

# Whiley Comparison

Vivian Stewart

January 27, 2016

## Abstract

## 1 Introduction

From an undergraduate perspective one might approach verification with optimistic positivity, “I can go about proving all of the programs I’ve contributed to, all of the weird algorithms I’ve always wondered about...” However the truth of Software Verification is in actuality much more of a field of research with very limited tools and a steep learning curve. Current efforts are impressive from an academic point of view and but require years of invested effort to learn the various analyses and how these fit together to prove correctness of program execution.

There are some interesting tools that have been developed recently, making for an easier more tractable learning curve and easing the burden of proof for the programmers whom may be a bit intimidated by formal specification or those who found other tools too much effort for meagre results.

As programmers we need some tools to assure us that, the code we work so long and hard on, does what we think it does. There are linter’s, Unit testing, Static Analysis tools, all of which will give you some idea about some aspect of code execution but Formal Verification promises to cover much greater ground with relative ease. First Order Logic is expressed alongside code to describe program execution which is then converted into verification conditions to be checked and asserted by an automated proof assistant. It is possible to completely or partially specify a program depending on critical importance correctness and impact of failure.

Formal Logic is at first difficult to grasp for the majority of programmers and even the tools discussed here in require some background in Mathematical Logic, proof by contradiction, contrapositive and induction etc. Induction seems to be a tricky one not only for human beings but for automated deduction also, for instance, Z3 doesn’t support induction (recursive) verification conditions, though there are structures available in Dafny for inductive proof, lemmas, functions, predicates (possibly in boogie?) One interesting thin that caught me out was the order of statements in the specification was important which I thought unusual as one would tend to think of certain mathematical operations as commutative or associative and mathematical definitions as being declarative. But it would seem that logical operations in specifications employ short circuit logic which makes order of premises important.

More often than not code made for one tool is easily transcribed to the other as long as there is a one to one mapping of structures used in formal verification of the program. The difference in range notation between the different tools is note worthy as in Dafny ranges can simply be seen mathematically  $0 \leq x < \text{length}$ , where as Whiley is  $x \text{ in } 0..\text{length}$  and Spec#  $x \text{ in } (0:\text{length})$  or  $x \text{ in } (0..\text{length} - 1)$ . In Whiley the  $a..b$  means mathematically  $[a, b)$ . In Spec#  $(a:b)$  means  $[a, b)$  but  $(a..b)$  means  $[a, b]$ . I personally prefer Dafny’s way for it’s

clarity though the others are short and concise, but this is a matter of taste. Potentially confusing off by one errors can be hard to see.

Intuitively one has to establish a correspondence between any of the variables that change in the loop(s) of the program. Knowing if there has been enough specification or even a complete specification is not immediate and it is easy to come up with only a partial specification. Hence, it raises two main questions: how to be sure that our specification is complete, and how complete must a specification be. For the first question, one must reason on some model of the specified program. For the second question, no definitive answer is known. It depends on the context in which the specification is written and the kind of properties that must be established. The amount of specification required for a given function is very different when verifying a given property for a given application in which calls to the function always occur in a well-defined context and when specifying it as a library function which should be callable in as many contexts as possible.

- Overflow, correspondence between bounded and unbounded types inside a specification...

- Termination...

- Null and non-null types...

- Simplification and its effects time taken to verify and translation into more/less verification conditions.

- Embedded code in specifications... lemmas, Pure Functions, Predicates...

## 2 Benchmarks

Selected benchmarks were intended to be simple since the verification of large blocks of code are exponentially asymptotic in logical deduction and/or not decidable in the worst case. The benchmarks range from simple to reasonably complex. Some of them were exercises recommended by David Pearce, where as others were suggested by Lindsey Grooves. To begin with, the main idea of this project was to create some small programs for use in lecture slides or as examples in tutorials.

### 2.1 palindrome

Scans through an array from the outer elements inwards to the centre, searching for a mismatch in values in which case returning false if there is such a mismatch and returning true if there is no such mismatch.

