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
Preliminary Report

Janice Chin

Supervisors: David Pearce and Lindsay Groves

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

Abstract

Contents

1 Introduction

The detection and elimination of bugs is important in software development as it improves the quality and reliability of software.

To help eliminate errors in code, the Whiley programming language was developed to verify code using formal specifications [1]. Ideally, programs written in Whiley should be error free [1]. 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 [1].

Listing 1 illustrates an example of a Whiley program with a function, `min()` which finds the minimum value of two integers. Executing `min(2, 3)` will return the integer, 2.

Listing 1: Whiley program for the min function

```
1 function min(int x, int y) -> (int r)
2 ensures r == x ==> x <= y
3 ensures r == y ==> y <= x:
4 if x <= y:
5    return x
6 else:
7    return y
```

Currently, the verifying compiler has limitations when evaluating complex pre- and post-conditions. For example in Listing 2, the post-condition is falsely identified as not holding by the verifying compiler even though the program does meet the post-condition.

Listing 2: Whiley program for the square function

```
1 function square(int x) -> (int r)
2    ensures r >= 0:
3    return x*x
```

Consequently, tests would need to be created to detect if there are further problems within the program. 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. Instead of manually creating tests, a tool could be created that automatically generates and executes tests on a program such as QuickCheck which was implemented in Haskell by Koen Claessen and John Hughes [2].

QuickCheck uses property-based testing and random testing [2]. Properties defined by the user are used to randomly generate and execute an arbitrary number of tests [2].

Random testing has been widely studied in the literature, leading to the creation of new variants and testing techniques including concolic testing [3], feedback-directed testing [4], [5] and mutation testing [6].

This project aims to implement an automated test-case generator for the Whiley programming language, based on the QuickCheck tool. As a result, the automatic test-case generator will improve software quality and increase confidence in unverifiable code.

Currently, a working implementation of QuickCheck for Whiley has been completed which can generate and execute tests using random and exhaustive test-case generation. Constraints applied to integers and array sizes on nominal types have also been implemented to limit the range of integers and size of arrays generated.

2 Background Survey

There exists a number of different test-case generation tools for different languages such as QuickCheck for Haskell [2], Concolic Unit Testing Engine (CUTE) for C [3] and Randoop for Java [5] and .NET [4].

QuickCheck for Whiley is specifically created for the Whiley programming language and uses different techniques to generate test data instead of just one technique.

2.1 Whiley

Whiley is a hybrid imperative and functional programming language developed by David Pearce [7]. The presentation of Whiley resembles an imperative language like Python whereas the core structure is functional and pure [7].

Functions in Whiley are pure therefore, an input will always result in the same output without any observable side effects [1]. Methods are impure so may have side effects on input parameters or other aspects of the program [1]. Functions, methods and other compound data types are passed by value meaning they are a copy of the original variable when they are passed in an argument to a function or method [7].

A key component in Whiley is the automatic verifying compiler which is used in conjunction with explicit specifications to verify programs are correct [7]. Specifications can be written as pre-conditions (`requires` clauses) and post-conditions (`ensures` clauses) in a function or method declaration [7]. Specifications can also be written as invariants using `where` clauses on user defined types (nominal type) or as an invariant on a loop [7].

To be able to execute a Whiley file, it must first be compiled into a WyIL file which is the bytecode form of Whiley in the Whiley Intermediate Language [7]. During compilation, the verifying compiler checks all specifications are met and checks for common errors such as division by zero [7]. Any errors or failed specifications are detected are reported to the developer to fix. Counter examples are also used within Whiley to notify how the specification can fail however, this functionality is limited to examples within a small range. The WyIL file can then be executed if it verifies successfully or compiled without verification.

2.2 QuickCheck

QuickCheck is a tool implemented by Koen Claessen and John Hughes [2] to test Haskell programs.

To check if a program is correct, QuickCheck uses property-based testing which uses conditions that will always hold true [2]. Users define the conditions as Haskell functions in terms of formal specifications to validate functions under test [2].

QuickCheck then generates a large number of test cases automatically using random testing [2]. Input values are randomly generated for each test (within a range) using generators that correspond to the input value's type [2]. QuickCheck contains generators for most of Haskell's predefined types and for functions [2]. User-defined types require the tester to specify their own custom generators with a bounded size if the type is recursive such as a binary tree. [2].

