FormTester: Effective Integration of Model-Based and Manually Specified Test Cases

Rahul Dixit, Christof Lutteroth and Gerald Weber
Department of Computer Science
University of Auckland, Auckland, New Zealand
Email: rahul.dixit.zero@gmail.com

***Abstract*—Whilst Model Based Testing (MBT) is an improvement over manual test specification, the leap from it to MBT can be hard. Only recently MBT tools for web applications have emerged that can recover models from existing manually specified test cases. However, there are further requirements for supporting both MBT and manually specified tests. First, we need support for the generation of test initialization procedures. Also, we want to identify areas of the system that are not testable due to defects. We present FormTester, a new MBT tool addressing these limitations. An evaluation with real web applications shows that FormTester helps to reduce the time spent on developing test cases.**

I. Introduction

In manual test specification, test cases and test initialization procedures are scripted manually. The test cases execute the initialization procedures and call an underlying adapter to drive the SUT; this is still the prevailing approach in commercial environments at present. In Model Based Testing (MBT), one develops a model of the system under test (SUT) and then generates test cases from it instead of scripting them individually [1]. This saves time and reduces the technical skill level required for manual test specification. It is also easier to maintain an SUT model and generate test cases from it than to maintain a repository of test cases themselves.

Only recently MBT tools have started to derive an SUT model from manual test cases and eliminate the need for specification of SUT models separately from test cases by testers [2]. However, there are scenarios where it is desirable to use MBT alongside manually specified test cases, e.g. when adding a test case for every defect in test-driven development. Current MBT tools such as GUITAR [3], NModel [4], Tonella et. al's statistical tester [5] and Testilizer [2] do not specifically facilitate the partial automation of manually specified test cases, e.g. the automation of only the test initialization procedures. Test initialization procedures are a sequence of steps that bring the web application to a required page from its start state, in order to test a particular function. Manually specified test cases are not executable on their own and require such initialization before they can be run. Writing initialization procedures is a time consuming activity. We present FormTester[1], a tool that can generate test initialization procedures for manually specified test cases based on an SUT model.

[1]Tool demo video: http://www.youtube.com/watch?v=T6D5fhR_ptk

Finally, existing MBT tools do not identify areas of the web application that are not testable due to defects, which may in turn conceal other defects. FormTester combines online testing with a reachability analysis of the SUT model, resulting in *reachability marking*. Pages and actions rendered unreachable due to a defect are identified, as well as the problems that need to be addressed in order to make the unreachable parts accessible. Reachability marking also enables using online test termination criteria based on coverage, i.e. online testing can be stopped once all reachable parts of the web application have been tested.

II. Tool Overview

FormTester implements three phases of the MBT process: Model Development, Test Case Generation and Test Case Execution [6] as displayed in Figure 1.

During the Model Development phase, test cases are used to reverse engineer an SUT model of the web application. The visualizer produces a diagram of the model that can be verified with developers to confirm model accuracy. Manual inspection of models is the most common form of model verification [7]. Testers may then change the test cases and this process is repeated to refine the model. Furthermore, the model can be annotated with probabilistic information to guide test case generation.

During the Test Case Generation phase, two kinds of outputs are generated: Test initialisation procedures and test cases. The online testing component traverses the SUT model to execute online tests, and when it identifies transitions rendered unreachable due to defects, these are marked. During the Test Execution phase, an Adapter is implemented that is common to both offline and online components, and is used to drive the GUI using high level commands.

A. Model Development: Recovering Models from Manually Specified Tests

Manual test cases are used to reverse engineer the SUT model. They contain information that describe page to page movements during a test, actions performed on pages and test data used. One can use this information to reverse engineer a Finite State Machine (FSM) model of the web application.

