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 Final Report

Janice Chin

Supervisor: David Pearce

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

Abstract

Whiley is a programming language that verifies code using formal specifications to improve software quality. Whiley contains a verifying compiler which can identify common bugs. However, the compiler is limited in its ability to discover bugs and may take a long time to verify large programs. Testing is another approach to find bugs. However, testing Whiley programs requires users to manually write code which can be a valuable but time-consuming process. This report details an automatic test case generation tool developed for Whiley based on an existing tool, QuickCheck. QuickCheck for Whiley has been implemented for random and exhaustive test-case generation with different data types generated. Optimisations were made to the tool by implementing function optimisation and memoisation. The tool was evaluated using several test suites to analyse the performance and ability of the tool to test programs. Several bugs were discovered within the test suites and subsequently fixed.

Acknowledgments

I would like to thank my supervisor, Dr David Pearce for the opportunity to work on this project. He provided invaluable feedback, guidance and support throughout the project and made this an overall, positive experience.

Contents

Chapter 1

Introduction

Software quality has been a growing concern in software engineering due to the high cost of software failures. In 1999, software failures cost NASA $1.6 billion dollars [1]. A core aspect of software quality is the detection and elimination of bugs to improve the quality and reliability of software. Commonly, software engineers would manually write tests to discover bugs in their code [2]. However, it is impossible to write tests for every execution path of a program. A comprehensive report in 2002 noted that inadequate software testing costs $59.5 billion dollars in the U.S. alone [1]. Another solution to improve software quality is software verification which ensures that code meets the specifications defined for all possible inputs. It can automatically detect and eliminate specific errors in code and, thus prevent these errors from ever occurring.

To help eliminate errors in code, the Whiley programming language was developed to verify code using formal specifications [3]. 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. Whiley has been applied to a variety of cases such as its use as a teaching tool for the course *"SWEN224: Formal Foundations of Programming"* as well as executing Whiley code on embedded systems [4]. Whiley can also be translated into programs in C, Java and JavaScript.

Listing 1.1 illustrates a Whiley function `min()`, which finds the minimum value of two integers, `x` and `y` and returns the minimum as the integer, `r`. The function's postcondition is written using two ensures clauses which checks `r` is `x` if `x` is the minimum or `r` is `y` if `y` is the minimum. For example, executing `min(2, 3)` will return the integer, 2.

```
function min(int x, int y) -> (int r)
ensures x <= y ==> r == x
ensures y <= x ==> r == y:
if x <= y:
   return x
else:
   return y
```

Listing 1.1: Whiley program for the min function

Currently, the verifying compiler has limitations when evaluating complex pre- and post-conditions. For example in Listing 1.2, the post-condition is falsely identified as not holding by the verifying compiler even though the program does meet the post-condition. The verifying compiler is unable to verify the program meets its specification as it cannot reason about non-linear arithmetic. The verifying compiler is a work in progress, as it is difficult to design a system to check programs are correct for all possible inputs using formal specifications.

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

Listing 1.2: Whiley program for the square function

Software verification is still a developing field in research and thus, is limited in the errors it can eliminate. An easier alternative to detect bugs would be to write and execute tests for the program. Ideally, tests would cover all statements within a program by enumerating all possible inputs. However, writing and running tests can be tedious and costly. Testers must have an in depth understanding of a program and carefully select a set of test scenarios to adequately test a program [2]. The test scenarios then need to be written, executed and ideally evaluated against a performance measure such as code coverage [2]. It is also difficult to write tests for all possible execution paths and inputs 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.

A simple technique to automatically generate tests is random testing. Random testing is where data is randomly generated to test a program. An important tool in random testing is QuickCheck which was implemented in Haskell by Koen Claessen and John Hughes [5]. QuickCheck's popularity has resulted in its implementation in other programming languages including Erlang, Java and C [6]. Random testing has also been widely studied in the literature, leading to the creation of new variants and testing techniques including concolic testing [7], feedback-directed testing [8, 9], and mutation testing [10]. These techniques have different approaches to testing, ranging from validating written assertions to executing a series of functions.

This project aims to implement an automated test case generator for the Whiley programming language, based on the QuickCheck tool. Whiley would work well with the tool as test inputs and outputs can be checked against formal specifications using the verifying compiler. Furthermore, users would benefit from having both testing and verification for detecting bugs. As a result, the automatic test case generator will improve software quality and increase confidence in unverifiable code.

1.1 Contributions

The following contributions were made to this project:

1. A tool was developed in Java, called QuickCheck for Whiley (WyQC) to automatically test Whiley programs, using random or exhaustive testing.

2. Optimisations to the tool were made which include function call optimisation, memoisation and bounding using integer ranges.

3. Several experiments were conducted on various test suites to measure the performance.

4. Bugs were discovered within the test suites executed which were then recorded and remedied.

Chapter 2

Background

This chapter gives a brief overview of the Whiley programming language and explains several automatic test generation tools. Each tool uses different approaches in order to test programs in their targeted language.

2.1 Whiley

Whiley is a hybrid imperative and functional programming language developed by David Pearce [11]. The presentation of Whiley resembles an imperative language using indentation syntax similar to Python whereas the core structure is functional and pure [11].

Programs defined in Whiley use methods and/or functions. Methods are impure so may have side effects on input parameters or other aspects of the program [3]. Functions in Whiley are pure meaning an input will always result in the same output without any observable side effects [3]. Functions, methods and data types are passed by value meaning that a copy of the original value is passed as an argument to a function or method [11]. Multiple return values are also allowed for functions and methods [12].

Data types in Whiley include the common data types: integers, booleans, bytes and arrays. Other data types in Whiley include nominals, records, unions, references and functions as exemplified in Listing 2.1. A nominal creates an alias of an existing type i.e. type synonyms in Haskell. Records are similar to structures in C and references are similar to pointers in C. A union allows a variable to contain different types throughout a program through Whiley's flow typing system [12]. For example in line 6, the return type of `makeCell()` is a union therefore, either `cell` or `null` could be returned.

A key component in Whiley is the automatic verifying compiler which is used in conjunction with explicit specifications to verify programs are correct [11]. Specifications can be written as pre-conditions (`requires` clauses) and post-conditions (`ensures` clauses) in a function or method declaration [11]. Specifications can also be written as invariants using `where` clauses on user defined types (nominal type) or as an invariant on a loop [11].

To be able to execute a Whiley file, it must first be compiled into the Whiley Intermediate Language (WyIL) file which is the bytecode form of Whiley [11]. During compilation, the verifying compiler checks each function and method meets its specifications, independently of each other and checks for common errors such as division by zero [11, 12]. Any errors or failed specifications are detected and 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.

Currently, the verifying compiler cannot detect all possible errors. For example, Whiley

```
type nat is (int n) where n > 0 // Nominal
type cell is {int x, int y} // Record
type func_t is function(int) -> int // Function

// A function that returns a union, either a cell or null
method makeCell(&int startPtr, &int endPtr) -> cell|null:
   int start = *startPtr
   int end = *endPtr
   if start > 0 && end > 0:
      nat xPos = start
      nat yPos = end
      return {x: xPos, y: yPos}
   else:
      return null

function foo(func_t func) -> int:
   return func(1)
```

Listing 2.1: Using different types in a Whiley program.

```
function infiniteLoop():
   int i = 0
   int n = 10
   while i < n:
      i = i - 1
```

Listing 2.2: Whiley program that will run infinitely if `infiniteLoop()` is called.

cannot detect if loops terminate as shown in Listing 2.2. Therefore, a user could execute `infiniteLoop()` to cause the program to run indefinitely. Furthermore, the verifying compiler is unable to verify large programs as it takes too long to verify.

2.2 Random Testing with QuickCheck

QuickCheck is a tool implemented by Koen Claessen and John Hughes [5] to test Haskell programs. QuickCheck is widely used in the Haskell community and industry, testing large software including the Galois Connections' Cryptol compiler [13]. To check if a program is correct, QuickCheck uses property-based testing which uses conditions that will always hold true. Users define a property as a Haskell function which contains a condition as a formal specification. The condition contains function calls to one or more functions that are then validated under test. In comparison, properties in Whiley are written within the functions/methods as pre-conditions and post-conditions. Properties for individual functions/methods can be checked without having to write an external function/method.

QuickCheck generates a large number of test cases automatically with random testing. Each generator in QuickCheck is an instance of the type class `Arbitrary` where it generates a value based off a random number. Input values are then randomly generated for each test using generators that correspond to the input value's type. For numeric types, a generator generates a value within a defined range such as between -5 and 5 for integers as shown in Listing 2.3. For non-numeric types, a generator uniformly selects a value that can be generated for that type such as generating true or false for a boolean.

QuickCheck contains generators for most of Haskell's predefined types and for functions. User-defined types require the tester to specify their own custom generators. The author claims it is due to the difficulty of constructing a generator for a type without knowing the distribution of values required for the tests [5].

The distribution of test data is an inherent problem with random testing. Ideally, test data should adhere to the distribution of actual data which is often uniformly distributed. For example, executing tests for Listing 2.6 could be skewed towards generating short lists. Therefore, QuickCheck allows testers to control the distribution of data by creating a custom generator for the data type with weighted frequencies on the methods used to generate the data. Listing 2.4 shows a custom list generator where it is three times more likely for a generated list to be appended with another element than the list ending. On average, the list generated will have three elements.

```
instance Arbitrary Int where
    arbitrary = choose(−5, 5)
```

Listing 2.3: Integer generator for QuickCheck

```
instance Arbitrary a => Arbitrary [a] where
    arbitrary = frequency [(1, return []),
                           (3, liftM2 (:) arbitrary arbitrary) ]
```

Listing 2.4: Custom list generator with weighted frequencies for QuickCheck

An example of using QuickCheck would be reversing a list. Firstly, a property is defined in Listing 2.5 where the list xs, should be the same as reversing xs twice.

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

Listing 2.5: 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 2.6.

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

Listing 2.6: Executing tests to check a list is reversed

A large number of test cases are created where random input values are generated to check the user-defined property holds. After each test, the input values generated are discarded, allowing previous input values to be generated and tested again. Subsequently, QuickCheck reports various test statistics including the number of tests that have passed or failed.

2.3 Concolic Testing with Concolic Unit Testing Engine (CUTE)

Random testing may generate inputs that follow the same execution path and therefore, might be considered redundant to testing [7]. For example, inputs may never execute the

```
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 }
```

Listing 2.7: Example C program

if block in lines 4 and 5 in Listing 2.7. Furthermore, the probability of selecting inputs that will detect errors using random testing could be small [7].

Concolic Unit Testing Engine (CUTE) addresses these problems in random testing by generating test inputs using concolic testing, which is a combination of concrete and symbolic execution [7]. Concrete execution is where a program is executed using concrete input values. For example in Listing 2.7, the function `foo` could be executed with the concrete values 5 for x and 10 for y. Symbolic execution is where a program is executed using named symbols instead of actual values. CUTE aims to provide input values to explore all feasible execution paths of a program and thus achieve high path coverage. This is by incrementally generating concrete input values using symbolic execution to discover feasible paths. CUTE has been applied to C programs.