A problem with random testing is the distribution of test data. Ideally, the distribution should adhere to the distribution of actual data which is often uniformly distributed [2]. For example, executing tests for Listing 4 could be skewed towards test data with short lists. The distribution of data is controlled by the tester by creating a custom generator for the data type with weighted frequencies on the methods used to generate the data [2].

An example of using QuickCheck would be reversing a list.

Firstly, a property is defined in Listing 3 where the list `xs`, should be the same as reversing `xs` twice.

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

Listing 3: Property for reversing a list in QuickCheck

QuickCheck is then executed by importing the property and passing it into the interpreter as shown in Listing 4.

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

Listing 4: Executing tests to check a list is reversed

A large number of test cases is created where random input values are generated to check the user-defined property holds. QuickCheck reports various test statistics including the number of tests that have passed or failed [2].

2.3 Concolic Unit Testing Engine (CUTE)

Concolic Unit Testing Engine (CUTE) generates test inputs using a combination of symbolic and concrete execution [3]. The aim of CUTE is to provide input values to explore all feasible execution paths of a program and thus achieve high path coverage [3]. CUTE has been applied to C programs [3].

Random testing may generate inputs with the same behaviour leading to its redundancy [3]. Furthermore, the probability of selecting inputs that will detect errors using random testing is small [3]. Therefore, CUTE incrementally generates concrete inputs using symbolic execution to discover feasible paths [3].

Firstly, CUTE executes the program with arbitrary inputs using concrete and symbolic execution concurrently [3]. In concrete execution, the program is executed normally with the input values [3]. At the same time, symbolic execution is run through the same path [3]. Using the symbolic variables, constraints from branching expressions are discovered and stored [3]. The last constraint applied is negated so that the next test run will explore a different feasible path [3]. The constraint is solved to limit and determine the possible inputs to execute in the next test [3].

For example, we execute Listing 5 and create the random values `x = y = 1`. In concrete execution, `z` is set to 1 whereas in symbolic execution, `z` is set to `x * y`. At line 2, `x != 2` so the condition fails. As the program went through one path where `x != 2`, the condition is then negated and solved so the next test goes through a different path by setting `x` to 2 and `y` to 1. The test will then fail at line 4 as `x >=z`. Therefore, CUTE will then find values that solve the constraints `x == 2` and `x < z` such as `x = 2` and `y = 2`. Running this test will then exhibit the error on line 5.

Listing 5: Example C program

```
1 void foo(int x, int y) {
2     int z = x * y;
3     if(x == 2){
4         if(x < z){
5             ERROR;
6         }
7     }
8 }
```

2.4 Randoop

Randoop which stands for random tester for object-oriented programs, generates unit tests using feedback-directed random test generation [4], [5]. Method sequences from previous tests are used to help generate subsequent tests [4], [5]. As a result, a test suite is outputted with successful and unsuccessful tests [4], [5].

Randoop tests classes by executing a sequence of methods as a test [5]. Method sequences are created incrementally by randomly selecting method calls and using arguments from previous sequences [5]. The sequence is then executed and checked against contracts [4]. Contracts are built into Randoop or optionally defined by the user [4]. If a sequence breaks a contract, then the test is outputted as a contract-violating test [4]. If sequence was successful, then the test is outputted as a regression test [4]. Successful sequences that are not redundant and can be extended are used in subsequent tests [4].

For example, executing Randoop on the Java program in Listing 6 will produce several tests based on different method sequences. One successful list of method sequences could be Listing 7, which can be re-used in subsequent sequences. It is outputted into a regression test as shown in Listing 8. By extending the method sequences, another list of method sequences is created such as Listing 9. This list of method sequences throws an error as the `equals()` method is incorrect as `a1` is not equal to `b1` and thus, is a contract-violating test that will be outputted.

