Using Testing Techniques to Classify User Interface Designs

Abigail Cauchi
University of Malta
Msida, Malta
abigail.cauchi@um.edu.mt

Gordon Pace
University of Malta
Msida, Malta
gordon.pace@um.edu.mt

ABSTRACT

Number entry systems in medical devices such as infusion pumps are used to input drug doses that will be administered to patients. They are safety critical since if the drug dose is too high or too low, this may cause harm to patients.

Previous work shows that number entry systems with the same hardware layout can have software that is implemented in different ways. This means that devices with the same hardware layout may lead to different results after the same keystrokes are pressed. Previous work also shows that choosing the best software implementation over the worst can reduce the likelihood of human error eight-fold in directional number entry systems.

Determining whether a software implementation abides by the requirements is a time consuming task for regulatory bodies and hospital procurement departments. In this paper we show how software testing techniques can be used to classify various software implementations in order to determine whether the given number entry system satisfies specifications.

ACM Classification Keywords

H.5.2. User Interfaces: Evaluation/methodology

Author Keywords

Medical Devices; User Interfaces; Testing; Program Classification

INTRODUCTION

As computer systems become increasingly pervasive, their regulation, especially in the context of safety critical software becomes increasingly important. For instance, in the domain of medical devices, regulatory bodies such as the US Food and Drug Administration (FDA) publish standards for approval of device use in medical contexts.

Narrowing our focus to number entry systems, such as those used to input drug doses on infusion pumps, one finds that standardising number entry systems to make the ones that look the same (in hardware) behave the same, is important to prevent human errors that can lead to harm. From a regulatory perspective however, this is not an easy task. Existing standards such as ISO 62366 [1] are a requirement for medical device approval by the FDA [12] but this standard does not mention safety critical number entry systems design, and furthermore, does not go into any detail of how number entry systems should be implemented.

The US FDA's 510k process [12] currently only requires submission of documentation about the device and does not require submission of the physical device for inspection. From a human-computer interaction perspective, this makes it difficult to spot some obvious interaction issues that might be unsafe. Checking details manually, such as how the number entry system is implemented can lengthen the approval process significantly leaving manufacturers unable to put their device on the market sooner and introducing new challenges to the market.

Given that there are several ways how number entry systems can be implemented (in software) depending on what choices are made, the challenge in such contexts goes beyond that of correctness, breaching the frontier into human-error resilience. Cauchi *et al.* [2] show that choosing the best over the worst design improves out by 10 error rates by a factor of eight. It is important to select the best implementation over the worst, however enforcing a standardised implementation for safety critical uses would be labour intensive if done manually.

Regulatory and procurement processes would thus frequently want to classify a new machine in order to be able to assess its expected resilience to human-error, in procurement supporting the choice of one machine from another, and in the case of regulations, potentially triggering the need of further procedural safety measures that might need to be enforced. Since different entities might adopt different classes depending on intended use, expecting devices to be documented for all such possible classes by the manufacturer is not realistic.

In this work we address the problem of devices which look physically identical or similar, but react differently to identical user interaction patterns. By providing a means of classifying such systems based on their behaviour, we can enforce standard software interaction design, or at least use such information to select devices implementing interaction patterns which are known to be safer for a particular context.

Testing is the most well established approach used to validate the correctness of a computer system against a specification. Effectively, a test suite can be seen as a means of classifying systems into two partitions — those that satisfy the specification, and those that do not. However, the partitions defined by testing need not be these. Furthermore, the use of multiple properties can be used to provide a more fine grained partitioning of possible systems.

In this paper, we exploit this view of testing and adopt an unorthodox use of testing in order to enable the classification of systems with respect to arbitrary classifications. We apply this approach on directional number entry systems classifying them into categories as identified by previous studies such as [5]. We show how, using standard testing techniques, we can take an executable model of a medical device and categorise it automatically in an empirical manner. This approach can be used to support a more informed choice amongst devices, even when they lack detailed documentation and information indicating whether or not they fall under these categories.

