Using PVSio-web to demonstrate software issues in medical user interfaces

Paolo Masci[1,⋆], Patrick Oladimeji[2], Paul Curzon[1], and Harold Thimbleby[2]

[1]Queen Mary University of London, United Kingdom
{p.m.masci,p.curzon}@qmul.ac.uk
[2]Swansea University, United Kingdom
{p.oladimeji,h.thimbleby}@swansea.ac.uk

Abstract. We have used formal methods technology to investigate software and user interface design issues that may induce use error in medical devices. Our approach is based on mathematical models that capture safety concerns related to the use of a device. We analysed nine commercial medical devices from six manufacturers with our approach, and precisely identified 30 design issues. All identified issues can induce use errors that could lead to adverse clinical consequences, such as numbers being incorrectly entered. An issue with formal approaches is in making results accessible to developers, human factors experts and clinicians. In this paper, we use our tool PVSio-web to demonstrate the identified issues: PVSio-web allows us to generate realistic and interactive user interface prototypes from the same mathematical models used for analysis. Users can explore the behaviour of the prototypes by pressing buttons on realistic user interfaces that reproduce the functionality and visual representation of the real devices. Users can examine the device behaviour resulting from any interaction. Key sequences identified from analysis can be used to explore in detail the identified design issues in an accessible way.

Demo video: "Design issues in medical user interfaces"[§]

1 Introduction

According to the US Food and Drug Administration (FDA) many problems reported in incidents involving medical devices are due to use errors and software defects [2]. For example, in the USA, a recent study conducted by FDA software engineers revealed that software-related recalls had almost doubled in just over five years: 14% in 2005 to nearly 25% in 2011 [16]. While it is usual to attribute software failures to coding errors, the FDA study highlighted that the largest class of problems were actually caused by logic errors in software design.

In the present work, a series of realistic prototypes, based on nine real devices from six different manufacturers, have been developed to demonstrate identified

⋆ Corresponding author.

§ https://www.youtube.com/watch?v=T0QmUe0bwL8

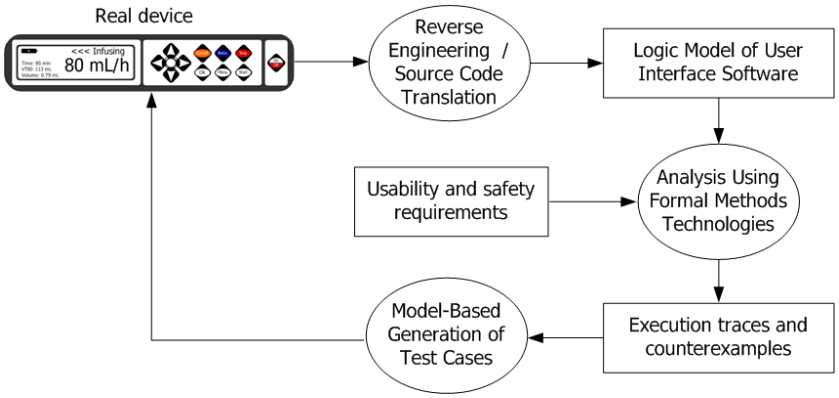

Fig. 1: Our approach for the analysis of user interface software.

relations between *logic errors in software design* and device behaviours that induce use errors (such as accidentally entering a number 10 times higher than intended) that have potential adverse clinical consequences. These logic errors are due either to incomplete or erroneous system requirements and specification, or design features chosen without considering well-known usability heuristics. All demonstrated issues were validated, and can be reproduced on the real devices using the input key sequences presented in our demonstrative video. Some example issues and input key sequences are illustrated in Section 4. The prototypes are run in our tool PVSio-web [12], a graphical environment based on the formal verification system PVS [13]. PVS is a well known industrial-level theorem prover that enables mechanised verification of potentially infinite-state systems. It is based on a typed higher-order logic, and its specification language has many features similar to those of imperative programming languages such as C++.

Formal tools typically produce results in a textual form that is usually inaccessible to engineers, and certainly to human factors specialists or clinicians. In contrast, PVSio-web can be used to examine and realistically animate the results of analysis performed with formal verification tools such as PVS. By using PVSio-web this barrier is eliminated because issues can be explored and demonstrated by interacting with a realistic user interface and watching the device behaviour that results from the interaction. The tool can be used for generating test cases for developed devices, or used at earlier phases of design (e.g., to animate a formally specified functional specification) when the final device is not yet available.

