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

Inferring invariants from postconditions in Whiley

Simon Pope

Supervisors: Lindsay Groves, David Pearce

October 21, 2018

Submitted in partial fulfilment of the requirements for
Bachelor of Engineering (Hons) in Software Engineering.

Abstract

For the formal verification of programs, loop invariants must be used to ensure the verifier understands the properties of a loop. These invariants are often trivial, and many are common between loops. It would be easier for programmers if these types of invariants did not have to be written into code, but were instead generated by the compiler. One way to do this is to infer the loop invariants from the postconditions of the containing function or method. This project looks at implementing this for the Whiley language. This is assisted by techniques of static analysis to better shape mutation.

Acknowledgments

Thank you to my supervisors Lindsay Groves and David Pearce.

Contents

Figures

Chapter 1

Introduction

One of the major concerns in programming is ensuring the correctness of programs. A solution to this issue is formal verification. This uses assertions to describe the relationships between variables and the properties of their values [1]. In languages such as Whiley[2] or Dafny[3], these assertions come in the form of preconditions and postconditions, which are used to specify what a function or method does. The precondition indicates the state of the argument variables on entry, and the postcondition indicates the state of the variables at the end of the function or method.

For example, take the Whiley code in Figure 1.1. The *nextHour* function takes an hour in 24 hour format and increments it by 1, returning it to 0 if it goes over 23. On line 2, there is the precondition that the *hour* argument is between 0 and 23 inclusive. On lines 3 and 4 are the postcondition that the returned value, *newHour*, is between 0 and 23 inclusive, and that it is either the old *hour + 1*, or *hour* is 23 and *newHour* is 0.

```
1  function nextHour(int hour) -> (int newHour)
2  requires 0 <= hour && hour <= 23
3  ensures 0 <= newHour && newHour <= 23
4  ensures newHour == hour + 1 || (newHour == 0 && hour == 23):
5      if hour < 23:
6          newHour = hour + 1
7      else:
8          newHour = 0
9      return newHour
```

Figure 1.1: The *nextHour* function.

In an automated verification system, like Whiley or Dafny, there is a verifier which attempts to ensure that a function or method, given the precondition, upholds the postcondition. One of the more complicated aspects of verification is the handling of loops. It is difficult for the verifier to understand the properties of a loop and what it achieves, meaning that often the function or method cannot be verified. In addition, it is difficult for the verifier to ensure that the loop will not cause runtime errors, such as array index out of bounds.

To account for this, the programmer must specify loop invariants. These are assertions which state the properties of the loop. A loop invariant must be upheld on entry to the loop, and after each iteration of the loop. This means that on exit from the loop, the invariant will be upheld, in addition to the negation of the loop condition. Note that even if the loop is not entered, the loop invariant must still hold on exit. The verifier is able to examine the loop and verify whether the invariants hold. It is then able to use this knowledge of the loop to verify whether the postcondition holds in regards to the rest of the function.

Loop invariants are often trivial and repetitive. This means it would be much easier for the programmer if there was a tool which generated the invariants for them. It is an undecidable problem to completely generate invariants from the code. Instead, various heuristic methods must be used.

In this project, I aim to implement a useful tool for generating loop invariants in the programming language Whiley. Whiley is a language which combines functional and object oriented programming, and supports formal verification. I aim to adapt and further the work of Carlo A. Furia and Bertrand Meyer's *Inferring loop invariants using postconditions* [4]. The paper provides a different approach to most other efforts in generating loop invariants. Most previous attempts have looked at analysing the bodies of loops to ascertain what they do, using various forms of static analysis. Instead, Furia and Meyer looked at the postcondition for the function the loop is in, to get an idea of the goal of the loop. They then used various methods to mutate postconditions to be more appropriate for loop invariants.

In developing their new method for loop invariant generation, Furia and Meyer completely moved away from the old methods of static analysis. This project looks at extending Furia and Meyer's work by incorporating static analysis to better shape the mutation of postconditions. One notable method of static analysis used is using back propagation to find the postcondition of the loop, rather than the function as a whole.

1.1 Contributions

I present an approach to generating loop invariants based on Furia and Meyer's *Inferring loop invariants using postconditions* [4]. My key contributions lie in the following techniques:

Implementation in context of Whiley I provide a tool for generating loop invariants within the Whiley language, allowing for users to benefit from loop invariant generation in their own programs.

Extension of algorithm I show how integrating static analysis of the enclosing function can improve mutation of postconditions.

Small empirical study I demonstrate the usefulness of my tool by applying it to a suite of Whiley programs.

1.2 Overview

The Background chapter details information about the Whiley language, and research into loop invariant generation. The Design chapter gives an overview of the techniques used in the tool, and why they were chosen. The Implementation section provides a more in depth look at the techniques and how they were implemented. The Evaluation chapter discusses the successes and shortcomings of the tool.

Chapter 2

Background

2.1 Whiley

Whiley is a language which combines functional and object-oriented programming [2]. "All compound structures have *value semantics*, meaning they are always passed by value." [5] This makes Whiley easier to reason about than languages which use pass by reference, as in Whiley structures cannot be altered outside of the scope they were declared in. Whiley uses a syntax similar to Python 3. This means that statement bodies are started with a colon, all expressions in the body are indented four spaces more than the statement, and the body ends when the indentation returns to that of the statement. Unlike Python however, Whiley has a strong type system.

```
 1  function max(int[] a) -> (int m)
 2  requires |a| > 0
 3  ensures some { k in 0 .. |a| | m == a[k] }
 4  ensures all { k in 0 .. |a| | m >= a[k] }:
 5      m = a[0]
 6      int i = 1
 7      while i < |a|
 8      where 1 <= i && i <= |a|
 9      where some { k in 0 .. |a| | m == a[k] }
10      where all { k in 0 .. i | m >= a[k] }:
11          if a[i] > m:
12              m = a[i]
13          i = i + 1
14      return m
```

Figure 2.1: The *max* function.

The function *max* (see Figure 2.1) takes an array of integers, and returns the highest value in the array. The returned variable (*int m*) is declared after the function's arguments (*int[] a*). Note that multiple variables can be returned by a single function or method in Whiley (though there is only one in this example).

The precondition of this function can be seen on line 2, with the keyword *requires*. The postcondition of the function can be seen on lines 3 and 4, denoted by *ensures*. Note that preconditions and postconditions can be written across multiple lines by using the *requires* or *ensures* keyword again. These expressions are conjunctive. On line 3, the first postcondition expression is an existential quantifier, denoted by the *some* keyword. This says that there exists a value of *k*, between 0 and the length of array *a*, for which *m* is equal to the value of array *a* at index *k*. When expressing the range of the quantifier, the lower bound is soft and the upper bound is hard. On line 4, the second postcondition expression is a universal quantifier.

On line 7 there is a while loop, and on lines 8, 9, and 10, there are its loop invariants, with the keyword *where*. Loop invariants, too, can be declared across multiple lines, and are conjunctive.

In this example, the returned variable, *m*, is the one declared at the top of the function. However, this does not always have to be the case. The returned variables only have to match the type of the declared returns.

2.2 Generating simple loop invariants for Whiley

Barnett's paper [6] is very relevant useful. It was written as part of a previous year's ENGR 489 project, and a precursor to this one. Barnett has created a tool which generates a small number of specific loop invariants using static analysis. These have been chosen as they are commonly used, and trivial to the Whiley programmer. Three types of invariants can be generated by the tool: upper and lower bounds, equal array length, and iterative array assignment.