RELATED WORK

Oladimeji et al. [11] carried out a lab study to find out which number entry interfaces (different hardware layouts) are more resilient to human error and the findings from that study showed that the directional number entry systems, that we talk about in this paper, perform best in this regard. In previous work by Cauchi et al. [3], different properties for directional number entry systems were identified and the Stochastic Key Slip Simulation method was introduced in order to find out which combination of properties for directional number entry systems makes the design more resilient to human error. The Stochastic Key Slip Simulation method was run with data that was collected through an empirical study in [5] in order to make it more accurate and suitable for the medical domain. Given that these studies give indication as to what properties directional number entry systems should have, the work we present in this paper provides a method by which we can classify different designs into the different properties in order to find out whether they obey the properties that they should.

The US FDA, the regulatory body for approving medical devices for market in the US, have been exploring the use of model based approaches for evaluating properties about medical device safety [13]. Work by Zhang et al. [13] is focused towards creating a generic model of an infusion pump with the aim of making it available to manufacturers for use when developing infusion pumps. This work is still in the research phase and using these model based approaches is not mandatory for approval of medical devices. The drawback of Zhang et al.'s approach is that this restricts the freedom of manufacturers in what they can create in their pumps, making competition between manufacturers more difficult since all infusion pump code should be generated using the same model. In our approach of using testing to classify user interfaces, models of different pumps can vary but safety critical aspects can be classified in order to ensure that they meet regulations.

Harrison et al. [7] and Masci et al. [10] use formal methods, namely model checking and theorem proving in order to verify interaction design properties about infusion pumps. Both these approaches either require the code of the infusion pump or the deduction of the system models through reverse engineering. Because of Intellectual Property rights, it is difficult to obtain source code of infusion pumps, therefore it is very difficult to automatically model the system and verify that safe interaction design properties hold within it. Since our method employs a black box approach to testing and classification, an execution model of the infusion pumps is sufficient to be able to classify different designs.

Key Press	Zimed AD	BBraun Infusomat Space
	0	0
◄	00	00
▲	10	10
▲	20	20
▲	30	30
▲	40	40
▲	50	50
◄	050	050
▼	950	000.1

Figure 2. A key sequence being input into Zimed AD and BBraun Infusomat Space from the same key sequence. This table shows the change in displays after pressing the key in the "Key Press" column. In the first row there is no key press to show the starting displays of the two devices.

NUMBER ENTRY USING DIRECTIONAL PADS

One use of directional pads is for entering numbers in devices. A directional number entry interface consists of a display that shows the current value and a cursor that highlights the digit that is currently selected. The ▲ and ▼ buttons change the highlighted digit and the ◄ and ► buttons move the cursor between digits. An OK button confirms the input number. A typical directional number entry interface can be seen in Figure 1.

Although the hardware of directional number entry systems is fixed, there are interaction design decisions to be taken when implementing the behaviour of the buttons through software. These decisions are important and choosing a good design over a bad design can reduce the likelihood of human error eight-fold [2].

This work was motivated by an observation of entering the dose of 950 mL into two infusion pumps that are currently on the market, the BBraun Infusomat Space and the Zimed AD. Both infusion pumps use a directional number entry system. From a display showing 0 on both devices (where the underline indicates the digit that is highlighted by the cursor), pressing the plausible key sequence ◄ ▲ ▲ ▲ ▲ ▲ ◄ ▼ on the BBraun Infusomat Space and Zimed AD simultaneously resulted in the Zimed AD display showing 950 at the end of the key sequence and the BBraun Infusomat Space showing 000.1. This behaviour is illustrated in Figure 2 that shows how the displays of both devices change when each of the keys are pressed.