Firstly, CUTE executes the program with randomly generated inputs using concrete and symbolic execution concurrently. In concrete execution, the program is executed normally with the input values. At the same time, symbolic execution is run through the same path with symbolic variables to discover and store constraints from branching expressions. The test is successful if no errors occurred during execution. Preparation for the next test run involves negating one of the discovered constraints so that a different path is explored. The negated constraint is solved to determine the possible input values to execute in the next test. If the constraint cannot be solved, then the constraint is simplified by replacing any variables with the concrete value used in the current test. The next test is then executed with the new input values that meet the negated constraint.

For example, we execute Listing 2.7 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 the path where `x != 2`, a different path will be taken in the next test. This is determined by negating and solving the condition by setting `x` to 2 and leaving `y` as 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.

CUTE discovers errors in a program by executing paths instead of checking for program correctness. Therefore, a program that has no errors during execution may be invalid as it does not follow its specifications. Furthermore, CUTE is limited by the constraint solver's ability to solve constraints, which follows a similar issue to the verifying compiler in Whiley.

2.4 Feedback-directed Random Test Generation with Randoop

Randoop which stands for random tester for object-oriented programs, generates unit tests using feedback-directed random test generation [8, 9]. Feedback-directed random test generation involves randomly selecting methods and using method values from executing previous successful tests in subsequent tests [9]. As a result, a test suite is outputted with successful and unsuccessful tests [8, 9].

A problem with automatic test generation in Java is the concept of a test oracle: how to determine whether a test is successful or not. Without a test oracle, a user may not know if their program is exhibiting the correct behaviour when tested. In comparison, we can define a test oracle in Whiley using pre- and post-conditions. Creating a test in Java also involves a lot of overhead work by setting up objects and calling appropriate methods before creating the assert statements to check methods under test. Furthermore, each method call can have various side effects as methods are impure. Therefore, the expected behaviour of a method is dependent on where and when a method is called, making it difficult to test a program is correct in respect to their assertions.

Randoop specifies a test oracle for Java in terms of contracts. Contracts act as invariants where they are checked to hold before and after a method call. A contract takes the current state of the system as input and returns whether the system satisfies or violates the contract. Contracts are built into Randoop or optionally defined by the user. Several of Randoop's built-in contracts include checking two equivalent objects, `o.equals(o)` returns true and calling `toString()` does not throw an exception [8]. Creating a contract in Randoop requires creating a new class that implements an interface [9]. Contracts are also used to create assertions after each method call to verify if a test was successful [8]. If a test failed, then the test is outputted as a contract-violating test [8]. Contract-violating tests may indicate a potential problem in the code. However, the failed test might be valid in some cases. For example, an exception could exhibit the correct behaviour as it is always caught in a try-catch block. If a test was successful, then the test is outputted as a regression test [8]. These regression tests can then be reused to test future or different implementations of the program [8].

Testing in Java requires the construction of an object from a class's constructor. Each object can contain an arbitrary number of fields, constructors and methods that can be invoked. To account for the object oriented nature of Java, Randoop creates a unit test incrementally by randomly generating a sequence of methods and assertions [8]. Method sequences consist of a series of randomly selected public methods and valid method sequences from previous tests [8]. A new method sequence is created by first selecting a random public method from a list of classes. The public method requires suitable input arguments to be able to execute it. If the input argument is a primitive type, a random value is selected from a fixed set of primitive values. If the input argument is an object, a random value is selected either from the current method sequence, from a previous method sequence or `null`. Using a value from a previous method sequence will add the method sequence to the current method sequence. Building upon previous method sequences helps to develop new method sequences to test since Java has restrictions on when methods can be called. Values from previous method sequences are only used if they are not filtered out by user defined filters or Randoop's inbuilt filters. This prunes the search space by removing potentially redundant and illegal inputs from execution [8].

An example of executing Randoop, is on the Java program in Listing 2.8 where several tests are produced based on different method sequences. Firstly, we have a valid method sequence in Listing 2.9 that creates a value `a1` to use in subsequent method sequences. Extending this sequence we can generate a successful method sequence in Listing 2.10 which uses `a1` from Listing 2.9 to create `b1`. It is outputted into a regression test as shown in Listing 2.11. Another method sequence could be Listing 2.12, which also uses `a1` from Listing 2.9. It also creates its own value `a2` to be able to create `b1`. This method sequence throws an error on the `equals()` method as it is incorrect. `b1` should be equal to itself therefore, this sequence is a contract-violating test that will be outputted.

```java
public class A {
   public int val;

   public A(int val){ this.val = val; }
}

public class B {
   public int x;
   public int y;

   public B(int x, int y){ this.x = x; this.y = y; }

   @Override
   public boolean equals(Object obj){
      if(this.x == obj.y && this.y == obj.x){
         return true;
      }
      return false;
   }
}
```

Listing 2.8: Example Java program

```java
A a1 = new A(1);
```

Listing 2.9: A valid method sequence that can be reused

```java
A a1 = new A(1);
B b1 = new B(a1.val, 1);
b1.equals(b1)
```

Listing 2.10: A successful method sequence, for testing Listing 2.8

```java
@Test
public void test1() {
   A a1 = new A(1);
   B b1 = new B(a1.val, 1);
   assertTrue(b1.equals(b1));
}
```

Listing 2.11: Regression test outputted for the successful method sequence in Listing 2.10

```java
A a1 = new A(1);
A a2 = new A(2);
B b1 = new B(a1.val, a2.val);
b1.equals(b1);
```

Listing 2.12: Contract violating method sequence for testing Listing 2.8

Chapter 3

Design

This chapter explains how QuickCheck for Whiley tests Whiley programs, the test techniques used for generating tests and the generators created for generating different data types.

3.1 Architecture

QuickCheck for Whiley (WyQC) involves several key steps to automatically test a Whiley program. An overview of the process is shown in Figure 3.1 which uses the Whiley program `foo.whiley` in Listing 3.1 as an example. Listing 3.1 will be used as an example in the following steps.

Step 1. Firstly, the user must compile a Whiley program into the WyIL format using the Whiley Compiler.

For example, the Whiley program `foo.whiley` will be compiled into `foo.wyil`.

Step 2. The tool reads the WyIL file and discovers functions and methods to test. Each function and method from top down is tested by the number of tests set by the user. A test case generator is created for the function/method using the test case generation technique (random or exhaustive) set by the user. Within the test case generator are generators corresponding to each argument in the function/method.

For example, the function `foo` is discovered from `foo.wyil`. A random test case generator is created for the function `foo`.

Step 3. The test case generator is used to generate inputs for testing.

Within the test case generator for `foo` is an array generator which generates integer arrays for the argument `int[] arr`. Possible inputs that could be generated include `[1,2,3]`, `[1,1]` or `[1,2]`.

Step 4. Whiley's interpreter filters the generated inputs to check they are valid. Valid inputs are those that satisfy the functions/methods pre-conditions and the constraints on the input types used. Only valid inputs are executed in functions/methods under test.

The generated inputs `[1,2,3]`, `[1,1]` and `[1,2]` are filtered using the function's pre-condition (in line 2 of Listing 3.1) stating that the input array must be less than a length of 3. Therefore, the input `[1,2,3]` is discarded from testing.

Step 5. The function/method is then tested by passing in the validated inputs as arguments.

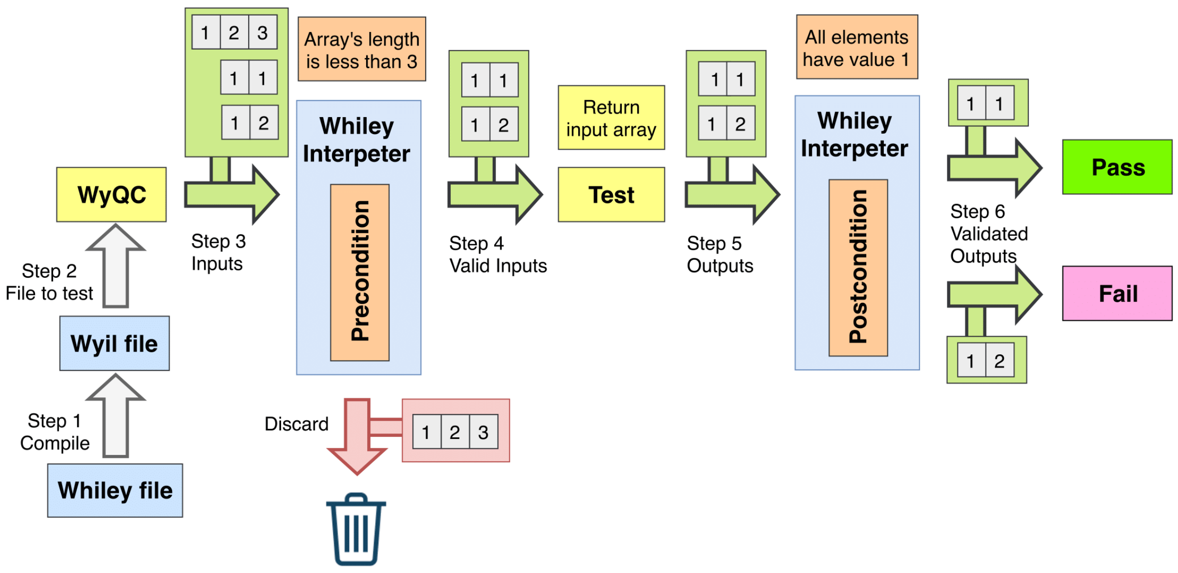


Figure 3.1: QuickCheck for Whiley process using `foo.whiley` from Listing 3.1

To test `foo`, the input `[1,1]` is passed in and executed, returning `[1,1]`. Similarly, executing with the input `[1,2]` returns the input `[1,2]`.

Step 6. Return values from the test are then validated using Whiley's interpreter.

After executing `foo`, the output is checked with the function's postcondition (in line 3 of Listing 3.1) stating that every element in the array must be the value 1.

Step 7. A successful test is when the return values meet the function's post-conditions and constraints on the outputs' type.

Executing `foo` with the input `[1,1]` results in a successful test as the output `[1,1]` meets the function's postcondition. In comparison, execution with the input `[1,2]` will result in a failed test as the output `[1,2]` does not meet the function's postcondition.

```
1 function foo(int[] arr) -> (int[] r)
2 requires |arr| < 3
3 ensures all {i in 0..|r| | r[i] == 1}:
4    return arr
```
Listing 3.1: The Whiley program, `foo.whiley` tested in Figure 3.1

3.1.1 Data types generated

Different generators are created for the different types defined in Whiley. Each generator generates a value for their corresponding type. Figure 3.2 illustrates the structure of the generators used.

Primitive Types

Generators for primitive data types directly generate a value associated to the type.

Null Generate a `null` value.