Barnett's tool traverses the Whiley Abstract Syntax Tree once, and for each loop it finds, it attempts to generate invariants. For each type of possible invariant, there is a set of requirements that needs to be met. For each type, the tool does a run through of the loop (and preceding function where required), and checks if it meets all the requirements to add that invariant.

Barnett's paper provides a very descriptive insight into generating loop invariants in Whiley through static analysis. This provides a good basis for a useful loop invariant generation tool, but it is limited in its generality. The techniques only work in specific scenarios, so it is not as applicable to more complicated or unique Whiley programs.

2.2.1 Upper and lower bounds

The bounds of a loop are important invariants, as they are needed in many loops. They are used when a loop has a counting variable which is being incremented or decremented. They establish a range for the loop. This is most useful when the loop works with arrays, as it can ensure that the variable will always be within the array range.

Barnett establishes the bounds of the loop by looking at the value of the counting variable on entry to the loop. The final value of counting variable is found by looking at the loop condition. It is then checked that the loop is going in the correct direction (incrementing or decrementing) for the counting variable to be between the starting and final value.

For example, in Figure 2.2, the generated loop bounds can be seen on line 7. These were calculated from the *i*'s initial value on line 4, the loop condition on line 6, and *i*'s increment on line 11.

2.2.2 Equal array length

When two arrays are being compared, or if an array is having its values mutated into another array, it is important that the lengths of both arrays are equal. Barnett accomplishes this by looking at the length of arrays when they are created, and checking if their lengths are the same. If they are, he then checks whether the array lengths are not changed throughout the

```
 1 function arrayMultiply(int[] a, int x) -> (int[] r)
 2 ensures |r| == |a|
 3 ensures all { k in 0..|a| | r[k] == a[k] * 5 }:
 4     int i = 0
 5     r = [0; |a|]
 6     while i < |a|
 7     where 0 <= i && i <= |a|
 8     where |r| == |a|
 9     where all { k in 0..i | r[k] == a[k] * 5 }:
10         r[i] = a[i] * 5
11         i = i + 1
12     return r
```

Figure 2.2: The *arrayMultiply* function with invariants generated using Barnett's methods.

loop. As an example, in Figure 2.2, the generated equal array length loop invariant can be seen on line 8. This was calculated using the array generation on line 5.

2.2.3 Iterative array assignment

It is common to need to assign values to an array and ensure that they match something. To do this, you need to have a universal quantifier that ranges from one end of the array to the counting variable. On exit of the loop, this ensures that all values between the lower and upper bounds of the loop have been assigned a value (which is generally relative to its index in the array, and often to another array). Barnett does this by ensuring that in each iteration of the loop, the array assignment is made.

For example, a generated universal quantifier loop invariant can be seen on line 9 of Figure 2.2. The array assignment on line 10 is turned into an equality and used as the quantifier's expression. The range of the quantifier is the values of the array index that have been covered. In this case, the array index is *i*. *i*'s initial value is determined from line 4, and it's determined to be increasing from line 11, giving the range *0..i*. It should be noted that iterative array assignment requires the index variable to be iterative, i.e. every iteration of the loop it is incremented by 1 (or decremented by 1 in a reverse iteration).

2.3 Inferring loop invariants using postconditions

The work of Furia and Meyer has formed a basis for this project [4]. It is one of the first papers looking into different ways of generating loop invariants, as most prior work focused on static analysis of the loop body. It explains that the postcondition can be seen as the goal of the loop, giving a better understanding of what invariants are needed.

The paper gives a number of ways that postconditions can be mutated into invariants. Furia and Meyer's tool goes through each function. It then mutates the postcondition into a large number of candidate invariants and goes through each candidate one by one. The invariant is added to every loop in the function, and a verification attempt is made. The loop invariant is removed from any loop it doesn't hold in, and they move on to the next candidate. This continues until the postcondition is held, or all fail.

This paper provides a useful insight into more general methods of loop invariant generation. It allows for more complex and unique invariants to be generated. However, it is mostly based on making educated guesses about what loop invariants are required, and lacks in depth analysis of the loop. In addition, it only generates loop invariants required for the postcondition, it is not able to generate those required for safe runtime.

2.3.1 Constant relaxation

Constant relaxation is where a constant in the postcondition is replaced with a variable. The loop aims to fulfill the postcondition, but it generally won't do it until the loop exits. This means that the invariant expression needs to be a weakened form of the postcondition. Generally, on exit of the loop, the variable will be equal to the original constant. Note that a constant in this sense refers to a value which is constant for the duration of the loop, not necessarily a constant in the traditional sense.

As an example, the function *arrayOfIndices* (see Figure 2.3) takes a integer argument. It creates an array of that length, sets each value in the array to be its index, and returns the array. This function has the postcondition (line 4) that for all values of k from 0 up to the length of array a, the value of array a at index k is k.

```
1  function arrayOfIndices(int len) -> (int[] a)
2  requires len > 0
3  ensures |a| == len
4  ensures all { k in 0..|a| | a[k] == k }:
5      a = [0; len]
6      int i = 0
7      while i < |a|
8      where 0 <= i && i <= |a|
9      where |a| == len:
10         a[i] = i
11         i = i + 1
12     return a
```

Figure 2.3: The function *arrayOfIndices*.

Using constant relaxation, this postcondition is mutated to a loop invariant (see Figure 2.4) by substituting the variable i (counting variable for loop) for the constant $|a|$. This means that for all values of k from 0 up to i, the value of array a at index k is k. As k only goes up to i, the invariant is only checking the values at indexes of array a that have already been set by the loop. When the loop exits, i is equal to $|a|$, meaning that the loop invariant is the same as the postcondition.

```
1      where all { k in 0..i | a[k] == k }
```

Figure 2.4: The invariant generated for method *arrayOfIndices* using constant relaxation.

2.3.2 Term dropping

Term dropping is removing part of the postcondition. This is useful when the loop itself doesn't fulfill the whole postcondition, but just a part of it. It is also useful when a postcondition is a state that is reached within the loop, instead of being invariant.

For example, in the *allLessThan* function in Figure 2.5, the expression on line 7 has been mutated from the postcondition with assistance from term dropping. The postcondition expression on line 3 has been dropped, because it is an event which occurs, rather than an invariant of the loop. In addition, the implication part of the postcondition expression on line 2 has also been dropped. This is because the function does not return *true* until after the loop, so it is not part of the loop invariant.

```
 1  function allLessThan(int[] a, int x) -> (bool r)
 2  ensures r ==> all { k in 0..|a| | a[k] < x }
 3  ensures !r ==> some { k in 0..|a| | a[k] >= x }:
 4      int i = 0
 5      while i < |a|
 6      where 0 <= i && i <= |a|
 7      where all { k in 0..i | a[k] < x }:
 8          if a[i] >= x:
 9              return false
10          i = i + 1
11      return true
```

Figure 2.5: The *allLessThan* function.

2.4 Inferring loop invariants by mutation, dynamic analysis, and static checking

Galeotti, Furia, et al. wrote a paper about a loop invariant generation tool they had created for Java called *DYNAMATE* [7]. While Java itself does not utilise formal verification, the *ESC/Java2* [8] tool is used to provide this functionality. Galeotti et al.'s tool incorporates the methods used in *Inferring loop invariants using postconditions* [4]. It also utilises another tool, *DAIKON* [9]. *DAIKON*, which uses dynamic analysis to generate loop invariant candidates. Most notably, this paper demonstrates a method of verifying generated candidates.