Starting from 0 on both devices, both devices work in the same way until the final key press of ▼. This is where the two number entry systems are implemented to do different things that result in the different values. There is a significant differ-

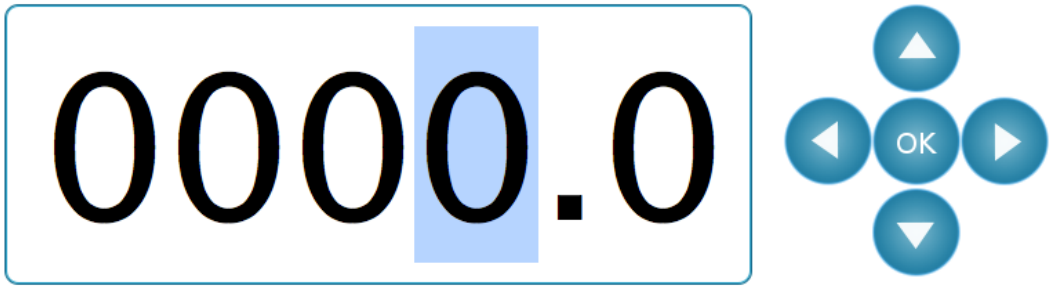

Figure 1. A directional number entry interface with a cursor highlighting a digit. In this system, ▲ and ▼ buttons manipulate the highlighted digit, ◄ ► buttons move the cursor between digits, and the OK button confirms the number.

ence in results between the two devices since these devices are safety critical, this difference could be harmful, or even fatal.

The problem is larger than directional number pad software implementations working differently between different devices by different manufactures. The directional number pad implementation of the device is implemented in the device firmware and the manufacturers upgrade firmware. Analysing the directional pad number entry system of different firmware versions of one of the commercial infusion pumps, resulted that the firmware upgrade changed the directional pad number entry system implementation. This results in devices that look identical and work differently. In a hospital, since devices are in use by patients, the device upgrade process is gradual. It happens that devices that look identical, work differently because of the firmware version installed. Nurses are overworked, and hospital wards are very busy [9], having different implementations of directional number entry systems can lead to unnecessary incidents that can be fatal.

These differences between software implementations of directional number pads comes down to the various interaction design decisions that are taken when developing the software. Taking the case of number entry systems using directional pads, interaction design decisions can be classified by the following example features:

- **Cursor edge cases** — When the cursor is at a display edge (leftmost or rightmost position), what happens when going beyond that edge? For instance, a machine with display size Δ is said to use *cursor wraparound* if starting from a non-error state (i) pressing the left or right button will always leave the machine in a non-error state; (ii) if the cursor is at position p, pressing right will move the cursor to position $(p+1) \mod \Delta$; and (iii) if the cursor is at position p, pressing left will move the cursor to position $(p-1) \mod \Delta$.

- **Digit limits** — When reaching the minimum or maximum number for a particular digit (0 or 9) what happens when attempting to go beyond it? For instance, a machine is said to use *digit wraparound* if starting from a non-error state (i) pressing up or down will always leave the machine in a non-error state; (ii) if the digit under the cursor is d, then pressing up will change it to $(d+1) \mod 10$; (iii) if the digit under the cursor is d, then pressing down will change it to $(d-1) \mod 10$; and (iv) digits not under the cursor will remain unchanged upon pressing the up or down button.

- **Arithmetic behaviour** — Another option for what happens upon attempting to go above 9 or below 0 for a particular digit, is to have the machine considering the whole display as a number and performing an increment or decrement upon pressing the up or down button. For example, if the display shows `0 9` and ▲ is pressed, the display will now show `1 0`. In this case, one still has distinct design options as to how the machine will function when there is overflow or underflow on the *whole* display. In practice, logical options possible are: (i) putting the machine in an error mode; (ii) setting the display to zero; (iii) setting the display to a string of 9s (the maximum it can display); (iv) nothing happens when trying to go below zero or above the maximum value; (v) the machine performs arithmetic modulo 10^{Δ} (recall that Δ is the size of the display).

