Using Gherkin to extract Tests and Monitors for Safer Medical Device Interaction Design

Abigail Cauchi
University of Malta
Msida, Malta
abigail@cauchi.net

Christian Colombo
University of Malta
Msida, Malta
christian.colombo@um.edu.mt

Adrian Francalanza
University of Malta
Msida, Malta
adrian.francalanza@um.edu.mt

Mark Micallef
University of Malta
Msida, Malta
mark.micallef@um.edu.mt

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

ABSTRACT
Number entry systems on medical devices are safety critical and it is important to get them right. Interaction design teams can be multidisciplinary, and in this work we present a process where the requirements of the system are drawn up using a Controlled Natural Language (CNL) that is understandable by non-technical experts or clients. These CNL requirements can also be directly used by the Quality Assurance (QA) team to test the system and monitor whether or not the system runs as it should once deployed. Since commonly, systems are too complex to test all possible execution paths before deployment, monitoring the system at runtime is useful in order to check that the system is running correctly. If at runtime, it is discovered that an anomaly is detected, the relevant personnel is notified through a report in natural language.

ACM Classification Keywords
H.5.2. User Interfaces: Evaluation/methodology

Author Keywords
Medical Devices; User Interfaces; Testing; Runtime Verification

INTRODUCTION
In healthcare, a significant contributory factor to unnecessary fatalities is the administering of incorrect drug doses. Vincente et al. [10] estimate the probability of fatal number-entry errors on Patient Controlled Analgesia (PCA) pumps (ones controlling pain, typically delivering opiates) as between 1 in 33,000 to 1 in 338,800 (the large uncertainty is due to estimating reporting rates — many errors are not reported).

Paste the appropriate copyright statement here. ACM now supports three different copyright statements:
• ACM copyright: ACM holds the copyright on the work. This is the historical approach.
• License: The author(s) retain copyright, but ACM receives an exclusive publication license.
• Open Access: The author(s) wish to pay for the work to be open access. The additional fee must be paid to ACM.
This text field is large enough to hold the appropriate release statement assuming it is single spaced.
Every submission will be assigned their own unique DOI string to be included here.

To address these worrying figures, within the field of Human-Computer Interaction (HCI), experts carry out studies to find out how to design software better; results [9, 1] show that the consideration of interaction design properties positively impacts safety critical software. However, problems might arise when experts come up with recommendations that need to be communicated to software developers. In particular, since HCI experts do not always understand code*there is a communication gap between non-technical experts and developers when it comes to discussing and agreeing on interaction design properties that should be present within the system.*

Communicating these HCI requirements does not guarantee that they are correctly implemented. Therefore another matter of concern is *enabling non-technical experts to check that the interaction design properties they communicated have in fact been properly included in the final artefact.*

Such challenges are not new to software development: when a client approaches a software house with a system in mind, this has to be properly communicated to technical personnel, a contract is drawn up documenting the features and expected behaviour of the system, and eventually signed. Once the system is completed, the client has to sign it off, accepting it as the embodiment of the originally described system.

Gherkin[1] is a quasi-natural language with very little structure that is used in the software development industry. This enables the client to describe the behaviour of the system in mind in a way that developers can understand. Moreover, Gherkin allows developers to build code directly connected to the client-written description to automatically provide assurance that the specification has indeed been correctly followed. We note that although we will be using Gherkin as our reference language throughout the paper, one can use any other suitable language, possibly one with inbuilt specialised concepts for the domain at hand (e.g., number entry systems).

Motivated by the need for better safety for patients where risks are As Low As Reasonably Practical, ALARP (UK Health & Safety At Work Act (1974) and under similar legislation in

[1] https://github.com/cucumber/cucumber/wiki/Gherkin

other countries), we present a way of exporting the experience of the software development industry for added assurance in quality of medical devices. In particular, we attempt to address the two challenges outlined above: one being that of facilitating the communication between non-technical experts and software developers, and the other being that of providing the means for non-technical experts to check that their recommendations are well-implemented in the resulting artefact.

The main contribution of this paper is an approach that (i) makes communicating interaction design principles with software development teams easier (ii) provides a testing framework to test that the interaction design principles are implemented in the software as specified (iii) automatically generates runtime monitors that ensure that the software is behaving as it should during runtime (iv) provides a feedback mechanism that reports when something does not work as it should during runtime[2]. To illustrate our approach, we show how it can be applied to number entry systems, but this does not limit the generality of the approach.

