Formal Verification with Dafny

Cedric Abano cabano@pennington.edu
Gordon Chu c.gordo97@gmail.com
Gabriel Eiseman hacatu5000@gmail.com
Justin Fu justinfu21@gmail.com
Tiffany Yu tyu@pingry.org

Abstract

Computer programming and software engineering have found their way into many facets of the current world. In certain industries, correct computer programs are needed for basic operation. This absolute correctness could be proven by brute force, which involves passing in all possible test cases through a program. This technique is largely impractical, especially for complex programs that do not have a bounded input range. Alternatively, a more theoretical and feasible approach can be taken, using mathematics and logical reasoning to prove the correctness of a program. Formal verification is a field of computer science dealing with the construction of mathematical proofs of programs' correctness. This paper explores the current state of formal verification through Dafny, a static program verifier created by Microsoft Research. Dafny was used to implement five common algorithms and compared with lower-level methods of formal verification. Although there were qualities about Dafny that were difficult to work with, ultimately self-verification of programs has positive prospects for future developments.

1. Introduction

1.1 Why is it important to write correct programs?

With the growing reliance on software for functionality in various industries, computer programs have become increasingly involved in operations that are responsible for the wellbeing of individuals. This enlarged scope of influence entails a moral obligation for programmers to write code that works correctly under all conditions. Programs must be correct for all inputs, not just expected ones, as even the smallest error can have disastrous consequences. In 2003, a flawed assumption in FirstEnergy's estimator program modeling the North American electric grid led to a blackout that lasted 26 hours and affected 50 million persons in Ohio, Michigan, Pennsylvania, New York, Vermont, Massachusetts, Connecticut, New Jersey, and Ontario.[1, 2] In 2007, Qantas flight 72 from Singapore to Perth unexpectedly nose-dived twice due to a faulty stabilization algorithm that failed to consider two consecutive sensor errors, injuring 119 passengers and crew members.[3] These incidents are just a few instances from a long list of occurrences where incorrect code crippled the integrity of a large scale system, demonstrating the importance of writing code that is correct in all cases at all times.

1.2 The Software Development Cycle

While there are several different models of the software development cycle, the steps in each can be generalized to four main categories.[4]

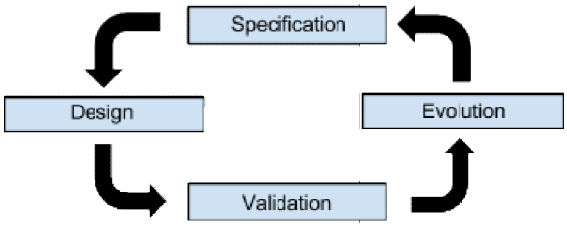

Figure 1. The Software Development Cycle

During the specification phase of software engineering, data is gathered about the problem to be solved. Developers describe all aspects of the issue formally and gain an understanding of the subtle intricacies accompanying it. A written specification is often created at this stage, describing how the program is to behave under various conditions.

After the specification is completed, developers proceed to the design phase where they consider various algorithms to solve the problem efficiently. This can involve using pre-existing algorithms or creating entirely new algorithms to approach to the problem. The majority of coding is done during this stage.

Following the software design phase is the validation phase. During validation, developers write test code to ensure their program works. Such code usually consists of testing standard input and edge cases.

Once a program passes its validation tests, the development cycle is repeated in order to refine the program. The code is improved by repeatedly reevaluating its specification, design, and validation.

A final note on the software development cycle is that the model shown above is not discreet. These phases are loosely defined and occur on a continuum, commonly overlapping in the real world.

1.3 Limitations of Validation

The primary component of the software development cycle this paper will scrutinize is validation. When writing tests, programmers often use ad hoc logic to convince themselves that their code is correct. It is not feasible to test all inputs for nontrivial programs, and thus validation tests usually check a limited number of cases meant to represent a sample of probable inputs. If a programmer's code passes these tests, they believe that their implementation is correct, even though there is insufficient evidence to make such a claim.

Furthermore, debugging, which is the process of fixing erroneous programs in order to satisfy its tests, is a time-consuming and costly process. Programmers spend as much as 49.9% of their time debugging their code, accounting for a total of $156 billion worldwide.[5]

1.4 Formal Verification

Various resources are funneled into writing correct programs, yet the standard method used in software development lacks mathematical rigor. An effort introduced to remedy this discrepancy is formal verification. Formal verification is a systematic approach to proving that a program is correct with regards to its specification.[6] This paper will explore the methodology of formal verification and demonstrate its current capabilities in determining a program's correctness through a cutting-edge verification tool called Dafny.

2. Background

Up to this point, the word "correct" has been used informally to describe programs. The following sections will develop a precise definition of correctness, building up the definition through the mathematical notation of propositional logic and employing it in Dafny.

2.1 A Brief Introduction to Propositional Logic

It is often difficult to know for certain, that a given algorithm is correct, especially if that algorithm is particularly lengthy. In this situation, mathematics provides some certainty in determining an algorithm's correctness, specifically with a branch of mathematics known as propositional logic.

Like all forms of logic, propositional logic concerns mathematical proofs. Propositional variables, or *atoms*, denoted by capital letters (A, B, X) represent true or false conditions and make up statements, or *formulae*, in propositional logic. More complex conditions are derived from these simple statements using relationships, or *connectors*, such as a disjunction, conjunction, negation, etc. (Figure 2).

Conjunction: A ∧ B denotes *A AND B*
Disjunction: A ∨ B denotes *A OR B*
Negation: ¬A denotes *NOT A*
Implication: A⇒B denotes *A IMPLIES B*, equivalent to *¬A ∨ B*
∀x denotes FOR ALL "X"
∃x denotes There EXISTS a variable X

Figure 2. Notation of Propositional Logic

The formulae created by these atoms and connectives can be evaluated, and the *truth value*s of these formulae, can be found by trivially following the relationships that the connectives state. In some cases, a truth table may be used to determine all possible truth values of a formula, given different combinations of truth states of each atom (Table 2).

True is denoted with a *1*
False is denoted with a *0*

A	B	¬A	A ∘ B	A⇒B
0	0	1	0	1
0	1	1	0	1
1	0	0	0	0
1	1	0	1	1

Figure 3. A truth table a few formula

Mathematical principles of substitution, double-negation, commutativity, associativity, and distributivity can aid in the construction or deconstruction of propositional formulae. Moreover, with the use of these principles, complex formulae, such as ¬(A ∘ B) ∨ C ⇒D, can be broken down into more fundamental formulae, A ∨ B ∨ ¬C ∨ D, and vice versa.[7]

The principles of propositional logic, while having applications in mathematics, also have a relevant application in computer science. Propositional logic forms the basis of automatic theorem provers and other code verification tools.