TESTING FOR CLASSIFICATION

In order to classify arbitrary 5-key machine implementations, we have built a test suite, with a set of tests for each class of machines such as the ones identified in Section 3. Although, in practice, the tests cannot be effectively run on the actual machine, one would be able to run them on a simulation of the machine — in our case written in Haskell [8]. For a particular implementation of a machine, we assume that we have a Haskell type which represents the internal state of such a machine[1]. In particular, since the type represents a 5-key machine, we will assume that is an instance of the `FiveButtonMachine` Haskell type class which explains how the machine is initialised, its interaction, and how to access its output as seen in Listing 1.

Our implementation automatically induces additional functions on such machines such as (i) `run` which takes a machine state and a sequence of key presses, returns the state of the machine after pressing those keys; (ii) `@@` which, given a machine and display position, returns the digit at the given position; (iii) `delta` which, given a machine returns a unit value at the

[1]The state will typically include the output of the machine (display, cursor and error state) but also any additional memory the machine may have e.g. a machine may allow the clearing of the error state through a double press of the *ok* key, in which case it would have to internally remember whether the last key pressed was the *ok* key.

```
class FiveButtonMachine a where
  initialise :: a
  getDigitDisplay :: a -> DigitDisplay
  getCursorPosition :: a -> Position
  getErrorMode :: a -> Bool
  up, down, left, right, ok :: a -> a
```

Listing 1. Five Button Machine

position of the cursor (e.g. if the cursor lies in the third least significant digit position, the function would return 100).

To test these machines, we will be using the random testing library QuickCheck [6]. The only remaining requirement on the machine implementation is a means of generating arbitrary instances of the machine type (by making it an instance of the QuickCheck `Arbitrary` type class). Based on such a machine implementation, we have implemented tests which can be run to check whether a given machine implementation is of a particular type or not.

Consider the class of machines which implement *cursor wraparound*, which was characterised as three properties explaining how, starting from a non-error state, the error state and cursor position change upon pressing the left and right button. This can be seen in Listing 2.

The specification defines a QuickCheck property `prop_is_cursor_wraparound`, which given a machine state `st`, checks that: if `st` is not in error mode, then (i) moving left or right will not raise the error flag; (ii) moving left changes the cursor position modulo the display size; and (iii) similarly moving right changes the cursor position. QuickCheck can then be used to test whether the property is true for arbitrary instances of the machine state (the machine state `st`) in the following manner:

```
Main> quickCheck prop_is_cursor_wraparound
+++ OK, passed 100 tests.
```

If we run this on a machine that does not change the cursor position when an attempt is made to go beyond the display limits, QuickCheck would identify a counter-example, thus allowing us to deduce that the machine is not in the class of cursor wraparound ones:

```
Main> quickCheck prop_is_cursor_wraparound
*** Failed! Falsifiable (after 1 test):
Machine in non-error mode with display <5832[6]>
```

Similarly, consider the class of *digit wraparound* machines. Such machines would have to pass tests corresponding to the properties identified in Section 3. For instance, the property that states that provided the machine starts off in a non-error mode, then pressing up or down will not change a digit if it does not fall under the cursor would be implemented as follows — note that in this case, we let QuickCheck quantify not only on the machine state, but also on the display positions for which we will check that they have not changed as in Listing 3.

Finally, consider arithmetic machines — to check that a machine is in this class includes checking that the test which ensures that pressing up works arithmetically as long as it would not exceed the the maximum displayed by the machine as in Listing 4. Note that the `delta` is a function which indicates the numeric amount to be added or subtracted (arithmetically) in a particular machine state.

As already discussed, arithmetic machines can be split into a number of subcases, depending on how they react to over and underflow. For instance, the subcase of overflow and underflow being handled modulo 10^{Δ} where Δ is the size of the display is given in Listing 5. The first two lines define when a machine state is in danger of overflow if the up button is pressed or underflow if the down key is pressed. The property then specifies that if the machine is not in an error state and would overflow upon pressing ▲or underflow upon pressing ▼, then the value afterward pressing that key would work using arithmetic modulo 10^{Δ}.