1. A test case generator builds executions of a method that satisfy the precondition.

2. Loop invariant candidates are generated using mutation of postconditions and *DAIKON*.

3. Candidates are dynamically tested against the executions.

4. Surviving loop invariants are statically checked using *ESC/Java2*. This establishes whether they are valid, or not decidable.

5. *ESC/Java2*'s static analysis is used to attempt correctness proof of the program against its specification using the valid generated loop invariants.

6. If the correctness proof fails, the tool returns to step 1. In place of the postcondition, the candidates which weren't able to validated are mutated.

This paper provides some useful techniques for verification of loop invariants. However, a number of techniques used are not applicable to Whiley. Whiley uses an automated theorem prover, and does not have static analysis like *ESC/Java2*. Whiley's theorem prover only verifies full programs, and not specific invariants.

Chapter 3

Design

3.1 Requirements

This project aimed to provide a useful tool for Whiley programmers by generating invariants for them, limiting how explicit they have to be. While there was a focus on generating loop invariants in a more general fashion, it was also important to generate specific invariants that would be useful. A critical requirement of this project was to not negatively impact the programmer. The loop invariant generator must not stop programs from working that would have worked otherwise. In addition, the runtime of the loop invariant generator should not be excessive.

3.1.1 Generality and usefulness

One of the benefits of generating loop invariants from postconditions is that the invariants can be generated more generally. The postcondition can show what the goal of the loop is, which can be mutated to show what the loop does. This in contrast to the normal methods of static analysis of the loop body to see what the loop does. This looks for specific aspects and generates invariants accordingly.

Generality is good, as it is less limiting on the types of programs which can have loop invariants generated. However, to create a loop invariant generation tool which is useful, it is important to generate invariants which are commonly required. Not all of these can be generated from postconditions, so must be specifically discovered using static analysis.

Therefore, in this project, there was the need to balance generality and specificity. Where possible, loop invariants were generated in a general fashion. However, a number of loop invariants were identified which were commonly required to be generated specifically. This identification was guided by analysis of available Whiley programs, and David Barnett's paper [6].

3.1.2 Backwards compatibility and runtime

The loop invariant generator tool is meant to make it easier and less trivial to program in Whiley. As such, it is critical that the tool does not cause issues for the programmer. Any program that works without the loop invariant generator must work with it. In addition, the tool should run quickly. There is little point using the tool to generate invariants if it would be faster for the programmer to do it themselves.

3.2 Whiley compiler architecture

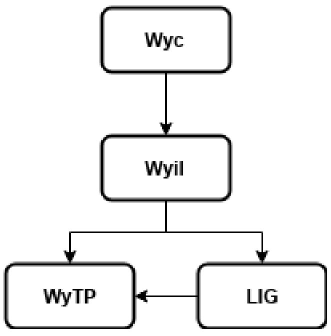

Figure 3.1: The Whiley compiler architecture with loop invariant generator.

The loop invariant generator (LIG) fits in as a plugin to the Whiley compiler collection. (see Figure 3.1). The Whiley source code is complied by the Whiley Compiler (WyC) into Whiley Intermediate Language (WyIL), in the form of the Abstract Syntax Tree. Various checks are done on the AST, including type checking and definite assignment.

Normally at this point, the WyIL would be converted into Whiley Assertion Language (WyAL), and passed into the Whiley Theorem Prover (WyTP). Instead, the WyIL is passed to the LIG, and loop invariant generation begins.

3.3 Loop invariant generation

In addition to postcondition mutation, a number of techniques are used to generate invariants, shape invariant generation, and verify generated invariants are valid. Mutating postconditions involves making educated guesses about what a loop does, guided by how loops commonly work. These techniques provided a more accurate analysis of the loop, and what it achieves.

As these techniques were added and expanded, various methods of mutation were found to no longer be required. The loop invariants that they had guessed were instead confirmed by other means. One example of this is logical implications. Sometimes, the postcondition would contain a logical implication, while the loop invariant would only require the implied expression. This mutation was guessed using term dropping, but with the introduction of back propagation, it was no longer required. One of the new techniques, back propagation, could discover the instances where the full implication was needed, and those where just the implied expression was required.

3.3.1 Method

1. The tool traverses the Abstract Syntax Tree, looking for loops (see Figure 3.2).
2. When it finds a loop, the loop is analysed for invariant generation.
   (a) Back propagation is used to find loop's postcondition.
   (b) Static analysis is done on the loop body and the statements preceding the loop.
   (c) The loop's postcondition is mutated to generate invariants.
3. The tool continues traversing the Abstract Syntax Tree, looking for more loops.

4. Once a full traversal has been made, the tool attempts to verify the generated loop invariants for the program.

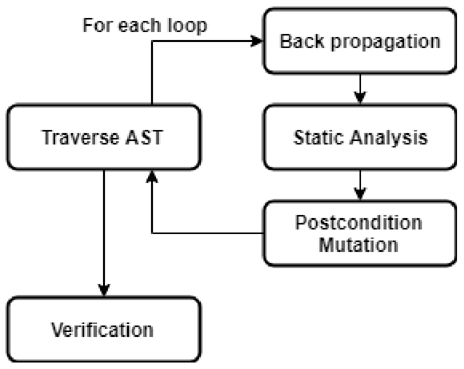

Figure 3.2: Flow diagram of invariant generation.

3.3.2 Back propagation

The postcondition for a function is not always the same as the postcondition for the loop. There can be statements after a loop which affect the postcondition, so it is important to know what the loop's postcondition is. The tool accomplishes this by using methods of finding the weakest precondition through back propagation. The weakest precondition for a statement is a property that needs to be satisfied, so that the statement, and all following statements, will lead to the postcondition being satisfied. The weakest precondition also ensures that there will not be runtime errors with the following statements. For example, if a statement contains an array access, its weakest precondition would include that the index valid.

The tool goes backwards from the end of the function, analysing each statement to discover its weakest precondition, until the loop is encountered. The weakest precondition for a statement is equivalent to the postcondition of the statement preceding it. This is useful for loop invariant generation, as it makes it possible to ascertain the postcondition for the loop.

Another benefit of back propagation is discovering variables which are used but aren't in the postcondition. For example, in Figure 3.3, the postcondition refers to the variable *r*, where as the variable *max* is actually used.

3.3.3 Static analysis

There are a number of loop invariants which are commonly required to ensure that array accesses within the loop body are in bounds. As these are related to the loop body, they are not generally found in the postcondition. However, they are necessary for verification of the program, and are usually trivial, so it is useful for the tool to be able to generate them. In addition, the information gleaned during their generation can be used to guide the mutation of postconditions.

