the capabilities of the real pump user interface. Initially, the cursor is on the units digit. The device has an auxiliary memory for restoring the last displayed value when the action-effect of pressing a button causes the displayed number to overshoot the maximum or the minimum values handled by the device.

The BBraun pump is capable of handling a range of infusions that go beyond the display capabilities. Namely, while the display can show up to five digits, the actual range displayed forms a sliding window between thousandths and ten-thousands. A non-zero digit in some position can prevent access to other positions. The window does not slide uniformly, and there are in fact three possible ranges: *hundredths to tens* (e.g., 99.99), *tenths to hundreds* (e.g., 999.9), and *units to ten-thousands* (e.g., 99999); note that the first two ranges are four digits wide, whereas the third is five wide, so it is not possible to enter 9999.9, for example. Furthermore, in the *hundredths to tens* range, the lowest non-zero value the device allows to be displayed is 0.1, which is *not* the lowest syntactically valid value in that range (i.e., 0.01); values between 0 and 0.1 cannot be entered or displayed, in fact.

Specification. The device behaviour is specified as a state machine. State transitions correspond to the action-effect of clicking one of the four arrow buttons on the device user interface: functions `bbraun_up` and `bbraun_down` model the effect of clicking the up and down button; the effect of clicking the left and right buttons is modelled through functions `bbraun_left` and `bbraun_right`. When a button is pressed and held down, the button behaves as in the case of iterative button clicks.

The state of the device user interface, which is shown in Listing 1, is a record type (`bbraun_state`) defining the minimal information needed to specify the behaviour of the number entry system of the device. Type `bbraun_state` consists of three fields: the current display value, of type `bbraun_real`, which defines the domain of the numbers handled by the device when in rate mode (`bbraun_real: TYPE = [0..max_display]` where `max_display = 99999`); the current cursor position, of type `bbraun_cursor`; and the current content of the memory, of type `bbraun_memory`. In the specification, we use the constant `NA` for specifying a clear memory.

Listing 1: Type definition for the BBraun state

```
1 bbraun_state: TYPE =
2  [# display: bbraun_real,
3     cursor : bbraun_cursor,
4     memory : bbraun_memory #];
```

The state machine defining the overall behaviour of the device user interface is given in Listing 2. It is specified in SAL with a *module* that includes the initial state of the device user interface (in the `INITIALIZATION` section), and the state transitions, given as guarded commands (in the `TRANSITION` section). The model is initialised so that the value of the display is *0*, the cursor is in the position *0* and the memory is clear. The input variable `event` represents button clicks (`up, down, left, right`). Each guarded command specifies a state transition that is triggered by the corresponding event (primed variables represent new values). Thus, our model `bbraun_device` generates all possible sequences of button clicks and the associated changes of the device state derived from a specific initial state.

Listing 2: State machine for the BBraun

```
1 bbraun_device : MODULE =
2 BEGIN
3 INPUT event: Event
4 OUTPUT st: bbraun_state
```

```
5  INITIALIZATION st = (# display := 0,
6                       cursor  := 0,
7                       memory  := NA #);
8  TRANSITION
9  [  event = up    --> st' = bbraun_up(st)
10 [] event = down  --> st' = bbraun_dn(st)
11 [] event = left  --> st' = bbraun_lf(st)
12 [] event = right --> st' = bbraun_rt(st)
13 ] END
```

In the following, a detailed illustration of the developed transition functions is presented. The functions model button clicks. In the specification, `pow10` is used to compute the value of powers of ten to a natural number (`pow10(n)` $= 10^n$).

bbraun_up This function models the action-effect of clicking the *up* arrow button. Up button clicks are ignored and the device emits a beep when the number on the display is already the maximum allowed value (99999). Otherwise, when the up button is clicked, the device displays a new value obtained by adding 10^{cursor} to the value currently displayed (see Listing 3, lines 7–15). There are a number of exceptions to this basic behaviour. First, the precision of the value displayed by the device after an up button click is as follows: below 100, the precision is of two fractional digits; between 100 and 1000, the precision is limited to one fractional digit; above 1000, the fractional part is always discarded. The precision of the digits is obtained with the `floor` function (see Listing 3, lines 14–15) that discards the fractional part of the number. Second, the actual number displayed after an up button click depends on the content of the device memory. Namely, if the device memory is not empty, then the up button acts like a *recall memory* button (see Listing 3, lines 23–26).

More precisely, the display value is updated according to the following rules when the up button is clicked.

If the current displayed value plus 10^{cursor} overshoots the maximum value (99999), then the displayed value is stored in memory and the display gets updated with the maximum value (Listing 3, lines 17–22); for instance, when the display shows 90010 and the cursor is on the ten thousands digit, if the up button is clicked then 90010 is stored in memory and the display shows 99999.

If the current displayed value plus 10^{cursor} *does not overshoot* the maximum value, the effect of the up button click depends on the content of the device memory (Listing 3, lines 23–27). If the memory stores a number, then the up button acts as a recall memory button (e.g., if the memory contains 100, an up button click will change the value shown on the display to 100, regardless of the number currently shown on the display); otherwise, the displayed number is increased by 10^{cursor} (e.g., when the display is 10 and the cursor is on the hundreds decimal, if the up button is clicked then the new displayed value is 110). The memory is cleared in either case.

Listing 3: BBraun model, *up* button clicks

```
1  bbraun_up(st: bbraun_state): bbraun_state =
2  IF display(st) = max_display THEN st
3  ELSE
4    LET val: bbraun_real = display(st),
5        cur: bbraun_cursor = cursor(st),
6        mem: bbraun_memory = memory(st),
7        new_val: real =
8         IF  val + pow10(cur) < 0.1 THEN 0.1
9         ELSIF val + pow10(cur) >= 0.1
10         AND val + pow10(cur) < 100
11        THEN val + pow10(cur)
12        ELSIF val + pow10(cur) >= 100
13         AND val + pow10(cur) < 1000
14        THEN floor((val + pow10(cur))*10)/10
15        ELSE floor(val + pow10(cur)) ENDIF
16   IN
17 IF new_val > max_display
18 THEN st WITH .display := max_display,
19         WITH .memory := IF valid?(mem)
20                         THEN memory(st)
21                         ELSE new_mem(val)
22                         ENDIF
23 ELSE st WITH .display := IF valid?(mem)
24                          THEN value(mem)
25                          ELSE new_val
26                          ENDIF,
27         WITH .memory := NA ENDIF ENDIF;
```

bbraun_down This function models the action-effect of clicking the *down* arrow button. If the number displayed by the device is zero, then down button clicks are ignored. Otherwise, when the *down* button is clicked, the device computes a new value by subtracting 10^{cursor} to the value currently displayed (see Listing 4, line 10). As for the up button, there are several exceptions to this basic behaviour. First, the precision of the value displayed by the device after a down button click has the same constraints explained for the up button clicks (these limits are specified in Listing 4, lines 10–18): below 100, the precision is of two fractional digits; between 100 and 1000, the precision is limited to one fractional digit; above 1000, the fractional part is always discarded. Additionally, for the down button, the device restricts the minimum value that can be entered according to the cursor position (see Listing 4, lines 7–9). Namely, if the cursor is on the thousands digit, the minimum value is 1; otherwise, the minimum allowed value is 0.1. Second, the actual number displayed after a down button click depends on the content of the device memory. As for up button clicks, if the device memory is not empty, then the down button acts like a *recall memory* button.

In more detail, the display value is updated according to the following rules when the down button is clicked.

