VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wānanga o te Ūpoko o te Ika a Māui

School of Engineering and Computer Science
Te Kura Mātai Pūkaha, Pūrorohiko

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@ecs.vuw.ac.nz

QuickCheck for Whiley

Janice Chin

Supervisors: David Pearce, Lindsay Groves

Submitted in partial fulfilment of the requirements for
Bachelor of Engineering with Honours.

Abstract

This document is a project proposal for the project, QuickCheck for Whiley. The project aims to improve software quality of Whiley programs by implementing an automated test-case generator for Whiley. The solution will be evaluated by checking if it can detect bugs in a small benchmark set. No special resources are required for this project.

1. Introduction

Testing is an important process in software development as it helps detect the presence of bugs. However, writing and running tests can be tedious and costly. Furthermore, it is difficult to write tests for all possible cases and be able to detect all bugs.

An automated test-case generator called QuickCheck was implemented in Haskell by Koen Claessen and John Hughes which alleviates these issues.

This lightweight tool employs two testing ideas. Firstly, QuickCheck uses property-based testing which uses conditions that will always hold true. Users define the conditions in terms of formal specifications to validate functions under test.

Secondly, QuickCheck generates test cases automatically using random testing. Input values are randomly generated for each test (within a range) using generators that correspond to the input value's type. QuickCheck contains generators for most of Haskell's predefined types and for functions. It requires the developer to specify their own generators for user-defined types.

An example of using QuickCheck would be reversing a list.

Firstly, a property is defined where the list xs, should be the same as reversing xs twice.

```
propReverseTwice xs = reverse (reverse xs) == xs
```

QuickCheck is then executed by importing the property and passing into the interpreter:

```
Main:> quickCheck propReverseTwice
Ok: passed 100 tests.
```

This will create a large number of test cases by using the user-defined property and random input values to check the property holds. QuickCheck reports various test statistics including the number of tests that have passed or failed.

QuickCheck has been re-implemented in other languages including Scala, Java and C++. A commercial version of QuickCheck in Erlang called Quviq QuickCheck is co-founded by one of the original developers, John Hughes and is an extension of the original QuickCheck for Haskell.

This project is about implementing an automated test-case generator for the programming language, Whiley based on the QuickCheck tool.

2. The Problem

Whiley is a programming language, developed to verify code and eliminate errors using formal specifications. Ideally, programs written in Whiley should be error free. To achieve this goal, Whiley contains a verifying compiler which employs the use of specifications written by a developer to check for common errors such as accessing an index of an array which is outside its boundaries.

Below is an example of a Whiley program with a function, `min()` which finds the minimum value of two integers. This program prints out "The smallest integer is 3" when executed.

```
1  import std::io
2  import std::ascii
3
4  method main():
5      int x = 5
6      int y = 3
7      int smallest = min(x, y)
8      io::print("The smallest integer is ")
9      io::println(smallest)
10
11 function min(int x, int y) -> (int r)
12 ensures r == x ==> x <= y
13 ensures r == y ==> y <= x:
14     if x <= y:
15         return x
16     else:
17         return y
```

Currently, the verifying compiler has limitations when evaluating complex pre- and post-conditions. For example, a post-condition could be falsely identified as not holding by the verifying compiler even though the program does meet the post-condition. Therefore, this project aims to implement an automated test-case generator in Whiley to improve software quality and increase confidence in unverifiable code.

3. Proposed Solution

The automated test-case generator needs to read a Whiley program and then generate tests for functions in the program. Input values that adhere to the function's preconditions will be used in testing. Strategies for generating input values include random generation and dynamic generation. The success of each test is determined by using the function's postconditions. The test results should show the percentage of tests that passed and the inputs used in the failed tests.

Several components developed for the Whiley language will need to be used for the automated test-case generator.

After basic test-case generation is completed, more complex types can be generated for testing and/or further enhancements to the tool can be made.

Possible enhancements include:

- Weight on the frequency of test values
- Shrinking failed test cases
- Classifying tests
- Using dynamic symbolic execution

Regular meetings (weekly or fortnightly) with the primary supervisor, David Pearce will be held for assistance and guidance in the project.

Timeline

A gantt chart of the project can be found on the project's repository.

Trimester 1

Output	Estimated Time	Start Date	Complete by
Produce bibliography	2 weeks	19/3/18	9/4/18 (Week 5)
Produce project proposal	2 weeks	19/3/18	9/4/18 (Week 5)
Implement an automated test-case generator for Whiley programs for the primitive types: bool, byte, int, null	3 weeks	19/3/18	16/4/18 (Week 6)
Implement automated test-case generation for more complex types: void, array, union and records	3 weeks	16/4/18	14/5/18 (Week 9)
Produce preliminary report	3 weeks	14/5/18	10/6/18 (Week 12)
Extend the automatic test generator to be able to generate other types or use methods for better test case distribution (weighting, classification)	3 weeks	11/6/18	2/7/18 (Exam period)

Trimester 2

Output	Estimated Time	Start Date	Complete by
Produce slides for presentation of preliminary report	1 week	2/7/18	9/7/18 (Break)
Extend the automatic test generator to include other methods of testing (symbolic) or be able to generate other types	3 weeks	9/7/18	30/7/18 (Week 4)
Produce draft of final report	6 weeks	30/7/18	15/9/18 (Week 7)
Complete implementation of automated test generator	3 weeks	15/9/18	5/10/18 (Week 10)
Finalise final report	2 weeks	6/10/18	21/10/18 (Week 12)
Prepare slides for final presentation	3 weeks	22/10/18	16/11/18 (Exam period)

4. Evaluating your Solution

This project will be evaluated by checking if the developed tool can detect bugs that have been inserted into a small benchmark set.

5. Resource Requirements

No special resources are required. Only the use of the ECS laboratories is required.