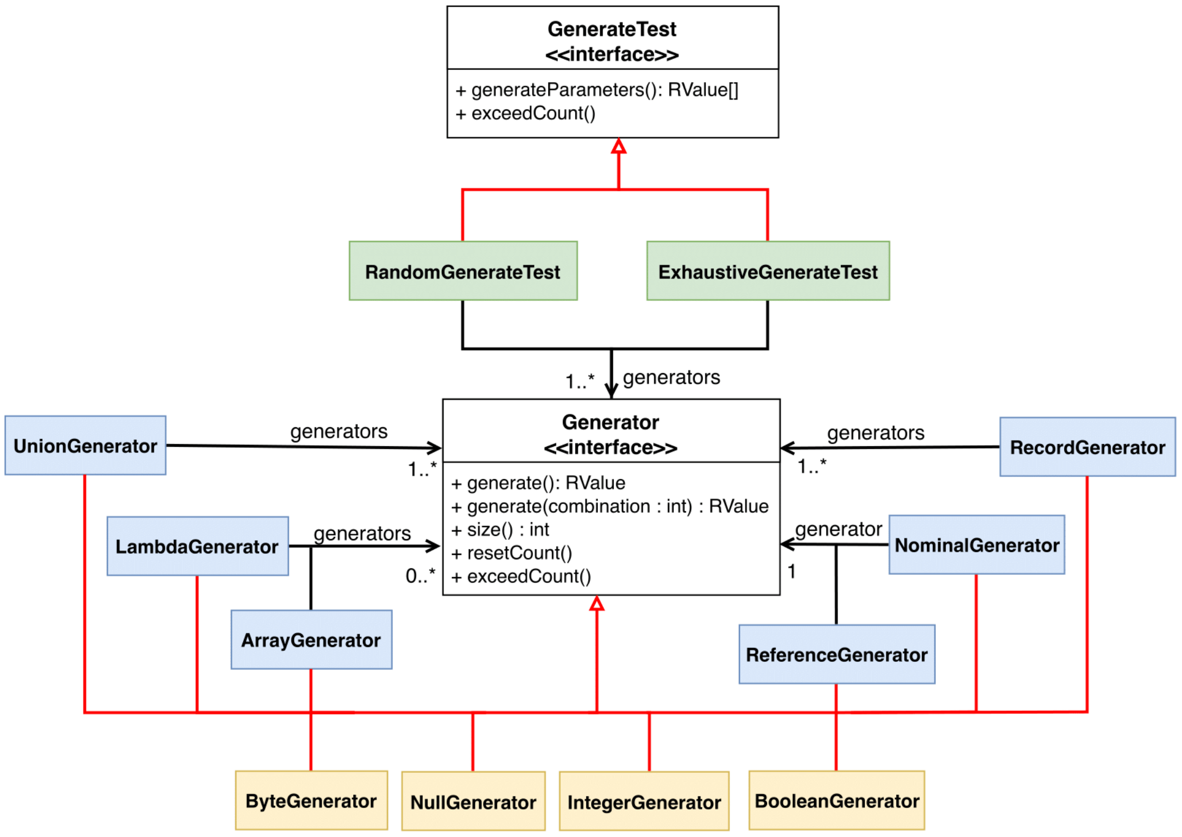


Figure 3.2: Class diagram of Generators in QuickCheck for Whiley

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 type constraints applied to the integer.

Byte Generates a byte, which is a sequence of eight bits followed by the letter `b` [3]. It is representative of a decimal number between 0 and 255.

Complex Types

Generators for complex data types require another generator to be able to generate a value for the type.

Array Generates an array with a specific element type between a bounded size range. Multi-dimensional arrays such as two-dimensional arrays can also be generated.

Nominal A nominal represents a user-defined type by redefining a type with a different name [3]. 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 applied to it [3].

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 [3].” A closed record contains a fixed number of fields and is generated by generating values that correspond to each field. Generators corresponding to each field are required as each field may have different types.

Whiley also allows open records, which are similar to a closed record. Open records can have additional fields that are unknown as they are not part of the record's type definition. An example of an open record is shown in Listing 3.2.

```
type ORecord is {int num, ...}

ORecords a = {num: 1}
ORecord b = {num: 2, anotherField: true}
```

Listing 3.2: Whiley program with open records

Open records are not generated by the tool. See Section 6.1 for a discussion on the work required to implement this.

Union A union is made up of multiple types where the union's value corresponds 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 its declared types. A generator corresponding to each type is required in the union generator. Recursive structures using unions have also been implemented, for example:

```
type list is {int value, list next} | null
```

Reference A reference is an address towards a cell that contains a value of a particular type. References generated point to a unique cell. A limitation is that references cannot alias the same cell as discussed in Subsection 4.2.3.

Lambda Function/Method Whiley allows functions/methods to be passed as an argument into another function or method. A generated lambda will return a tuple of values which correspond to the return types including their type constraints if they exist. The returned values are implemented in a similar manner to how records are generated. It does not manipulate or account for the value inputted into the lambda.

3.1.2 Test Case Generation Techniques

QuickCheck for Whiley currently employs random and exhaustive test case generation to generate test cases. A test case in QuickCheck for Whiley tests one function or method using a set of generated input values by calling `generateParameters()`. Each test case generator requires a list of generators that correspond to each input type as shown in Figure 3.2. Both random and exhaustive test case generation are supported to allow users to select whether they wish to randomly or exhaustively generate inputs. Random test case generation explores more of the search space by generating a range of input values compared to exhaustive test case generation. Exhaustive test case generation iterates through all possible input values allowing input values to be consistently generated.

Random Test Case Generation

Input values for a function or method can be randomly generated. To randomly generate a value for a particular type, the type is quantified into a numeric range of valid values. Initially, a random integer is pseudorandomly created and used to select a value out of the range of valid values for the generator. However, a problem was encountered when generating input values. Previous combinations of inputs were not recorded which may lead to the same test being executed as the same inputs were used.

Therefore, Knuth's Algorithm S from [14] was implemented so that every test has a unique combination of input values. Every combination of input values is represented as an integer. The algorithm requires the number of tests to be specified to randomly and uniformly select integers. Each integer is then used to get the input values for the test. Creating a map from every integer to their corresponding input values is expensive and impractical. Instead, the integers are passed into generators where the corresponding input value is then generated. Using this algorithm should improve performance as redundant inputs are no longer generated. In comparison with naive random generation, generating unique inputs would require recording the inputs and discarding inputs that were the same. Finding a unique input becomes more expensive as the search space becomes smaller.

To generate a random set of input values, the number of possible input combinations was calculated by getting the number of combinations, `size()` for each generator. A set of input combination numbers is then pseudorandomly generated using Knuth's Algorithm S. Input values are then generated for each combination number by calling `generate(int combination)` on each generator. For example, the `bothPositive` function in Listing 3.3 there are four possible input combinations as shown in Table 3.1. Calling `generate(2)` on the random test case generator would generate the second combination, which is the input value `(true, false)`.

Combination Number	Input Values
1	`(true, true)`
2	`(true, false)`
3	`(false, true)`
4	`(false, false)`

Table 3.1: All possible input values for Listing 3.3

```
function bothPositive(bool a, bool b):
    return a && b
```

Listing 3.3: Whiley program with two boolean arguments.

Exhaustive Test Case Generation

Input values for a function or method can be exhaustively generated. Exhaustive test generation uses the `size()` and `exceedCount()` methods to generate the correct value as shown in Figure 3.2. The number of unique values that can be generated is determined by using the `size()` method. `exceedCount()` is used to check if the number of tests exceed the number of unique values that can be generated. This may occur when the number of tests set by the user is greater than the number of possible input values that can be generated. For example, testing a function with one boolean argument would only have two possible input values `true` and `false`. If all input combinations have been tested, the tool will stop generating tests.

Integers are generated by starting from the lower limit. By default, the generation of integer values starts at `-5` and continues up to the limit `5`. These integer limits can be configured by the user when they execute the tool. Arrays are generated by starting from the smallest possible array size, which is usually an empty array.

Chapter 4

Implementation

This chapter explains the implementation details in QuickCheck for Whiley which includes decisions for generating data types and performance optimisations made.

4.1 Overview

QuickCheck for Whiley has been implemented based on the design in Section 3.1. Different generators have been implemented which correspond to primitive and complex data types. The implementation of these generators involve several design decisions about limiting values generated, fairness in unions and aliasing in references discussed in Section 4.2. The tool has also been extended with various performance improvements which include analysing type constraints, optimising functions and memoisation discussed in Section 4.3. The user can decide whether they wish to activate these features and other settings for the tool using the command line. Unit tests have also been written to check the tool generates the values expected.

An example of using the tool is by testing the file `abs.wyil` which contains the Whiley program in Listing 4.1. To execute the tool, we can 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 case generation with an integer range between `-5` and `5`. After executing the tests, several statistics are outputted as shown in Listing 4.2. These statistics include the number and percentage of tests that passed, were skipped or failed and the execution time of testing the function/method. Failed test inputs, corresponding outputs and test failure causes are also displayed. In Listing 4.1, tests failed due to certain inputs not meeting the postcondition of `abs` as identified in Listing 4.2.

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

Listing 4.1: A Whiley program for an incorrect implementation of the `abs` function. This is compiled into the file, `abs.wyil`

```
Name of the function/method: abs
Failed Input: [-5]
Failed Output: [-5]
Postcondition failed java.lang.AssertionError
Failed Input: [-4]
Failed Output: [-4]
Postcondition failed java.lang.AssertionError
Failed Input: [-3]
Failed Output: [-3]
Postcondition failed java.lang.AssertionError
Failed Input: [-2]
Failed Output: [-2]
Postcondition failed java.lang.AssertionError
Failed Input: [-1]
Failed Output: [-1]
Postcondition failed java.lang.AssertionError
Tested all possible combinations
Failed: 5 passed (50.00 %), 5 failed (50.00 %),
0 skipped (0.00 %), ran 10 tests
Execution time: 156 milliseconds
Some tests failed.
```

Listing 4.2: Results of executing the tool on Listing 4.1

4.2 Generation Basics

4.2.1 Data Types — Limiting values generated

Some types can have large ranges, specifically integers, arrays and recursive types. These types can negatively impact the time taken for the tool to generate values as the number of combinations significantly increase and more generators are required. Therefore, ranges are used to bound integers, arrays and recursive types so that a feasible number of values are generated.

- Integers are bounded using fixed lower and upper bounds to reduce the number of possible combinations. As integers are a core data type within Whiley, the limits used for integers can be configured as shown previously when executing the tool in the command line.
- Arrays have been implemented so that each element in an array has a generator for the array's type. For example, an integer array with two elements would have two integer generators. Therefore, the length of an array is currently limited to a maximum length of three elements to reduce the number of generators created. Subsequently, the performance cost of generating larger arrays is reduced. A setting in the command line for configuring the length of an array has not been implemented.
- Recursive structures have been implemented by recursively creating generators. For example, we have the following recursive type, `Node` which creates a recursive record following a linked list. A union generator is created that can generate either null or a record as illustrated in Figure 4.1.

```
type Node is null | {int value, Node next}
```

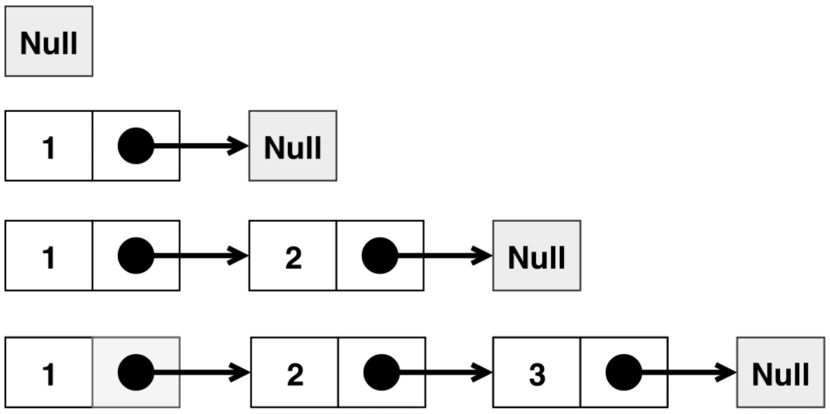


Figure 4.1: Recursive records

Recursive structures can be infinitely long, so their maximum depth must be bounded. Due to the use of Java's runtime stack, the maximum depth is set to three levels to prevent the tool from crashing due to a StackOverflow.