If either the current display value is the minimum allowed rate or the current display minus 10^{cursor} is zero then the display value is updated to zero and the device memory is cleared (see Listing 4, lines 20–22);

If the current display minus 10^{cursor} overshoots the minimum allowed rate, then the display is updated to the minimum allowed rate, and the device memory is either updated with 10^{cursor} or kept unchanged (see Listing 4, lines 23–28). The device memory is updated with 10^{cursor} when the memory is clear and the cursor is on an integer digit (e.g., when the display shows 10 and the cursor is on the thousands digit, if the down button is clicked then the display will show 0.1 and the value 1000 is stored in memory); the device memory is unchanged when the cursor is on a fractional digit (e.g., when the display shows 0.11 and the cursor is on the first fractional digit, if the down button is clicked then the display will show 0.1 and the memory is unchanged);

If the current display minus 10^{cursor} *does not overshoot* the minimum allowed rate, the behaviour of the down button depends on the device memory (see Listing 4, lines 29–34). If the memory contains a number, then the down button acts as a recall memory button (e.g., if the memory contains 910, a down button click will change the value shown on the display to 910, regardless of the number currently shown on the display[1]); otherwise, the displayed number is decreased by 10^{cursor}.

Listing 4: BBraun model, *down* button clicks

```
1  bbraun_down(st: bbraun_state):bbraun_state =
2  IF display(st) = 0 THEN st
3  ELSE
4  LET val: bbraun_real = display(st),
5      cur: bbraun_cursor = cursor(st),
6      mem: bbraun_memory = memory(st),
7      min_val: bbraun_real =
8         IF cur >= 3 AND val >= 1
9         THEN 1 ELSE 0.1 ENDIF,
10     new_val: real =
11        IF  val - pow10(cur) < 0.1 THEN 0
12        ELSIF val - pow10(cur) >= 0.1
13         AND val - pow10(cur) < 100
14        THEN val - pow10(cur)
15        ELSIF val - pow10(cur) >= 100
16         AND val - pow10(cur) < 1000
17        THEN floor((val - pow10(cur))*10)/10
18        ELSE floor(val - pow10(cur)) ENDIF
19 IN
20 IF val = min_val OR new_val = 0
21 THEN st WITH .display := 0
22        WITH .memory := NA
23 ELSIF new_val < min_val
24 THEN st WITH .display := min_val,
25        WITH .memory  :=
26         IF cur >= 0
27         THEN new_mem(pow10(cur))
28         ELSE mem ENDIF
29 ELSE st WITH .display :=
30         IF valid?(mem)
31         THEN value(mem)
32         ELSE min_infuse(min_val)
33                         (new_val) ENDIF,
34        WITH .memory := NA ENDIF ENDIF;
```

[1] Due to the constraints imposed by the functionalities of the other buttons, the down button *may* act as recall memory only when the display shows 99999.

bbraun_left This function (shown in Listing 5) models the effect of clicking the *left* arrow key button. If the cursor is on the ten-thousands digit, left button clicks are ignored (the maximum rate that can be entered is 99999). If the cursor position is not on the most significant (integer) digit, then left button clicks move the cursor left one position and clear the device memory.

Listing 5: BBraun model, *left* button clicks

```
1 bbraun_left(st: bbraun_state):bbraun_state =
2 IF cursor(st) = 4 THEN st
3 ELSE st WITH .cursor := cursor(st) + 1
4         WITH .memory := NA ENDIF;
```

bbraun_right This function (shown in Listing 6) models the effect of clicking the *right* arrow key button. The behaviour of the right button depends on the value currently displayed by the device. Namely, the device ignores right button clicks when the operation would hide non-zero digits at the left-most position of the display, that is either when the displayed value is above 1000 and the cursor position is on the units, or when the displayed value is between 100 and 1000 and the cursor position is on the first decimal digit. The device ignores right button clicks also when the cursor is on the second decimal digit (the maximum precision of the device is limited to two decimal digits). In all other cases, right button clicks move the cursor right one position and clear the device memory.

Listing 6: BBraun model, *right* button clicks

```
1 bbraun_right(st: bbraun_state):bbraun_state=
2 IF (display(st) >= 1000 AND cursor(st) = 0)
3  OR (display(st)>=100 AND display(st)<1000
4  AND cursor(st) = -1) OR (cursor(st) <= -2)
5 THEN st
6 ELSE st WITH .cursor := cursor(st) - 1
7         WITH .memory := NA ENDIF;
```

Fig. 2: Alaris GP programmable infusion device.

5.2 Alaris GP

The number entry system of the user interface on the Alaris GP has four buttons (see Figure 2). A pair of buttons is used to increase the value displayed and a second pair is used to decrease the value displayed. In each pair of buttons, one causes a change ten times bigger than the change caused by the other button. Typically, clicking either single chevron (arrow) key changes the last digit of the value displayed, and clicking either double chevron key changes the second to last digit of the value displayed. In this device, when a chevron button is *pressed & held* down, the display value is iteratively changed, and the device dynamically selects *step multipliers* to scale the amount by which the displayed number is increased or decreased. One multiplier leads to changes that are ten times larger than the other multiplier. The step multipliers are automatically selected by the device during the interaction. The selection depends on the value currently displayed by the device and on the amount of time a button has been held down by the user.

Specification. The behaviour of the Alaris pump is specified as a state machine. Differently from the BBraun model, here state transitions correspond to the action-effect of *button clicks* (i.e., a key is pressed and released immediately), and *button press & hold* (i.e., a key is pressed & held down for a certain amount of time, and then released). A convenient way to specify these behaviours is by splitting the definition of each key k into a pair of functions, one modelling the action of pressing (these functions will be named `alaris_press_k` in the model) and the action of releasing (`alaris_release_k` in the model) the button.

The device state, is a record type (`alaris_state`) defining the minimal information needed to specify the behaviour of the number entry system of the device. Type `alaris_state`, as shown in Listing 7, consists of three fields: `display`, which models the current display value; `timer`, a discrete timer with resolution of seconds whose value is used to model the amount of time a button is held down; `multiplier`, which models the step multiplier used by the interactive number entry system of the device. The display value is of type `alaris_real`, a bounded real number that models the actual domain handled by the device when in rate mode (`alaris_real:TYPE = [0..max_rate]`, where the maximum rate `max_rate` is 1200). The discrete timer is of type `alaris_timer`, a natural number below `max_timer` (5, in this case). The multiplier can be either small or large: the small multiplier is the constant 1 (the symbolic name `x1` will be used in the model for this constant), the large multiplier is 10 (the symbolic name `x10` will be used in the model for this constant). Initially, the display shows the number 0, the multiplier is `x1`, and the timer is 5.

Listing 7: Type definition of the alaris state

```
1 alaris_state: TYPE =
2  [# display   : alaris_real,
3     timer     : alaris_timer,
4     multiplier: alaris_multiplier #];
```

The state machine defining the overall behaviour of the device is in Listing 8. The transition system is initialised so that the value displayed is `0`, the initial timer is `max_timer` and the step multiplier is `x1`. The input variable `event` represents key presses (`press_up`, `press_down`, `press_UP`, `press_DOWN`) and key releases (`release_key`). At each step, `event` can take any of theses values, thus modelling arbitrary key sequences. The output variable `st` represents the pump state. Each guarded command specifies a state transition that is triggered by the corresponding event (primed variables represent new values).

Listing 8: State machine for the Alaris

```
1 alaris_device : MODULE =
2 BEGIN
3 INPUT event: Event
4 OUTPUT st: alaris_state
5 INITIALIZATION
6 st = (# display  := 0,
7         timer    := max_timer,
```

```
8          multiplier := x1 #);
9  TRANSITION
10 [  event = release_key
11     --> st' = alaris_release_key(st);
12 [] event = press_up
13     --> st' = alaris_press_up(st);
14 [] event = press_down
15     --> st' = alaris_press_down(st);
16 [] event = press_UP
17     --> st' = alaris_press_UP(st);
18 [] event = press_DOWN
19     --> st' = alaris_press_DOWN(st) ] END
```

The model generates all possible sequences of key press and key release and the associated changes of the device state derived from a specific initial state. For a correct definition of the transition system, we need to consider key press and release sequences such that no two keys are being pressed at the same time — the real device enforces this constraint by ignoring subsequent key presses until the first key has been released. For that to be the case, a key release event must precede any press of a different key in our model. This constraint can be imposed through an *observer* module that encapsulates the constraints and allows only legal press-release sequences. The observer module is shown in Listing 9. In the module, the output variable `prev_event` represents the previous event (either press or release) and the output variable `ok` is *true* only for legal sequences. The observer is thus composed to the device model for enforcing these constraints (this will be done when creating the Alaris system model for the predictability analysis, in section 6.3).