2.2 Specifications, Preconditions and Postconditions

Concerning correctness from a programmer's perspective, specifications are used to define a program's behavior with particular sets of preconditions and postconditions. Preconditions are assumptions that must be true when a program is invoked and postconditions are claims that must hold true after a program terminates. For the purposes of verification, it is important to write meaningful specifications that characterize the behavior of a program. Postconditions should reflect a defining property or relationship of a program's manipulated data after it is executed and preconditions should ensure that these postconditions could be reached. Specifications are an integral component of formal verification, used to establish the definition of correctness.

2.3 Hoare Logic and Defining Correctness

In the late 1960s, Sir Charles Antony Richard Hoare formalized a model for reasoning about the correctness of programs. The model, which came to be known as Hoare logic, outlined a schema for proving that a program implementation adheres to its specifications. The general methodology behind a Hoare logic proof of a program is to:

1. Start with a set of axioms that are assumed to be true
2. Using propositional logic, develop a series of claims from these axioms and the computer code of the program
3. Demonstrate that the program's postconditions are true at the end of the program

Through this application of propositional logic to computer programs, Hoare helped to define correctness by connecting a program's preconditions (*P*), code (*Q*), and postconditions (*R*).

"If the assertion *P* is true before the initiation of a program *Q*, then the assertion *R* will be true on its completion."[8] This relationship is called a *Hoare Triple*, and is denoted as:

$$P\ \{Q\}\ R$$

Figure 4. Hoare Triples

A rigorous definition of correctness can be obtained from the definition of a Hoare triplet.

A program that executes code *Q*, with a specification consisting of a set of preconditions *P* and set of postconditions *R*, and terminates[*] is correct with respect to its specification {*P*, *R*} if:

$$P \circ Q \Rightarrow R$$

Figure 5. Defining Correctness

[*] Hoare logic can only prove the partial correctness of a program, as opposed to the total correctness of a program. A Hoare triplet makes no claim that the code *Q* terminates. If the program never terminates, then it is impossible to assert the postconditions *R*. Therefore, the definition of correctness has been amended to account for this oversight. Hoare explained this shortcoming himself in his paper [5] (refer to General Reservations section of his paper).

The advantage of this definition is that if a program adheres to it, then it will work as the specification states *in all cases at all times*. Proving a given program correct through this model eliminates the need for test code and debugging, saving valuable resources.

2.4 Correctness versus Intent

It is important to make a distinction between correctness and intent. While a program may be correct with respect to a certain specification, if that specification is written in a manner that does not guarantee the programmer's intended outcome, then the program, although "correct," may not perform the intended task. Thus, in order to write correct programs, it is essential to begin with appropriate specifications, which must outline the program's preconditions and postconditions such that they properly express the intent of the programmer.

Below is a simple program, written using basic Dafny constructs, that is correct with respect to its specifications but not intent. This also serves as a cursory introduction to Dafny syntax, which will be covered later in more detail.

```
//this should subtract two integers
method Subtract(a: int, b: int)
returns(c: int)
  ensures c == a + b;
{
  return a + b;
}
```

Figure 6. Correct with respect to a specification but not intent

Figure 6 presents a method called *Subtract* that is intended to subtract two numbers as described in the comment[†] on the first line. The second line of the code states that the *Subtract* method takes an input of two integers *a* and *b*, and outputs, or *returns*, a third integer *c*. On the third line, the *ensures* keyword specifies a postcondition that the integer *c*, which the *Subtract* method returns, must be equal to the sum of the two parameters *a* and *b*. The body of the method, which is inside the curly braces, states that the method will return the sum of *a* and *b*. When run through Dafny, this program verifies; however, there is a glaring disparity between the intent of the method and its specification. The method is supposed to return the difference of two numbers as indicated by the comment before the method, yet it returns their sum. This demonstrates that while a program may be correct with respect to its specifications (in this case verified by Dafny), that it can still not perform as the programmer intended it to. Although this example of the *Subtract* method is a trivial one, aligning specifications with intent is critical to preventing logic errors which can be difficult to detect in a large codebase.

2.5 Why Dafny?

Using Hoare logic or formal verification can at times involve rather lengthy, sometimes handwritten proofs. In an attempt to mitigate some of the tedious work involved with mathematically verifying the correctness of a program, Dafny was created to automate some of these processes. Dafny is an automated verification language meant to aid software engineers in the design of correct code.[9] There are very few other tools which offer similar functionality; SLAyer is another Microsoft project to verify code, but it is more low-level than Dafny, working directly with the heap. Using familiar C-like syntax and annotations, Dafny provides an easily understandable platform for the user to write mathematically correct programs.

Behind the scenes, Dafny converts programs for its users into the mathematical expressions of Hoare logic with the aid of a program called Boogie, and then it sends the code to an automatic proving program called Z3. This conversion process benefits Dafny users in eliminating the need to write out long proofs in confusing notation while still being able to verify their programs. As a result, Dafny requires that programmers use a strict form of syntax in order to properly convert their code into mathematical expressions. Thus, Dafny proves its programs

[†] Comments in Dafny are denoted as lines that begin with two forward slashes "//" and do not affect a program's behavior. A comment is used here merely to clarify the human intent.

for every possible input, not just a few as a traditional test suite which just runs the program would. Proving a program meets its specification is not possible in the general case, however, so Dafny might fail to verify a correct program. If this were possible, Dafny could be used to verify a program that proved P = NP or some other unsolved mathematical problem.[9]

2.6 Dafny Syntax

As stated above, Dafny uses syntax more similar to other programming languages, but it requires a few unique tools to aid in formal verification. The most important of these are explained below, and the relevant keywords along with other keywords are glossed in Figure 7.

Annotations are one tool that Dafny provides its users to aid in the implementation of a program. Annotations are statements that are not a part of an actual program but provide specifications and information about the methods in the programs; in Dafny, however, they serve more than a superficial purpose. By using annotations, Dafny is able to verify the correctness of a program more directly by considering the *annotations* within the functions.

Two of the most fundamental annotations in Dafny are the preconditions and postconditions. In most programming languages, preconditions and postconditions are typically stated in comments, but they do not strictly bind the method. In Dafny, preconditions, expressed with the keyword *requires*, and postconditions, expressed with the keyword *ensures*, are taken into consideration when proving whether the method in question is correct. Preconditions must be met when a method is called and can prevent errors that may arise within the method if they were not set forth previously. Postconditions specify truths that must follow the program execution. Given by the programmer, these postconditions allow Dafny to check if a program is correct.

In order to help prove methods, Dafny also utilizes assertions, given with an *assert* statement, which are expressions within the method's code that, when reached, must hold true for the method to be correct, no matter what the variables actual values are. These assertions allow Dafny to verify certain sections of the method, which then assists in proving the postcondition.