In the next section we introduce medical device number entry systems in order to describe the case study, then we go on to show how specification properties can be effectively communicated with software developers and verified. Next, we discuss the concepts of runtime monitoring through the case study of medical device number entry systems and show how tests can be converted into runtime monitors. Finally, we discuss our conclusions and future work.

MEDICAL DEVICE NUMBER ENTRY SYSTEMS

As a case study, we will be taking the case of directional pads that are used for number entry (drug dose entry) on medical devices. In medical devices, number entry is safety critical because if the input drug dose is incorrect it may have serious consequences on the health of the patient.

Oladimeji et al. [9] described a few of the possible hardware layouts and carried out a lab experiment to determine which layout is best at reducing the likelihood of human error. From Oladimeji et al.'s work [9] we see that directional pads are best for entering numbers in medical devices since they showed the lowest error rate in the study. 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, different software implementations can result in instances where the same keys can be pushed on devices that look identical but the resulting outputs can be different. Cauchi et al. highlight this in their work [1], and identify the following principles to minimise the magnitude of data entry errors:

[2]The feedback would require some additional features in the design of the device, i.e., a means of sending violation reports back to the developers, but we do not go into this issue in this paper.

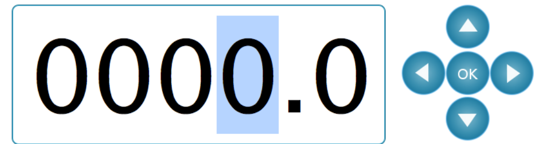

Figure 1. A 5-Key interface with a cursor highlighting a digit. In this system, ▲ and ▼ buttons manipulate the highlighted digit, ◄ ► buttons move the cursor between digits, and the OK button confirms the number.

1. When a cursor is highlighting a digit that is at 9 and ▲ is pressed, then the display should work arithmetically and add 1 to the appropriate digit, e.g., 23496 becomes 23506.
2. When a cursor is at the leftmost position and ◄ is pressed, the display should not change. Similarly if the cursor is at the rightmost position and the ► is pressed.
3. If a user presses a button and the display does not change then the user should be alerted.

Given such a list of principles (with a more comprehensive one in [2]), the following sections attempt to tackle the problem of effectively communicating them to software developers who in turn have to provide solid evidence that the principles have indeed been correctly implemented.

SPECIFICATION AS A COMMUNICATION MEDIUM

In the software development industry, Controlled Natural Languages (CNLs) such as Gherkin are sometimes used to help clients communicate with developers and draw up specifications. A Gherkin specification consists of a number of *features* that are in turn made up of *scenarios* written in statements (referred to as *steps*) that start with the keywords *Given*, *When*, *Then* and *And*. The *Given* keyword is used to describe a precondition, i.e., the context in which the scenario is relevant. The *When* statement describes the action that can be performed when the precondition holds, while the *Then* statement describes the postcondition that should hold following the action. Finally, the *And* keyword is used as a joining mechanism when the author of a scenario needs to split a precondition, action or postcondition over multiple steps.

As an example of how Gherkin can be used to describe interaction design properties, we consider directional number entry systems that are commonly used in medical devices. Taking the example of property 2 from the previous section, an interaction designer and a software developer discuss the property and draw up a Gherkin specification as follows:

```
1 Scenario: Going beyond left boundary
2 Given cursor is on leftmost position
3  When Left is pressed
4  Then Cursor position stays the same
5  And Displayed number stays the same
6  And User is alerted
```

Similarly to the above, a Gherkin scenario can be drawn up for all the features of the number entry system, forming a complete specification.

VERIFYING THE SPECIFICATION

While having an easy to use specification language that doubles up as a communication tool is useful, it is desirable to

have a direct way of confirming that the described features have indeed been implemented. From the possible verification techniques available, in this paper we concentrate on those which can check the implementation directly, i.e., leaving out techniques such as model checking and favouring instead approaches such as testing.

Gherkin scripts are interpreted by a tool called Cucumber[3], which comes with Eclipse IDE support to facilitate the creation of tests. As shown in Figure 2, a method is automatically generated for each unique step definition (corresponding to a line) in the Gherkin specification. Subsequently, the software developer implements the methods in the step definitions file to do what is required in each step. Methods for the *Given* parts of scenarios, set up the system for the test by driving the system to a point where the precondition holds. Methods for *When* statements perform the actions to be tested, while methods for *Then* statements evaluate assertions.