2 Our formal methods approach

The analysis approach involves the following steps (see Figure 1):

- First, a logic model of the device user interface software is developed. It defines how the device supports user actions (e.g., the effect of pressing a button on the user interface), and what feedback (e.g., messages displayed on the screen) is provided to the user in response to user actions or internal device events. These models are obtained from the source code of the device software, or by reverse engineering the device behaviour through systematic interaction with the device.
- Second, the developed model is verified against relevant safety and usability requirements using formal methods technologies — the PVS theorem prover, in this case. Example requirements are: visibility of relevant information about the device state; feedback about what is the current device state and what has been achieved; ability to undo the effect of actions. An illustration of analysed requirements and techniques used to analyse the requirements is in our previous work [3, 6–8]. Within this step, realistic prototypes of the device are automatically generated from the same PVS models using the PVSio-web prototyping environment. The prototypes facilitate model debugging and analysis of conjectures about the device behaviour specified in the model.
- Third, test cases are generated using a semi-automatic approach using graph exploration techniques on the developed model. The aim of these test cases is to validate the model behaviour against the real device, and to check that design issues identified during the analysis apply to the real device and not only to the model. Within this step, prototypes developed with PVSio-web are used to demonstrate the behaviour of the device for the generated input key sequences, and engage with domain and clinical experts to discuss the consequences of identified problematic behaviours.

2.1 Reverse Engineered Models

To perform reverse engineering in a systematic way, we use an iterative approach that involves a direct evaluation of the device behaviour. An initial model is specified that describes the device behaviour according to the documentation provided with the device, and based on execution traces obtained by interacting with the real device. The model is then analysed within PVS to perform basic sanity checks on the model. These include: coverage of conditions; disjointness of conditions; and consistent use of types. To facilitate this analysis, we include pre- and post-conditions for each software feature specified in the PVS model — this is done using predicate subtypes [14], a PVS language mechanism that narrows down the domain of types used in the model. Test cases are generated for each pre- and post-conditions to validate that the behaviour described in the model is consistent with that of the real device. These test cases are given in terms of key presses that can be used with the real device to trigger specific features of the software. Whenever a test case fails on the real device, a discrepancy is identified between the model and the real device. These discrepancies allow us to identify additional conditions that were not taken into account in the model, or actual coding errors in the real device. In the former case, the model is refined,

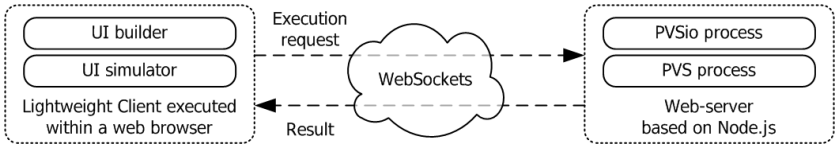

Fig. 2: PVSio-web architecture.

and the process iterated until all tests succeed. In the latter case, the issue is discussed with software engineers, and verified solutions can be identified using the PVS theorem prover.

2.2 Source Code Models

A set of guidelines is used to translate software source code into PVS specifications in a systematic way. The guidelines cover a substantial set of object-oriented language constructs typically used in core software modules. In the following we summarise the guidelines we used for translating C++ source code into PVS specifications.

- Numeric types such as double, float and integer are mimicked in PVS using subtyping [14, 15] — this is needed because the native PVS types are mathematical Reals and Integers.
- Classes and structures are mimicked in PVS using records and datatypes.
- Class variables are emulated in PVS using fields of a user-defined record type, `state`, that contains a field for each class variable;
- Class functions operating on objects are emulated in PVS as higher-order functions. That is, a PVS function is defined that takes the same types and number of arguments of the corresponding class function; this PVS function returns a new function type that takes a single argument of type `state`, which emulates the implicit argument of class functions. This modelling approach allows us to maintain the original signature of class functions.
- Conditional statements have identical counterparts in the PVS specification language; iterative statements are translated using recursive functions; sequential statements and local variables used for computation are mocked using the PVS `LET-IN` construct that binds expressions to local names.

3 PVSio-web

PVSio-web [12] is our graphical tool for rapid prototyping of user interfaces. The tool can be used at any stage of the development life-cycle for the following purposes: (i) rapid generation of realistic prototypes for exploring design alternatives; (ii) debugging or testing of device models; (iii) demonstration of features of device models; (iv) demonstration of analysis results.