4.2.2 Union Types — Fairness

Generating a union type involves generating a value that matches any of the types in the union. For example, in Listing 4.3 we can generate either a boolean or an integer value.

```
type Val is bool|int
```

Listing 4.3: A union type that can hold a boolean or integer.

A union generator is created that contains both a boolean and an integer generator. When generating values exhaustively, one approach was to generate all values for each generator in turn. Therefore, a generator for `Val` would generate all possible integer values first, then generate all possible boolean values.

A problem with this approach was that values generated could be skewed towards generating values for only one type. Consequently, one type in a union may not be generated and thus not tested. To solve this problem, different types in a union type are generated equally. 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.

In Listing 4.3, `Val` can hold either a boolean or an integer value. Booleans only have two possible values, true or false whereas integers can have a larger range of values. Therefore, when `generate()` is called on the union generator for the first time, a boolean is returned. For the next two `generate()` calls, an integer would be returned, then a boolean. All subsequent `generate()` calls would then return an integer as there are no other possible boolean values. This ensures that both boolean and integer values are generated and thus tested.

4.2.3 References — Aliasing

References in the Whiley programming language can be aliased which means that references with different names can access the same memory cell. Changing the value in a memory cell would affect all named references to the cell. For example, we could have a function `swap` that swaps the boolean value of two cells as shown in Listing 4.4. Figure 4.2 shows all six possible input combinations as `x` and `y` can point to different cells or the same cell. If `x` and `y` point to the same cell, changing a value in `x` would affect `y` and vice versa.

```
method swap(&bool x, &bool y):
    bool temp = *x
    *x = *y
    *y = temp
```

Listing 4.4: Swap values of two boolean referenced variables.

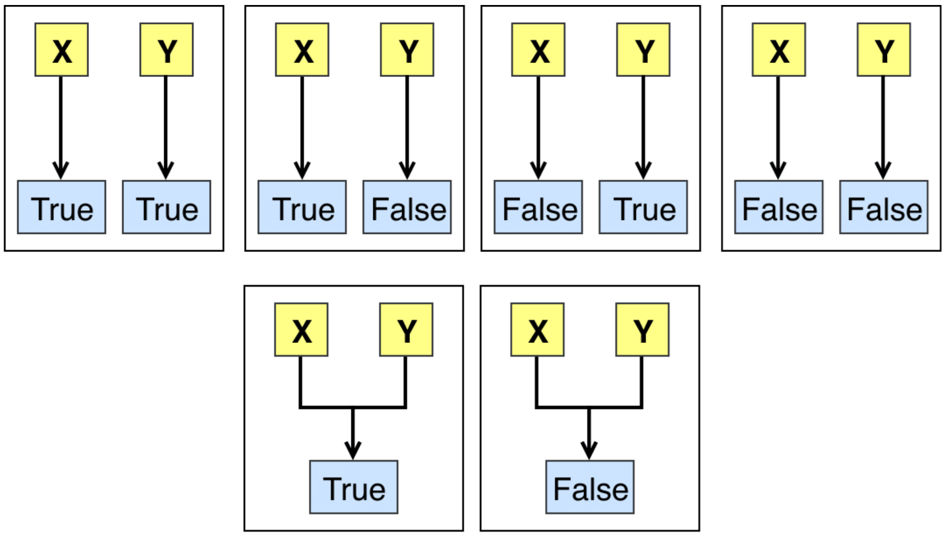


Figure 4.2: Input combinations when calling `swap` for `x` and `y`.

A limitation in the tool is that references generated cannot point to the same memory cell. This means that the tool cannot generate the last two inputs in Figure 4.2. Therefore, obscure bugs involving references pointing to the same cell would not be discovered. This limitation is due to the design of the generators in the tool, as each generator will generate values for its type. With aliasing, all values generated would need to be tracked across all generators, as each value has its own memory cell. Furthermore, the tool generates input values by using a combination number out of a finite number of combinations. With aliasing, the number of possible combinations is increased by an arbitrary amount as a reference could point to any number of existing memory cells. Solving this problem would require redefining how generators work within the tool.

4.3 Optimising Performance

4.3.1 Type Constraint Analysis

Nominal types can have type constraints applied to them. When a nominal generator generates a value, it must first validate the value using the type constraints applied. If the value is invalid, a new value is generated. This process continues until a valid value is generated or an error is thrown as no possible values can be generated. An alternative design was to generate and discard the generated value if it did not meet its type constraints when checking the precondition. Checking type constraints in the nominal generator ensures that tests are not discarded due to invalid values generated.

Generating and checking for values using the type constraint is costly in terms of performance. Therefore, it would be beneficial to reduce the number of invalid values generated to improve the performance of the tool. One method to remove invalid values is to shrink the ranges used for limiting values as discussed in Subsection 4.2.1, thus preventing the generation of invalid values. For example, in Listing 4.5 we have the nominal `nat` which has the type constraint `x >= 0` to ensure it is a positive integer.

```
type nat is (int x) where x >= 0
```

Listing 4.5: A nominal with a type constraint

A nominal generator for `nat` could generate negative integers which would then have to be discarded by the tool as they do not match the type constraint. Ideally, the generator would not generate any values below 0 as they would be invalid for the type constraint. By limiting the range of integers that can be generated for `nat` we can ensure that integers generated are always greater than or equal to 0.

This is implemented by evaluating the type constraints for a nominal type. Existing code from [15] from the work in [16] was used to create the integer ranges. Only type constraints of the following format are evaluated: `x op c`.

x Named variable in the nominal type.

op An operator out of the set {>, <, >=, <=, ==, !=}

c A constant number, set as the upper or lower limit of a integer range.

Each constraint is evaluated to determine if an integer range can be created. The is by evaluating the expressions on the left and right hand side of the constraint. If the expression contains a constant then the expression is executed to get the concrete value. Otherwise the expression is checked to ensure it is a named variable in the nominal. An integer range is created by using the operator and the concrete value discovered. The integer range is then applied to the child generators in the nominal generator to constrain the values they can generate by intersecting the range with the child generator's range. For example, the type constraint on `nat` is `x >= 0` which would have an integer range created between 0 and infinity. Within the nominal generator for `nat`, would be an integer generator which has an integer range set by the user such as between -5 and 5. The discovered integer range, `[0,infinity]` is then intersected with the integer generator, `[-5,5]` to get a new range from 0 to 5. This ensures that only valid integer values would be generated. An error is thrown if the lower range is greater than the upper range as no possible integers can be generated.

More complex type constraints are not evaluated due to a number of issues. Firstly, constraints with multiple parameters such as `x < y`, cannot be evaluated as the integer range would be dependent on another value that is generated. Another constraint that is difficult to evaluate is when multiple constants are applied to different sides of a constraint such as `x - 2 > 10`. This requires reversing operations so that all constants are evaluated on one side of the constraint. Evaluating `x - 2 > 10` would require reversing the `-2` to evaluate `x > 10 + 2` instead.

Currently, integer ranges are only applied to constraints on nominal types. However, this could be extended to arguments using a function or method's precondition.

4.3.2 Function Optimisation

When executing a program, functions and methods often call other functions and methods. For example, we have in Listing 4.6 the `main` method which calls the function `sumPositive` to check if the sum of two numbers is positive. Calling and executing a function or method can be expensive as they can contain loops or recursive calls. Testing `main` involves executing `sumPositive` which has a cost of O(n) as it has to iterate through a loop `n` times.

Therefore, it would be beneficial to skip execution by generating the return value of the function or method. In this example, we optimise `sumPositive` when testing `main`.

```
1  function sumPositive(int x, int y) -> (bool positive)
2  requires x >= 0
3  ensures positive <==> x + y > 0:
4     int i = 0
5     while i < x:
6        y = y + 1
7        i = i + 1
8     return y
9
10 method main():
11    sumPositive(10, 5)
```

Listing 4.6: Whiley program where the function `sumPositive` is called.

This has been implemented by using random test case generation to generate the return values. When a function/method is called, random test case generation is used to generate the return values of the function/method instead of executing the function/method. Return values are then validated using the Whiley interpreter against the postcondition of the function/method and the type invariants. In Listing 4.6, optimising `sumPositive` involves generating boolean values for its return parameter `positive` that is then validated using its postcondition in line 3. The random test generator has a fixed number of attempts at generating valid return values as the cost of generation can become more expensive than executing the function. The user can configure this in the command line.

Recursion A problem encountered when implementing function optimisation is when a function is recursively called within its postconditions. For example, Listing 4.7 shows the factorial function called recursively in the postcondition in line 3. Function optimisation could cause this function to infinitely run as it would attempt to optimise the function by generating values for the postcondition. As the postcondition is also used for checking the generated values are valid, then the function will have to call itself repeatedly in an infinite loop. Therefore, recursive function calls are not optimised by caching the name of the functions called to prevent this problem from occurring.

```
1 function factorial(int a) -> (int r)
2 requires a >= 1
3 ensures r == 1 || r == a * factorial(a-1):
4 if a == 1:
5    return 1
6 else:
7    return a * factorial(a - 1)
```

Listing 4.7: Whiley program for the factorial function

4.3.3 Memoisation

Memoisation is used to optimise the performance of the tool by caching inputs and outputs of functions. When a function is called with unique inputs, the function is executed normally. The inputs are then stored in a table with the corresponding return values of the function. When the function is called again with the same inputs, the cached output is returned instead of executing the function. Consequently, the time spent executing a test is reduced at the cost of using more memory.

To illustrate using Listing 4.7, firstly we execute `factorial(3)` and cache its output. When calling `factorial(4)` we would not need to execute `factorial(4-1)` at line 7. Instead, the cached value for `factorial(3)` is returned thus avoiding the need to execute `factorial(3)` again.

Another example is shown in Listing 4.8 where `summation(5)` is executed repeatedly in the function `bar()`. When testing `bar()` using memoisation, `summation(5)` is executed for the first test and its output is cached. For subsequent tests with different integers of `val` such as `val=1, val=2`, the cached value for `summation(5)` is returned instead. Therefore, `summation(5)` is executed only once instead of for every test. This is then added to the input parameter, `val` and returned from `bar`.

```
function bar(int val) -> (int r):
   return summation(5) + val;

function summation(int b) -> (int r):
   int i = 0
   int ans = 0
   while i < b:
      ans = ans + i
      i = i + 1
   return ans
```

Listing 4.8: Whiley program for the bar function that uses `factorial` from Listing 4.7

Memoisation is especially effective at reducing the number of recursive calls made to a function. Memoisation can also help to test functions that cannot be tested as they take too long to execute such as `factorial(30)`. Methods are not cached as they have side effects and may use variables in the global scope. Therefore, executing a method with the same input may result in a different output.

4.4 Limitations of Testing Methods

Testing functions with the tool is easier than methods as functions do not have side effects. Functions are not influenced by external resources therefore, the same input results in the same output. In comparison, there are several limitations for testing methods with the tool.