DISCUSSION AND CONCLUSIONS

In critical systems, differences in software implementations can cause harm and, in some cases, even deaths[4, 3]. Since directional number entry systems can have software implementation with different interpretations of what key presses perform means that keying in the same key presses from the starting screen can lead to different results. Moreover, when medical device firmware is upgraded, some of the critical number entry design features may change, and nurses in busy hospital wards are faced with devices that physically look identical but behave differently when the same buttons are pressed — this is hazardous. By enabling the procurement of devices with similar interaction patterns such problems can be reduced.

Given the large number of interaction-design solutions even for a simple 5-button device (e.g. Cauchi et al. [2]) shows that directional number pads can be implemented in 32 different ways) and choosing the best design over the worst is important since it reduces the likelihood of human error eight-fold, the need to be able to classify implementations based on their behaviour and thus assess their safety is crucial.

In this work we have presented an atypical use of testing — that of classifying number entry systems. By building a test suite using Quickcheck to test various software designs on a simulation of directional number entry systems, it is possible to identify which of the various features of directional number entry system the system under analysis has. In the process of using testing for classification, a regulatory body may provide a specification of the feature or class of interfaces in the form of properties which functions should satisfy. QuickCheck then

```
prop_is_cursor_wraparound st =
 not (getErrorMode st) ==>
   let st_l = left st
       st_r = right st
   in    not (getErrorMode st_l) .&&. not (getErrorMode st_r)   --(i)
    .&&. getCursor st_l == (getCursor st - 1) `mod` displaySize --(ii)
    .&&. getCursor st_r == (getCursor st + 1) `mod` displaySize --(iii)
```

Listing 2. Cursor Wraparound Property

```
prop_is_digit_wraparound_other_digits st p' =
  not (getErrorMode st) && p' /= getCursor st ==>
    let st_u = up st
        st_d = down st
        d    = getDigitDisplay st @@ p'
    in    getDigitDisplay st_u @@ p' == d
     .&&. getDigitDisplay st_d @@ p' == d
```

Listing 3. Digit Wraparound Property

```
prop_is_arithmetic_normal_up st =
  (not (getErrorMode st) && (n+delta st <= maximumDisplayed))
    ==> (not (getErrorMode st') .&&. n'==n+delta st)
  where
    n = digitsToValue (getDigitDisplay st)
    st' = up st
    n' = digitsToValue (getDigitDisplay st')
```

Listing 4. Arithmetic Property

```
overflow st = digitsToValue (getDigitDisplay st)+delta st > maximumDisplayed
underflow st = digitsToValue (getDigitDisplay st)-delta st < 0

prop_is_arithmetic_overflow_wraparound st = (
   not (getErrorMode st) && overflow st ==>
      digitsToValue (getDigitDisplay (up st)) ==
      digitsToValue (getDigitDisplay st)+delta st-10^displaySize)
 ) .&&. (
   not (getErrorMode st) && underflow st ==>
      digitsToValue (getDigitDisplay (down st)) ==
      digitsToValue (getDigitDisplay st)-delta st+10^displaySize
 )
```

Listing 5. Arithmetic Property Over/Underflow

tests that the properties hold in a large number of randomly generated cases.

One advantage of our approach is that it allows automated classification of black-box systems, without the need of looking into the implementation or trusting the machine documentation (if it does indeed document the interaction model). This ensures generalisability requiring just the specification of the (software) interface and classification of any other class of (user) interfaces. On the other hand, the solution still suffers from the Achilles heel of testing — not being able to discover counter-examples does not imply their absence. Therefore, although a machine does not behave according to a particular feature, a test suite might still fail to show this. In the domain of 5-button systems, in which the state space of the system is not that large, this is not such an issue. However, in larger interfaces such issues might arise, in which case better designed test suites would be required in order to give more faith in the analysis.