Listing 6: Example Java class

```
1  public class A {
2      public A(){ }
3
4      @Override
5      public boolean equals(Object obj){
6          return true;
7      }
8  }
9
10 public class B{
11     public B(){ }
12 }
```

Listing 7: Successful method sequence for testing Listing 6

```
1 A a1 = new A();
2 a1.equals(a1);
```

Listing 8: Test output for the successful method sequence in Listing 7

```
1 @Test
2 public void test1() {
3     A a1 = new A();
4     assertTrue(a1.equals(a1));
5 }
```

Listing 9: Contract violating method sequence for testing Listing 6

```
1 A a1 = new A();
2 a1.equals(a1);
3 B b1 = new B();
4 a1.equals(b1);
```

3 Work Completed

An implementation of the tool has been created with generation of core and more complex data types. The tool has also been extended to use integer range analysis to constrain input values generated for better performance. Unit tests have also been written to check the tool generates the values expected.

The tool can be executed via the command line with several arguments. To use the tool, a user must pass in the path of a WyIL file and specify the test case generation technique to use, number of tests they wish to generate and ranges for generating integer values. Ranges are used to limit the integers generated between an inclusive, minimum number and an exclusive, upper number.

Whiley's interpreter is used to validate inputs and outputs from tests. Valid inputs are based on verifying pre-conditions and constraints on the input type used. Only valid inputs are used in tests. A successful test is checked by verifying outputs with the function's post-conditions and constraints on the outputs' type.

Currently, only functions within the program with a post-condition can be tested as they can be verified.

An example of using the tool would be to execute a file `abs.wyil` which contains the following code in Listing 10. To execute the tool, we run in the command line:

```
java QuickCheck abs.wyil exhaustive 10 -5 5
```

This tells QuickCheck for Whiley to generate 10 tests using exhaustive test generation with an integer range between -5 and 5. When the tool is executed, it outputs several statistics for the user as shown in Listing 11.

Listing 10: Whiley program for an incorrect implementation of the abs function

```
1 function abs(int x) -> (int r)
2 ensures r >= 0
3 ensures r == x || r == -x:
4    return x
```

Listing 11: Results of executing the tool on Listing 10

```
Failed Input: [-5] Output: [-5]
Failed Input: [-4] Output: [-4]
Failed Input: [-3] Output: [-3]
Failed Input: [-2] Output: [-2]
Failed Input: [-1] Output: [-1]
Failed: 5 passed (50.00 %), 5 failed (50.00 %), ran 10 tests
```

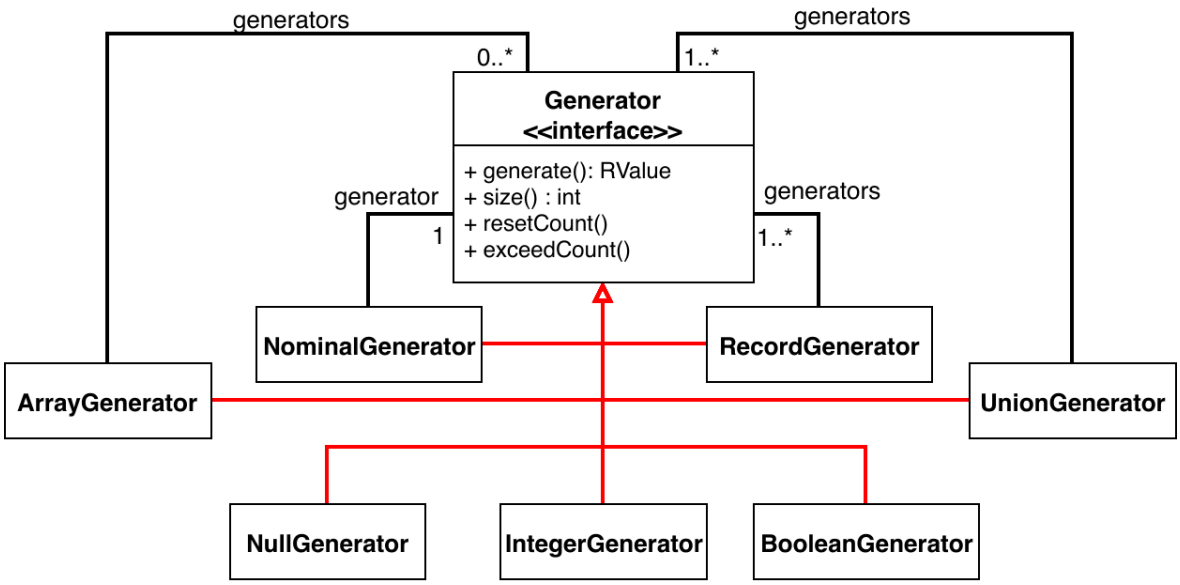
3.1 Test Case Generation Techniques

QuickCheck for Whiley currently employs random test-case and exhaustive test case generation to generate test cases:

3.1.1 Random Test Case Generation

A fixed number of tests determined by the user is randomly generated. Previous inputs are not considered which may lead to the same inputs being generated. Knuth's Algorithm S from [8] was considered to generate unique inputs to solve this problem but has not been implemented yet.