Our tool builds on and extends PVSio [11], the textual simulation environment of PVS. Using PVSio-web, users control PVSio simulations by interacting

Fig. 3: Snapshot of the graphical editor bundled with PVSio-web. Shaded areas over the picture represent interactive areas for input and output widgets. The form shown upfront allows designers to configure the selected interactive area.

with buttons and keys of a realistic picture of the device being simulated, and watch the effect of the interactions on the device screen, as in the real device. As illustrated in Figure 2, the architecture of PVSio-web has two main components: a front-end client, used to interact with the tool; and a back-end server hosting PVS and PVSio. The functionalities of the PVSio-web front-end are now further illustrated.

3.1 Graphical editor

A snapshot of the graphical editor is in Figure 3. Using the graphical editor, designers can select a picture of the device being modelled in PVS, and define interactive areas over the picture for identifying input widgets (e.g., buttons, keys) and output widgets (e.g., displays, LEDs). For input widgets, designers can specify which user gestures need to be captured (e.g., clicks), and which PVSio commands need to be invoked for each gesture (e.g., in the picture in Figure 3, function `click_up` of the PVS model is executed when the user clicks on the interactive area defined over button [∧]). For output widgets, designers can associate the widget with state variables of the PVS model (e.g., in Figure 3 a state variable `display` representing information about the infusion rate is

Fig. 4: Example interaction with the PVSio-web simulator: the user clicks on virtual device buttons, and watches feedback on the virtual device display.

associated to a display area). The PVS model can also be viewed and edited within the graphical editor, using the *PVS editor* bundled with PVSio-web.

3.2 Simulator

The PVSio-web graphical simulation environment captures user gestures on interactive areas, and renders the current value of state variables during the simulation. Gestures are automatically translated into PVSio commands for invoking the evaluation and execution of functions in the PVS model. These commands are generated on the basis of the current value of state variables in the PVS model, and the association between user gestures and PVSio commands defined using the PVSio-web graphical editor.

A snapshot of the graphical simulation environment during the execution of a simulation is in Figure 4. It shows an example interaction with the prototype: the user clicks on the virtual device button [^], and the effect of this action is watched on the virtual device display.

4 Demonstration of software design issues

The developed prototypes demonstrate over 30 software design issues inducing use errors that have potential adverse clinical consequences. The demonstrated device behaviours can be reproduced on commercial medical devices in use in hospitals across the US and UK. Several issues are common in devices from different manufacturers, and across different device types. Here, representative examples of identified issues are presented. The full list of identified issues and

an explanation of the consequences of the issues is in our demonstrative video. A detailed discussion of tools and techniques used to identify these software issues is in [1,3,4,7–9].

- **Decimal point key presses ignored without any warning.** The device registers a key sequence such as [1][0][0][•][1] as 1,001 (instead of 100.1) without warning. This may cause missing decimal point errors which results in the transcribed number being ten times larger than that intended.
- **Ill-formed values accepted and displayed.** The device shows fractional numbers without a leading zero (e.g., .9 instead of 0.9), or integer numbers with a leading zero (e.g., 09 instead of 9). These cases violate the recommendations issued by the Institute for Safe Medication Practices (ISMP) and may cause numbers to be misread and misinterpreted [5].
- **Entered values ignored without any warning.** If the user fails to confirm the entered value or pauses data entry for a period of time, the device discards the entered value without any warning. This may cause misconfiguration of device parameters.
- **Unintended values rollover.** When the maximum value is overshot with a key press, the value rolls over to the minimum value and vice versa. This may cause misconfiguration of device parameters. Recently, a safety notice [10] involved this design issue. According to the safety notice, there were reported incidents of users accidentally misprogrammed an insulin pump to deliver the maximum bolus amount because of this design issue. One of these incidents resulted in severe hypoglycemia.

5 Conclusions

This paper presents the use of formal methods technologies for modelling, simulating and testing safety critical interactive interfaces such as those of commercial infusion devices in use in hospitals. Over 30 software design issues are demonstrated that may create traps for the users that can lead to use error during interactions with the device. Occurrences of these errors can have consequences on patient safety — therefore, these issues should be identified and dealt with.