External inputs Methods may require external data such as a file to execute correctly. Files are usually imported by passing in a filename as a string argument into the `main` method. However, this is not possible to test using QuickCheck for Whiley as arguments to method are generated by the tool. Consequently, the method cannot be tested as the generated value which represents the filename does not exist in the file system.

Shared global variables Each method and function is tested independently from each other with their own call stack for storing variables. In some cases, methods may share and modify a common global variable. Therefore, it would be beneficial to test a method in the presence of other method calls. The order of method calls is important as the global variable may affect the output of a method. Calling a method with the same input could cause it to behave differently and may produce different outputs.

Chapter 5

Evaluation

This chapter explains the experiments conducted to measure the performance of QuickCheck for Whiley as well as highlight the bugs discovered by the tool.

5.1 Evaluation setup

The tool has been evaluated by testing a variety of Whiley programs. The Whiley programs will judge the tool's ability to generate suitable input values for testing, ability to validate a program's correctness and ability to identify bugs in the program.

Several existing test suites with Whiley programs are used to evaluate the tool. The test suites are the Whiley Compiler tests from the WhileyCompiler repository [17] and the Whiley Benchmark tests from the WyBench repository [18] as described in Section 5.2.

The test suites are executed using four different configurations to check the performance of the tool across different factors. The configurations are:

1. No optimisation techniques are applied
2. Memoisation is applied
3. Function optimisation is applied
4. Memoisation and function optimisation is applied

Java Unit (JUnit) test suites were created from the Whiley test suites to use for experimentation. Executing the JUnit test suite involves compiling each Whiley file without verification. Each compiled Whiley file is then tested twice using different integer ranges, a positive integer range and a negative range. The first integer range is for generating negative integer values, between -5 (inclusive) and 0 (exclusive). The second integer range is for generating positive integer values, between 0 (inclusive) and 5 (exclusive). The reason for splitting into two integer ranges is to reduce the possible number of input value combinations. In total, 200 tests will be executed equally across the two execution runs. Each JUnit test has a ten minute timeout to prevent a long running test from skewing the performance results. Exhaustive test generation and random test generation are used, where a seed is used for the latter to ensure test results are consistent.

The tool is executed seven times for each configuration on a test suite. The first and second run are discarded to ensure all libraries required are fully loaded into Java's virtual machine [19]. For each run, the execution time, number of errors, number of failures, number of skipped tests and number of passed tests is recorded from the JUnit test suite results. Five runs are measured so that the tool reaches a steady state of performance and to reduce the variance from non-deterministic factors [19]. The mean, standard deviation and coefficient of variation (standard deviation/mean) for determining variation between runs is then calculated across multiple runs. Finally, the mean and the coefficient of variation is reported.

5.2 Test Suites

Whiley Compiler Valid Tests. The Whiley compiler valid tests consists of 524 test cases from the WhileyCompiler repository [17] that checks all aspects of the Whiley language such as records, recursive structures and invariants. All test cases should be testable by QuickCheck for Whiley and pass for all valid inputs.

Whiley Compiler Invalid Tests. The Whiley compiler invalid tests consists of 321 test cases from the WhileyCompiler repository [17]. A successful test is when a syntactic error or verification error is raised during testing. Syntactic errors occur during the compilation of the program. Verification errors occur as an invalid test should fail for some valid input.

Whiley Benchmark Tests. The Whiley benchmark test suite includes 28 benchmarks as described in Table 5.1 from the WyBench repository [18]. The benchmark tests are representative of common problems that can be solved using Whiley, such as merge sort in `010_sort` and a binary search tree in `013_btree`. Several of these tests take too long to verify so are not verified when compiled. The benchmark tests will help test the runtime performance of QuickCheck for Whiley as the test cases are larger than the test cases in the Whiley Compiler tests. All benchmark test cases should be testable by QuickCheck for Whiley and pass for all valid inputs.

5.3 Evaluation results

5.3.1 Whiley Compiler Valid Tests

The performance of this test suite is shown in Figure 5.1 and Figure 5.2. Tests with memoisation and no optimisation had the same number of tests pass and fail, with 502/514 tests passing (97.67%) for exhaustive testing and 496/514 tests passing (96.50%) for random testing. Tests failed as inputs generated for the integer ranges were not valid for the preconditions in the function/method. The test `Lifetime_Lambda_Valid_4` also failed as the tool was not able to create lambda methods that took the same lambda method type as an argument. For random testing, one test `ListAccess_Valid_8` timed out after 10 minutes (600 seconds) due to an infinite loop for larger bytes. No optimisation performed the best for exhaustive testing running at 86.2 seconds for exhaustive testing. Memoisation performed the best for random testing at 1156.3 seconds. This is followed closely by memoisation at 89.584 seconds for exhaustive testing and no optimisation for random testing at 1318.6 seconds.

Running the tool with only function optimisation resulted in 418/514 tests pass (81.32 %) with 84 more failed tests. This is due to the assert/assume statements used to check the output of a called function as discussed in Subsection 5.4.2. The functions optimised do not have postconditions therefore, the output generated did not match the assert/assume statement. Function optimisation also takes longer at 450.7 seconds and 1470.5 compared to no optimisation at 86.2 seconds and 1318.6 seconds for exhaustive and random testing respectively. This is due to the generation of values in function optimisation. When function optimisation is applied with memoisation, the tests run faster at 306.4 seconds for exhaustive

Name	Description
`001_average`	Average over integer array.
`003_gcd`	Classical GCD algorithm.
`004_matrix`	Straightforward matrix multiplication.
`007_regex`	Regular expression matching.
`008_scc`	Tarjan's algorithm for finding strongly connected components.
`009_lz77`	LZ77 compression / decompression.
`010_sort`	Merge Sort.
`011_codejam`	Solution for Google CodeJam problem.
`012_cyclic`	Cyclic buffer.
`013_btree`	Binary search tree with insertion / lookup.
`014_lights`	Traffic lights sequence generator.
`015_cashtill`	Simple change determination algorithm.
`016_date`	Gregorian dates.
`017_math`	Simple math algorithms.
`018_heap`	Binary heap data structure.
`022_cars`	Controlling cars on bridge problem.
`023_microwave`	Classical microwave state machine.
`024_bits`	Algorithms for bit arrays.
`026_reverse`	Reversing an array.
`027_c_string`	Model of C strings.
`028_flag`	Dutch national flag Problem.
`029_bipmatch`	Perfect matching for bipartite graphs.
`030_fractions`	Big Rationals.
`032_arrlist`	`ArrayList` implementation.
`101_interpreter`	WHILE language interpreter.
`102_conway`	Conway's Game of Life.
`104_tictactoe`	Tic-Tac-Toe.
`107_minesweeper`	Minesweeper.

Table 5.1: Description of the Whiley benchmark suite.

testing compared with no optimisation. This is due to outputs being cached in memoisation and then returned instead of executing the function again which improves the speed of execution. However for random testing, function optimisation with memoisation took just as long as tests with function optimisation only at 1469.7 seconds. Function optimisation with memoisation also passes one more test than using only function optimisation because a previously cached value was returned instead of a value generated from function optimisation.

The generation of recursive types in random testing seems to cause the tests to take longer than exhaustive testing. This is due to long recursive values generated in random testing whereas exhaustive generation starts generation at shorter recursive values. The tests, `RecursiveType_Valid_23` and `RecursiveType_Valid_26` execute for 100 seconds each in random testing compared to exhaustive testing at 0.1 seconds.

5.3.2 Whiley Compiler Invalid Tests

The performance of this test suite is shown in Figure 5.3 and Figure 5.4. Surprisingly, all tests passed for the Whiley invalid tests for all configurations. Both random and exhaustive testing has similar execution times between nine and ten seconds. 115/355 tests (32.39 %) were identified to have bugs in their program as they failed for some valid input. This is due

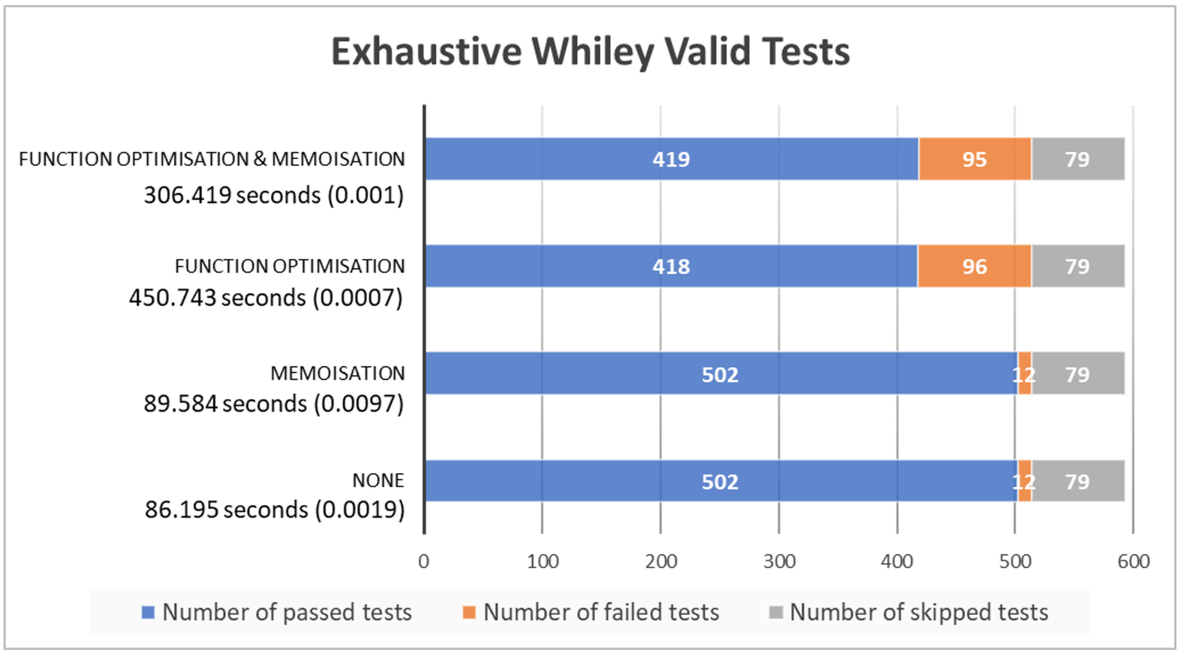


Figure 5.1: Statistics for the Whiley Valid Tests running with exhaustive testing. Underneath each configuration is the average time to run the tests and the variation coefficient.