For example, in Figure 3.4, the generated loop invariants can be seen in comments. The first, on line 6, was generated using static analysis. The second, on line 7, was generated by mutating the postcondition, using information from the static analysis.

```
function max(int[] a) -> (int r)
requires |a| > 0
ensures all { k in 0..|a| | r >= a[k] }
ensures some { k in 0..|a| | r == a[k] }:
    int max = a[0]
    int i = 1
    while i < |a|
    where 1 <= i && i <= |a|
    where all { k in 0..i | max >= a[k] }
    where some { k in 0..i | max == a[k] }:
        if a[i] > max:
            max = a[i]
        i = i + 1
    return max
```

Figure 3.3: The *max* function.

```
function allLessThan(int[] a, int x) -> (bool r)
ensures r ==> all { k in 0..|a| | a[k] < x }
ensures !r ==> some { k in 0..|a| | a[k] >= x }:
    int i = 0
    while i < |a|:
    //where 0 <= i && i <= |a|
    //where all { k in 0..i | a[k] < x }
        if a[i] >= x:
            return false
        i = i + 1
    return true
```

Figure 3.4: The *allLessThan* function with generated invariants in comments.

3.3.4 Verifying generated invariants

When generating loop invariants, it is important to ensure that they are valid. This is done in two ways, inspired by Galeotti et al.'s *DYNAMATE* [7]. However, it does not use their techniques of test case generation and dynamic checking. In addition, only one run through is made, and all mutations are made in that one go. These changes to *DYNAMATE*'s method are due to the requirement to limit the runtime of the loop invariant generation tool.

The first method uses static analysis to validate candidates. In the process of generating certain loop invariant candidates, some candidates can be confirmed to be valid. If certain criteria are met (explained in Section 4.2), and the properties of the candidate are predictable, the invariant can be validated. It is added to a list of valid loop invariants. Remaining candidates that can't be confirmed in this way are attempted to be verified using the second method.

The second method uses Whiley's theorem prover to verify the candidates. However, the theorem prover verifies whole programs, rather than single loop invariants. This means the all required loop invariants must have been generated for the program to verify. In addition, the theorem prover can be slow, so it is preferable to verify candidates using static analysis.

Chapter 4

Implementation

This section explains how the various techniques of the tool were implemented. To show how these techniques come together, there is an example in Figure 4.1. First, back propagation finds the loop's postcondition, which can be seen on line 11. Next, static analysis generates the loop invariants seen on line 6, and confirms they are valid. Then, using information from static analysis, the loop's postcondition is mutated to give the loop invariant on line 7. Finally, the program is verified, showing that the generated loop invariant is valid.

```
1  function allLessThan(int[] a, int x) -> (bool r)
2  ensures r ==> all { k in 0..|a| | a[k] < x }
3  ensures !r ==> some { k in 0..|a| | a[k] >= x }:
4      int i = 0
5      while i < |a|:
6      //where 0 <= i && i <= |a|
7      //where all { k in 0..i | a[k] < x }
8          if a[i] >= x:
9              return false
10         i = i + 1
11     //all { k in 0..|a| | a[k] < x }
12     return true
```

Figure 4.1: The *allLessThan* function, with generated loop invariants and weakest precondition in comments.

4.1 Back propagation

Back propagation starts from after the end of the function. At this point, the weakest precondition (*wp*) is the postcondition. The tool goes backwards through the function, analysing each statement. Depending on the statement, the *wp* may be modified. Back propagation continues analysing statements until the bottom of the loop is encountered. Note that for the ease of understanding, examples in this section do not have loops in them, as they tend to be complicated.

4.1.1 Variable initialisation and assignment

When variable initialisation or assignment is encountered, all instances of the variable in the *wp* are replaced with the assigned expression. For example, in Figure 4.2, the *wp* can be seen on line 3. When the statement on line 2 is analysed, all instances of r in the *wp* are replaced with $y + 3$, resulting in the assignment statement's *wp* on line 1.

```
1  //y + 3 == x * 5
2  r = y + 3
3  //r == x * 5
```

Figure 4.2: The weakest precondition of an assignment statement.

4.1.2 If/else statement

When an if/else statement is encountered, both the true (if) and false (else) branches are analysed using the current *wp* (see Figure 4.3). The resultant *wp* is *(condition && wp(true branch))* || *(!condition && wp(false branch))*. Note than when there is no else block, the current *wp* is used for *wp(false branch)* (see Figure 4.4).

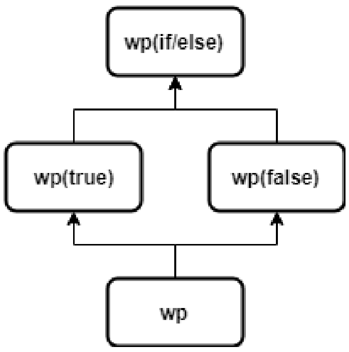

Figure 4.3: Back propagation flow for if/else statement.

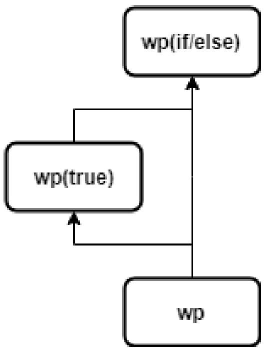

Figure 4.4: Back propagation flow for if statement.

```
1  function greater(int x, int y) -> (int r)
2  ensures r == x || r == y
3  ensures r >= x && r >= y:
4      /*(x >= y && (x == x || x == y) && x >= x && x >= y) ||
5      (x >= y && (y == x || y == y) && y >= x && y >= y)*/
6      if x >= y:
7          //(x == x || x == y) && x >= x && x >= y
8          r = x
9      else:
10         //(y == x || y == y) && y >= x && y >= y
11         r = y
12     //(r == x || r == y) && r >= x && r >= y
13     return r
14     //(r == x || r == y) && r >= x && r >= y
```

Figure 4.5: The *greater* function with weakest preconditions in comments.

As an example, take the function *greater* in Figure 4.5. When the if/else block is encountered, the *wp* is that seen on Line 12. This *wp* is then used by both the true branch (*wp(true)* on line 7) and the false branch (*wp(false)* on line 10). These two *wp*s are then used to make the *wp* for the whole if/else statement (across lines 4 and 5).

4.1.3 Switch statement

Back propagating switch statements works in a similar way to if/else statements. The current *wp* is put into each branch, and the branch *wp*s are combined as a union. The resultant *wp* is *(case 1 && wp(case 1))* || *(case 2 && wp(case 2))* || ... || *((!case 1 && !case 2 && ...) && (default case))*. In Whiley, switch statements do not fall through as they do in languages such as C or *Java*.

4.1.4 Return statement

When a return statement is encountered, the postcondition is substituted for the *wp*. This is because the original postcondition must be upheld at the point of return. The variables which are returned are substituted for those that they represent in the postcondition.

```
1  function abs(int x) -> (int r)
2  ensures r >= 0
3  ensures r == x || r == -x:
4      /*(x < 0 && -x >= 0 && (-x == x || -x == -x)) ||
5      (!(x < 0) && x >= 0 && (x == x || x == -x))*/
6      if x < 0:
7          //-x >= 0 && (-x == x || -x == -x)
8          return -x
9      //x >= 0 && (x == x || x == -x)
10     return x
11     //r >= 0 && (r == x || r == -x)
```

Figure 4.6: The *abs* function with weakest preconditions in comments.

When back propagating function *abs* in Figure 4.6, the return statement on line 10 has resulted in the *wp* on line 9, where all instances of *r* in the postcondition have been replaced with *x*. This can similarly be seen with the return statement on line 8 and its *wp* on line 7.