Interestingly, our approach does not distinguish between machines which were not intended to have a particular feature from ones which were intended to be, but do not due to a wrong implementation. Clearly, from a regulatory or safety perspective, either type of machine should not be acceptable. During our implementation of sample machines, the test suite helped us discover instances which were intended to implement a particular feature but did not, due of bugs.

As an extension of this work, we aim to apply this to a number of other classes of interface, and see how the approach can be generalised to product lines (which considers systems in which features can be composed into products — collections of which, resulting from the composition of different features result in product families e.g. the different flavours of an operating system being a product family. Furthermore, we intend to apply the testing language Gherkin (a business readable domain specific language for test specification) to enable features to be specified by non-technical experts.

ACKNOWLEDGMENTS

Project GOMTA financed by the Malta Council for Science & Technology through the National Research & Innovation Programme 2013.

REFERENCES

1. BS EN. 2008. EN ISO 62366:2008 Medical devices - Application of usability engineering to medical devices. (2008).
2. A. Cauchi, Andy Gimblett, Harold Thimbleby, Paul Curzon, and Paolo Masci. 2012a. Safer 5-key number entry user interfaces using Differential Formal Analysis. *Proceedings of HCI 2012, The 26th BCS Conference on Human Computer Interaction* (2012), 29–38.
3. Abigail Cauchi, Andy Gimblett, Harold Thimbleby, Paul Curzon, and Paolo Masci. 2012b. Safer "5-key" Number Entry User Interfaces Using Differential Formal Analysis. In *Proceedings of the 26th Annual BCS Interaction Specialist Group Conference on People and Computers (BCS-HCI '12)*. British Computer Society, Swinton, UK, UK, 29–38.
4. Abigail Cauchi, Patrick Oladimeji, Gerrit Niezen, and Harry Thimbleby. 2014. Triangulating Empirical and Analytic Techniques for Improving Number Entry User Interfaces. In *Proceedings of the 2014 ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '14)*. ACM, New York, NY, USA, 243–252.
5. Abigail Cauchi, Harold Thimbleby, Patrick Oladimeji, and Michael Harrison. 2013. Using Medical Device Logs for Improving Medical Device Design. In *Proceedings of the 2013 IEEE International Conference on Healthcare Informatics (ICHI '13)*. IEEE Computer Society, Washington, DC, USA, 56–65.
6. Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000.* 268–279.
7. Michael D Harrison, José Creissac Campos, and Paolo Masci. 2015. Reusing models and properties in the analysis of similar interactive devices. *Innovations in Systems and Software Engineering* 11, 2 (2015), 95–111.
8. Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. 2007. A history of Haskell: being lazy with class. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*. ACM, New York, NY, USA, 12–1–12–55.
9. Ross J. Koppel and Suzanne Gordon. 2012. *First, Do Less Harm: Confronting the Inconvenient Problems of Patient Safety (The Culture and Politics of Health Care Work)*. ILR Press, New York, New York, USA. 280 pages.
10. Paolo Masci, Rimvydas Rukšėnas, Patrick Oladimeji, Abigail Cauchi, Andy Gimblett, Yunqiu Li, Paul Curzon, and Harold Thimbleby. 2015. The benefits of formalising design guidelines: A case study on the predictability of drug infusion pumps. *Innovations in Systems and Software Engineering* 11, 2 (2015), 73–93.
11. Patrick Oladimeji, Harold Thimbleby, and Anna Cox. 2011. 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)*. Springer-Verlag, Berlin, Heidelberg, 178–185.
12. US FDA. 2013. 510(k) Submission Process. (2013). http://www.fda.gov/medicaldevices///deviceregulationandguidance/howtomarketyourdevice///premarketsubmissions/premarketnotification510k/
13. Yi Zhang, Paul L Jones, and Raoul Jetley. 2010. A hazard analysis for a generic insulin infusion pump. *Journal of diabetes science and technology* 4, 2 (2010), 263–283.