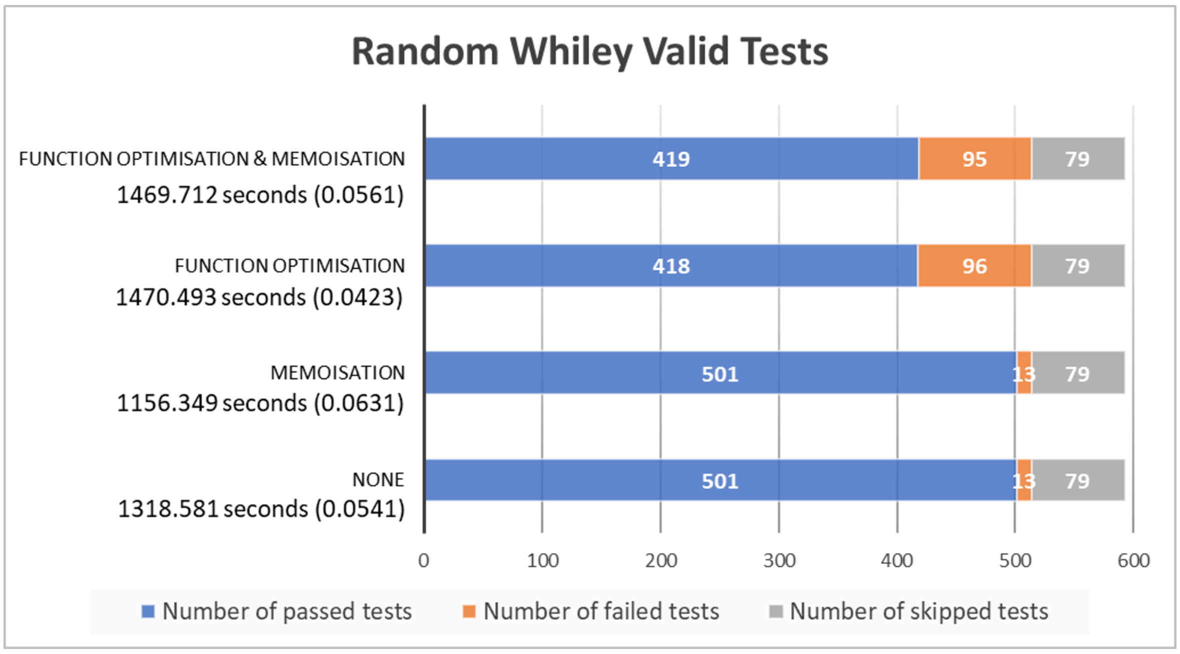


Figure 5.2: Statistics for the Whiley Valid Tests running with random testing. Underneath each configuration is the average time to run the tests and the variation coefficient.

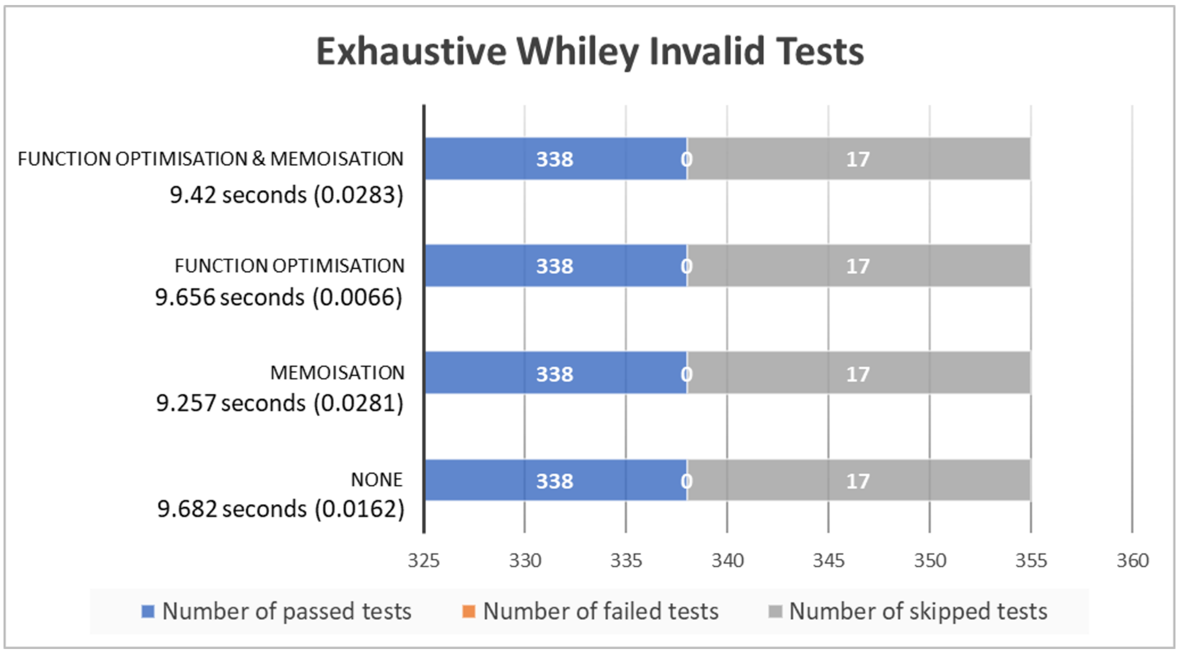


Figure 5.3: Statistics for the Whiley Invalid Tests running with exhaustive testing. Underneath each configuration is the average time to run the tests and the variation coefficient.

to incorrect specifications or an incorrect function/method body in the test. For example, the test `ConstrainedInt_Invalid_1` fails as the value returned may not meet return type's constraint. The remaining tests passed as they were unable to be compiled.

5.3.3 Whiley Benchmark Tests

The performance of this test suite is shown in Figure 5.5 and Figure 5.6. Tests from `016_date` and `029_bipmatch` have not been included in the performance test as they take too long to execute.

Tests with memoisation and no optimisation had the same number of tests pass and fail, with 13/27 tests passing (48.15%). Some tests fail as they require reading a file from the file system. The filepath is a string argument that is generated by the tool and may not be a valid filepath. Other tests failed due to inputs generated not meeting the preconditions of the functions/methods for the integer ranges given. For random testing, one test failed as the test timed out after 10 minutes.

Running with function optimisation has 10/27 (37.04%) pass with three more tests failing. Two tests failed for function optimisation and one test failed for function optimisation with memoisation as the tests timed out after 10 minutes. The other tests failed due to a value from an optimised function not meeting the postcondition of the outer function, or a precondition of another function called as discussed in Subsection 5.4.2. In terms of time, the longest configuration is with function optimisation only at 1539.8 seconds for exhaustive testing and no optimisation at 2412.1 seconds for random testing. The next slowest configuration is with function optimisation and memoisation at 1104.311 seconds for exhaustive testing and only function optimisation at 2258.2 seconds for random testing. Next, no optimisation executed in 380.2 seconds for exhaustive testing and function optimisation with memoisation at 2189.4 seconds for random testing. Memoisation performs better than all other configurations at 81.606 seconds for exhaustive testing and 1445.8 seconds for memoisation. For exhaustive testing, it performed four times faster than when executing tests

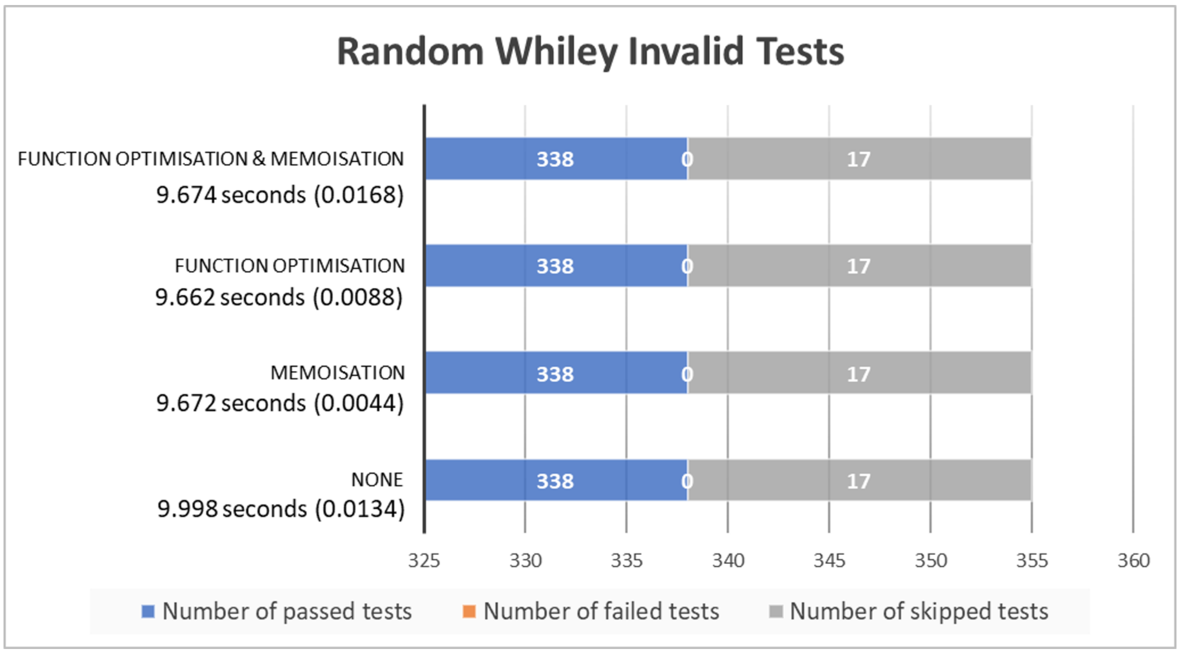


Figure 5.4: Statistics for the Whiley Invalid Tests running with random testing. Underneath each configuration is the average time to run the tests and the variation coefficient.

with no optimisation. When function optimisation is applied with memoisation, the tests run faster for both test generation techniques compared with no optimisation at the cost of test performance.

There is a discrepancy between the execution time between exhaustive and random testing. This is due to the generation of large values which can cause loops within the tests to execute for a long time. For example, `004_matrix` can take 400 seconds in random testing compared to 20 seconds in exhaustive testing.

5.4 Discussion

5.4.1 Fixing Bugs Identified

As a result of running QuickCheck for Whiley for the evaluation, several bugs were discovered within the Whiley programs in the various test suites. Subsequently, these bugs were logged as issues in their relevant GitHub repositories with counter examples generated by the tool. Bug fixes were then created as pull requests as shown in Figure 5.7 for the benchmark tests and applied after review from the repository owner for example, in Figure 5.8. Details of the bugs and fixes applied is shown in Table 5.2 [17, 18].

The bugs identified ranged from `NegativeArraySizeException` errors to failing postconditions. Solutions to the problems involve fixing existing specifications, adding preconditions or adding type invariants. In total, 14 problems were identified and fixed across 12 programs. Three programs are from the Whiley compiler valid tests and eight programs are from the Whiley benchmark tests. Furthermore, an incorrect specification was discovered and fixed in the `append` function of the `ascii` file in the Whiley standard library [20].