The presented results were obtained from retrospective analysis of existing device designs. Our analysis tools and techniques, however, can be used at earlier stages of the development process when the software for the real device is not yet implemented. While traditionally there has been poor communication between formal methods experts and human factors experts, our tool and analysis methods take a step towards bridging this gap, as they provide an accessible approach for reasoning about the safety and usability of a product using rapid prototyping techniques and realistic simulations.

Acknowledgements. Paul Jones and Yi Zhang (FDA), Julian Goldman and Dave Arney (Massachusetts General Hospital MD PnP Lab, mdpnp.org), Marc Bloom and staff members of the Washington Adventist Hospital, and Paul Lee (Morriston Hospital, Swansea) helped us to validate our findings. This work is supported by EPSRC as part of CHI+MED (Computer-Human Interaction for Medical Devices [EP/G059063/1]).

References

1. A. Cauchi, A. Gimblett, H.W. Thimbleby, P. Curzon, and P. Masci. Safer 5-key number entry user interfaces using differential formal analysis. In *26th Annual BCS Interaction Specialist Group Conference on People and Computers (BCS-HCI)*, pages 29–38. British Computer Society, 2012.
2. Center for Devices and Radiological Health, US Food and Drug Administration. *White Paper: Infusion Pump Improvement Initiative*, 2010.
3. M.D. Harrison, J.C. Campos, and P. Masci. Reusing models and properties in the analysis of similar interactive devices. *Innovations in Systems and Software Engineering*, pages 1–17, 2013.
4. M.D. Harrison, P. Masci, J.C. Campos, and P. Curzon. Demonstrating that medical devices satisfy user related safety requirements. In *4th International Symposium on Foundations of Healthcare Information Engineering and Systems*, 2014.
5. Institute for Safe Medication Practices (ISMP). List of error-prone abbreviations, symbols and dose designations, 2006.
6. P. Masci, A. Ayoub, P. Curzon, M.D. Harrison, I. Lee, and H.W. Thimbleby. Verification of interactive software for medical devices: Pca infusion pumps and fda regulation as an example. In *EICS2013, 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. ACM Digital Library, 2013.
7. P. Masci, R. Rukšėnas, P. Oladimeji, A. Cauchi, A. Gimblett, Y. Li, P. Curzon, and H.W. Thimbleby. On formalising interactive number entry on infusion pumps. *ECEASST*, 45, 2011.
8. P. Masci, R. Rukšėnas, Oladimeji P., A. Cauchi, A. Gimblett, Y. Li, P. Curzon, and H.W. Thimbleby. The benefits of formalising design guidelines: A case study on the predictability of drug infusion pumps. *Innovations in Systems and Software Engineering*, pages 1–21, 2013.
9. P. Masci, Y. Zhang, P. Jones, P. Curzon, and H. Thimbleby. Formal verification of medical device user interfaces using pvs. In *ETAPS/FASE2014, 17th International Conference on Fundamental Approaches to Software Engineering*, Berlin, Heidelberg, 2014. Springer-Verlag.
10. Medtronic. Device safety information: accidental misprogramming of insulin delivery. http://www.medtronicdiabetes.com, March 2014. Report # 930M12226-011.
11. C. Munoz. Rapid prototyping in PVS. *National Institute of Aerospace, Hampton, VA, USA, Tech. Rep. NIA*, 3, 2003.
12. P. Oladimeji, P. Masci, P. Curzon, and H.W. Thimbleby. PVSio-web: a tool for rapid prototyping device user interfaces in PVS. In *FMIS2013, 5th Intl. Workshop on Formal Methods for Interactive Systems*, 2013. Tool available at http://pvsioweb.org/.
13. S. Owre, S. Rajan, J. Rushby, N. Shankar, and M.K. Srivas. PVS: combining specification, proof checking, and model checking. In *Computer-Aided Verification*, number 1102 in Lecture Notes in Computer Science. Springer-Verlag, 1996.
14. J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in pvs. *IEEE Transactions on Software Engineering*, 24(9):709–720, 1998.
15. N. Shankar and S. Owre. Principles and pragmatics of subtyping in PVS. In *Recent Trends in Algebraic Development Techniques*, pages 37–52. Springer, 2000.
16. Simone, L.K. Software-Related Recalls: An Analysis of Records. *Biomedical Instrumentation & Technology*, 47(6):514522, 2013. doi:10.2345/0899-8205-47.6.514.