### 2.2 firstIndexOf

Finds the first index where the specified element is found in the given array of integers, searching through the array from the zero index in the positive direction. returns immediately the item is found, otherwise traverses the entire array then returns -1 to represent the absence of the value in the array.

### 2.3 lastIndexOf

Finds the last index where the specified element is found in the given array of integers, searching through the array from the the highest index in the negative direction. A lot like the firstIndexOf function in that the entire array is traversed if the value is not found but quits early if the value is found.

## 2.4 max

Searches the entire list in order to find and return the largest integer found. Accumulating the maximum value evaluated so far, as the algorithm traverses the array.

## 2.5 occurrences

Returns the number of times a given value is encountered while traversing the array.

## 2.6 strlen

The **strlen** function is intended to mirror the C function of the same name. It counts the number of characters in a string (array) of characters, having come to the end of the list when encountering a null character. The null character is usually denoted as `'\0'`, but since there is no `char` type currently available in Whiley, instead we have to use an abstraction of this data type. Ascii characters are bytes (8 bits) their range of values is between 0 and 255 inclusive and the null character is 0.

## 2.7 linearSearch

Much like the `firstIndexOf` function but iterates through a sorted list and can therefore quit early when the element matching the value is either found or the value is exceeded by the current element. Previous versions found an index of insertion into the sorted part of the array.

## 2.8 binarySearch

Using the divide and conquer method to find the index of an element in the given array matching the given value.

## 2.9 append

Adds a single element to the end of the array by making a new array that is one element larger than the original and copies the original array in order to the lower elements and inserting the given value in the last index.

## 2.10 remove

The `remove` function should remove the item at the given index from the given array of integers and return the resulting array otherwise unchanged. The resulting array is of course one element shorter in length. This is done by creating a new array one smaller in size and then copying across the elements before the given index directly, followed by copying across elements above the index to a position one index lower and overwriting the removed element.

## 2.11 copy

Copies a region of the source array into a same sized region of the destination.

## 2.12 displace

rotates a region of the array by one place forward

### 2.13 insert

This function should insert the item at the given index from the items array. The resulting array is of course one element longer in length.

### 2.14 insertionSort

...

## 3 Results & Discussion

Env.	Palin.	FIO	LIO	Max	Occ	Slen	LSrch
Whiley	✓	✓	✓	✓	✗	✓	✓
Dafny	✓	✓	✓	✓	✓	✓	✓

Env.	BSrch	Appd	Remv	Copy	Dplc	Inst	InSrt
Whiley	✗	✓	✗	✗	✗	✗	✗
Dafny	✓	✓	✓	✓	✓	✓	✓

the loop invariants tended to be reflected in the postconditions.

Z3 has no inductive proof mechanism and I suspect this is why Dafny has such a messy inductive definition syntax that I also suspect derives from boogie rather than Z3.

Whiley allows you to do more with less, minimal syntax but more expressive. (IMPORTANT!! flexible language constructs)...

Dafny has more complicated syntax to deal with memory management i.e pointers/references, lots to remember and coordinate.

The tools Dafny provides are for the development of proofs in a “top-down” manner, and which allow us to concentrate on the “architecture” of the proof. Unlike Frama-C.

With Frama-C in particular, it is a 2-valued logic with only total functions. Consequently, expressions are never “undefined”. Having only total functions implies that one can write terms such as  $1/0$ , or  $*p$  when  $p$  is null.

Specifications in Frama-C can have implicit casts between C-types and Mathematical types (something to watch out for.) Weird:

- $5/3$  is 1 and  $5 \% 3$  is 2;
- $(-5)/3$  is -1 and  $(-5) \% 3$  is -2;
- $5/(-3)$  is -1 and  $5 \% (-3)$  is 2;
- $(-5)/(-3)$  is 1 and  $(-5) \% (-3)$  is -2.

Frama-C When no ‘assigns’ clauses are specified, the function is allowed to modify every visible variable.

**Termintes** It is possible to relax a particular function’s specification by providing a formula that describes the conditions in which the function is guaranteed to terminate.