4.1.5 Array access

When an array access is encountered within a statement, an additional condition is added to the *wp*. The index of the array that is being accessed must be between 0 and the length of the array. This is to ensure that there are no runtime errors in the program. An example can be seen in Figure 4.7. On line 1, in addition to the substitution made due to the assignment, it can be seen that a condition has been added that *i* is within the bounds of array *a*.

```
1  //a[i] + 3 == 5 && 0 <= i && i < |a|
2  x = a[i] + 3
3  //x == 5
```

Figure 4.7: The weakest precondition of a statement with an array access.

4.1.6 While loops

While loops are an problem for back propagation. The reasons for loop invariants are that loops are difficult to reason for verification, and this makes them an issue for back propagation too. For this reason, loops are ignored for the purpose of back propagation.

4.1.7 Function and method calls

Method calls also cause issues. It is too complex to reason what is returned by them. As such, they are ignored. This is not an issue with function calls, as they may be used in loop invariants. However, with both function and method calls, the precondition is added to the *wp*. The arguments which are passed to the function or method are substituted into it.

4.1.8 On the fly simplification

As can be seen in the examples used (most notably Sections 4.1.2 and 4.1.3), back propagation can result in a long and complicated *wp*. Therefore, whenever the *wp* is modified, a simplifier is run over the new *wp*. This is a basic evaluation of the *wp*, which can determine whether parts of it are always true or false, and also remove superfluous parts. On the fly simplification is work now to save more work later. Various statements can cause the complexity of the postcondition to increase greatly, so by reducing its complexity at each statement, there is less room for it to balloon out [10].

On the fly simplification is made up of three parts. The first part is a definite value checker for integer comparison expressions (==, !=, <, <=, >, >=). If both sides have definite values,i.e. all constants and no variables, the expression can be determined to be true or false.

The second part analyses logical implications (*A* ==> *B*). If the implication's condition (*A*) is true, then the logical implication expression is replaced with the implied expression (*B*). This is because the condition is always true, so *B* is always required to hold. If the implication's condition is false, the logical implication expression is replaced with true. This is because the implied expression (*B*) will never be required to hold, so the implication expression will always be true.

The third part looks to remove or combine comparison expressions in conjunctions and disjunctions. It analyses each statement against the others. If two comparisons both have the same left and right hand sides, they are combined into a new expression. In conjunctions, this is the intersection between the expressions (see Figure 4.1). In disjunctions, it is the union (see Figure 4.2). Note that when checking if the left and right hand sides of two expressions are the same, it also compares left to right and vice versa. If they are the same, the expression is changed to be the other way round. For example, *x <= y && y > x* would be analysed as *x <= y && x < y*.

	x==y	**x!=y**	**x<y**	**x<=y**	**x>y**	**x>=y**
x==y	x==y					
x!=y	false	x!=y				
x<y	false	x<y	x<y			
x<=y	x==y	x<y	x<y	x<=y		
x>y	false	x>y	false	false	x>y	
x>=y	x==y	x>y	false	x==y	false	x>=y

Table 4.1: Resultant expression for two conjunct expressions.

	x==y	**x!=y**	**x<y**	**x<=y**	**x>y**	**x>=y**
x==y	x==y					
x!=y	true	x!=y				
x<y	x<=y	x!=y	x<y			
x<=y	x<=y	true	x<=y	x<=y		
x>y	x>=y	x!=y	x!=y	true	x>y	
x>=y	x>=y	true	true	true	x>=y	x>=y

Table 4.2: Resultant expression for two disjunct expressions.

The fourth part once again simplifies conjunctions and disjunctions. If any expression in a conjunction is true, it is removed. If any expression is false, the whole conjunction is false. For a disjunction, if any expression is true, the whole disjunction is true. If any expression is false, it is removed.

As an example of back propagation, in Figure 4.8 there is the *greater* function as seen in Figure 4.5. However, it also includes simplification of the *wp*. On line 11, there can be seen a simplification of the *wp* on line 12. In 4.9 is an explanation of how this particular simplification was made, ordered from top to bottom.

```
1   function greater(int x, int y) -> (int r)
2   ensures r == x || r == y
3   ensures r >= x && r >= y:
4       //true
5       //(x >= y && x >= y) || (!(x >= y) && y >= x)
6       if x >= y:
7           //x >= y
8           //(x == x || x == y) && x >= x && x >= y
9           r = x
10      else:
11          //y >= x
12          //(y == x || y == y) && y >= x && y >= y
13          r = y
14      //(r == x || r == y) && r >= x && r >= y
15      return r
16      //(r == x || r == y) && r >= x && r >= y
```

Figure 4.8: The *greater* function with simplification of weakest preconditions in comments.

4.2 Static analysis

The static analysis is based on Barnett's methods in *Generating simple loop invariants for Whiley* [6]. His techniques for generating array bounds and equal array length have been adapted to work within the tool. For each loop, the loop condition, loop body, and preceding statements in the function are all analysed. When generating loop bounds, the analysis is split into three parts: finding the counting variable, finding the loop direction, and generating the loop bounds. The counting variable and loop direction are also used in postcondition mutation.

```
(y == x || y == y) && y >= x && y >= y
//y == y is true, as is y >= y
(y == x || true) && y >= x && true
//if one disjunct is true, the whole disjunct is true
true && y >= x && true
//all trues in a conjunction may be ignored
y >= x
```

Figure 4.9: The simplification process.

4.2.1 Counting variable

A counting variable is an integer that is used within some loops. It is the variable which is changed each iteration, and once it reaches a certain value, the loop is exited. To find counting variable candidates (CVC), the loop condition is analysed to look for integer variables in integer inequalities. For example, in Figure 4.10, there are two integer inequalities. In these two instances, the CVCs are *i* and *r*.

```
while i < |a| && r >= 0:
```

Figure 4.10: An example loop condition.

4.2.2 Direction

Once the CVCs have been found, their direction needs to ascertained. To do this, the loop body is analysed to look for assignments to them. It is important that the CVC is consistently incremented or decremented by 1 every iteration of the loop, as it is generally used in iterative analysis of or assignment to an array. The loop direction is the change of CVC, i.e. 1 or -1. If it is not one of those, it is removed as a candidate.

In addition, the condition which provided the CVC is checked to see whether the loop direction matches the bound. With a loop direction of 1, the CVC would need to be less than, or less than or equal to, the other value. With a loop direction of -1, the counting variable would need to be greater than, or greater than or equal to, the other value. If the condition doesn't match the direction, the counting variable is removed as a candidate.

For example, in Figure 4.11, the counting variable candidates are *i*, *j*, *k*, and *l*. It can be seen that *i* is incremented by 1 each iteration of the loop, so its direction is 1. It can also be seen that *j* is decremented by 2 each iteration, so it is removed as a counting variable candidate. The variable *k* is decremented by 1. However, this is not guaranteed to occur each iteration, so *k* is also removed as a candidate. *l* is decremented by 1 each iteration, but its direction is not correct for its condition, so it too is removed as a candidate.

4.2.3 Loop bounds

After the candidates and their directions have been established, a loop invariant can be generated from the upper and the lower bounds of the CVC in the loop. This is expressed in the form *lower <= CVC && CVC <= upper*.

When the loop direction is 1, the lower bound is the value of the CVC on entry to the loop, and the upper bound is the value of the CVC on exit. When the loop direction is -1, the lower bound is the exit value, and the upper bound is the entry value.

```
function loop(int[] a):
    int i = 0
    int j = 10
    int k = |a| - 1
    int l = 5
    while i < |a| && j > 5 && k >= 0 && l <= 10:
        i = i + 1
        j = j - 2
        if a[k] > a[i]:
            k = k - 1
        l = l - 1
```

Figure 4.11: A nonsense function to show loop directions.