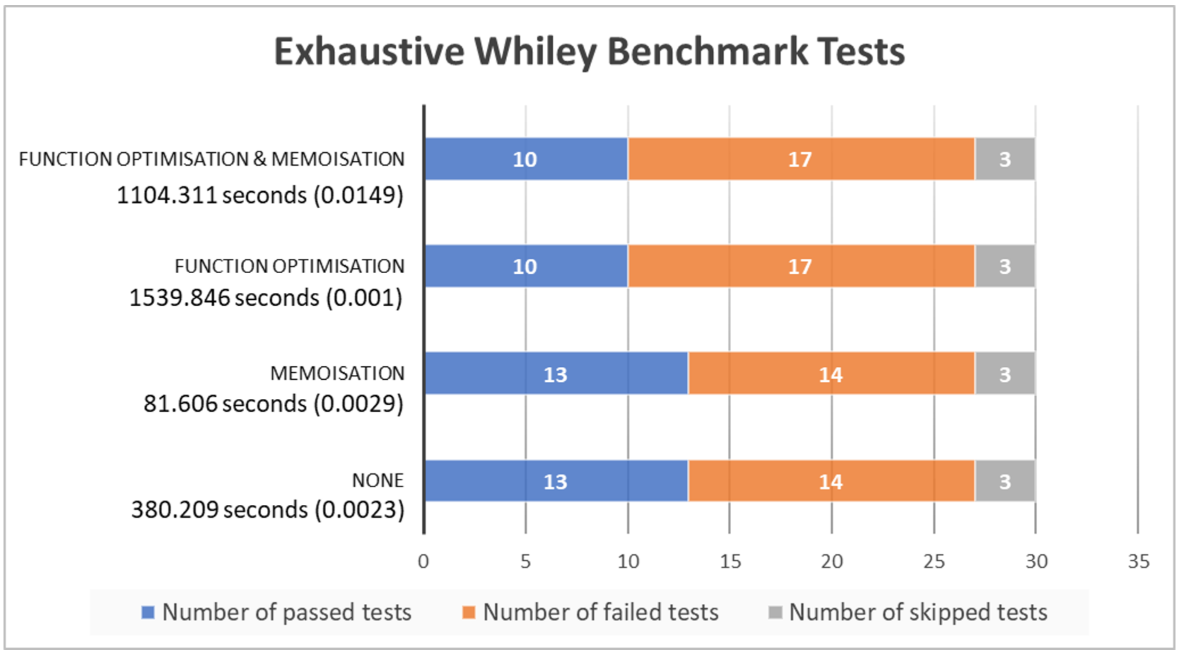


Figure 5.5: Statistics for the Whiley Benchmark Tests running with exhaustive testing. Underneath each configuration is the average time to run the tests and the variation coefficient.

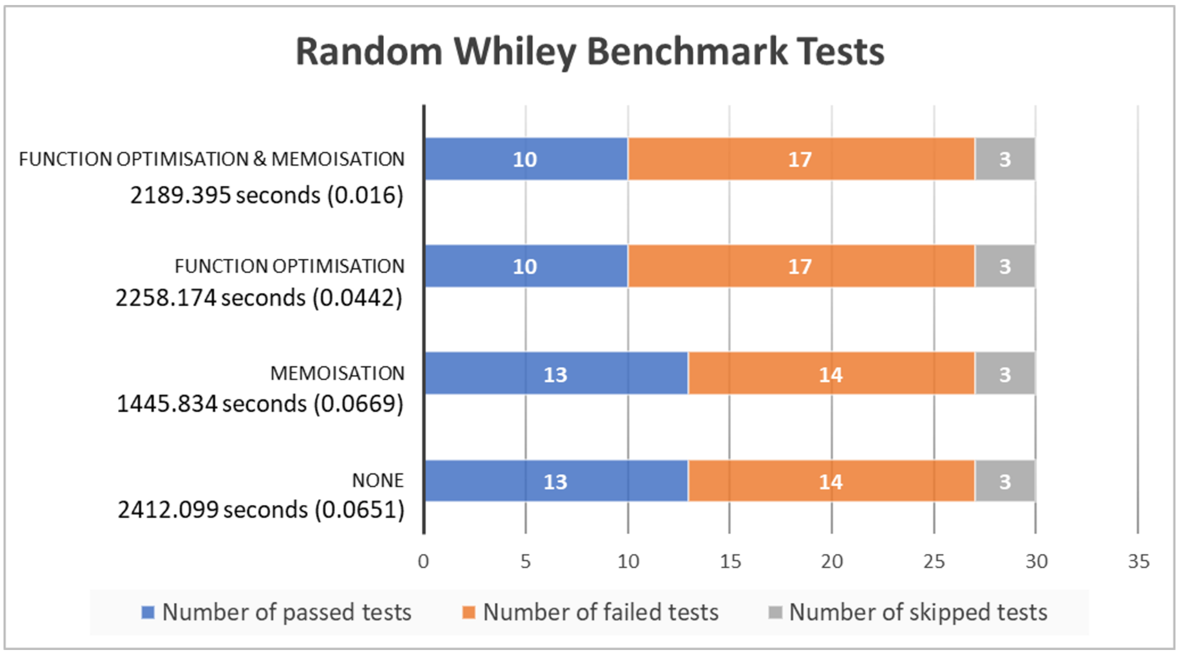
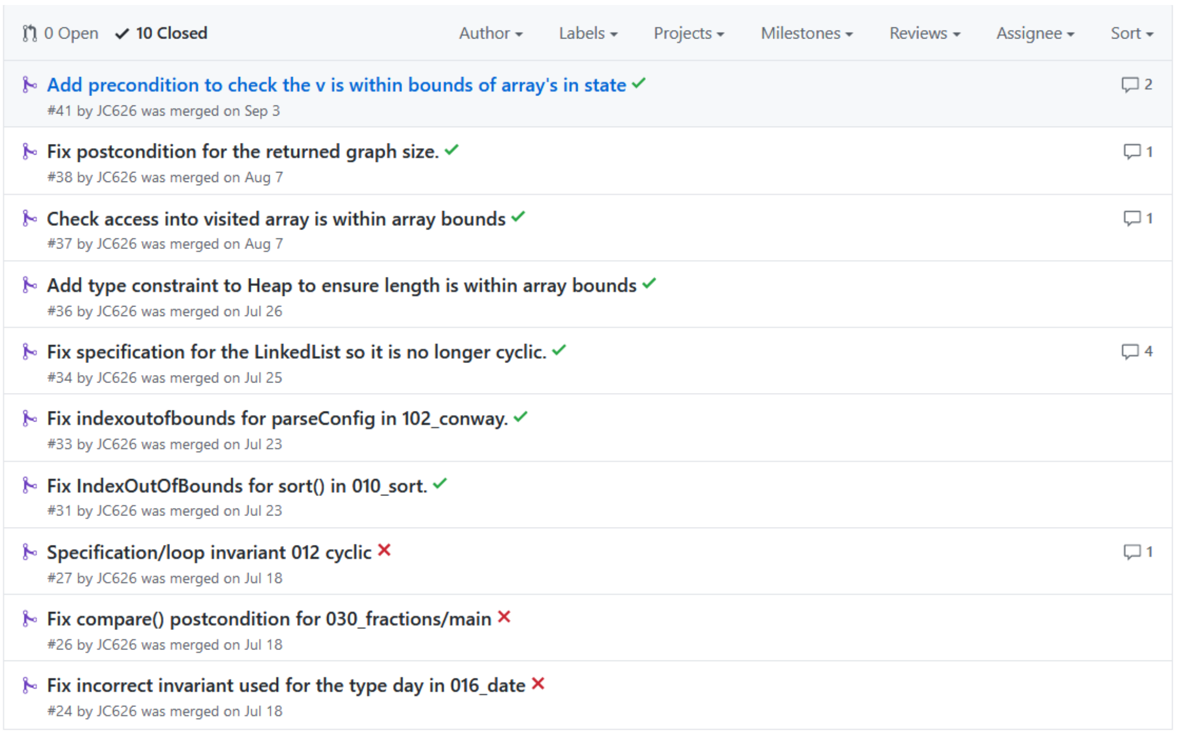


Figure 5.6: Statistics for the Whiley Benchmark Tests running with random testing. Underneath each configuration is the average time to run the tests and the variation coefficient.


0 Open ✓ 10 Closed Author Labels Projects Milestones Reviews Assignee Sort

Add precondition to check the v is within bounds of array's in state ✓ 2
#41 by JC626 was merged on Sep 3

Fix postcondition for the returned graph size. ✓ 1
#38 by JC626 was merged on Aug 7

Check access into visited array is within array bounds ✓ 1
#37 by JC626 was merged on Aug 7

Add type constraint to Heap to ensure length is within array bounds ✓
#36 by JC626 was merged on Jul 26

Fix specification for the LinkedList so it is no longer cyclic. ✓ 4
#34 by JC626 was merged on Jul 25

Fix indexoutofbounds for parseConfig in 102_conway. ✓
#33 by JC626 was merged on Jul 23

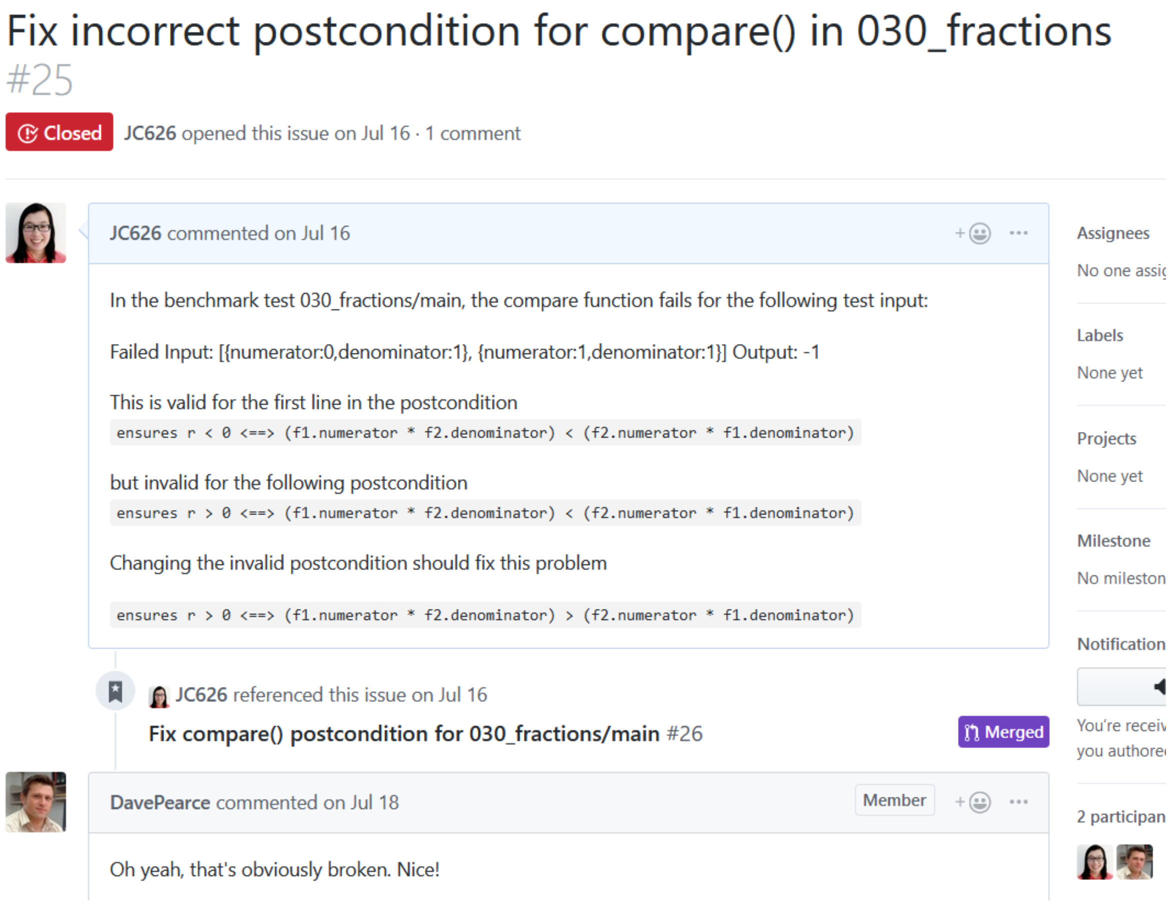
Fix IndexOutOfBounds for sort() in 010_sort. ✓
#31 by JC626 was merged on Jul 23

Specification/loop invariant 012 cyclic ✗ 1
#27 by JC626 was merged on Jul 18

Fix compare() postcondition for 030_fractions/main ✗
#26 by JC626 was merged on Jul 18

Fix incorrect invariant used for the type day in 016_date ✗
#24 by JC626 was merged on Jul 18

Figure 5.7: GitHub pull requests that were merged for the Whiley benchmark test suite [21].