In the case of this example, the step definitions are implemented as seen in Figure 3. Each method in the class *StepDefinitions* is annotated with a keyword and regular expression that is related to the Gherkin code. Through method *cursor_on_leftmost_position()*, the number entry system satisfies the precondition specified by the *Given* keyword. In the method *left_is_pressed()* that is related to the *When* keyword, the current cursor position and displayed number are logged and the left action is simulated on the system. The next three methods *cursor_position_stays_the_same()*, *displayed_number_stays_the_same()* and *user_is_alerted()* are used to carry out the assertions for the property being checked.

When the tests are executed, the corresponding step definition for each line in the scenario is executed in the order they are listed in the feature file. When run, the Gherkin plugin outputs the number of tests that passed or failed and also outputs the sequence in which the step definition methods were called.

Using this approach, the original specification writers, i.e., the non-technical experts, can be shown concrete evidence that each scenario has indeed been correctly incorporated in the implementation. However, as with all testing techniques, with this approach, we cannot guarantee that the system will run according to the specification in all possible runtime executions. Software testing only checks that a specific program path satisfies a specification but it cannot guarantee that the specification holds along all possible execution paths.

RUNTIME MONITORING

Runtime monitoring [7] has been used in safety, business, or mission-critical systems to ensure that they continue to behave as expected beyond the testing phase, i.e., after deployment. Monitoring typically depends on some form of source or byte code manipulation to extract relevant runtime events, that are subsequently checked against the specification. Using runtime monitoring, the program execution path that is running is checked to determine whether or not a specification holds.

A major hurdle for the adoption of this technique (based on industrial experience of the authors [4, 3]), is to reformulate the specification into a form accepted by the monitoring tool. In this paper, we present an approach where we attempt to automatically extract the monitoring specification from the test specification. In this way, we get communication, specification, testing and runtime monitoring all integrated in a single approach. If at runtime, an anomaly is detected, it is logged and the relevant personnel are notified as necessary.

The main difference between tests and runtime monitors is that while the former drives and checks the system, the latter listens to — as opposed to drives — and checks the system. This constitutes a major difference since defining how to reach a state is significantly different from checking whether the state has indeed been reached. In the case of our example, if we want to test a property about the cursor being on the leftmost position, in the test we assign the cursor to the leftmost position. In the monitor's code, the assignment statement in the test code has to be translated to a condition which checks whether the cursor is indeed on the leftmost position. Therefore our task is to attempt to extract information from the test that defines the how, and generate a monitor that defines the condition.

In what follows, we give an overview of how tests can be converted into monitors referring back to the number entry system example.

Translating tests to runtime monitors

Figures 4 and 5 show the resulting monitors from the translated tests. Starting with the easy part of the translation first, we note that the *Then* part remains the same across both testing and runtime monitoring. Therefore this will be copied directly from the step definitions.[4]

Next, we consider the *When* component of the specification. Recall that in the monitoring context we need to listen to the action rather than trigger it. For this purpose we employ Aspect Oriented Programming (AOP) technology [6] which instruments monitoring code in the system code, providing access to the relevant events. Referring back to the running example, consider the step definition of the *When* part which calls method *left*. Using AOP we can intercept calls to this method using what is known as a *pointcut* so that control is transferred to the monitor upon the call to the method. AOP provides several pointcut options such as *before* and *after* the method call. However, since we need to check for both the precondition and the postcondition, we opt for the *around* pointcut which gives us access to both the before the start and the end of the method. Figure 4 shows how we have copied the code from the step definition (line 30 and 31), and called the *proceed* keyword to perform the intercepted call to method *left*. Subsequently, ignoring line 36, the assertions are copied from the step definitions annotated with the *Then* keyword in the feature file.

Finally, we come to translating the precondition part annotated by the *Given* keyword. The step definition describes one way

[3]Cucumber has been implemented in a variety of languages but there are similar tools such as SpecFlow which serve the same purpose.

[4]We note that this direct correspondence between testing assertions and monitoring ones is not as straightforward in general. However, for the case study under consideration this is the case.

```
@Given("^cursor on leftmost position$")
public void cursor_on_leftmost_position() throws Throwable {
    // Express the Regexp above with the code you wish you had
    throw new PendingException();
}

@When("^left is pressed$")
public void left_is_pressed() throws Throwable {
    // Express the Regexp above with the code you wish you had
    throw new PendingException();
}

@Then("^cursor position stays the same$")
public void cursor_position_stays_the_same() throws Throwable {
    // Express the Regexp above with the code you wish you had
    throw new PendingException();
}

@Then("^displayed number stays the same$")
public void displayed_number_stays_the_same() throws Throwable {
    // Express the Regexp above with the code you wish you had
    throw new PendingException();
}

@Then("^user is alerted$")
public void user_is_alerted() throws Throwable {
    // Express the Regexp above with the code you wish you had
    throw new PendingException();
}
```