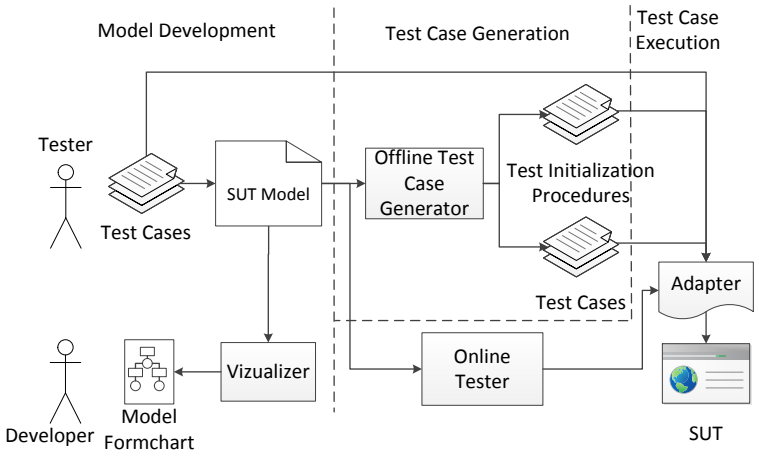

Fig. 1. Toolchain Overview

The format of test cases is an adapted Gherkin input file. Gherkin is a business readable, domain specific language that describes test cases in structured English using 'Given, When, Then' sentences. An example Gherkin feature file is shown in Listing 1. A start page is given in the 'Given' sentence of the scenario. An action to be taken is defined in the 'When' sentence, which may include field data. Following this, the expected output page is defined in the 'Then' sentence, including output field information. This describes a set of transitions, which can be annotated with probabilities to characterize real use. The transitions can be visualized in a formchart model, where pages are represented by ellipses and actions by boxes [8]. Figure 2 shows a probabilistic formchart SUT model, recovered partially from the test case in Listing 1.

B. Test Case Generation

The Offline Test Case Generator generates test initialization procedures as a sequence of steps performed to bring the web application into a desired state, using a shortest paths algorithm. Test initialization procedures are generated for the original manual test cases used to reverse engineer the model. This allows the manual test cases to be used as automation tests as well as SUT model specification.

Listing 1. Feature File

```
Scenario: Go to Login Page
Given the start page: Main Page
When I perform the following action: Go to Login Page
When selecting the action has a probability of: 0.6
When I enter the following input data into the input
    fields:
|Field Name | login link                    |
|Field Type | Link                          |
|XPath      | //li[@id='pt-login']//a       |
|Data       | click                         |
Then I am taken to the output page: Login Page
Then going to the page has a probability of: 1.0
```

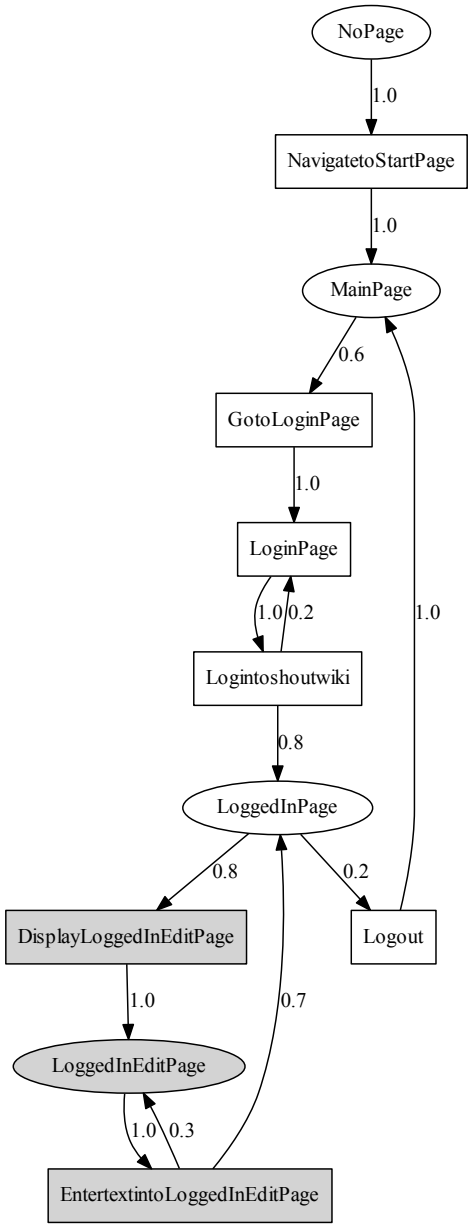

Fig. 2. Formchart with unreachable parts in gray

These procedures are called from the manual test cases using the 'Given the start page' sentence. The model is traversed to compute shortest paths from the start page to every page in the model using the Dijkstra algorithm. The generated paths are stored in a source file, which is referenced when executing the tests. These paths are used when running the tests to bring the test into a desired state.