Figure 1: Class diagram of Generators in QuickCheck for Whiley



3.1.2 Exhaustive Test Case Generation

A fixed number of tests determined by the user are exhaustively generated, where ranges are used to limit integers and the size of arrays during generation. This is so that a feasible number of combinations can be generated. If the number of tests exceed the number of input combinations, the tool will cycle around and generate inputs exhaustively from the starting input. Integers are generated starting from the lower limit. Arrays are generated starting from the smallest possible array size, which is usually an empty array.

3.2 Data types generated

Different generators are created for the different types defined in Whiley. Figure 1 illustrates the structure of the generators used. Each generator generates a value. It also has a size for the number of input combinations and `resetCount()` and `exceedCount()` methods for monitoring input combinations generated for test-case generation.

Null Generate a `null` value.

Boolean Generates a Boolean which is either `true` or `false`.

Integer Generates an integer between a bounded range. This range may be modified based on constraints applied to the integer.

Array Generates an array with a specific element type between a bounded size range. The size of arrays is currently limited to a maximum size of three elements due to the performance cost of generating larger arrays.

Nominal A nominal represents a user-defined type by redefining a type with a different name [1]. Therefore, a generator for a nominal wraps a generator for a different type to generate a value. A nominal can also have a type constraint/invariant applied to it [1].

Record "A record type describes the set of all compound values made from one or more fields, each of which has a unique name and a corresponding type [1]." A closed record

contains a fixed number of fields whereas an open record can have any number of fields with different types. A closed record is generated by generating values that correspond to each field. Generators corresponding to each field is required as each field may have different types. Generation for open records has not been implemented as there can be an arbitrary number of fields added to a record. This could be a future extension to the tool.

Union A union is made up of multiple types where the value can correspond to any one of the types declared. For example, the type `bool|int` can hold either a boolean or an integer value. A union is generated by generating a value from one of the declared types. A generator corresponding to each type is required.

Values for the union type could be skewed towards generating values for only type. It was important that the different types for each union are generated fairly. Therefore, the union generator was implemented so that a value from a different type would be generated each time by iterating through the generators within the union generator.

3.3 Integer Range Analysis

Generating and checking for invalid inputs is costly in terms of performance. Therefore, it would be beneficial to reduce the number of invalid inputs generated to improve the performance of the tool. One method to remove invalid inputs for integers is to shrink the ranges used during generation thus no longer generate invalid inputs.

For example, define the nominal type, `type nat is (int x) where x >= 0` and execute the tool for an integer range between -5 and 5. Invalid inputs are numbers below 0. An integer generator for this input will only generate numbers between 0 and 5, removing the need to generate then discard invalid inputs below 0.

This is implemented by evaluating the constraints within a nominal type to discover integer range generated from the constraint. Existing code from [9] was used to create the integer ranges which is based off [10]. The discovered integer range is passed down in the nominal generator until it reaches the relevant integer or array generator where it is intersected with an existing range. If the ranges are invalid, i.e. the lower range is greater than the upper range then an error is thrown.

Integer ranges are only applied to constraints that follow the format: `x op c` where `x` is the named variable in the nominal type, `op` is an operator out of the set {>, <, >=, <=, &&, ||, ==} and `c` is a constant number.

3.4 Evaluating the tool

The tool has been evaluated by executing a variety of tests notably, the test suite for the Whiley Compiler containing tests for valid inputs and outputs and tests for invalid inputs and outputs [11]. Valid tests should always pass for valid inputs whereas invalid tests should always fail for some valid input. A successful implementation of the tool should pass all tests.