Another fundamental annotation in Dafny is the loop invariant. Loop invariants, given with an *invariant* statement, are expressions that must hold true before the initialization of the loop, throughout all iterations, and upon termination. At the end of the loop, invariants also provide information that assist in verifying the overall method and the postconditions. Invariants are often restatements of the postconditions in terms of intermediate variables, and by outlining these steps, Dafny is able to verify that these statements which hold constant for the duration of the loop, implying the postcondition.

Besides annotations functions and predicates also serve as tools for program verification. Functions consist of only one expression and are used only as tools to help verify methods. In addition, unlike methods, functions can be used directly in annotations. Predicates are functions that return a boolean value.[10]

bold denotes keywords
italics represent custom code

requires *boolean expression (preconditions)*
ensures *boolean expression* (*postconditions)*
assert *boolean expression*
invariant *boolean expression (that holds before, during, and at the conclusion of a while loop)*
decreases *ranking function*
forall *counter* :: *range* ⇒ *boolean expression*
exists *value* :: *range* ⇒ *boolean expression*
reads *mutable object*
modifies *mutable object*

Figure 7. Dafny Verification Syntax

3. Method

In order to prove a program correct, Dafny checks that two conditions are met: that the user's specifications are satisfied and that all loops must terminate. For this project, a verification process known as the technique of

weakest preconditions was used to create the annotations that Dafny required to prove implementations of several algorithms. This technique looks at the program's postconditions and traces the code backwards, logically deducing the conditions that must be met in order to verify the postconditions. Building upon each step, this process will eventually lead back to the beginning of the program and the minimum conditions required for it to run correctly, which are the weakest preconditions.

In addition, ranking functions must be created to ensure loop termination. Also known as "variants" because they change between loop iterations, ranking functions are expressions that show that some value decreases within the loop. The value of this expression must decrease at least by one and is bounded by the set of natural numbers (positive integers). By verifying that such a function exists, Dafny can ensure that the loop eventually terminates.[11]

3.1 Factorial

The first algorithm implemented was the factorial algorithm, which given a nonnegative integer *x* computes *x!* and returns the value. First, a basic implementation of factorial was written in Dafny without any annotations (Figure 8).

```
method Factorial(x: int) returns (y: int)

{     y := 1;
      var z := 0;
      while(z != x)
      {
            z := z + 1;
            y := y * z;
      }
}
```

Figure 8. Basic Implementation of Factorial Algorithm in Dafny

In order to verify this implementation of the factorial algorithm in the method *Factorial()*, the function *Fact()* was created and implemented correctly to return *x!* (Figure 9).

```
//correctly implements x!
function method Fact(x: int): int
      requires x >= 0;
{
      if x < 2 then 1 else x * Fact(x - 1)
}
```

Figure 9. Implementation of Function *Fact()*

The correct postcondition of the method *Factorial()* is *y == x!* or *y == Fact(x)*, properly expressing the program's intent to compute the factorial of the given integer. This allows the following postcondition to be added (Figure 10):

```
method Factorial(x: int) returns (y: int)
      ensures y == Fact(x);
{                                          postcondition
      y := 1;
      var z := 0;
      while(z != x)
      {
            z := z + 1;
            y := y * z;
      }
}
```

Figure 10. Postcondition of *Factorial()*

This postcondition must be fulfilled at the end of the program, after the loop terminates. When the loop terminates, the loop condition (*z != x)* is no longer true; however, this negation is not enough to satisfy the postcondition. Thus, the appropriate expression *y == Fact(z)* is added as a loop invariant, which must be true at the end of the *while* loop. Thus, the conditions that must be true at the end of the program are (Figure 11):

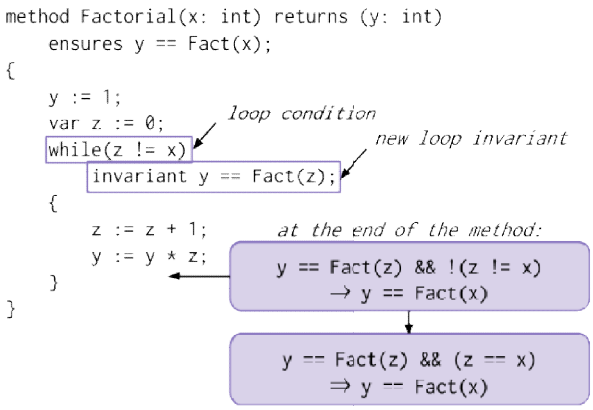

Figure 11. Invariant *y == Fact(z)* at End of Loop

Loop invariants must hold true throughout each iteration of the loop, in which the variable *z* is a counter that increments from *0* to *x*. The loop is traced backwards step by step until the beginning of the loop is reached. Whenever a variable is reassigned, the reassigned value of the variable is substituted for all instances of the original variable in the invariant expression. The conditions true at the beginning of the loop (which are the loop invariant and the loop condition) must be able to prove that the expression with substituted values is true, showing that the loop invariant will hold at the beginning of the loop (Figure 12).

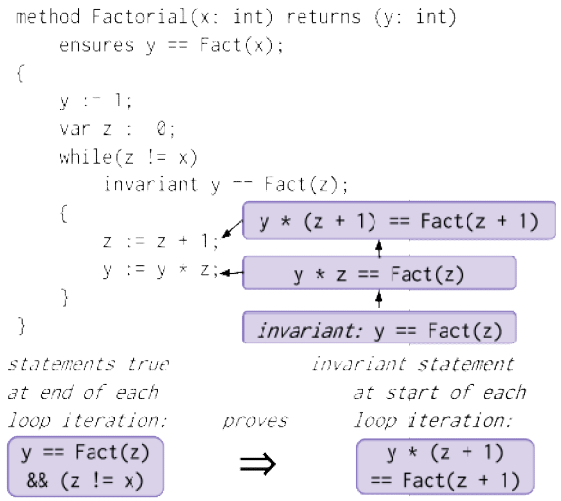

Figure 12. Invariant *y* == *Fact(z)* throughout Loop

The loop condition is extraneous information, but these statements algebraically prove the invariant to hold true throughout each iteration of the loop.