Entry value

The tool analyses the CVC's initial value, and any changes made to it before the loop. For example, in Figure 4.11, i's value on entry to the loop is 0. i's direction in the loop is 1, so the lower bound of the loop is $i >= 0$.

Exit value

On exit from the loop, the loop's condition will have been negated. As the CVC's value only changes by 1 each iteration, and its direction is known, its value on exit can be ascertained from the condition expression which provided it. For example, in Figure 4.11, when looking for i's upper bound, the expression $i < |a|$ is analysed. It is known that on the previous iteration of the loop, the expression held. Now, with i having been incremented by 1, it no longer does. It can be inferred that $i < |a| + 1$, or, $i <= |a|$, i.e. the upper bound.

It should be noted that loop could be exited due to another part of the condition not holding. However, even if this does happen, the CVC will still be within the bound of its derived exit value.

Bounds verification

There is the possibility that the entry value of the CVC is outside of the bound derived from the exit value, and that the loop is never entered. This means that the bound is invalid. To account for this, the upper and lower bounds are analysed to check that the lower is, in fact, lower than the upper. If it can be ensured, the bounds are added to the list of validated loop invariants. If it can't be, the bounds are each added to the list of candidate invariants.

4.2.4 Equal array length

To find arrays with equal lengths, first the loop's function's precondition is analysed to find if any of the arguments are arrays with equal length. Next, the statements preceding the loop are analysed. If an array generator is used, its length argument is looked at to see if it is based off another array's. If so, an expression is generated representing their relative lengths, and is added to a list. If an array is assigned to another array variable, i.e. array copy, an equal array length expression is added to the list.

Through the remaining analysis of the statements, if one of the array variables referenced in an expression is assigned a different array, that expression is removed from the list (and the new array assignment is evaluated as above). Thus, when the loop is reached, it is confirmed that all remaining array expressions are valid. As such, they are added to the list of validated loop invariants. For example, in Figure 4.12, there is an array generation on line 5. From this, an expression is generated, which can be seen on line 10.

```
1  function arrayAdd(int[] a, int x) -> (int[] r)
2      ensures |r| == |a| + 1
3      ensures all { k in 0 .. |a| | r[k] == a[k] }
4      ensures r[|a|] == x:
5      r = [0; |a| + 1]
6      int i = 0
7      while i < |a|
8      where 0 <= i && i <= |a|
9      where all { k in 0 .. i | r[k] == a[k] }:
10     //where |r| == |a| + 1
11         r[i] = a[i]
12         i = i + 1
13     r[|a|] = x
14     return r
```

Figure 4.12: The *arrayAdd* function with generated loop invariant in comment.

4.3 Mutating postconditions

Once the postcondition of the loop has been discovered using back propagation, it needs to be mutated. The postcondition is split into its individual conjunct expressions, and each is analysed to see if it should be mutated. If the expression is not mutated, it is added to the list of loop invariant candidates. If the expression is mutated, the mutated expression is added to the list of candidates, along with the original expression.

4.3.1 Term dropping

As each conjunct expression in the postcondition is treated as a separate candidate, this accounts for the majority of term dropping mutation. Some of this also occurs due to back propagation. For example, in Figure 4.13, back propagation has determined that the postcondition conjunct expression on line 3 is not required. As true is always returned after the loop, *!r* is false, and the logical implication will always hold (see Section 4.1.8 for more details).

4.3.2 Constant relaxation

If the analysed expression is a universal quantifier, it is considered for mutation by constant relaxation. For this to occur, there needs to be at least one valid counting variable candidate (CVC) for the loop. For each valid CVC, a loop invariant candidate is produced.

Based on the loop direction, the range of the quantifier is changed. If the loop direction is 1, the new range is from the original lower end up to the CVC. As an example, in Figure 4.14, the expression being mutated is *all {k in 0..|a| | a[k] != x}*. The mutated expression can be seen on line 7, with the range *0..i*.

```
function allLessThan(int[] a, int x) -> (bool r)
ensures r ==> all { k in 0..|a| | a[k] < x }
ensures !r ==> some { k in 0..|a| | a[k] >= x }:
    int i = 0
    while i < |a|:
    //where 0 <= i && i <= |a|
    //where all { k in 0..i | a[k] < x }
        if a[i] >= x:
            return false
        i = i + 1
    //all { k in 0..|a| | a[k] < x }
    return true
```

Figure 4.13: The *allLessThan* function, with generated loop invariants and weakest precondition in comments.

If the loop direction is -1, the new range is from the CVC, plus 1, to the original upper end. For example, in Figure 4.15, the expression being mutated is the same as in the last example. However, as the loop direction is -1, there is a different resultant mutated expression. This can be seen on line 7, with the range $i + 1..|a|$.

```
function indexOf(int[] a, int x) -> (int r)
ensures r >= 0 ==> a[r] == x && all {k in 0..r | a[k] != x}
ensures r == -1 ==> all {k in 0..|a| | a[k] != x}:
    int i = 0
    while i < |a|
    where 0 <= i && i <= |a|:
    //where all {k in 0..i | a[k] != x}
        if a[i] == x:
            return i
        i = i + 1
    return -1
```

Figure 4.14: The function *indexOf*, with generated loop invariant in comment.

4.4 Verification

Once loop invariant candidates have been generated for all loops, the tool attempts to verify which ones are valid. Firstly, all of the loop invariants which were validated through static analysis are added to the program. Next, all possible combinations of candidates, including none, are tested by temporarily adding them to the program, and putting the program into the Whiley theorem prover. This continues until the program successfully verifies with a combination, or all combinations fail. The number of possible combination is 2^n, where n is the number of candidates. For example if there are three candidates, x, y, and z, there are eight possible combinations: $\{\}$, $\{x\}$, $\{y\}$, $\{z\}$, $\{x, y\}$, $\{x, z\}$, $\{y, z\}$, and $\{x, y, z\}$.

```
function lastIndexOf(int[] a, int x) -> (int r)
ensures r >= 0 ==> a[r] == x && all {k in r + 1..|a| | a[k] != x}
ensures r == -1 ==> all {k in 0..|a| | a[k] != x}:
    int i = |a| - 1
    while i >= 0
    where -1 <= i && i < |a|:
    //where all {k in i + 1..|a| | a[k] != x}
        if a[i] == x:
            return i
        i = i - 1
    return -1
```

Figure 4.15: The function *lastIndexOf*, with generated loop invariant in comment.

Chapter 5

Evaluation

The generality, usefulness, and runtime of the program were evaluated using a test set of Whiley programs. The backwards compatibility was evaluated using a validation set. This section describes these evaluations and their results.

5.1 Methodology

The test and validation sets were created from the Whiley test suite. This is a collection of 604 Whiley programs which are used in the development of Whiley itself. Each program is aimed at testing a specific aspect of Whiley, such as functions, postconditions, properties, or while loops. Across the suite there is a good representation of Whiley's various components.

A simple tool was developed which went through all of the test suite programs and found the ones with while loops and loop invariants. There were 92 of these. These files were the validation set. A JUnit test was written to make sure the programs still verified when run through the loop invariant generator. The test set used the same files as the validation set, however the loop invariants were removed from all of them. Another JUnit test was written to run the loop invariant generator over the programs in the test set to see if it could generate the required loop invariants for verification.