**Assertions** when the analyzer is not able to determine that an assertion always holds, it may be able to produce a pre-condition for the function that would, if it was added to the function’s contract, ensure that the assertion was verified.

**Overflow** overflow is/must be handled.

```
1 inductive reachable{L} (list* root, list* node) {  
2   case root_reachable{L}:  
3     \forallall list* root; reachable(root,root);  
4   case next_reachable{L}: // L indicates a Label -> memory state  
5     \forallall list* root, *node;  
6     \valid(root) ==> reachable(root -> next, node) ==>  
7       reachable(root,node);
```

```
1 requires \valid(p) && \valid(q);  
2 ensures *p <= *q; // pointers  
3 behavior p_minimum:  
4   assumes *p < *q;  
5   ensures *p == \old(*p) && *q == \old(*q);  
6 behavior q_minimum:  
7   assumes *p >= *q;  
8   ensures *p == \old(*q) && *q == \old(*p);  
9 complete behaviors p_minimum, q_minimum;  
10 disjoint behaviors p_minimum, q_minimum;
```

No verifier can tell you whether your code doesn't work the according to the specification or the specification doesn't describe what the code does, as this depends on the intention of the user/developer. Frama-C ACSL specification language lacks flexible language constructs, very cumbersome/specialised syntax.

All languages need different forms of verification and to different degrees. Sensitivity to approach to the problem.

### 3.1 Whiley

Whiley is very easy to use and shows promise as a tool to verify programs in such a way that the accompanying specification scales with complexity of the program to be verified. Whiley offers a simplified program and specification definition. However, Whiley's underlying prover needs some work to bring Whiley up to the level attained by other tools. Arrays in Whiley are not reference or pointer types, there is no possibility of the array being null, also copying of an array is just a matter of assigning that array to a different variable, bypassing the issue of specifying array copying loops. Some basic arithmetic and region ranges seem to be difficult for the whiley constraint solver (WyCS.)

### 3.2 Dafny

Dafny is a sophisticated tool/language for verification based on Microsofts' Z3 prover and Boogie with Monodevelop/Visual Studio. Though tests were performed on the command line to gain feedback from Dafny. Dafny can use pure functional code to help prove the correctness of imperative code for instance, only functions can used in specification unless qualified as 'function method'. These pure function definitions require very minimal specification in order to provide lemmas, axioms and other mathematical constructs to prove correct execution. One thing that struck me as odd was the use of short circuit logic in the specifications which implies a certain order to the specifications statements. Dafny doesn't require return statements in the usual way and defines a return variable(s) that is assigned to, and if necessary a 'return' can be called for early return from the call. Explicit and obvious reference to the returned variable was very convenient. All function/method parameters are immutable unless specified otherwise. Inside the specifications arrays are treated differently from other containers but are implicitly convertible to the built in sequence type if needed. Which allows for the very convenient syntax: `arr1[m..n] == arr2[m..n]` for an adjacent element comparison avoiding the need for long winded predicates. But not so convenient in that a sequence cannot be easily converted back to an array. Arrays

are always possibly null in Dafny and there are attempts in the specification language to mitigate the issue of reference aliasing and other such memory management quirks (modified clause.) Unlike Whyle arrays are of reference type and need to be null checked.

### 3.3 Frama-C

When it comes to verifying C code Frama-C seems to be a current industry favourite. But I found the user gtk interface to be cumbersome and confusing. The ACSL language would appear to appeal more to an expert C programmer which I am not. Much expertise is needed to master Frama-C and every aspect and plugin have large manuals and tutorials available, all very heavy in detail. What I really wanted was a more general tool that would help point out where errors in my reasoning were evident. Frama-C offers only the vaguest of clues and errors in my reasoning are not always obvious. The issues of bounded types are dealt with up front and checked with the WP plugin through an RTE (runtime error) option that injects (over/under)flow checks. Specifications are meant to be written in a style that complements C and is analogous to C, even mirroring C conventions and C arithmetic. The lack of consolidation makes the various plugins difficult to integrate into a workflow. The ACSL specification language is littered with predefined predicates and detailed set of proof tools such as user defined predicates, behaviours, inductive(recursive) functions and axioms etc. Arrays were a very sticky issue in Frama-C and involve proof of non-null and the validation of the range of indices and elements. For loop invariants, proof of termination is provided by the loop 'variant' key word, which is hard to see in amongst loop 'invariant' clauses in the same comment code block. Also there are assigns clauses that define the frame of modification for both the function and loop invariants (loop assigns.) since unlike other tools Frama-C has no knowledge of what has been modified in the function or inside the loop (side effects). Since the C language does not have formal semantics one should take the verification of C code with a grain of salt, horribly coded nonsense with undefined results can still be verified.