In addition, loop invariants must hold true before the loop, regardless of whether the loop is entered or not. The following proof checks the invariant (Figure 13):

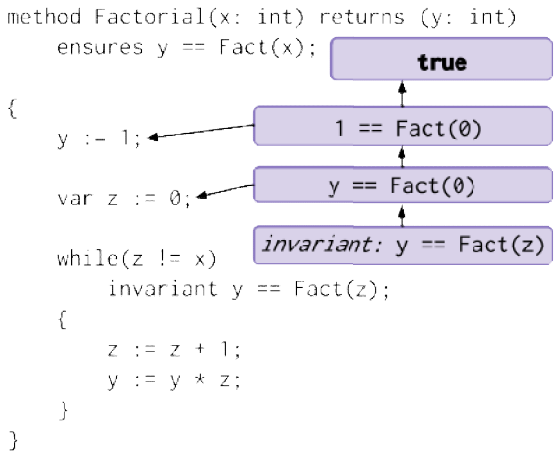

Figure 13. Invariant *y – Fact(z)* Before Loop

It is also necessary to prove termination of the loop. As the counter variable *z* increments with each iteration, the ranking function that is expressed in the loop is *x* - *z*. As a ranking function, which decreases during the loop and is bounded by 0, this statement also implies another loop invariant, *0* <= *x* - *z* or *z* <= *x*, as shown below (Figure 14):

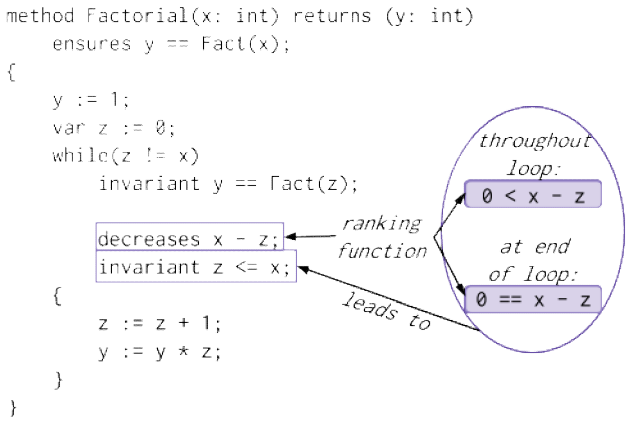

Figure 14. Ranking Function *x – z* and Invariant *z* <= *x*

This invariant is verified upon termination of the loop as a result of the loop condition (Figure 15):

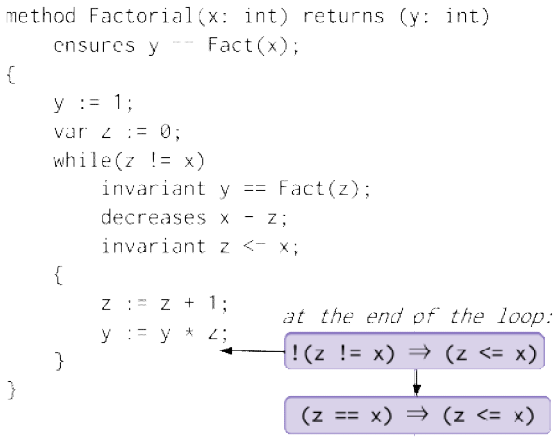

Figure 15. Invariant *z* <= *x* at End of Loop

The invariant is also proved through each loop iteration by taking the ranking function into account (Figure 16):

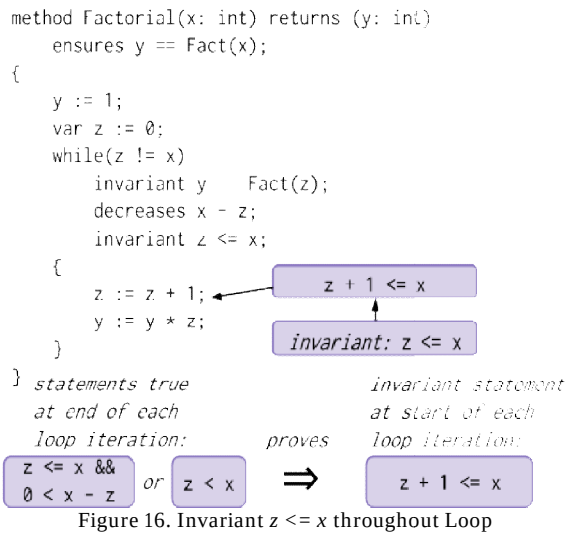

Figure 16. Invariant *z* <= *x* throughout Loop

This invariant must also hold true before the loop (Figure 17):

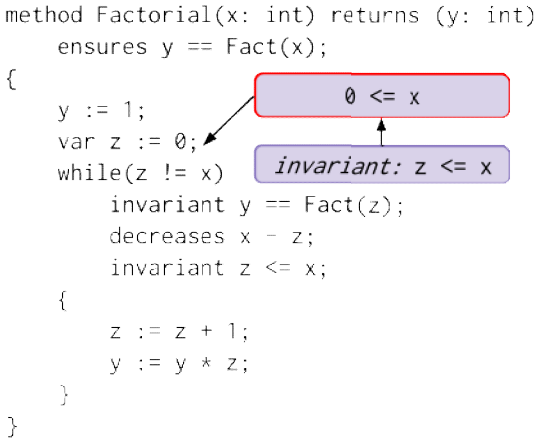

Figure 17. Invariant *z* <= *x* Before Loop

The expression outlined in red must be true for the loop invariant to hold. Thus, this final statement:

$$x >= 0$$

must be the weakest precondition of the program. Putting the above logic together in one code, the following implementation is created (Figure 18):

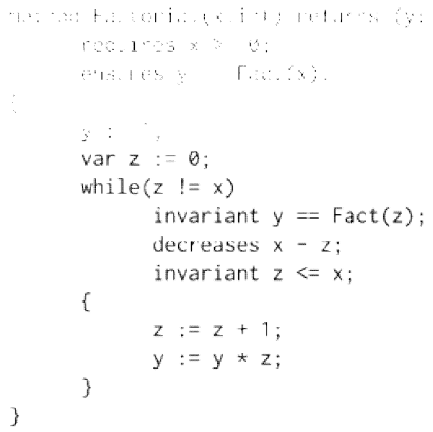

Figure 18. Complete Implementation of Factorial Algorithm

3.2 Sorting Algorithms

The following five algorithms require a significantly larger number of intermediate steps, so it is impractical to write out every step as with the factorial algorithm. However, the implementations of these algorithms were formulated with the same logical thought process. Each algorithm is explained and written out in pseudo-code below.

In addition, like the function *Fact()* used for the factorial implementation, the predicate *sorted()* was created to verify the postconditions of the four sorting algorithms (Figure 19):

```
predicate sorted(a: array<int>, low: int, high: int)
      requires a != null;
      requires 0 <= low <= high <= a.Length;
      reads a;
{
      forall j, k :: low <= j < k < high ==> a[j] <= a[k]
}
```

Figure 19. *sorted()* predicate

3.21 Selection Sort

In selection sort, the array of integers is initially traversed from the first element to the last element to find the minimum value. The minimum value is then swapped with the first element of the array, placing it in its correct position. The array is then traversed from the second element to the end, finding the minimum value in this section of the array (which is the second least value of the entire array) and swapping it with the second element of the array. The first two elements of the array are now correctly sorted. This process continues with each index except for the last of the array, sorting the elements of the array in ascending order (Figure 20).

```
method index_min (array, start):
  index = start
  min_index = start
  while index < array.length
    if array[index] < array[min_index]:
      min_index = index
      index = index + 1
  return min_index

method selection_sort (array):
  sorted_to = 1
  while sorted_to < array.length:
    swap array[sorted_to] and index_min
(array, sorted_to)
```