Listing 9: Alaris *observer* module

```
1  alaris_constraint : MODULE =
2  BEGIN
3  INPUT event: Event
4  OUTPUT prev_event: Event
5  OUTPUT ok: boolean
6  INITIALIZATION
7       ok = true; prev_event = release;
8  TRANSITION
9      [ ok AND event /= prev_event -->
10       ok' = (event = release
11                XOR prev_event = release);
12      prev_event' = event
13     [] ELSE --> ] END
```

In the following, the transition functions for key presses and key releases are illustrated in detail. In the specification function `trim` is used to enforce the range limits imposed by the real device when in *rate* mode. Namely, `trim(x)` returns x if $0 \leq x \leq max\_rate$, otherwise the function returns either 0 (if $x < 0$) or max_rate (if $x > max\_rate$).

alaris_press_up This function (shown in Listing 10) models the action-effect of pressing the *slow up* chevron key, which increases the number shown on the display of the device according to the following rules:

- if the number on the display is below one hundred, then the fractional part of the number is increased to the next decimal (see Listing 10, lines 5–6); for instance, if the display shows 9.1 and the small increase button is clicked, the display becomes 9.2;
- if the number is between one hundred and one thousand, then the unit digit of the number is increased to the next unit digit (see Listing 10, lines 7–9); for instance, if the display shows 123 and the small increase button is clicked, the display becomes 124);
- if the number is above one thousand, then the tens digit of the number is increased to the next tens digit (see Listing 10, lines 10–11); for instance, if the display shows 1080 and the small increase button is clicked, the display becomes 1090).

When pressed & held down, the slow up button iteratively executes button clicks. The step multiplier is always `x1` when interacting with this button.

Listing 10: Alaris model, *slow up* button presses

```
1  alaris_press_up(st: alaris_state)
2   : alaris_state =
3  LET m: alaris_multiplier = x1,
4      d: alaris_real =
5   IF display(st) < 100
6   THEN trim(floor((display(st)*10)+m) / 10)
7   ELSIF display(st) >= 100
8           AND display(st) < 1000
9   THEN trim(floor((display(st)) + m))
10  ELSE trim((floor(display(st)/10)+m)
11                * 10) ENDIF
12 IN st WITH .display := d;
```

alaris_press_UP This function (shown in Listing 11) models the action-effect of pressing the *fast up* chevron key, which increases the number shown on the display of the device. The increase is larger than that of the slow up key, and depends on the value of a step multiplier that is automatically selected by the device depending on the interaction. If the fast up button is pressed & held down for more than five consecutive display changes, then the multiplier changes from small (`x1`) to large (`x10`) either when the displayed number is below one hundred and a multiple of ten, or when the display is a multiple of one hundred. The discrete timer included in the `alaris_state` is used to support modelling this behaviour. When the fast up button is clicked (instead of press & hold actions) the multiplier never changes. In the following, the action-effect of pressing the fast up key when a given step multiplier is selected by the device is explained in detail.

When the selected step multiplier is `x1` (i.e., when the fast up button is either clicked or pressed & held

down at most five consecutive display changes), the number on the display is modified according to the following rules (as shown in Listing 11, lines 16–23; the multiplier is $s = 1$ in this case):

- if the number on the display is below one hundred, then the number is increased to the next unit digit (see Listing 11, lines 16–17); for instance, if the display shows 9.1 and the big increase button is pressed, the display becomes 10;
- if the number is between one hundred and one thousand, then the tens digit of the number is increased to the next tens digit (see Listing 11, lines 18–21); for instance, if the display shows 123 and the big increase button is pressed, the display becomes 130;
- if the number is above one thousand, then the hundreds digit of the number is increased to the next hundreds digit (see Listing 11, lines 22–23); for instance, if the display shows 1080 and the big increase button is pressed, the display becomes 1100

When the selected step multiplier is `x10` (i.e., when the fast up button is pressed & held down for more than five consecutive display changes), the number on the display is modified as follows (as shown in Listing 11, lines 16–23; the multiplier is $s = 10$ in this case):

- if the number on the display is below one hundred, then the number is increased by tens (see Listing 11, lines 16–17); for instance, if the display shows 30 and the big increase button is pressed, then the display becomes 40;
- if the displayed number is between one hundred and one thousand, then the number is increased by hundreds (see Listing 11, lines 18–21); for instance, if the display shows 300 and the big increase button is pressed, then the display becomes 400;
- if the displayed number is above one thousand, the number can be incremented by thousands (see Listing 11, lines 22–23); due to limits imposed on infusion rates, this type of increment is actually disabled on the real device.

For the analysed Alaris model, an additional constraint on the behaviour of the pump user interface is that the `x10` multiplier can be selected by the device only on numbers that are either a multiple of ten (when the display value is below 100) or a multiple of one hundred (when the display value is above 100). This is reflected in the conditions at lines 9–12 in the specification shown in Listing 11.

Listing 11: Alaris model, *fast up* button presses

```
1 alaris_press_UP(st: alaris_state)
2  : alaris_state =
3 LET t: alaris_timer =
4       IF timer(st) - 1 >= 0
5       THEN timer(st) - 1
6       ELSE timer(st) ENDIF,
7     s: alaris_multiplier  =
8       IF t = 0 AND multiplier(st) = x1
9        AND ((display(st) < 100
10       AND fractional(display(st),10) = 0)
11       OR (display(st) > 100
12       AND fractional(display(st),100)=0))
13      THEN x10
14      ELSE multiplier(st) ENDIF,
15    d: alaris_real =
16      IF display(st) < 100
17      THEN trim(floor(display(st))+s)
18      ELSIF display(st) >= 100
19              AND display(st) < 1000
20      THEN trim((floor(display(st)/10)+s)
21                  * 10)
22      ELSE trim((floor(display(st)/100)+s)
23                  * 100) ENDIF
24 IN (# display    := d,
25      timer      := t,
26      multiplier := s #);
```

alaris_press_dn This function models the action-effect of pressing the *slow down* chevron key. The function decreases the number shown on the display of the device according to rules that are almost symmetric to those of the slow up chevron:

- if the number on the display is below one hundred, then the fractional part of the number is decreased to the next decimal (see Listing 12, lines 5–6); for instance, if the display shows 9.1 and the small decrease button is pressed, the display becomes 9;
- if the number is between one hundred and one thousand, then the unit digit of the number is decreased to the next unit digit (see Listing 12, lines 7–9); for instance, if the display shows 123 and the small decrease button is pressed, the display becomes 122;
- if the number is above one thousand, then the tens digit of the number is decreased to the next tens digit (see Listing 12, lines 10–11); for instance, if the display shows 1080 and the small decrease button is pressed, the display becomes 1070.

When pressed & held down, the slow down button iteratively executes button clicks. The step multiplier is always `x1` when interacting with this button.

Listing 12: Alaris model, *slow down* button presses

```
1 alaris_press_down(st: alaris_state)
2  : alaris_state =
3 LET  m: alaris_multiplier = x1,
4      d: alaris_real =
5  IF display(st) < 100
6  THEN trim((ceil(display(st)*10)-m) / 10)
7  ELSIF display(st) >= 100
8          AND display(st)<1000
```

```
 9  THEN trim(ceil(display(st)-m))
10  ELSE trim((ceil(display(st)/10)-m)
11               * 10) ENDIF
12 IN st WITH .display := d;
```

alaris_press_DOWN The function models the action-effect of pressing the *fast down* chevron key, which decreases the number shown on the display. Similarly to the fast up key, the actual decrease depends on a step multiplier, which is automatically selected by the device as the button is pressed & held down. The multiplier is automatically changed from small (x1) to large (x10) either when the displayed number is below one hundred and a multiple of ten, or when the display is a multiple of one hundred. In the following, a detailed illustration of the action-effect of pressing the fast down key when a given step multiplier is selected by the device is presented.