### 3.4 Spec#

Spec# having the same underlying workings as Dafny (a.k.a Boogie and Z3) if there are many differences between the two.

## 4 Appendix

```

1 // Status wyc-37: verified [96203ms]
2 // wyc-36: verified [43551ms] -54.73 percent.
3 function isPalindrome(int[] chars) -> (bool r)
4   ensures r <=> all { x in 0..|chars|
5     | chars[x] == chars[|chars| - (x + 1)] }
6 :
7   int i = 0
8   int j = |chars|
9
10  while i < j
11    where i + j == |chars| && i >= 0
12    where all { k in 0..i | chars[k] == chars[|chars| - (k+1)] }
13  :
14    j = j - 1
15    if chars[i] != chars[j]
16    :
17      return false
18    i = i + 1
19
20  return true

```

```

1 // Status: verifies and compiles
2 // Result holds iff array is a palindrome :)
3 method isPalindrome(chars: array<int>) returns (r: bool)
4   requires chars != null
5   ensures r <=> forall x: int :: 0 <= x < chars.Length
6     ==> chars[x] == chars[chars.Length - (x + 1)]
7 {
8   var i: nat := 0;
9   var j: nat := chars.Length;
10  //
11  while i < j
12    invariant i + j == chars.Length && i >= 0
13    invariant forall k: int :: 0 <= k < i
14      ==> chars[k] == chars[chars.Length - (k + 1)]
15  {
16    j := j - 1;
17    if chars[i] != chars[j]
18    {
19      return false;
20    }
21    i := i + 1;
22  }
23  return true;
24 }

```

```

1 // Status wyc-37: verifies wyc-36: verifies
2 function firstIndexOf(int[] items, int item) -> (int r)
3 // If result is positive, element at that position must match item
4 ensures r >= 0 ==> items[r] == item
5 // If result is positive, no element at lesser position matches item
6 ensures r >= 0 ==> no { k in 0..r | items[k] == item }
7 // If result is negative, no element matches item
8 ensures r < 0 ==> no { k in 0..|items| | items[k] == item }
9 :
10  int i = 0
11  while i < |items|
12    // i is increasing and no element at greater position matches item
13    where 0 <= i && i <= |items|
14    where no { k in 0..i | items[k] == item }
15  :
16    if items[i] == item
17    :
18      return i
19    i = i + 1
20  // didn't find item in entire list
21  return -1

```

```

1 // Status: verifies
2 method firstIndexOf(items: array<int>, item: int) returns (r: int)
3   requires items != null
4   ensures r < items.Length
5   // If result is positive, element at that position must match item
6   ensures r >= 0 ==> items[r] == item
7   // If result is positive, no element at lesser position matches item
8   ensures r >= 0 ==> forall k: nat :: k < r ==> items[k] != item
9   // If result is negative, no element matches item
10  ensures r < 0 ==> forall k: nat :: k < items.Length
11    ==> items[k] != item
12 {
13   var i: int := 0;
14   while i < items.Length
15     // i is increasing and no element at greater position matches item
16     invariant 0 <= i <= items.Length
17     invariant forall k: nat :: k < i ==> items[k] != item
18   {
19     if items[i] == item
20     { return i; }
21     i := i + 1;
22   } // didn't find item in entire list
23   return -1;
24 }