The test set provided a range of required loop invariants. Of the 92 programs, 48 had postconditions and 44 did not. As the Whiley test suite is used for testing Whiley itself, there were a number of programs aimed at various edge cases. This allowed for a more thorough testing of the loop invariant generator, as it wasn't only dealing with predictable programs.

5.2 Results

The loop invariant generation tool was able to make 68 of the 92 programs in the test set verify. 34 out of the 48 programs with postconditions were verified, as well as 34 of the 44 programs without. A breakdown of the number of invariants generated in the process can be seen in Figure 5.1 on page 24. Each column in the graph represents the invariants generated for a program in the test set. Invariants are split into three categories: validated (blue), accepted (green), and rejected (red).

Validated invariants are those generated and validated by static analysis. Accepted invariants are generated candidates that have been verified by the Whiley theorem prover. Rejected invariants are candidates which were not verified. A red dot is displayed above programs which failed to verify. It should be noted that, if a program failed to verify, all candidates are rejected, as the Whiley theorem prover verifies a whole program and not specific invariants.

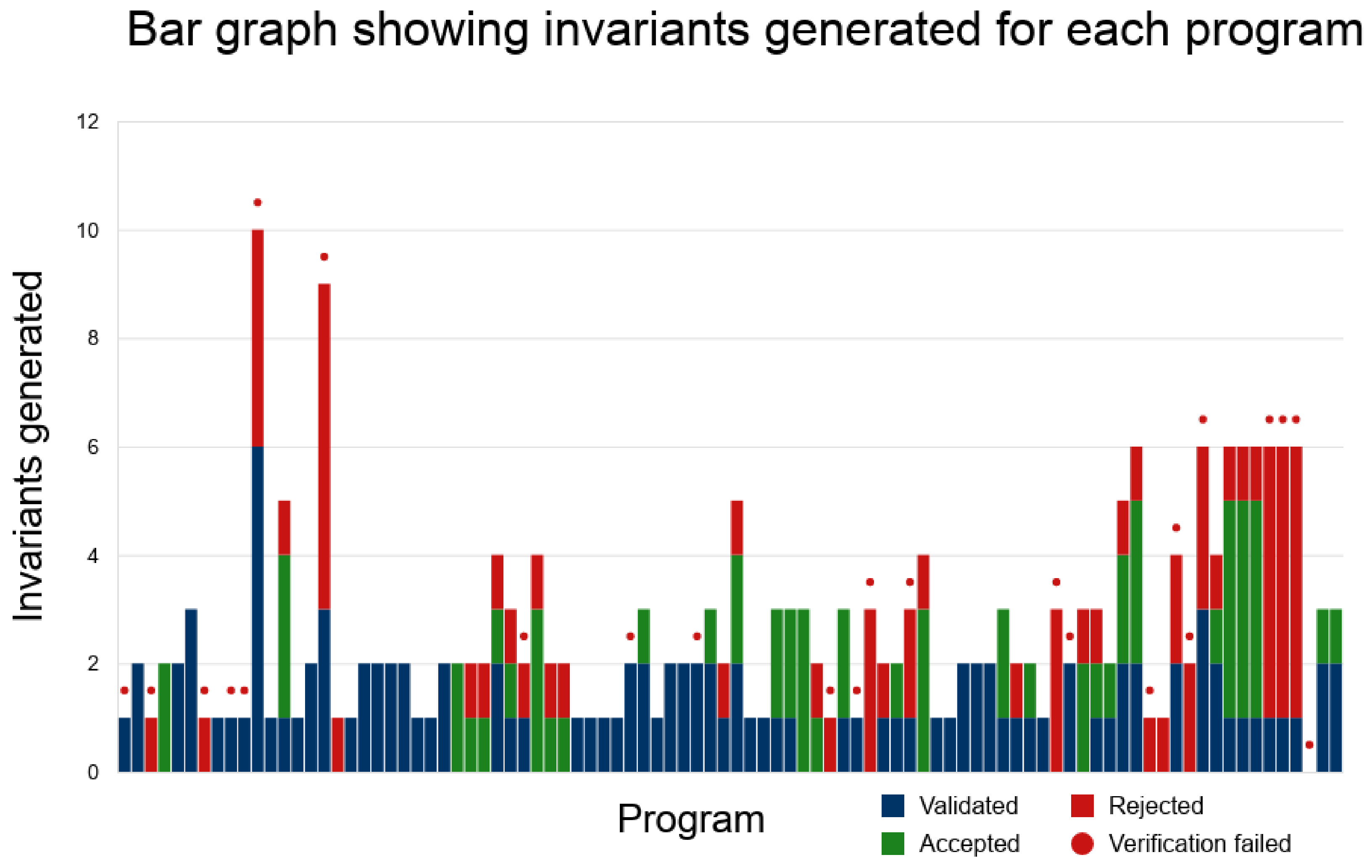

Figure 5.1: Bar graph showing invariants generated for each program

Figure 5.1 shows that the majority of programs had validated loop invariants. Static analysis generated loop bounds for most programs, and many also had equal array lengths. Another observation is that programs with a larger number of candidates tend to fail verification. This is most likely due to the fact that they are more complicated problems with greater requirements. However, programs with candidates that did verify tended to have a larger portion of candidates accepted than rejected. This suggests that the systems in place for limiting the number of candidates are effective.

The loop invariant generator provides a good balance between generality and usefulness. The majority of simple, common style programs in the test set verify, though there is room to improve (see Section 5.4). The loop invariant generator was also able to make numerous complex programs in the test set verify. This means that the tool has applicability across a range of programming styles and complexities.

The loop invariant generator was only able to verify 91 of the 92 programs in the validation set. The one failure was due to a bug with loop invariant conjunctions in Whiley which meant that a generated and validated loop invariant was considered invalid. This means that the backwards compatibility requirement for this project is not met.

5.3 Time taken

The loop invariant generator was run over each program in the test set, and the runtime was recorded. This was done ten times, and the mean time was calculated for each program (t_{lig}). The same was done for the Whiley Theorem Prover over the validation set, i.e. how long a program would normally take to verify if the programmer supplied all the invariants (t_{norm}). The runtimes of both were compared in various ways, as can be seen in Figure 5.1.

Comparison	$\Sigma t_{lig} / \Sigma t_{norm}$	$mean(t_{lig} / t_{norm})$	$median(t_{lig} / t_{norm})$
Result (3sf)	8.92	5.02	1.95

Table 5.1: Comparison of runtimes.

There was not a huge correlation between the comparative runtime of the loop invariant generator over a program (t_{lig}/t_{norm}) and whether the program was verified (see Figure 5.2). The exception was for the programs where $t_{lig} < t_{norm}$, which had a verification rate of around half that of the rest of the programs. This is most likely due to the Whiley Theorem Prover quickly rejecting these programs. The two programs with the greatest comparative runtimes (62.3 and 52.6 (3sf)) both failed to verify.

Range (t_{lig}/t_{norm})	< 1	1 - 1.3	1.3 - 2	2 - 3	3 - 8	> 8
No. Programs	18	15	15	15	16	13
No. Verified	8	14	10	14	11	11

Table 5.2: Comparison of runtimes.