When the selected step multiplier is x1 (i.e., when the fast down button is either clicked or pressed & held down up to five consecutive display changes), the function decreases the number shown on the display of the device according to the following rules (as shown in Listing 13, lines 21–28; the multiplier is $s = 1$):

- if the number on the display is below one hundred, then the fractional part of the number is decreased to the next unit digit (see Listing 13, lines 21–22); for instance, if the display shows 9.1 and the big decrease button is pressed, the display becomes 9;
- if the number is between one hundred and one thousand, then the tens digit of the number is decreased to the next tens digit (see Listing 13, lines 23–26); for instance, if the display shows 123 and the big decrease button is pressed, the display becomes 120;
- if the number is above one thousand, then the hundreds digit of the number is decreased to the next hundreds digit (see Listing 13, lines 27–28); for instance, if the display shows 1080 and the big decrease button is pressed, the display becomes 1000.

When the selected step multiplier is x10 (i.e., when the fast down button is pressed & held down more than five consecutive display changes), the number on the display is modified as follows (as shown in Listing 13, lines 21–28; the multiplier is $s = 10$):

- if the number on the display is below one hundred, then the number is decreased by tens (see Listing 13, lines 21–22); for instance, if the display shows 30 and the big decrease button is pressed, then the display becomes 20;
- if the displayed number is between one hundred and one hundred, then the number is decreased by hundreds (see Listing 13, lines 23–26); for instance, if the display shows 300 and the big decrease button is pressed, then the display becomes 200.
- if the displayed number is above one thousand, the number can be decremented by thousands (see Listing 11, lines 27–28); due to limits imposed on infusion rates, this type of increment is actually disabled on the real device.

Listing 13: Alaris model, *fast down* button presses

```
 1 alaris_press_DOWN(st: alaris_state)
 2  : alaris_state =
 3 LET t: alaris_timer =
 4       IF timer(st) - 1 >= 0
 5       THEN timer(st) - 1
 6       ELSE timer(st) ENDIF,
 7     s: alaris_multiplier  =
 8       IF t = 0 AND multiplier(st) = x1
 9        AND ((display(st) > 10
10        AND display(st) < 100
11        AND fractional(display(st),10) = 0)
12        OR (display(st) > 100
13        AND fractional(display(st),100)=0))
14       THEN x10
15       ELSIF t = 0 AND multiplier(st) = x10
16        AND (display(st) = 10
17        OR display(st) = 100)
18        THEN x1
19       ELSE multiplier(st) ENDIF,
20     d: alaris_real =
21       IF display(st) < 100
22       THEN trim(ceil(display(st)) - s)
23       ELSIF display(st) >= 100
24              AND display(st) < 1000
25       THEN trim((ceil(display(st)/10)-s)
26                        * 10)
27       ELSE trim((ceil(display(st)/100)-s)
28                        * 100) ENDIF
29 IN (# display    := d,
30       timer      := t,
31       multiplier := s #);
```

alaris_release_key In the considered Alaris pump, the action-effect of releasing a button is identical for all buttons. A single function is therefore used to model the effect of releasing any chevron key: the displayed value is left unchanged and the timer is reset to its initial value (max_timer). The higher order logic specification is in Listing 14.

Listing 14: Alaris model, *release key*

```
1 alaris_release_key(st: alaris_state):
2     alaris_state =
3 st WITH .timer := max_timer
4  .WITH multiplier := x1;
```

6 Analysis

The considered definition of predictability concerns whether it is possible to tell what state the number entry system of the device is in from the current state externalised by the device through the user interface. The number entry system of the device is not predictable if there is more than one possible state the device could move to as a result of some action when decisions are taken solely on the basis of the current observable state.

This form of predictability has been formalised in[14, 13] through the Program-Interpretation-Effect (PIE) framework [14,15], which describes interactive systems in terms of sequence of commands issued by users (denominated *programs*), device states perceived by the users (denominated *effects*), and relations between command sequences and their effects on perceived device states (denominated *interpretations*). Following the notation of the PIE framework, predictability is defined as follows:

$$predictable(e) \triangleq \forall p, q, r \in \mathcal{P} : (I(p) = e = I(q)) \Rightarrow (I(pr) = I(qr))$$

where: $\mathcal{P}$ is the set of sequences of key-presses that can be performed on the device user interface; $I : \mathcal{P} \rightarrow \mathcal{E}$ is the set of all possible computations performed by the device, where $\mathcal{E}$ is the set of observable states of the device.

A bisimulation-based approach is used to analyse predictability. A model, hereafter called the *prediction model*, that encapsulates the user's knowledge of the device according to considered definition of predictability is defined. An equivalence relation is thus established between the observable device states specified in the prediction model and in the device models. If the two models always *match*, that is the values of the corresponding variables in the device and prediction models are equal in all the reachable states of the device, then the (number entry system of the) device is predictable.

In the following, an approach is presented for deriving a prediction model from a device model.

6.1 Prediction model

The prediction model is a deterministic model that describes the behaviour of the system on the basis of information resources externalised by the device through its user interface (e.g., visible and audible cues). The prediction model can be assimilated to a mental model (i.e., a representation of the users' understanding of the system behaviour) of an idealised expert user that knows the functionalities of the device perfectly but makes decisions only on the basis of the current observable state. The concern with an idealised expert user means that, if predictability fails, any user equal or less experienced than the ideal (that is, any normal human user) will certainly be unable to predict the next state — thus the definition is conservative.

The prediction model encapsulates the following hypotheses about the user's knowledge: (1) the user makes decisions only on the basis of observable information provided by the device through its user interface; (2) the user has no memory of past device states or history of performed actions; (3) the user has a correct understanding of the functionalities of the device.

The prediction model and the device model share several behaviours. A procedure for building the prediction model from a device model follows.

The initial prediction model is a simplified device model. The behaviour if this simplified model is obtained from the specification of the device behaviour by removing from it all behaviours that are not observable from the device user interface. For the definition of predictability considered here, the observable state includes only the persistent state of the devices: the display value for the Alaris GP; the display value and the cursor position for the BBraun Infusomat Space. For these devices, therefore, transition functions in the prediction model are defined on the basis of information on the displays. They are obtained from the transition functions of the device by discarding all effects and all conditions over resources other than those included in the observable state.

Iterative model refinement is performed for eliminating mismatches due to oversimplification of the device behaviour in the initial prediction model. In some situations, discarding all conditions over hidden resources may lead to false positive, as the conditions discarded could have been replaced by additional conditions over the current observable state. Because of the hypotheses encapsulated in the prediction model, new conditions can be added that use only the current observable state of the device. For the considered devices, new behaviours added in the refinement steps are therefore still based on information reported on the device displays only. New conditions are included in the prediction model until either the prediction model and the device model always match (in this case, the device is predictable), or until a situation is found where a refinement cannot be found that resolves the mismatch. In this second case, we say that the device is *not predictable*.

Any divergence between the two models corresponds to a situation where two observationally equivalent device states lead to different observational states (when

hidden state variables are not considered). That is a transition from two states that are identical when comparing just observable information resources leads to two states with different observable information. This corresponds to situations where the system given by the parallel composition of the prediction model and the device model has non-deterministic behaviour. Dix calls *ambiguous* these states causing non-deterministic behaviour in the composed system model, because more than one observable state can be mapped to them [13]. The set of ambiguous states define the areas of unpredictability of the device. In order to identify precisely the areas of unpredictability, a systematic exploration of the state space of the composed system is required. In the following section, the SAL model checker is used for this purpose.

Before proceeding with the verification in SAL, it is worth noting that the approach for building the prediction model can be easily implemented in different languages and verification tools. A drawback of the approach is that several iterations may be required to refine the initial prediction model. This typically happens when the user interface has complex behaviours linked to hidden state variables. The need of several iterations is therefore a symptom that the user interface behaviour may need to be revised in any case (even if predictability succeeds at the end), as the mental model that users would need to develop is likely to be too complex. We recall that the predictability analysis performed here is performed to help discover potential problems with interaction design in safety-critical parts of a user interface, such as the interactive number entry system of an infusion pump, rather that the whole user interface behaviour.