Figure 20. Selection Sort Pseudo-Code

3.22 Bubble Sort

The bubble sort algorithm traverses the array of integers beginning at the last index of the array. The value of the last index is compared to the value that precedes it in the array. If the preceding value is larger than the last value, then the two elements are swapped. Then, the second-to-last value is compared to the value that precedes it, and so on. In this way, the minimum value of the array ends up as the first index of the array. This process is then repeated from beginning at the end of the array up to the second element of the array. The second least value of the entire array ends up in the second index of the array, so that the first two elements are sorted correctly. The next iteration goes from the end to the third element, then from the end to the fourth element and so on, ultimately sorting the whole array (Figure 21).

```
method bubble_sort (array):
  sorted_to = 1
  while sorted_to < array.length:
    index = array.length - 1
    while 0 <= index:
      if array[index - 1] >
array[index]:
       swap array[index - 1] and
array[index]
     index = index - 1
    sorted_to = sorted_to + 1
```

Figure 21. Bubble Sort Pseudo-Code

3.23 Insertion Sort

Insertion sort begins by assuming the first element of the array is “sorted” and maintains that the first part of the array is in ascending order. Each element in the second part of the array is then inserted into the first part of the array in the correct position, increasing the length of the sorted portion of the array until the entire array is sorted. This is accomplished by comparing each element to the elements before it, shifting each element over one index to the right if the element at the index a being considered is greater than the one being read. The correct index for this element is found when it is less than the element after it and greater than the element before it. The element is then inserted into the array at this index, maintaining the sorted portion of the array and increasing its length (Figure 22).

```
method insert_sorted (index1, array):
  index2 = index1 - 1
  value = array[index1]
  while 0 <= index2 and array[index2] >
value:
       array[index2 + 1] = array[index2]
       index2 = index2 - 1
       index2 = index2 + 1
       array[index2] = value

method insertion_sort (array):
  sorted_to = 1
  while sorted_to < array.length:
       insert_sorted (sorted_to, array)
       sorted_to = sorted_to + 1
```

Figure 22. Insertion Sort Pseudo-Code

3.24 Quick Sort

Quick sort utilizes an algorithm noticeably different from selection, bubble, and insertion sort. Unlike the other sorting algorithms, quick sort relies heavily on recursion to sort the array. The algorithm first selects a pivot point, which is an element in the array. This point can be chosen at random; however, implementations may select specific pivots in order to increase the algorithm's efficiency. Then, all elements less than the pivot are placed to the left side of the pivot, and all elements equal to or greater than the pivot are moved to the right of the pivot. Once the array is divided into these two sections, the quick sort algorithm is called on the segments before and after the pivot. As quick sort repeats this process, the intervals over which the algorithm is called decrease in size, eventually reaching lengths of one or two elements. At that point, the array is sorted when put in place in respect to the pivot, and once completed for all iterations of quick sort, the array is sorted (Figure 23).

```
method partition (array, start, end):
  pivot = start
  index = start + 1
  while index < end:
  if array[index] <= array[pivot]:
    swap array[index] and array[pivot]
    pivot = index
    index = index + 1
return pivot
```

```
method quicksort (array, start, end):
  if start==end:
    return
  otherwise:
    pivot = partition (array, start,
end)
    quicksort (start, pivot)
    quicksort (pivot + 1, end)
```

Figure 23. Quick Sort Pseudo-Code

3.3 Minimum Contiguous Subarray

The minimum contiguous subarray problem is different from the previous four algorithms in that it is not a sorting algorithm. Given an array, this algorithm should locate the subarray of consecutive values within the array that produces the lowest possible sum. There are no bounds in terms of the size of the subarray: the subarray can span the full length of the array or, if all elements are positive, can contain no elements.

One way to find the minimum contiguous subarray is through brute force. By looping through the array for start and end values, the algorithm can systematically check each and every possible subarray, storing only the value of the minimum subarray checked to that point. This, however, is largely inefficient, as it allocates the computer's resources towards checking every possibility.

A more time-efficient algorithm was soon developed, commonly referred to as Kadane's algorithm. The array starts with the first element and compares it with 0. If the number is less than zero, it is stored in a temporary variable. This temporary variable is then compared with the sum of the minimum subarray to that point, originally set as zero. In the second iteration, which evaluates the second element, the temporary value is combined with the second element and compared with zero. If negative, this new value is stored in the temporary variable, which is then compared with the minimum subarray to that point.

The logic behind this is as follows: a subarray followed by a negative value has a greater sum than a subarray which contains the original subarray and that negative value, while a subarray followed by a positive value has a

smaller sum than a subarray which contains both the original subarray and that positive value. In order to identify the minimum contiguous subarray, the minimum subarray to that point must be able to accommodate additions which would decrease its value without allowing values in which would increase its value. The temporary variable, the intermediary step in Kadane's algorithm, establishes this. Beginning with the first negative number, the temporary variable stores that and any following numbers, as long as the value remains negative. If the value ever reaches zero or higher, the temporary variable reinitializes itself to zero, because any subarray containing the elements which the temporary variable encompasses cannot be the minimum. This also allows the array to add negative elements, and the final value is stored in the variable for the minimum subarray to that point (Figure 24).

```
method minimum_subarray (array):
  min_here = 0
  min_so_far = 0
for element in array:
 min_here = min (0, min_here + x)
 min_so_far = min(min_so_far, min_here)
return min_so_far
```

Figure 24. Minimum Contiguous Subarray Pseudo-Code

4. Results and Discussion

4.1 Selection Sort

This implementation of selection sort (Figure 25) has the preconditions *(P)* that the given array is not null and the length of the array is greater than or equal to one (line 2), and the postcondition *(Q)* that all the elements of the array are sorted in ascending order.

The loop invariants that must hold true so that the program is correct (*Q*) refer to the instance variables *index*, *counter*, and *minIndex*. The variable *index* must stay within the bounds of 0 and the length of the array *a* inclusive (line 10) and the variables *counter* and *minIndex* must stay within the bounds of *index* and the length of the array *a* inclusive (lines 19 and 20). In the inner while loop, as *counter* is incremented, the value at *minIndex* of *a* must be less than or equal to the value of *a* at each index between *index* and *counter*, implying that for all the values checked so far, the value at *minIndex* is the smallest (line 21). The outer loop ensures that all values of indices before *index*, which are the values that have already been checked, have been put into ascending order (line 11). In addition, all the values of *a* at indices before *index* are less than or equal to all the values of *a* at indices after *index*, ensuring that the first part of the array is sorted in relation to the second part of the array (line 12).