Once done, the remainder of the test is executed directly from

the manual test case, which calls the adapter. This partial automation of manual test cases removes the need to specify separate SUT models other than manual test cases. This novel feature optimizes the path a test case takes to perform a test, increasing efficiency by removing redundant steps.

This partial automation of manual test cases allows specification of SUT models using manual test cases, without fully implementing them. This saves time in SUT model specification, and subsequent test initialization automation removes the need for complete test definition, saving further time.

Currently, FormTester automates initialization procedures for offline testing using shortest paths only. However, this can be extended in the future to include arbitrary paths to increase the scope of testing – other paths can uncover additional defects not identified by shortest paths alone. In online testing, all paths are randomized, so there is no such limitation.

Besides test initialization procedures, test case generation can be performed. Test cases are generated that are equivalent to the original manual test cases that were used to reverse engineer the SUT model. This is done by traversing th model using a breadth first search to identify combinations of paths through the model, and use these as tests. Additional tests for scenarios not originally specified can also be derived from the SUT model. Here, we specify the number of steps the test should run for, and traverse the model those many times (often in repeated sequences) to test new and original sequences.

C. Test Execution: Identifying Untestable Parts of a Web Application

In contrast to offline testing, the online testing component walks through the system based on the given transition probabilities between pages and actions. If possible, this walker continues to traverse the model, instead of restarting testing [9]. The logic to determine the next action or page to visit is then based on both probabilistic profiles of the model and how many times a node has been previously visited. This makes route selection more realistic.

The online testing component also handles defects along the way, identifying unreachable parts of the web application due to defects. For example, Figure 2 shows reachable parts of the web application in white and unreachable parts in gray, due to a defect in the only transition leading to the 'Logged in Edit Page'.

If a test fails on an action with a genuine defect, all incoming transitions of that action are checked to see if they cause a failure on that action, too. If all incoming transitions cause a failure, the action is marked as unreachable. This in turn may cause the page following the action to be marked as unreachable, and so on. As parts of the model are marked as unreachable, other parts are marked as well if there is no working transition to them left. A depth-first search is used for this reachability analysis.

III. Evaluation

A. Methodology

Ideally, the evaluation would be performed as a between-group experiment: one group of developers uses manual test specification to create a test suite for a representative set of web applications, while a second group with comparable skills and experience uses the FormTester tool to achieve the same for the same applications. The measurements for the two groups (manual vs. FormTester) are compared overall and for each of the applications. At this point, we present only a self-experiment, which serves as a pilot study to such a full evaluation in the future.

FormTester's developer implemented manual test specifications for two web applications, and used FormTester for one other web application, thus being the sole automator on all evaluated applications. We then compared the two conditions using four key metrics. The efficiency of test automation was measured using the metric 'Minutes per Test Case'. We measured this by tracking the total time spent automating a test suite and then dividing it by the total number of test cases developed. We also measured the 'Lines of Code' written to give a second indicator of effort required to automate. We measured the 'Number of Defects' found on all systems to give a comparison of defect detection rates.

We measured these metrics including and excluding project setup. Project setup refers to activities of setting up the code and test data infrastructure, and developing initial smoke tests. Although project setup effort can be high in MBT, this is amortized over the number of tests developed, so the overall effort can be lower than with manual methods.

B. Tested Applications

Manual test specification was performed on one small-sized project called 'Administration Console' and one mid-sized project called 'Self Service'. Self Service is a website where gym owners can log in and manage customer accounts. They can create customer accounts, update their payment details, update their billing schedules, and generate reports. Three developers developed this application over 2 years, writing 5990 lines of code. Administration Console is an administration tool for Self Service users, supporting the creation of Self Service user accounts and setting permissions specifying what Self Service users can do. Three developers developed this application over one year, writing 1461 lines of code.

FormTester was applied on 'Autobill Corrections'. As part of customer account management, invalid operations or transactions occur sometimes that can wrongly bill or debit a customer. Autobill Corrections corrects these invalid transactions by passing requests across a four user sequential workflow

system involving: i) Customer Service Representatives (CSRs who attend customer issues); ii) Team Leads (who direct CSRs); iii) Finance (who administer the change requests); and iv) Operations Manager (responsible for the overall team). This is a large-scale application developed by three developers over 1.5 years with 18022 lines of code. There are several special edge cases where error handling is required, which makes the application complex to test.