It is also worth noting that the procedure that refines the initial prediction model is essentially discovering invariants of the model (i.e., relations that always holds on all reachable states) between the values of fields in the observable state and the values of fields in the hidden state. All these invariants could have been found through static analysis of the specification of the device model rather than through iterative refinement. We have chosen the iterative procedure as it keeps the approach simple. Counterexamples generated by the model-checker are used as the basis for identifying the invariant — we will demonstrate how this can be done during the analysis of the Alaris pump in section 6.3.

6.2 Verification of the B-Braun number entry system

We describe here the analysis we carried out on the BBraun number entry model. The first step is to specifying the prediction model as a reduced version of the device model. To this aim, a new type is defined that models information resources in the observable device state (`bbraun_observable_state`, see Listing 15). For the BBraun, the observable state contains two fields: `display` of type `bbraun_real` and `cursor` of type `bbraun_cursor`.

Listing 15: Observable state of the BBraun

```
bbraun_observable_state: TYPE =
 [# display: bbraun_real,
    cursor : bbraun_cursor #];
```

The action-effect of button clicks are then defined as transition functions over observable states. The specification of the prediction model is obtained by discarding all predicates over resources other than the observable state (which is given by the display value and the cursor position for the BBraun), and all effects on resources other than those of the observable state. For the BBraun, predicates and effects on device memory are therefore removed.

The state machine of the prediction model is shown in Listing 16. Similarly to the device model, the overall behaviour of the prediction model is specified in SAL with a *module*, which defines the initial state of the prediction model (in the `INITIALIZATION` section), and the transitions, given as guarded commands. The initial state of the prediction module initially matches that of the device model. The input variable `event` represents button clicks, the input variable `st` is used to determine the current values of the display and cursor, and the output variable `predicted` represents the expected display value and cursor position as a result of button clicks according to the prediction model.

Listing 16: Prediction model for the BBraun

```
bbraun_prediction : MODULE =
BEGIN
INPUT event: Event, st: bbraun_state
OUTPUT predicted: bbraun_observable_state
INITIALIZATION
 predicted = (# display := display(st),
                cursor  := cursor(st) #);
TRANSITION
[  event = up -->
   predicted'=prediction_up(
               (# display := display(st),
                  cursor  := cursor(st) #))
[] event = down -->
   predicted'=prediction_down(
               (# display := display(st),
                  cursor  := cursor(st) #))
[] event = left -->
   predicted'=prediction_left(
               (# display := display(st),
                  cursor  := cursor(st) #))
[] event = right -->
```

```
22    predicted'=prediction_right(
23                  (# display := display(st),
24                     cursor  := cursor(st) #))
25 ] END
```

The specification of transition functions modelling up, down, left, and right button clicks is thus developed from the corresponding functions of the device model. As explained in section 6.1, conditions on state variables that are not part of the observable state are removed. The specification of `prediction_up` is shown in Listing 17. The specification of the other transition functions for the prediction model are omitted as they are obtained in a similar way.

Listing 17: Prediction model, *up* button clicks

```
1 prediction_up(st: bbraun_observable_state):
2   bbraun_observable_state =
3 IF display(st) = max_display THEN st
4 ELSE
5  LET val: bbraun_real = display(st),
6      cur: bbraun_cursor = cursor(st)
7      new_val: real =
8       IF  val + pow10(cur) < 0.1 THEN 0.1
9       ELSIF val + pow10(cur) >= 0.1
10       AND val + pow10(cur) < 100
11      THEN val + pow10(cur)
12      ELSIF val + pow10(cur) >= 100
13       AND val + pow10(cur) < 1000
14      THEN floor((val + pow10(cur))*10)/10
15      ELSE floor(val + pow10(cur)) ENDIF
16  IN
17 IF new_val > max_display
18 THEN st WITH .display := max_display
19 ELSE st WITH .display := new_val ENDIF;
```

From the specification above it can be noted that the calculation of `predicted` is based solely on the display value and the cursor position. When the calculation performed by the device model only uses information from the observable state, the logic behind the calculation in the prediction model is identical to that of the device model. That is, it is done according to the same rules as specified by the transition functions of the device model. This captures the hypothesis encapsulated in the definition of predictability that the user has a complete and correct understanding of the functionalities of the device but decisions are made only on the basis of the current observable device state.

Now, predictability can be checked by verifying that the device model and the prediction model match in all reachable states with respect to the observable state. The parallel composition of the two models generates all reachable states. In SAL, this can be done by specifying a composed module, `bbraun_system`, given by the synchronous composition of `bbraun_device` and `bbraun_prediction`. The specification of the composed system is then given in Listing 18 (the parallel composition is the symbol `||`).

Listing 18: BBraun system model

```
1 bbraun_system:
2 MODULE = bbraun_device || bbraun_prediction;
```

The predictability property can be then specified as a Linear Temporal Logic (LTL) property over all states reached by the composed system module. The verification checks that all reachable states have the same value for the observable components of the two models (*predicted* of the prediction model and the pair (*display(st)*, *cursor(st)*) of the device model).

Listing 19: Predictability for the BBraun

```
1 bbraun_predictable:
2 CLAIM bbraun_system |-
3  G ( predicted =
4           (# display := display(st),
5              cursor  := cursor(st) #));
```

The verification of this property with SAL produces a counterexample when *display(st) = 10*. Namely, starting from the initial state when the display is 0, the cursor is at position 0, and the memory is empty, the sequence of button clicks *up, left, down, up* produces the following state changes:

- *on the device model*: starting from 0, the up button click increments the number on the display by 1; then, by clicking the left button, the cursor highlights the tens digit (the display is still 1); by clicking the down button, the device overshoots the minimum value ($1 - 10 < 0$), thus the display shows a default minimum value (0.1) and stores 10 (i.e., *pow10(cursor)*) in memory; finally, by clicking the up button, the button click recalls and clears the memory, thus the display shows 10.
- *on the prediction model*: starting from 0, by clicking the up button, the device displays goes to 1; the left button click moves the cursor to the tens digit. Then, the down button click causes an overshoot of the minimum value ($1 - 10$ is a negative number), and the device displays the minimum default value (0.1). At this point, the cursor is still selecting the tens digit, and the display shows 0.1. If the decision is based on this information only, a *up* button click produces 10.1 ($10 + 0.1$)

6.3 Verification of the Alaris number entry system

We describe here the analysis carried out on the Alaris number entry model. Refinement is needed for the verification of the Alaris pump.

As for the verification of the other device, the first step is again to specifying the prediction model. The observable state (shown in Listing 20) is defined by a new type, `alaris_observable_state`, which includes only the value shown on the display in this case.

Listing 20: Observable state of the Alaris

```
1 alaris_observable_state: TYPE = alaris_real;
```

The action-effect of key presses are specified as transition functions over observable states. All predicates and all effects over step multipliers and timers are discarded in this case, as they are not externalised on the device user interface. The transition system of the Alaris prediction model (shown in Listing 21) is a state machine with an initialisation that copies the corresponding initial values of the device model, and a set of transition functions modelling the prediction for the press and release actions.

Listing 21: Prediction model for the Alaris

```
1 alaris_prediction : MODULE =
2 BEGIN
3 INPUT
4    event: Event; st: alaris_state;
5 OUTPUT
6    predicted: alaris_observable_state;
7 INITIALIZATION
8    predicted = display(st);
9 TRANSITION
10 [  event = press_UP -->
11    predicted' =
12       prediction_press_UP(display(st))
13 [] event = press_DOWN -->
14    predicted' =
15       prediction_press_DOWN(display(st))
16 [] event = press_up -->
17    predicted' =
18       prediction_press_up(display(st))
19 [] event = press_down -->
20    predicted' =
21       prediction_press_down(display(st))
22 [] event = release_key -->
23    predicted' =
24       prediction_release_key(display(st))
25 ]
26 END
```

The specification of the transition functions is thus generated from the corresponding function of the device model. The specification of `prediction_press_UP` is shown in Listing 22. It can be noted that the step multiplier is considered in the computation but its value is derived solely from the current observable display value. The specification of this transition function will be used throughout the rest of this section in the verification example. The specification of the other transition functions is omitted.