The crucial piece of information that Dafny needed to prove the postcondition was the invariant on line 12. Although this statement initially seemed intuitive, Dafny does not know what occurs within loops and so is unable to verify that the elements of sorted part of the array have not been rearranged. In ensuring that all the elements of the first part of the array are less than or equal to all the elements of the second part of the array, coupled with the *sorted()* invariant on line 11, Dafny can verify that the entire array is sorted and the postcondition is met.

4.2 Bubble Sort

Similar to selection sort, bubble sort (Figure 26) possesses the preconditions *(P)* that the array to be sorted is not null and has a length greater than 1. Its postcondition *(Q)* states that the input array is sorted according to the previously defined sorted predicate.

The invariants set the ranges for the counter variables *sortedUntil* and *i* to be maintained throughout the loop (lines 10 and 16). As the bubble sort implemented above bubbles smaller elements towards the left of the array, the *sorted()* invariant states that elements to the right of the counter *sortedUntil* are arranged in increasing numerical order (line 12). As the counter, *sortedUntil* increases throughout the loop, the *sorted()* invariant on line 12 gradually proves more and more of the array to be sorted, satisfying the postcondition at the end. Another invariant states that all elements to the left of *sortedUntil* are less than the elements to the right (line 11). This helps justify the truthfulness of the *sorted()* invariant on line 12.

```
method SelectionSort(a: array<int>)
  requires a != null && a.Length > 1;
  modifies a;
  ensures sorted(a, 0, a.Length);
{
  var index := 0;
  var counter := 0;
  var minIndex := 0;
  while(index < a.Length)
       invariant 0 <= index <= a.Length;
       invariant sorted(a, 0, index);
       invariant forall j, k :: 0 <= j < index <= k < a.Length ==> a[j] <= a[k];
  {
       minIndex := index;
       counter := index;
       while(counter < a.Length)
       invariant index <= counter <= a.Length;
       invariant index <= minIndex < a.Length;
       invariant forall k :: index <= k < counter ==> a[minIndex] <= a[k];
       {
         if(a[minIndex] >= a[counter])
         {
           minIndex := counter;
         }
         counter := counter + 1;
       }
       a[minIndex], a[index] := a[index], a[minIndex];
       index := index + 1;
  }
}
```

Figure 25. Implementation of Selection Sort

```
method BubbleSort(a: array<int>)
  requires a != null && a.Length > 1;
  modifies a;
  ensures sorted(a, 0, a.Length);
{
  var sortedUntil := 0;
  var i := a.Length - 1;

  while(sortedUntil < a.Length)
    invariant 0 <= sortedUntil <= a.Length;
    invariant forall j, k :: 0 <= j < sortedUntil <= k < a.Length ==> a[j] <= a[k];
    invariant sorted(a, 0, sortedUntil);
  {
    i := a.Length - 1;
    while(i > sortedUntil)
      invariant sortedUntil <= i < a.Length;
      invariant forall j, k :: 0 <= j < sortedUntil <= k < a.Length ==> a[j] <= a[k];
      invariant forall j :: i <= j < a.Length ==> a[i] <= a[j];
      invariant sorted(a, 0, sortedUntil);
      {
        if(a[i] <= a[i - 1])
        {
```

```
          a[i - 1], a[i] := a[i], a[i-1];
        }
        i := i - 1;
      }
      sortedUntil := sortedUntil + 1;
  }
}
```

Figure 26. Implementation of Bubble Sort

In the inner loop, the *sorted()* invariant shows up again, as justification of its truthfulness in the inner loop will help with its justification in the outer loop. Line 17, like line 11, helps in the proof of the *sorted()* invariant. Line 18, which states that all elements to the right of the counter *i* are greater than the element at index *i*, further qualifies the truthfulness of the *sorted()* invariant, allowing Dafny to prove the postcondition that the entire array is sorted.

4.3 Insertion Sort

The precondition *(P)* of this implementation of insertion sort (Figure 27) is that the given array *a* is not null (line 2), and its postcondition *(R)* is that the entire array is sorted into ascending order (line 3).

The expressions that must hold true within the program to prove it correct *(Q)* relate to the instance variables *index*, *counter*, and *temp*. The variable *index* must stay within the bounds of 1 and the length of the array *a* inclusive (line 12). In the inner loop, as *counter* is decreased, all the elements between the indices of *counter* and *index* must be greater than the value of *temp*, which is the initial value of *index* and the value currently being checked (line 21). This statement implies that the correct placement of *temp* in the first part of the array has not yet been found, and so the values at *counter* continue to be swapped with the succeeding element. This implementation ensures twice that the first part of the array *a* has been correctly put into ascending order, once in the outer loop and once in the inner loop (lines 13 and 20). This expression is checked within each loop to aid in inductively proving the postcondition, as the Dafny compiler does not automatically consider conditions that hold

```
method InsertionSort(a: array<int>)
  requires a != null && a.Length > 1;
  ensures sorted(a, 0, a.Length);
  modifies a;
{
  var index := 1;
  while(index < a.Length)
    invariant 1 <= index <= a.Length;
    invariant sorted(a, 0, index);
  {
    var counter := index - 1;
    var temp := a[index];
    a[index] := a[counter];
    while(counter >= 0 && temp < a[counter])
      invariant sorted(a, 0, index + 1);
      invariant forall k :: counter < k < index ==> a[k] > temp;
      {
        a[counter + 1] := a[counter];
        counter := counter - 1;
      }
    a[counter + 1] := temp;
    index := index + 1;
  }
}
```

Figure 27. Implementation of Insertion Sort

true within a given loop unless specified. In line 13, the expression checks the values from 0 up to but not including *index*. Line 20 ensures that the value at *index*, which has been reassigned to the preceding value, is greater than all values before it in the array *a*, verifying that the first part of the array is sorted correctly. It is initially unknown whether *temp* is less than, equal to, or greater than elements of the sorted section of the array, but these loop invariants ensure that when *temp* is put back into the array, it is in the correct position. These invariants ensure that the sorted portion of the array grows with each iteration and maintains its ascending order, fulfilling the method's postcondition.

4.4 Quick Sort

Quick sort (Figure 28) is the most complex sorting algorithm to prove of the four demonstrated, both as a result of the quantity of code and the recursion which it utilizes. Whereas the other three required invariants which logically implied the postcondition, the postconditions must be proved by the postconditions of two other instances of the quicksort method; furthermore, the iterative process must be shown to decrease, a challenge which is considerably more difficult within the context of a recursive method. In order to assert to Dafny that the reiterations of the quick sort method did not affect the elements outside the domain over which the method specifically sorted, a helper method to partition the array. The preconditions of quick sort (P) asserted not only that the array was not null (line 2) but also that the start and end elements of the region to be sorted were within the domain of the array (line 3) and that the elements to the left and the right of those bounds were less than and greater than those bounds, respectively (lines 4 and 5). The postconditions of quick sort (R) not only state the array is sorted within those bounds (line 7) but also that elements outside the bounds were not altered by the array and still remain sorted with respect to the sorted section (lines 8-10).