C. Results

There was an overall improvement in Minutes/Test (including project setup) of 30.86% over the average of the two manual projects. Even more notable was the improvement in Minutes/Test excluding project setup which was 58.06%. The higher time savings excluding project setup indicate that savings will be even more significant as the size of the application being tested increases.

The total number of functional defects found over the course of automation were similar in MBT and non-MBT web applications. In addition, the types of defects were also similar. These defects were both identified by offline and online testing. These results are detailed in Table I.

TABLE I
TRIAL RESULTS

	Admin Console (Manual)	Self Service (Manual)	Autobill Corrections (MBT)
Project setup (hours)	20	40	40
No. of tests	576	996	828
Total hours	120	217	122
Mins/Test (incl. proj. Setup)	12.5	13.07	8.84
Mins/Test (excl. proj. Setup)	10.42	10.66	4.42
Total lines of automation code	3160	4554	6442
Natural language specification code (SpecFlow)	357	399	2424
Implementation code (C#)	3160	4554	0
Generated code (C#)	0	0	2206
Adapter code	0	0	1812
Total no. of functional defects	5	5	4

D. Discussion

Due to the small scale of the evaluation, the observed benefits of FormTester were primarily an improved performance. We expect that other benefits can be measured in larger, future evaluations. The reductions in development time are mainly due to automated code generation: instead of writing binding methods for Gherkin steps and debugging them, the code is generated through MBT. The gains become greater as project size increases, as there is less effort in specifying additional model features compared to specifying a basic model from scratch.

However, the MBT automation was performed by the developer of FormTester, who knew it intricately and was able to debug errors easily when they occurred. Had a more novice tester performed the MBT, the results could have been different, with a lesser efficiency in tool usage observed. However, the way the tool is used is standard and similar to how non-MBT projects are automated, i.e. standard Gherkin constructs are used. This standardization mitigates this issue.

A threat to validity during the study was caused by limitations in our Gherkin parser implementation. Whilst, its ability to parse the Gherkin language constructs was adequate, it broke every time data was entered in an incorrect format, without supplying useful error messages to debug the problem. This made data entry into the SUT model much more time consuming than necessary. With better error handling and reporting capabilities, the time savings achieved by using FormTester would increase significantly. The fact that significant time savings were observed in spite of an effective parser, reinforces FormTesters effectiveness.

The amount and types of defects identified by MBT are similar to manual methods. This indicates that the MBT tool is similarly effective in detecting defects. It must be noted that defect counts are usually lower in MBT since static modeling of the system is likely to reveal defects before test execution begins. This may explain the slightly lower count of defects identified in MBT.

REFERENCES

[1] A. Hartman, M. Katara, and A. Paradkar, "Domain specific approaches to software test automation," in *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering: companion papers*. ACM, 2007, pp. 621–622.

[2] A. Milani Fard, M. Mirzaaghaei, and A. Mesbah, "Leveraging existing tests in automated test generation for web applications," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 67–78.

[3] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon, "Guitar: an innovative tool for automated testing of gui-driven software," *Automated Software Engineering*, pp. 1–41, 2013.

[4] J. Ernits, R. Roo, J. Jacky, and M. Veanes, "Model-based testing of web applications using nmodel," in *Testing of Software and Communication Systems*. Springer, 2009, pp. 211–216.

[5] P. Tonella and F. Ricca, "Statistical testing of web applications," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 16, no. 1-2, pp. 103–127, 2004.

[6] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. Morgan-Kaufmann, 2007, ch. 2.2, p. 27.

[7] B. Polgár, I. Ráth, Z. Szatmári, Á. Horváth, and I. Majzik, "Model-based integration, execution and certification of development tool-chains," *Model Driven Tool and Process Integration*, vol. 35, 2009.

[8] D. Draheim and G. Weber, *Form-Oriented Analysis*. Springer, 2005, ch. 2.1, p. 10.

[9] M. Veanes, C. Campbell, W. Schulte, and N. Tillmann, "Online testing with model programs," in *ACM SIGSOFT Software Engineering Notes*, vol. 30. ACM, 2005, pp. 273–282.