Listing 22: Prediction model, *fast up* button presses

```
1 prediction_press_UP
2 (val: alaris_observable_state)
3    : alaris_observable_state =
4 LET s: alaris_multiplier =
5   IF (display(st) < 100
6       AND fractional(display(st),10) = 0)
7       OR (display(st) > 100
8       AND fractional(display(st),100) = 0)
9   THEN x10 ELSE x1 ENDIF
10 IN
11  IF display(st) < 100
12  THEN trim(floor(display(st))+s)
13  ELSIF display(st) >= 100
14          AND display(st) < 1000
15  THEN trim((floor(display(st)/10)+s)
16                 * 10)
17  ELSE trim((floor(display(st)/100)+s)
18                 * 100) ENDIF
```

The specification of the whole system is the module composition (as shown in Listing 23, lines 1–3, where `alaris_constraint` is the observer module illustrated in section 5.2 that enforces legal keypress sequences). The predictability condition is formulated as a LTL formula (as shown in Listing 23, lines 4–6) that checks the behaviour of the Alaris number entry system for all keypress sequences.

Listing 23: Alaris, system model & predictability

```
1 alaris_system: MODULE =
2  alaris_constraint || alaris_device
3                    || alaris_prediction;
4 alaris_predictable: CLAIM
5 alaris_system
6   |- G (ok => (display(st)=predicted));
```

When verifying the predictability property with the above specification, the model checker immediately finds a counterexample after the initial state: the prediction model expects the display to show 10 (the step multiplier is `x10` according to the conditions in the press up function in the prediction model), while the display in the device model is actually 1 (the step multiplier is `x1` according to the press up function in the device model). This counterexample is an artefact of the model, and it is discharged by adding a condition when the display is zero — when the display is zero, the step multiplier is `x1` (see Listing 24, line 5).

Listing 24: Prediction model, first refinement

```
1 prediction_press_UP
2 (val: alaris_observable_state)
3    : alaris_observable_state =
4 LET s: alaris_multiplier =
5  IF display(st) = 0 THEN x1 % 1st refinement
6  ELSIF (display(st) < 100
7     AND fractional(display(st),10) = 0)
```

```
8      OR (display(st) > 100
9      AND fractional(display(st),100) = 0)
10    THEN x10 ELSE x1 ENDIF
11   IN ...
```

When performing a subsequent new verification with the refined prediction model, SAL identifies another counter example at a different observable state. The new counter example generated shows that, when the device is in a state where *display(st) = 10*, *multiplier(st) = x1*, and *timer(st) = 5*, if the fast up button is clicked (i.e., the sequence *press_UP, release_key* is performed in the model), then the device display becomes 11, while the prediction model expects 20. This counterexample is discharged by adding a relation between the display value and the multiplier: when the display is 10 then the step multiplier is *x1* (see Listing 25, line 5).

Listing 25: Prediction model, second refinement

```
1 prediction_press_UP
2 (val: alaris_observable_state)
3    : alaris_observable_state =
4 LET s: alaris_multiplier =
5  IF display(st) = 10   % 2nd refinement
6     OR display(st) = 0 THEN x1
7  ELSIF (display(st) < 100
8     AND fractional(display(st),10) = 0)
9     OR %...
```

After the second refinement, a new verification with SAL identifies another counterexample at the same observable state. This time, the prediction model expects 11 while the device model provides 20. This happens because of a sequence where the fast up button is pressed & held long enough to make the internal timer of the device 0. Therefore, this counterexample cannot be discharged if the decision is to be taken solely on the value of the current display — when the display is 10, the multiplier can be either `x1` or `x10` depending on the value of an hidden state variable.

7 Generating recommendations

Given that the modelled interactive number entry systems are not always predictable, two interesting question are worth answering:

(i) What design changes could be applied to make the design predictable? An answer to this question may provide useful insights to device manufacturers about the effect of different features in interaction design.

(ii) Under what conditions do they become predictable? An answer to this question would provide insights for user training, in that we can check whether a reasonably simple strategy exists (other than resetting the device and restart the programming task from the beginning) that allows one to circumvent the predictability issues evident in the analysis of the two pumps.

Based on these questions, recommendations can be given in the form of **verified design solutions** and **verified user strategies**.

7.1 Verified design solution for Alaris

The predictability issues of the Alaris pump analysed here are essentially linked to the step multiplier. The step multiplier is automatically selected by the device during the interactions according to rules linked to the interaction history and the current display value. Information about the selected step multiplier cannot be derived from the current persistent output of the device. Three possible design solutions follow. For each proposed solution, we checked with SAL that the predictability property holds.

Avoid using step multipliers. The simplest way to fix the interaction design of the pump and make it predictable in any situation would be to avoid the use of step multipliers. This solution would be consistent with the classic user interface design principle of making critical actions (in this case, programming the pump with a high rate) more difficult to perform [26,25]: the higher the number to be entered, the longer the interaction. The solution might be judged inconvenient by designers or operators, as it would increase the time to enter certain values in the pump, and thus make the overall programming task less efficient in certain situations. Though, an experiment run by Oladimeji et al. to compare error detection of two number entry interface styles pointed out that users frequently overshoot and undershoot their target numbers when using the specific Alaris interface considered in this work [27]. This hunt for the target number could be attributed to the use of step multipliers in the Alaris system.

Enhance the feedback. The feedback about the result of actions could be enhanced by providing information about the effect of key-presses on the step multiplier. The feasibility of this solution should be carefully evaluated, because the capabilities of the physical display might not allow a proper visualisation. There is experimental evidence that humans have numerous bottlenecks in performing simultaneous processing of several pieces of information, especially if the information is gathered from a single channel, e.g., only from the auditive or from the visual channel. A typical failure due to these bottlenecks is known as *attentional tunnelling*, i.e., the user "locks in" on specific information

and inadvertently drops other (possibly relevant) information [16]. Therefore, in such cases, rather than enhancing the feedback, a better solution is to reduce the functionalities of the device.

It is worth noting that a variant of the Alaris pump with an enhanced feedback has been actually implemented in a recent release of the firmware of this pump. Namely, in the new firmware, indicators in the form of double and single underlines are shown on the display to highlight which digit would change when any of the four chevron buttons are pressed. The fast up and fast down (i.e., the double chevron keys) will change the digit highlighted by the double underline, while the slow up and slow down (i.e., the single chevron keys) will change the digit highlighted by the single underline.

Allow users take active control of step multipliers. Changing the control widget used for number entry into one which would allow the user to explicitly control the step multipliers could also render the system predictable. Examples of such widgets are those used in traditional document scrolling tasks. These include but are not limited to pressure sensitive buttons, isometric joysticks or rotary encoders [18]. Using these types of widgets, the changes in the step multiplier could be mapped to active user actions of applying more pressure on a button or quickly turning a knob rather than the passive changes where changes in the step multiplier depends on the duration of the interaction.

7.2 Verified design solutions for BBraun

The predictability issues of the BBraun pump analysed here derive from the use of memory. In particular, the problem lies in the fact that the user is not able to tell whether the memory has been cleared or not from the persistent state of the device. Three possible design solutions follow. For each proposed solution, we checked with SAL that the predictability property holds.

Don't use memory. The use of memory usually makes devices unpredictable [13]. Without memory, the functionality of the device at certain boundary cases may need to be revised. For instance, when overshooting the maximum value, the *undo* feature available when using memory would not be available. This solution is thus linked to the following one, which deals with overshooting the maximum and minimum values that can be entered in the device.