In addition, the decreases statement (line 11) indicates that the recursive elements of the quicksort loop will terminate. Both the preconditions and postconditions hold more statements for Dafny to interpret than in the other sorting algorithms because they must function not only as preconditions and postconditions but also as invariants, confirming features that must hold for the duration of the loop. Outside of the preconditions and postconditions, the main method of quick sort holds no assertions or invariants; the majority of these are held within the partition helper method.

Within the partition method, the preconditions contained both the requirements that the array not be null and contain elements (line 26) as well as the additional statements asserted in the preconditions of the quick sort method (lines 27-29). Similarly, the postconditions of partition ensured that all elements to the left of the pivot and within the bounds specified were less than the pivot, and that the elements to the right of the pivot within the same bounds were greater than the pivot (lines 32, 33); this, complemented with the additional statements posed in the precondition (lines 34-36), demonstrated to Dafny that the quick sort instances called later on (lines 16 and 17) have confirmed preconditions. The invariants contained within both loops assert these additional statements (lines 44 - 46, 59 - 61). As for the main postconditions of this loop, they are confirmed through simple statements in the invariants (lines 42, 43, 56, and 57). The postconditions of the partition method enable the two later calls of the quick sort method to compile without calling errors due to unsatisfied postconditions. Furthermore, as each iteration of quicksort completes itself, the array within the bounds of that instance of quicksort is sorted.

4.5 Minimum Contiguous Subarray

The minimum contiguous subarray algorithm returns the smallest sum of a subarray of its input array (Figure 29). This input array may have negative numbers; if it does not, the minimum subarray is empty and has a sum of 0. The algorithm used is Kadane's algorithm, identical to the trapezoid algorithm developed by the group. This algorithm operates in linear time

```
method QuickSort(a: array<int>, start: int, end: int)
  requires a != null && a.Length >= 1;
  requires 0 <= start <= end <= a.Length;
  requires 0 <= start <= end < a.Length ==> forall j :: start <= j < end ==> a[j] < a[end];
  requires 0 < start <= end <= a.Length ==> forall j :: start <= j < end ==> a[start - 1] <= a[j];
  modifies a;
  ensures sorted(a, start, end);
  ensures forall j :: 0 <= j < start || end <= j < a.Length ==> old(a[j]) == a[j];
  ensures 0 <= start <= end < a.Length ==> forall j :: start <= j < end ==> a[j] < a[end];
  ensures 0 < start <= end <= a.Length ==> forall j :: start <= j < end ==> a[start - 1] <= a[j];
  decreases end - start;
{
  if(end - start > 1)
  {
    var pivot := partition(a, start, end);
    QuickSort(a, start, pivot);
    QuickSort(a, pivot + 1, end);
  }
  else
  {
    return;
  }
}

method partition(a: array<int>, start: int, end: int) returns (pivot: int)
  requires a != null && a.Length > 0;
  requires 0 <= start < end <= a.Length;
  requires 0 <= start <= end < a.Length ==> forall j :: start <= j < end ==> a[j] < a[end];
  requires 0 < start <= end <= a.Length ==> forall j :: start <= j < end ==> a[start - 1] <= a[j];
  modifies a;
  ensures 0 <= start <= pivot < end <= a.Length;
  ensures forall j :: start <= j < pivot ==> a[j] < a[pivot];
  ensures forall j :: pivot < j < end ==> a[pivot] <= a[j];
  ensures forall j :: 0 <= j < start || end <= j < a.Length ==> old(a[j]) == a[j];
  ensures 0 <= start <= end < a.Length ==> forall j :: start <= j < end ==> a[j] < a[end];
  ensures 0 < start <= end <= a.Length ==> forall j :: start <= j < end ==> a[start - 1] <= a[j];
{
  pivot := start;
  var index := start + 1;
  while(index < end)
    invariant start <= pivot < index <= end;
    invariant forall j :: start <= j < pivot ==> a[j] < a[pivot];
    invariant forall j :: pivot < j < index ==> a[pivot] <= a[j];
    invariant forall j :: 0 <= j < start || end <= j < a.Length ==> old(a[j]) == a[j];
    invariant 0 <= start <= end < a.Length ==> forall j :: start <= j < end ==> a[j] < a[end];
    invariant 0 < start <= end <= a.Length ==> forall j :: start <= j < end ==> a[start - 1] <= a[j];
  {
    if(a[index] < a[pivot])
    {
      assert 0 < start <= end <= a.Length ==> forall j :: start <= j < end ==> a[start - 1] <= a[j];
      var counter := index - 1;
      var temp := a[index];
      a[index] := a[counter];
      while(counter > pivot)
```

```
        invariant forall j :: start <= j < pivot ==> a[j] < a[pivot];
        invariant forall j :: pivot < j < index + 1 ==> a[pivot] <= a[j];
        invariant a[pivot] > temp;
        invariant forall j :: 0 <= j < start || end <= j < a.Length ==> old(a[j]) == a[j];
        invariant 0 <= start <= end < a.Length ==> forall j :: start <= j < end ==> a[j] < a[end];
        invariant 0 < start <= end <= a.Length ==> forall j :: start <= j < end ==> a[start - 1] <= a[j];
      {
        a[counter + 1] := a[counter];
        counter := counter - 1;
      }
      a[pivot + 1] := a[pivot];
      pivot := pivot + 1;
      a[pivot - 1] := temp;
    }
    index := index + 1;
  }
}
```

Figure 28. Implementation of Quick Sort

with respect to the length of the array, only looking at each element of the array *a* once.

This implementation of the algorithm utilizes two function methods: *sum()*, which adds the values of an array *a* from *i* to *j*, and *min()*, which returns the smaller of the two given numbers.

The precondition of the algorithm *(P)* is that the given array *a* is not null (line 9), and according to the postcondition *(R)*, the sum *m* returned must be less than or equal to the sum of every subarray of the array.

The loop invariants *(Q)* concern the variables *m*, *n*, and *x*. The variable *x* must stay within the bounds of 0 and the length of the array *a* inclusive (line 17). Each iteration of the loop considers a new element of *a* at index *x*, ensuring that *m* is the minimum value overall (line 18) and *n* is the minimum sum of any subarray ending at *x* (line 19). This last invariant concerning *n* represents a key insight for Dafny to prove that the implementation is correct.