To evaluate the Whiley Compiler test suite, the tool is executed twice using exhaustive test generation for each test. The first execution is with integer ranges between -5 (inclusive) and 0 (exclusive) whereas the second execution is for integer ranges between 0 (inclusive) and 5 (exclusive). Currently, the tool passes 80.69% (4 s.f) (422 out of 523 tests) of the valid tests and 93.49% (4 s.f.) (316 out of 338 tests) of the invalid tests [12]. Some of the tests may falsely pass by the tool due to the small domain of integer ranges used. Therefore, some tweaking will be required to determine a suitable integer range across all tests.

4 Future Plan

A working implementation has been completed halfway through the project. However, there are still several components that need to be completed. A timeline of these components is shown in Table 1

Output	**Estimated Time**	**Start Date**	**Complete by**
Optimise functions/method calls when testing a specific function/method to improve the performance of the tool.	3 weeks	11/6/18	2/7/18 (Exam period)
Produce slides for presentation of preliminary report	1 week	2/7/18	9/7/18 (Break)
Add mutation testing by mutating Whiley files and comparing results from the mutated files with the original file.	3 weeks	9/7/18	30/7/18 (Week 3)
Produce draft of final report	6 weeks	30/7/18	15/9/18 (Week 7)
Evaluate the automatic test-case generator using the different test-case generation techniques against a test suite.	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)

Table 1: Timeline

5 Request for Feedback

Feedback about using the Whiley Compiler test suite for evaluating the tool as discussed in Subsection 3.4 would be appreciated.

When evaluating the tool, I would like to know if the following method would be suitable. I could execute the tool multiple times on the same test suite using different test techniques and compare the results based on speed of execution, number of tests passed and number of errors detected. Alternative suggestions on how to evaluate the tool would be helpful.

References

[1] D. J. Pearce, "The Whiley Language Specification," 2014.

[2] K. Claessen and J. Hughes, "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs," *SIGPLAN Not.*, vol. 46, pp. 53–64, May 2011.

[3] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," *SIGSOFT Softw. Eng. Notes*, vol. 30, pp. 263–272, Sept. 2005.

[4] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, (Washington, DC, USA), pp. 75–84, IEEE Computer Society, 2007.

[5] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for java," in *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA '07, (New York, NY, USA), pp. 815–816, ACM, 2007.

[6] G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, (New York, NY, USA), pp. 416–419, ACM, 2011.

[7] D. J. Pearce and L. Groves, "Whiley: A platform for research in software verification," in *Software Language Engineering* (M. Erwig, R. F. Paige, and E. Van Wyk, eds.), (Cham), pp. 238–248, Springer International Publishing, 2013.

[8] D. E. Knuth, *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*, pp. 142–143. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[9] D. J. Pearce, "Whiley2EmbeddedC/IntegerRange.java at master," January 2017.

[10] D. J. Pearce, "Integer Range Analysis for Whiley on Embedded Systems," in *Proceedings of the 2015 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, ISORCW '15, (Washington, DC, USA), pp. 26–33, IEEE Computer Society, 2015.

[11] D. J. Pearce, "WhileyCompiler/tests at master," January 2018.

[12] J. Chin, "Documentation/Whiley QC Result Statistics.docx - master," Mat 2018.

A Project Proposal

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 and 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 [1] 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 [1].

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 [1].

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 [2].

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 [3].

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.

Listing 1: Whiley program for the min function

```
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.

An example of a function for test-case generation would be for the `min` function introduced in Listing 1.

Possible input values automatically generated include:

- min(4, 4) would result in r = 4
- min(6, 7) would result in r = 7
- min(10, 5) would result in r = 10
- min(-2, 8) would result in r = 8

Several components developed for the Whiley language will need to be used for the automated test-case generator. The Whiley interpreter will be used for checking candidate values meet precondition and validating test outputs meet the postconditions. It will also be used for analysing the program structure such as the names of functions declared and their parameters. The Whiley Compiler will be used for generating input data for different types.

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 3)
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.

Bibliography

[1] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of haskell programs," *SIGPLAN Not.*, vol. 46, pp. 53–64, May 2011.

[2] J. Hughes, "Quickcheck testing for fun and profit," in *Proceedings of the 9th International Conference on Practical Aspects of Declarative Languages*, PADL'07, (Berlin, Heidelberg), pp. 1–32, Springer-Verlag, 2007.

[3] D. Pearce, "The Whiley Language Specification," 2014.