The vast majority of the programs are run in an acceptable time frame. Problems only occur when the program is complex. However, there is room to improve the runtime. Most of the runtime is due to the Whiley Theorem Prover, so if more candidates can be validated, the runtime could be reduced considerably. This could be done by using another round of static analysis after postcondition mutation to attempt to validate target types of candidate. One example is using methods similar to Barnett's iterative array assignment [6] to verify quantifiers. In addition, this static analysis could be used to rule out candidates which can be found to definitely not be valid.

5.4 Problems encountered

In addition to evaluating the success of the tool on the number of test programs it was able to get to verify, it is important to understand why it wasn't able to get the others to. Another tool was created which printed the loop invariants generated for the failing programs. It also analysed the full version of the programs, before they had their invariants removed, to print out the invariants required for verification. This provided a good breakdown of the shortcomings of the tool. Following is a discussion of the main reasons that programs failed to verify.

5.4.1 Inferring type requirements

In Whiley, types may be declared which are a constrained form of another type. This can be seen in Figure 5.2 on line 1. Whiley also has implicit casts between these constrained types and the original. However, when verifying casts to the more constrained form, it needs to be proven that the value of the variable is within the constraints.

For example, on line 13 of Figure 5.2, the array of ints, *rs*, is returned. However, the return type of the function is an array of nats. Therefore, it needs to be confirmed that all of the values in *rs* are greater than or equal to 0. This is done by the loop invariant on line 9. This type of loop invariant is not generated by the tool, as analysis of types was not investigated. This could be implemented by identifying implicit casts as part of back propagation, and add the appropriate condition to the *wp*.

```
1  type nat is (int x) where x >= 0
2
3  function extract(int[] ls) -> nat[]:
4      int i = 0
5      int[] rs = [0;|ls|]
6      while i < |ls|
7      where i >= 0
8      where |rs| == |ls|
9      where all { j in 0..|rs| | rs[j] >= 0 }:
10         if ls[i] >= 0:
11             rs[i] = ls[i]
12         i = i + 1
13     return rs
```

Figure 5.2: The *extract* function.

5.4.2 Improved static analysis

The static analysis done by the tool looks for specific expected patterns, and does not allow for much deviation. There are a number of places where the static analysis could be made smarter. This could make use of a number of lessons learnt from the back propagation, especially substitution.

For example, in Figure 5.3, the required invariant can be seen on line 8. This would normally be generated by finding the value of *i* on entry to the loop. However, as the value of *i* is altered on line 6, the tool is unsure what it's value is. This can be fixed by substituting the change into the *i*'s value, rather than removing *i* as a CVC. This could either be left as is after the substitution, as *0 + 1*, or a basic evaluator could be employed, similar to on the fly simplification.

```
 1  function sum(int[] xs) -> int
 2  requires |xs| > 0:
 3      int i = 0
 4      int r = 0
 5      r = r + xs[i]
 6      i = i + 1
 7      while i < |xs|
 8      where i >= 0:
 9          r = r + xs[i]
10          i = i + 1
11      return r
```

Figure 5.3: The *sum* function.

Chapter 6

Conclusions

The loop invariant generator provides a useful and general tool which limits the need for programmers to specify loop invariants. Many common and trivial loop invariants are generated, as well as a number of more complex ones. This means the tool can generate invariants for, and verify, a range of programs. The tool mostly runs in an acceptable amount of time, though there is room for improvement.

Furia and Meyer's techniques of generating loop invariants by mutating postconditions [4] provided a good basis for the project. Most notably, their constant relaxation technique could generate one of the most commonly used loop invariants, a quantifier covering the range that the loop has covered. This is a more effective way of doing this than static analysis, as it directly relates to the postcondition for easier verification.

However, more work was required on top of Furia and Meyer's to make the tool useful. Their methods involved making a lot of educated guesses, and there were many areas where that was not the optimal way to go about loop invariant generation. By using static analysis of the loop body and the preceding statements in the function to guide the mutation, fewer guesses had to be made about what the loop did. In addition, static analysis could discover invariants that were required for clean execution, such as loop bounds.

Back propagation was an incredibly effective technique for extending the mutation of postconditions. It removed a large amount of the guesswork involved. The variables used in the loop invariants could be discovered, which was one of the limitations of Furia and Meyer's work. By discovering the postcondition of the loop rather than the function, what the loop achieved was much easier to guess. One of the useful effects of the back propagation was term dropping by path taking. When there are multiple return statements in a function or method, the postcondition must cover all of them. However, the loop does not always need to work for every return statement, and back propagation is able to discover which expressions in the postcondition are required.

6.1 Future work

Though the tool is able to generate a wide range of loop invariants in a reasonable time, it has a number of limitations. Many of these have been mentioned in previous sections, such as the limitations of static analysis in Section 5.4.2. In addition to these, there are two large scale improvements which could be made to the program.

6.1.1 Back propagation

Back propagation can be improved to better account for method calls and while loops. It should use a number of techniques to work out what they do, and recognise when it isn't able to. For method calls, the postcondition of the method could be analysed to see if a value, or at least a constrained range of values, can be gleaned.

For while loops, the back propagation could be shaped by the static analysis. If the loop direction and bounds are known, it can be determined what happens with the loop. This would help make back propagation possible for nested loops.

If back propagation can be improved sufficiently, it can also be used to find the precondition for a function. Instead of stopping when the bottom of the loop is encountered, it could continue to the top of the function.

6.1.2 Decomposition of program

As the whole program is verified at once, it is difficult to analyse what is wrong. It can also increase verification time, as everything must be verified. One way to overcome this issue is to break the program down into smaller programs, and verify each part separately. Functions which are independent of other parts of the program can be verified independently. This would allow for easier identification of where issues arise. More importantly, it would show what does work, even when some parts don't.

Bibliography

[1] C. A. R. Hoare, “An axiomatic basis for computer programming,”

[2] D. J. Pearce, “Whiley overview,” 2017. Available: http://whiley.org/about/overview/.

[3] C. Hawblitzel, J. Lorch, M. Moskal, and N. Swamy, “Dafny: A language and program verifier for functional correctness,” 2008. Available: https://www.microsoft.com/en-us/research/project/dafny-a-language-and-program-verifier-for-functional-correctness/.

[4] C. Furia and B. Meyer, “Inferring loop invariants from postconditions,” in *Fields of Logic and Computation*, pp. 277–300, Springer, 2010.

[5] D. J. Pearce, “Efficient value semantics for Whiley,” 2011. Available: http://whiley.org/2011/12/13/efficient-value-semantics-for-whiley/.

[6] D. Barnett, “Generating simple loop invariants for Whiley,” Victoria University of Wellington, 2017.

[7] J. Galeotti, C. Furia, E. May, G. Fraser, and A. Zeller, “Inferring loop invariants by mutation, dynamic analysis, and static checking,” in *IEEE Transactions on Software Engineering, vol. 41, no. 10*, pp. 1019–1037, 2015.

[8] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll, “Beyond assertions: advanced specification and verification with JML and ESC/Java2,” in *FMCO*, pp. 342–363, 2006.

[9] M. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, “Dynamically discovering likely program invariants to support program evolution,” in *IEEE Transactions on Software Engineering, vol. 27, no. 2*, pp. 99–123, 2001.

[10] S. Chandra, S. J. Fink, and M. Sridharan, “Snugglebug: A powerful approach to weakest preconditions,” in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.