Figure 2. Default step definition code generated by Cucumber

through which the state can be set up to satisfy the precondition. We note that while there can indeed be many ways in which the precondition can be satisfied, the test only provides one. The approach we take is that of — again using AOP — detecting when the set up sequence occurs during the system execution. The assumption here is that if we observe the steps defined in the *Given*, then the precondition will be satisfied. For instance, take *system.setCursor(0)* of the running example: We create a pointcut (see Figure 5) which listens for *setCursor* and ensures that the parameter is zero. When observing such a call, we deduce that at that point the state of the system is the same as when we run the *Given* step of the test. At this point we set a flag true to communicate this fact to the pointcut matching the *When* step. For this reason, in the *When* pointcut we only gather the information (lines 30 and 31 of Figure 4) and check the postcondition (lines 38-41) if the flag is true.

While the approach described so far works whenever a call to *left* immediately follows a call to *setCursor(0)*, we have to ensure that no other method changes the system state from the time we match *setCursor(0)* till the time we match *left*. Therefore, assuming that only *setCursor* can change the cursor position, whenever we match *setCursor* with a non-zero parameter, we set the flag to false. In a more general context, we can use static analysis to identify all the methods which modify variables accessed or modified by *setCursor* and reset the flag any time such methods are called.

Discussion

In this work we proposed using Gherkin as a communication language to communicate interaction design concepts to software developers. Using Gherkin for testing properties of reactive interface systems was found to be straightforward, with the Given-When-Then structure of the language matching the structure of the properties we were verifying.

Another concern that this work addresses is ensuring that interaction design properties behave as they should at runtime. We demonstrated how Gherkin step definitions can be translated to runtime monitors automatically in order to make the process of integrating runtime monitoring into medical devices (and other critical systems) seamless. The benefit of having runtime monitors checking that systems are working as they should is that if things go wrong, we have a way of giving both users and developers feedback. In this work, the feedback we provide is simply alerting when an interface does not behave according to specification. This feedback can be more sophisticated in that, from a user perspective, if a system does not behave as it should, there may be mechanisms implemented to get the attention of the user to ensure that no erroneous actions are made. From a manufacturer's perspective, bad states may be logged onto the device to be addressed.

The challenge with translating Gherkin step definitions to runtime monitors automatically lies in the pre-condition portion of tests. While tests set a system to a state satisfying the precondition, in runtime monitoring, a monitor waits for such a state to be reached. Therefore, the pre-condition in a test takes the form of a set of state-modifying statements, while the precondition in a runtime monitor is written out as a conditional statement. We overcome this problem by using aspect-oriented programming to detect the occurrence of the sequence of steps corresponding to the state-modifying statements.

In order to investigate whether runtime monitors introduced considerable runtime overheads, we used the number entry keystrokes of medical device logs from 19 medical infusion pumps that were running in a hospital for four years. We simulated the key presses on our number entry simulator and recorded the differences between running all the number entry inputs with and without runtime monitors. The introduction

of the runtime monitors increased the total running time of the simulations from 2ms to 167ms on a 2.4GHz processor and 8GB RAM. While this increase is substantial, the number is still small in absolute terms (particularly since it represents four years of pump usage) and therefore we do not expect it to affect the responsiveness of number-entry systems.

In practice, the instrumented code would be used on the devices themselves as a safer version of the firmware. With these overheads, we envisage no performance issues that would hamper such an approach.

RELATED WORK

The United States Food and Drug Administration (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 [11]. Work by Zhang et al. [11] is focused on creating a generic model of an infusion pump (without considering models of data entry systems) while the work we present in this paper goes into detail on number entry systems. Through our use of runtime verification, we also add the extra assurance that, during everyday use, systems work as specified.

Harrison et al. [5] and Masci et al. [8] use formal methods, namely model checking and theorem proving in order to verify interaction design properties of 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, obtaining source code of infusion pumps is not straightforward, thus making it difficult to automatically model the system and verify that safe interaction design properties hold within it. In our approach, it is possible for either manufacturers or regulation bodies such as the US FDA to monitor the software's behaviour at runtime. Using runtime verification techniques, it is possible for manufacturers to keep their code unexposed and provide an API with standardized method calls in order for an external body, such as the US FDA, to be able to monitor the infusion pump behaviour.