Avoid overshooting. Overshooting can be avoided by ignoring or blocking the button presses that cause overshooting. The specific solution for ignoring or blocking button presses needs to be carefully evaluated as there is a trade-off between *permissiveness* [35] and the number and types of audible/visual cues needed for capturing the user attention when overshooting. This solution implies that setting the maximum and minimum values on the interfaces would require more effort from the user in that it would need to be done by using a precise sequence that take the device to that value. However the possible consequence would be that they are more aware of setting values at these boundaries. This solution therefore supports the classic user interface design principle of making critical actions more difficult to perform so as to ensure when done they are done deliberately [26, 25].

Increase the visibility of the system state. This ensures that the display of the device shows sufficient information to understand the current state of the device. For instance, an indication that the memory of the device is not empty would help users identify situations where the device seamlessly changes the function associated to the buttons (e.g., when the down button becomes a recall memory button). As discussed for the other pump, this solution should be carefully evaluated in order to avoid *attentional tunnelling* [16].

7.3 Verified user strategies

When the design of a device cannot be changed (e.g., because the manufacturer has consciously chosen to lose predictability in favour of other features, or because the hospital ward needs to use the pumps while waiting the problem to be fixed), an important alternative is to offer *verified users strategies* to mitigate against the consequences of that loss. Strategies can be obtained by reasoning about the features driving the interactive behaviour of the pumps. In the following, we discuss simple verified strategy that can be used to make the analysed devices predictable.

It is worth remarking that the interaction strategies we define here are in fact workarounds to avoid certain unwanted behaviours of the device. As such, if the definition of predictability considered here is a desired feature of the device, the manufacturer should modify the interaction design to enable predictability without workarounds — systems that rely on error-free performance are doomed to failure [21].

BBraun user strategy. From the developed specification we can notice that, if the memory is clear, then the interactions with the pump become predictable. We verified this claim in SAL with the LTL property shown in Listing 26.

Listing 26: Verification of the BBraun user strategy

```
bbraun_predictable_weak: CLAIM
 bbraun_system |-
  G ( NOT valid?(memory(st))
       => display(st) = predicted );
```

The claim is successfully verified. By studying the specification, we can notice that a simple strategy can be defined to clear the content of the memory and hence guarantee predictable interactions in any situation with this model of the BBraun pump — clear the memory by changing the cursor position with the left and right arrow keys. It is interesting to note here that another simple strategy, one where the user remembers the previous state, fails with this design.

Alaris user strategy. From the developed specification, we can notice that the step multiplier can change only during press & hold interactions with the fast up and fast down keys. Therefore, if we limit the possible interactions by avoiding press & hold actions on the fast up and fast down buttons, the interaction becomes always predictable. We can verify this claim in SAL by defining a new module (`alaris_strategy`) that imposes such constraints on the interactions:

Listing 27: User strategy as observer module

```
alaris_strategy : MODULE =
BEGIN
INPUT event: Event
OUTPUT prev_event: Event
OUTPUT ok: boolean
INITIALIZATION
     ok = true;
     prev_event = release;
TRANSITION
    [ ok AND (prev_event = press_UP
            OR prev_event = press_DOWN) -->
           ok' = (event = release);
           prev_event' = event
    [] ELSE --> ] END
```

The predictability claim can then be verified against a system given by the parallel composition of the device model, the prediction model, and `alaris_strategy` (as shown in Listing 28).

Listing 28: Verification of the Alaris user strategy

```
alaris_system_new: MODULE
= alaris_constraint || alaris_device
                    || alaris_prediction
                    || alaris_strategy;
alaris_predictable_new: CLAIM
 alaris_system_new |-
  G (ok => (display(st) = predicted));
```

The claim is successfully verified — if only button clicks are used to interact with the device, then the device is predictable. We can notice also that, since the slow up and slow down chevron keys do not change the multiplier in any situation, an alternative strategy is also to use only such two keys.

8 Discussion and conclusions

In this paper, we have shown that formal methods can be used for studying predictability of detailed specifications of commercial interactive number entry systems: on the one hand, the mere exercise of building a formal specification of the interface gave us useful insights on possible design issues, even before performing the analysis with the model checking tool; on the other hand, the formal tool enabled us to explore all possible behaviours, thus allowing us to explore the validity of the proposed design modifications.

The increasing demand for advanced functionality forces single devices to be used for a wide variety of tasks, but under the fixed physical constraints of the devices. It is understandable that over time users (or organisations) would require more sophisticated interactive systems that assist their varied tasks. However, the required generality introduces inconsistent behaviour to the user interface, which is sometimes an obstacle to the user's mental model development. In addition, even if users have a complete and sound mental model of the system, the increasing number of hidden states that are inevitable with general-purpose systems makes it harder for them to predict the consequences of their actions. In fact, when devices are closely examined, there are many boundary cases where interactive functionality seems awkward; this compromises the predictability of the devices, and hence may lead to unnecessary hazards in use.

Several frameworks have been presented in the literature for human-machine interaction and human error in the context of interface design. However, none of them is currently widely accepted as a reference. Most of these frameworks are based on psychological assumptions. Although they help identify issues, of the human factors kind, only few of them provide means to identifying engineering solutions that can be used for improving device designs. Our work on predictability is a first step towards addressing this gap. As we have shown, the predictability analysis links to several high-level design principles illustrated in the HF75:2009 standard for human factors. The analysis can be mechanically performed with a verification tool. The analysis results help designers understand how to fix identified design

problems and help users overcome predictability issues when the design cannot be modified.

Failure of the predictability property holding is not necessarily in itself a criticism of any design. There are many trade offs in design, and loss of predictability because of the presence of other features may be less important in practice than the value of some other features to the user. Indeed there may be better definitions of predictability than we consider in this paper. The point of the paper, however, is to show that plausible interaction design properties can be formalised, that real devices can be very effectively analysed for their compliance to such properties, and that problems can be precisely identified **and hence fixed** where this is deemed appropriate given the trade-offs involved. Our definition of predictability is taken from the HCI research literature, so it is a meaningful property, and it is clear that loss of predictability as we define it will increase specified hazards in operation. Whether those hazards are somehow compensated for by other design features is an important question that lies beyond the scope of this paper.

In an idealised world many definitions of predictability would be compared, and the formal analyses based on them compared with empirical experiments. That would allow one to say definitively that certain conceptions of predictability are correlated with user performance and hence that specific design features can be identified to improve performance. However with human error rates being so low and devices so complex, it is unlikely that valid experiments could be performed — there are no easy empirical experiments that can explore the non-occurrence of problems! As an alternative, here we argue that what is important is that a formal analysis of predictability allows one to **reason** about performance. If certain features are identified as problematic or potentially problematic, then a developer can take steps to better manage these features. Relative to the definition of predictability one then avoids identified design problems.

In future, we plan to use empirical evaluation and hence develop an evidence-based balance between predictability and functionality. In particular, we aim to validate to which extent predictable number entry systems makes a difference when taken into account in the design.

Acknowledgements Funded as part of the CHI+MED: Multidisciplinary Computer-Human Interaction research for the design and safe use of interactive medical devices project, EPSRC Grant Number EP/G059063/1, and Extreme Reasoning, Grant Number EP/F02309X/1.