4.6 Discussion

Initially, the method by which Dafny proved programs was unclear. Without an understanding of the underlying principles behind Dafny, it was difficult to implement any of the algorithms. Selection sort, the first one completed, required the same amount of time to verify as bubble sort and insertion sort combined. By relating Dafny with Hoare logic principles and induction proofs, such as the technique of weakest preconditions, it became easier and quicker to formulate the necessary conditions and invariants to satisfy the algorithms' postconditions. This is one of the caveats of using Dafny: proving programs requires comprehension of the verification methods that Dafny automates. However, once

```
function method sum(a: array<int>,i: int,j: int): int
  decreases j;
  requires a != null;
  requires 0 <= i <= j <= a.Length;
  reads a;
{
  if j == i then 0 else sum(a, i, j - 1) + a[j - 1]
}

function method min(a:int,b:int):int {if a > b then b else a}

method min_subarray(a: array<int>) returns(m: int)
  requires a != null;
```

```
  ensures forall i, j :: 0 <= i <= j <= a.Length ==> sum(a, i, j) >= m;
{
  m := 0;
  var n, x := 0, 0;
  while(x < a.Length)
    invariant 0 <= x <= a.Length;
    invariant forall i, j :: 0 <= i <= j <= x ==> sum(a, i, j) >= m;
    invariant forall i :: 0 <= i <= x ==> sum(a, i, x) >= n;
  {
    n := min(0, n + a[x]);
    m := min(m, n);
    x := x + 1;
  }
}
```

Figure 29. Implementation of Minimum Contiguous Subarray

the basics have been learned, Dafny can provide a helpful tool for program verification.

One significant improvement that could be made in the future is for Dafny to indicate in more detail the causes of verification errors. The biggest challenge faced over the course of this project was that although Dafny was able to identify errors, it failed to specify exactly which conditions were false and for what reason. Moreover, at times it was hard to differentiate between errors in the code and simply a lack of invariants to assert the correctness of these algorithms. Thus, errors that might have been quickly noticed by other languages through test cases were in some instances difficult to identify. In addition, some statements that programmers can easily identify cannot be recognized by Dafny without an invariant, and some statements that the programmer might assume Dafny needs are not actually necessary at all. More specificity in describing the errors that occur would aid dramatically in speeding up the process of writing correct code in Dafny, as currently, it is much faster to write these programs in another language.

Despite the difficulties Dafny posed, the language as a whole demonstrated considerable effectiveness in achieving the goals for which it had been designed. Although the algorithms developed for sorting elements within an array have existed for many years, implementations by individual programmers are not foolproof; minor errors could incapacitate the code, particularly in specific test cases. Without specifically developing test cases, Dafny can identify the postconditions of each algorithm and, through induction, show that these postconditions may not hold for various instances. In several cases, errors that would have bypassed the compiler were stopped before compilation, enabling the programmer to address all issues at once as opposed to potentially missing some of the errors entirely.

In addition, the final implementations of the algorithms, with the exception of quick sort, were all less than thirty-five lines long. To write out the entire mathematical induction proof for these algorithms would fill several pages, with some intermediate steps that are evident enough not to require writing down every detail. The same thought process used with Hoare logic is needed to implement algorithms correctly in Dafny, but Dafny shortens this process significantly. Not only is the language more familiar to programmers than mathematical notation, but also the statements that Dafny requires are conclusions that it is incapable of generating without human insight, able to deduce the intermediate steps for the user. Furthermore, by addressing faults in the logic of their implementations, Dafny forces its users to develop stronger understandings of the underlying mechanisms of their algorithms.

5. Conclusions

In the modern world, computers have pervaded nearly all aspects of society, and so, it is incredibly important to ensure that programs are both functional and correct. One flaw in the

code could have potentially disastrous results, like an airplane malfunction or a stock market glitch. Companies are aware of the possible consequences and try to prevent them, typically by requiring new programs to be checked rigorously with a variety of test cases; however, formal verification along the lines of Hoare logic is not commonly used in this process. While a large sample of test cases can be sufficient to show that a program is correct in most circumstances, it cannot prove that a program will run correctly for *all* test cases. It may be more difficult to approach the concept of a program's correctness theoretically with mathematical logic than to simply generate a set of test inputs, but the increased complexity of formal verification is offset by the assurance of complete functionality that it can provide.

In addition, static verification languages like Dafny can simplify the process of formal verification with a language more familiar to programmers and automatic confirmation of certain steps. Despite Dafny's shortcomings, the language holds potential for future developments. As static verification languages continue to grow, they will likely prove increasingly capable of autonomously generating the insights for which Dafny requires the user. In the future, these languages may even go beyond identifying their own errors to being able to self-correct programs.

6. Acknowledgements

The authors would like to thank Dr. Santosh Nagarakatte of Rutgers Computer Science for providing valuable information regarding formal verification and processes in Dafny, as well as giving key insights for specific problems encountered along the course of this research project.

The authors would also like to acknowledge Dave Menendez, a PhD graduate student of Rutgers, who explained the fundamentals of propositional logic, and provided help for each of the algorithms written as a part of this project.

Additionally, the authors gratefully acknowledge their residential teaching assistant, Anthony Yang, for his guidance through every step of this research process and assistance in communication with the mentors.

Lastly, the authors would like to thank the assistant director of the Governor's School of Engineering and Technology, Jean Patrick Antoine, for organizing this program and providing the opportunity for this research project. Also, this opportunity would not have been possible without the help of Governor's School sponsors, Rutgers University, the State of New Jersey, Morgan Stanley, Lockheed Martin, Silver Line Windows, South Jersey Industries, The Provident Bank Foundation, and Novo Nordisk.

7. References

[1]"In-flight upset, 154 km west of Learmonth, Western Australia, 7 October 2008, VH-QPA, Airbus A330-303, " Australia Transportation Safety Board, 1-7, 96 (2011)

[2]"Final Report on the August 14, 2003 Blackout in the United States and Canada: Causes and Recommendations, " U.S.-Canada Power System Outage Task Force, 1-2 (2004).

[3]"Technical Analysis of the August 14, 2003 Blackout: What Happened, Why, and What Did We Learn?," North American Electric Reliability Council, (2004).

[4] N. M. A. Munassar, A. Govardhan, "A Comparison Between Five Models Of Software Engineering," IJCSI 5, 95-101 (2010).

[5] T. Britton, L. Jeng, G. Carver, P. Cheak, T. Katzenellenbogen, "Reversible Debugging Software," (2012).

[6] E. Roberts, "Hoare Logic," (2008), http://cs.stanford.edu/people/eroberts/courses/soco/projects/2008-09/tony-hoare/logic.html (July, 2014).

[7] H. K. Büning, T. Lettmann, Propositional Logic: Deduction and Algorithms (Cambridge University Press, Cambridge, UK, 1994), pp. 1-14.

[8] C. A. R. Hoare, “An Axiomatic Basis for Computer Programming,” Communications of the ACM, 576-583 (1969).

[9] “Dafny: A Language and Program Verifier for Functional Correctness,” http://research.microsoft.com/en-us/projects/dafny (July, 2014).

[10] H. Koenig, K. R. M. Leino, “Getting Started with Dafny, A Guide,” Software Safety and Security, 151-181 (2012).

[11] K. R. M. Leino, “Efficient Weakest Preconditions,” Information Processing Letters, 281-288 (2005).