CONCLUSIONS AND FUTURE WORK

It is important to get user interaction design right, especially in safety critical domains. The field of HCI is multidisciplinary — psychologists, designers and computer scientists work together to develop usable and useful software systems. For such a field, a CNL such as Gherkin may be useful for bridging the communication gap between non-technical experts and software developers. In this work we have shown how a CNL can be used to describe safety critical number entry systems design properties and we discussed how we can test these properties and monitor them at runtime. Since it is difficult to have complete code coverage when testing a system, runtime monitoring is very useful for critical systems. In this case, runtime monitors continuously check that the system works as it should, and if at runtime the program goes through an execution path where one of the requirements does not hold, it is logged and the device manufacturer can be notified.

For future work, we will be exploring the design of CNLs to make them more suitable for the domain at hand, possibly with inbuilt specialised concepts for the domain. In this work we have seen how this process works for a safety critical number entry systems and in the future we will be applying this to a more complex use case such as that of a business critical financial transaction system.

ACKNOWLEDGMENTS

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

REFERENCES

1. A. Cauchi, Andy Gimblett, Harold Thimbleby, Paul Curzon, and Paolo Masci. 2012. 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.
2. 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, 56–65.
3. Christian Colombo, Gordon Pace, and Patrick Abela. 2012. Safer asynchronous runtime monitoring using compensations. *Formal Methods in System Design* 41, 3 (2012), 269–294.
4. Christian Colombo, Gordon J. Pace, and Gerardo Schneider. 2008. Dynamic Event-Based Runtime Monitoring of Real-Time and Contextual Properties. In *Formal Methods for Industrial Critical Systems (FMICS) (Lecture Notes in Computer Science)*, Vol. 5596. L'Aquila, Italy, 135–149.
5. 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.
6. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-oriented programming. In *ECOOP 1997 Object-oriented programming*. Springer, 220–242.
7. Martin Leucker and Christian Schallhart. 2009. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming* 78, 5 (2009), 293–303.
8. 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.
9. 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, Berlin, Heidelberg, 178–185.

10. Kim J Vicente, Karima Kada-Bekhaled, Gillian Hillel, Andrea Cassano, and Beverley A Orser. 2003. Programming errors contribute to death from patient-controlled analgesia: case report and estimate of probability. *Canadian Journal of Anaesthesia* 50, 4 (2003), 328–332.

11. 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.

```java
3  public class StepDefinitions {
4
12     private fiveKey system = new fiveKey();
13
14     int cursorPos;
15     String displayedNumber;
16
17 @Given("^cursor on leftmost position$")
18 public void cursor_on_leftmost_position() throws Throwable {
19     system.setCursor(0);
20 }
21
22 @When("^left is pressed$")
23 public void left_is_pressed() throws Throwable {
24     this.cursorPos = system.getCursorPos();
25     this.displayedNumber = system.getDisplay();
26     system.left();
27 }
28
29 @Then("^cursor position stays the same$")
30 public void cursor_position_stays_the_same() throws Throwable {
31     Assert.assertTrue(this.cursorPos == system.getCursorPos());
32 }
33
34 @Then("^displayed number stays the same$")
35 public void displayed_number_stays_the_same() throws Throwable {
36     Assert.assertTrue(this.displayedNumber == system.getDisplay());
37 }
38
39 @Then("^user is alerted$")
40 public void user_is_alerted() throws Throwable {
41     Assert.assertTrue(system.isUserAlerted());
42 }
43
```

Figure 3. Developer or test engineers populate the generated skeleton code (as seen in Figure 2) to interact with the system during automated testing

```java
27 int around (fiveKey system): execution(* *.left(..)) && target(system)
28 {
29     if (noCursorWrapAroundPreCondition){
30         currentDisplay = system.getDisplay();
31         cursorPos = 0;
32     }
33
34     int ret = proceed(system);
35
36     if(noCursorWrapAroundPreCondition)
37     {
38         if ((system.getCursorPos() == cursorPos) && (system.isUserAlerted()) && (system.getDisplay().eq
39             System.out.println("Monitor says: OK");
40         else
41             System.out.println("Monitor says: OOPS!");
42     }
43     return ret;
44 }
```

Figure 4. Monitoring using Aspect Oriented Programming

```java
12 boolean noCursorWrapAroundPreCondition = false;
13
14 after(int pos): (execution(* *.setCursor(..)) && args(pos)){
15     if (pos == 0)
16         noCursorWrapAroundPreCondition = true;
17     else
18         noCursorWrapAroundPreCondition = false;
19 }
20
```

Figure 5. Setting flags