References

1. List of errorprone abbreviations, symbols and dose designations (2006). URL http://www.ismp.org/tools/abbreviations/
2. Abowd, G.D., Coutaz, J., Nigay, L.: Structuring the space of interactive system properties. In: Proceedings of the IFIP TC2/WG2.7 Working Conference on Engineering for Human-Computer Interaction, pp. 113–129. North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands (1992). URL http://portal.acm.org/citation.cfm?id=647103.717569
3. Arney, D., Jetley, R., Jones, P., Lee, I., Sokolsky, O.: Formal methods based development of a pca infusion pump reference model: Generic infusion pump (gip) project. Joint Workshop on High Confidence Medical Devices, Software, and Systems and Medical Device Plug-and-Play Interoperability **0**, 23–33 (2007). DOI http://doi.ieeecomputersociety.org/10.1109/HCMDSS-MDPnP.2007.36
4. B-Braun Melsungen AG: Infusomat space and accessory: Instruction for use
5. Back, J., Brumby, D.P., Cox, A.L.: Locked-out: investigating the effectiveness of system lockouts to reduce errors in routine tasks. In: Proceedings of the 28th of the international conference extended abstracts on Human factors in computing systems, CHI EA '10, pp. 3775–3780. ACM, New York, NY, USA (2010). DOI 10.1145/1753846.1754054. URL http://doi.acm.org/10.1145/1753846.1754054
6. Bass, E.J., Feigh, K.M., Gunter, E.L., Rushby, J.M.: Formal modeling and analysis for interactive hybrid systems. ECEASST **45** (2011)
7. Bolton, M.L., Bass, E.J.: Formally verifying human—automation interaction as part of a system model: limitations and tradeoffs. Innovations in Systems and Software Engineering **6**(3), 219–231 (2010). DOI 10.1007/s11334-010-0129-9. URL http://dx.doi.org/10.1007/s11334-010-0129-9
8. Campos, J.C., Harrison, M.D.: Interaction engineering using the ivy tool. In: Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems, EICS '09, pp. 35–44. ACM, New York, NY, USA (2009). DOI 10.1145/1570433.1570442. URL http://doi.acm.org/10.1145/1570433.1570442
9. Campos, J.C., Harrison, M.D.: Modelling and analysing the interactive behaviour of an infusion pump. ECEASST **45** (2011)
10. Cauchi, A., Gimblett, A., Thimbleby, A., Curzon, P., Masci, P.: Safer "5-key" number entry user interfaces using Differential Formal Analysis. 26th Annual Conference on Human-Computer Interaction, BCS-HCI (2012)
11. Degani, A., Heymann, M.: Formal verification of human-automation interaction. Human Factors **44**(1), 28–43 (2002)
12. Department fo Health and Human Services, US Food and Drug Administration. Total Product Life Cycle: Infusion Pump - Premarket Notification [510(k)] Submissions - Draft Guidance, April 2010.
13. Dix, A.J.: Formal methods for interactive systems. Computers and people series. Academic Press (1991). URL http://www.hiraeth.com/books/formal/
14. Dix, A.J., Runciman, C: Abstract models of interactive systems. People and Computers: Designing the Interface. Cambridge University Press, 13–22 (1985)
15. Harrison, M.D. and Thimbleby, H.: Abstract models of interactive systems. Proceedings British Computer

Society Conference on Human Computer Interaction (HCI'85), Cambridge University Press, 161–171 (1985)
16. Endsley, M.R., Bolte, B., Jones, D.G.: Designing for Situation Awareness: An Approach to User-Centered Design. Taylor and Francis (2003)
17. Health, C.: Alaris GP volumetric pump: Directions for use (2006)
18. Hinckley, K., Cutrell, E., Bathiche, S., Muss, T.: Quantitative analysis of scrolling techniques. In: Proceedings of the SIGCHI conference on Human factors in computing systems: Changing our world, changing ourselves, CHI '02, pp. 65–72. ACM, New York, NY, USA (2002). DOI 10.1145/503376.503389. URL http://doi.acm.org/10.1145/503376.503389
19. Javaux, D.: Explaining sarter and woods' classical results. In: Second Workshop on Human Error, Safety, and Software Design (1998)
20. Kim, B., Ayoub, A., Sokolsky, O., Lee, I., Jones, P., Zhang, Y., Jetley, R.: Safety-assured development of the gpca infusion pump software. In: Proceedings of the ninth ACM international conference on Embedded software, EMSOFT '11, pp. 155–164. ACM, New York, NY, USA (2011). DOI 10.1145/2038642.2038667. URL http://doi.acm.org/10.1145/2038642.2038667
21. Leape, L.: Error in medicine. Journal of the American Medical Association **272**(23), 1851–1857 (1994)
22. Masci, P., Rukšėnas, R., Oladimeji, P., Cauchi, A., Gimblett, A., Li, Y., Curzon, P., Thimbleby, H.: On formalising interactive number entry on infusion pumps. ECEASST **45** (2011)
23. Medicines and Healthcare products Regulatory Agency (MHRA): Device bulletin, infusion systems, db2003(02) v2.0 (2010). URL http://www.mhra.gov.uk/Publications/Safetyguidance/DeviceBulletins/CON007321
24. de Moura, L., Owre, S., Ruess, H., Rushby, J., Shankar, N., Sorea, M., Tiwari, A.: SAL 2. In: R. Alur, D.A. Peled (eds.) Computer Aided Verification: CAV 2004, *Lecture Notes in Computer Science*, vol. 3114, pp. 496–500. Springer-Verlag (2004)
25. Norman, D.A.: Design rules based on analyses of human error. Communications of the ACM **26**(4), 254–258 (1983). DOI http://doi.acm.org/10.1145/2163.358092
26. Norman, D.A.: The Design of Everyday Things, reprint paperback edn. Basic Books, New York (2002)
27. Oladimeji, P., Thimbleby, H., Cox, A.: Number entry interfaces and their effects on error detection. In: Proceedings of the 13th IFIP TC 13 international conference on Human-computer interaction - Volume Part IV, INTERACT'11, pp. 178–185. Springer-Verlag, Berlin, Heidelberg (2011). URL http://dl.acm.org/citation.cfm?id=2042283.2042302.
28. Perrow, C.: Normal accidents : living with high-risk technologies. Basic Books, New York (1984)
29. Rasmussen, J.: The role of error in organizing behaviour. Ergonomics **33**, 1185–1199 (1990)
30. Reason, J.: Human Error, 1 edn. Cambridge University Press (1990)
31. Rushby, J.: Using model checking to help discover mode confusions and other automation surprises. Reliability Engineering and System Safety **75**(2), 167–177 (2002). Available at http://www.csl.sri.com/users/rushby/abstracts/ress02
32. Rushby, J.M.: Modeling the human in human factors. In: Proceedings of the 20th International Conference on Computer Safety, Reliability and Security, SAFECOMP '01, pp. 86–91. Springer-Verlag, London, UK, UK (2001). URL http://dl.acm.org/citation.cfm?id=647399.724851
33. Ryan, M., Fiadeiro, J.L., Maibaum, T.S.E.: Sharing actions and attributes in modal action logic. In: TACS, pp. 569–593 (1991)
34. Thimbleby, H.: Generative user-engineering principles for user interface design. In: B. Shackel (ed.) Proceedings First IFIP Conference on Human Computer Interaction, INTERACT'84, vol. 2, pp. 102–107 (1984)
35. Thimbleby, H.: Permissive user interfaces. International Journal of Human-Computer Studies **54**(3), 333–350 (2001). DOI 10.1006/ijhc.2000.0442
36. Thimbleby, H.: Interaction Walkthrough: Evaluation of safety critical interactive systems. In: G. Doherty, A. Blandford (eds.) DSVIS 2006, The XIII International Workshop on Design, Specification and Verification of Interactive Systems, *Lecture Notes in Computer Science*, vol. 4323, pp. 52–66. Springer Verlag (2007)
37. Thimbleby, H., Harrison, M.D.: Formalising guidelines for the design of interactive systems. In: S. Cook, P. Johnson (eds.) Proceedings British Computer Society Conference on Human Computer Interaction, HCI'85, pp. 161–171. Cambridge University Press (1985)
38. Thimbleby, H.W., Gimblett, A.: Dependable keyed data entry for interactive systems. ECEASST **45** (2011)
39. Trafton, G.J., Monk, C.A.: Task interruptions. Reviews of Human Factors and Ergonomics **3**, 111–126(16) (2007). DOI doi:10.1518/155723408X299852. URL http://www.ingentaconnect.com/content/hfes/rhfe/2007/00000003/00000001/art00005
40. Tucker, A.L., Spear, S.J.: Operational failures and interruptions in hospital nursing. Health Services Research **41**(3p1), 643–662 (2006). DOI 10.1111/j.1475-6773.2006.00502.x. URL http://dx.doi.org/10.1111/j.1475-6773.2006.00502.x
41. Vincent, C.: Patient Safety, second edn. John Wiley & Sons (2011)