Fix incorrect postcondition for compare() in 030_fractions #25

Closed JC626 opened this issue on Jul 16 · 1 comment

JC626 commented on Jul 16

In the benchmark test 030_fractions/main, the compare function fails for the following test input:

Failed Input: [{numerator:0,denominator:1}, {numerator:1,denominator:1}] Output: -1

This is valid for the first line in the postcondition

```
ensures r < 0 <==> (f1.numerator * f2.denominator) < (f2.numerator * f1.denominator)
```

but invalid for the following postcondition

```
ensures r > 0 <==> (f1.numerator * f2.denominator) < (f2.numerator * f1.denominator)
```

Changing the invalid postcondition should fix this problem

```
ensures r > 0 <==> (f1.numerator * f2.denominator) > (f2.numerator * f1.denominator)
```

JC626 referenced this issue on Jul 16

Fix compare() postcondition for 030_fractions/main #26 Merged

DavePearce commented on Jul 18 Member

Oh yeah, that's obviously broken. Nice!

Figure 5.8: A GitHub issue for the failing valid test `030_fractions` [22].

File & Location	**Issue Identified**	**Solution**
ascii WyStd Issue 4	Some Whiley Benchmark tests were failing when the `append` function was executed.	Fix postcondition for `append`.
ConstrainedInt_Valid_20 WhileyCompiler Issue 855	Infinite loop for `f` function when input, `x` is negative	Add precondition to the input, so that x >= 0
DoWhile_Valid_9 WhileyCompiler Issue 857	NegativeArraySizeException on postcondition when an empty array is given to the `bubbleSort` function	Fix `sorted` property on postcondition to allow empty arrays
Record_Valid_5 WhileyCompiler Issue 859	NegativeArraySizeException on postcondition when an empty array is given to the `cards` function	Fix `sorted` property on postcondition to allow empty arrays
008_scc WyBench Pull Request 38	Fix `resize` function failing on postcondition for input g = [[0]], size = 0	Fix post-condition when the input graph does not need to be resized, as it is larger than the size given
008_scc WyBench Issue 40	ArrayIndexOutOfBounds for `visit` function failing such as the input, v=0, s = {graph:[], index:0, stack:{items:[], length:0}, cindex:0, rindex:[], visited:[], inComponent:[] }	Add type invariant to `State` and precondition to ensure `v` is within the bounds of the arrays in `s`
010_sort WyBench Issue 30	IndexOutOfBounds error on `sort` function	Add precondition to ensure indexes given are within array boundaries
016_date WyBench Issue 23	The `next` function failed for the input, date = {day:1,year:0,month:1}, got the output {day:2,year:0,month}.	Fix type invariant for `Date`.
018_heap WyBench Issue 35	NegativeArrayException when `length` on `Heap` is negative	Add invariant to type `Heap` to ensure `length` is within the array's length
029_bipmatch WyBench Issue 29	The `findMaximalMatching` function fails for the following input, g = [{N1:1, N2:1, edges: [{to:0, from:0}]}]	Fix invariant for the type `Graph`
029_bipmatch WyBench Pull Request 37	`find` function can cause IndexOutOfBounds when accessing the `from` array	Add precondition to ensure `visited` is within bounds of the array, `from`
030_fractions WyBench Issue 25	The `compare` function failed for the input, f1 = {numerator:0, denominator:1}, f2 = {numerator:1, denominator:1}, got the output -1.	Fix postcondition for `compare`.
032_arrlist WyBench Issue 28	Stackoverflow error occurs in `length` function if list is cyclic such as the list - {size:1, links: [{data:0, next:0}]}	Fix invariant in the type `LinkedList`
102_conway WyBench Issue 32	IndexOutOfBounds error on `parseConfig` function	Add precondition to check the length of the array

Table 5.2: Table of test programs fixed using QuickCheck for Whiley

5.4.2 Effectiveness of optimisation

Memoisation Memoisation can improve the performance of the tool for some cases. Fewer functions are executed as cached output is returned instead resulting in an overall better performance time without loss of test accuracy. Memoisation did not perform as well in random testing for the Whiley valid test suite compared to exhaustive testing as the cached values are less likely to be used in subsequent tests due to the random generation of values. In exhaustive testing, the cached values may be more likely to be used as input values are generated in a sequential order.

Function Optimisation In comparison, function optimisation resulted in a decrease in performance. The time taken to optimise a function was more than the time for executing the function normally. Furthermore, the accuracy of tests also decreased when function optimisation was used. This is because not all functions optimised have strong postconditions used when generating values for optimisation as was the case when executing the *Whiley Valid Tests*. If the value generated is used in an assert/assume statement or function/method, the value may not meet the assert/assume statement or the function/method's contract. Consequently, the test fails as the value is invalid. For example, testing `isOne()` with function optimisation from Listing 5.1 may result in failed tests. When `getValue()` is called in line 3, function optimisation generates an integer, such as 2 for the return value `r`. The return value 2 is then set to `val`. An error occurs when `val` is returned in line 4 as it does not meet the postcondition on line 2 where the return value must be 1. Alternatively, function optimisation could be used for discovering functions with weak postconditions.

```
1 function isOne() -> (int ans)
2 ensures ans == 1:
3    int val = getValue()
4    return val
5
6 function getValue() -> (int r):
7    return 1
```

Listing 5.1: Whiley program for testing `isOne()` with function optimisation

Chapter 6

Conclusions

6.1 Future Work

QuickCheck for Whiley supports most of the types and functionality found in the Whiley programming language. However, there are several extensions that could be implemented to improve the tool:

Controlling distribution of values generated. The user cannot control the distribution of random input values that are generated. To allow users to control the distribution of data, probabilities or frequencies could be added to each generator based on the value that can be generated.

Aliasing in reference generation. References generated by QuickCheck for Whiley currently do not support aliasing the same memory cell. Therefore, QuickCheck for Whiley could be extended by allowing references to be generated that point to the same memory cell.

Open record generation. Open records can contain an arbitrary number of unknown fields, each with an unspecified data type. An example of open records is shown in Listing 6.1.

```
type TestRecord is {int x, ...}

TestRecord a = {x: 1}
TestRecord b = {x: 2, isPositive: true}
```

Listing 6.1: Whiley program with open records

To implement this, the tool would need to be able to generate more fields for an open record. This requires knowing the number of extra fields that should be added, each with a corresponding type and label that needs to be generated. The problem with generating field labels is that there is an infinite number of possible labels that can be generated. Possible labels to generate can be guessed by looking at the labels for concrete fields in the program. Care should also be taken when naming the new fields in the record as the names should be unique.

Complex lambda generation. Currently, lambdas generated only contain a return statement with values that correspond to their return types. It would be beneficial to use more complex lambdas such as lambdas that use input variables or contains loops and conditional statements. This could be implemented by using existing functions/methods in the program that match the definition of the lambda type.

Regression testing. Test inputs generated by the tool are executed and then discarded for each test. It would be useful for the user if they could test their code by reusing the same inputs when they make changes in the future. This would require storing the test inputs generated and the corresponding outputs in a file. Therefore, a user could rerun the tests by using the stored file to check their code is correct with respect to the input and output stored.

Mutation testing. Another method for evaluating the tool's effectiveness would be to mutate a correct implementation of a program to generate more programs for testing. Mutating a program involves modifying aspects of a Whiley function or method such as changing an operator in an if statement. Subsequently, the tool's effectiveness would be evaluated on its ability to discover bugs in the mutated versions of the program. Tests generated by the tool should pass for the correct implementation of the function/method and ideally fail for the mutated versions.

6.2 Conclusion

For this project, the concept of automatic test case generation for Whiley programs has been realised by implementing and evaluating QuickCheck for Whiley. Users can easily test their Whiley program by adding specifications to functions and methods. The tool allows a user to utilise random or exhaustive test case generation to robustly test their Whiley programs. In addition, function optimisation and memoisation was added to the tool in order to improve the execution time of tests. A range of programs can be tested as shown by evaluating various Whiley test suites. As a result of evaluating the tool, bugs were discovered and fixed within the Whiley test suites and Whiley standard library. Memoisation was also identified as a good optimisation strategy whereas function optimisation was not. Overall, QuickCheck for Whiley helps achieve the goal of the Whiley programming language to improve software quality through the detection of bugs.

Bibliography

[1] S. Planning, "The economic impacts of inadequate infrastructure for software testing," *National Institute of Standards and Technology*, 2002.

[2] J. A. Whittaker, "What is software testing? And why is it so hard?," *IEEE Software*, vol. 17, pp. 70–79, Jan 2000.

[3] D. J. Pearce, "The Whiley Language Specification." `http://whiley.org/download/WhileyLanguageSpec.pdf`, 2014. [Online; accessed 9-April-2018].

[4] D. J. Pearce and L. Groves, "Designing a verifying compiler: Lessons learned from developing Whiley," *Science of Computer Programming*, vol. 113, pp. 191 – 220, 2015. Formal Techniques for Safety-Critical Systems.

[5] K. Claessen and J. Hughes, "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs," in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, (New York, NY, USA), pp. 268–279, ACM, 2000.

[6] J. Hughes, "Experiences with QuickCheck: testing the hard stuff and staying sane," in *A List of Successes That Can Change the World*, pp. 169–186, Springer, 2016.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] D. J. Pearce, "Getting Started with Whiley." `http://http://whiley.org/download/GettingStartedWithWhiley.pdf`, 2018. [Online; accessed 26-August-2018].

[13] 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.

[14] 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.

[15] D. J. Pearce, "Whiley2EmbeddedC/IntegerRange.java at master." `https://github.com/Whiley/Whiley2EmbeddedC/blob/master/src/main/java/wyec/lang/IntegerRange.java`, January 2017. [Online; accessed 23-May-2018].

[16] 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.

[17] D. J. Pearce, "WhileyCompiler/tests at develop." `https://github.com/Whiley/WhileyCompiler/tree/develop/tests`, August 2018. [Online; accessed 03-September-2018].

[18] D. J. Pearce, "WyBench/src/ at develop." `https://github.com/Whiley/WyBench/tree/develop/src`, August 2018. [Online; accessed 03-September-2018].

[19] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically Rigorous Java Performance Evaluation," in *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, (New York, NY, USA), pp. 57–76, ACM, 2007.

[20] D. J. Pearce, "WySTD." `https://github.com/Whiley/WySTD`, August 2018. [Online; accessed 03-September-2018].

[21] J. Chin, "Pull Requests - Whiley/WyBench." `https://github.com/Whiley/WyBench/pulls?q=is%3Apr+author%3AJC626+is%3Aclosed`, September 2018. [Online; accessed 13-October-2018].

[22] J. Chin, "Fix incorrect postcondition for compare() in 030_fractions." `https://github.com/Whiley/WyBench/issues/25`, August 2018. [Online; accessed 13-October-2018].