```

```

1 // Status wyc-37: verifies [1048ms]
2 // wyc-36: verifies [1429ms] -27.77 percent
3 function lastIndexOf(int[] items, int item) -> (int r)
4 // If result is positive, element at that position must match item
5 ensures r >= 0 ==> items[r] == item
6 // If result is positive, no element at greater position matches item
7 ensures r >= 0 ==> all { x in (r + 1)..|items| | items[x] != item }
8 // If result is negative, no element matches item
9 ensures r < 0 ==> no { x in 0..|items| | items[x] == item }
10 :
11  int i = |items|
12  while i >= 0
13    // i is decreasing and no element at greater position matches item
14    where 0 <= i && i <= |items|
15    where no { x in i..|items| | items[x] == item }
16  :
17    i = i - 1
18    if items[i] == item
19    :
20      return i
21  // didn't find item in entire list
22  return -1

```

```

1 // Status: verifies and compiles
2 method lastIndexOf(items: array<int>, item: int) returns (r: int)
3   requires items != null
4   ensures r < items.Length
5   // result is positive element at that position must match item
6   ensures r >= 0 ==> items[r] == item
7   // result is positive no element at greater position matches item
8   ensures r >= 0 ==> forall x :: r < x < items.Length
9     ==> items[x] != item
10  // If result is negative, no element matches item
11  ensures r < 0 ==> forall x :: 0 <= x < items.Length
12    ==> items[x] != item
13 {
14   r := items.Length;
15   while r > 0
16     // no element at greater position matches item
17     invariant 0 <= r <= items.Length
18     decreases r
19     invariant forall x :: r <= x < items.Length ==> items[x] != item
20   {
21     r := r - 1;
22     if items[r] == item { return; }
23   }
24   // didn't find item in entire list
25   r := -1;
26 }

```

```

1 // Status wyc-37: verifies. wyc-36: verifies.
2 public function maxArray(int[] items) -> (int max)
3   requires items != null
4   ensures some { k in 0..items | max == items[k] }
5   ensures all { k in 0..items | max >= items[k] }
6 :
7   int i = 1
8   int r = items[0]
9
10  while i < items.Length
11    where 0 < i && i <= items.Length
12    where some { k in 0..i | r == items[k] }
13    where all { k in 0..i | r >= items[k] }
14  :
15    if items[i] > r:
16      r = items[i]
17    i = i + 1
18
19  return r

```

```

1 // Status: verified
2 method maxArray(items: array<int>) returns (r: int)
3   requires items != null
4   requires items.Length > 0
5   ensures forall k: nat :: 0 <= k < items.Length ==> r >= items[k]
6 {
7   var i: int := 1;
8   r := items[0];
9
10  while i < items.Length
11    invariant 0 < i <= items.Length
12    invariant forall k: nat :: k < i ==> r >= items[k]
13  {
14    if items[i] > r
15      { r := items[i]; }
16    i := i + 1;
17  }
18 }

```

```

1 // Status wyc-37: wyc-36: loop invariant
2 // not restored line: 16
3 function occurrences(int[] items, int item) -> (int r)
4   // some number of occurrences of item
5   ensures r > 0 ==> some { k in 0..items |
6     | items[k] == item }
7   // no occurrences of item
8   ensures r == 0 ==> all { k in 0..items |
9     | items[k] != item }
10 :
11   int i = 0
12   int count = 0
13   //
14   while i < items.Length
15     // i is increasing and there could be elements that match
16     where 0 <= i && i <= items.Length
17     where count > 0 ==> some { k in 0..i
18       | items[k] == item }
19     where count == 0 ==> all { k in 0..i
20       | items[k] != item }
21 :
22   if items[i] == item
23   :
24     count = count + 1
25   i = i + 1
26   return count

```

```

1 function count(items: seq<int>, item: int): nat
2   decreases items
3 {
4   if items.Length == 0 then 0 else
5     if items[0] == item
6       then (1 + count(items[1..], item))
7     else count(items[1..], item)
8 }
9 // Status: ???
10 method occurrences(items: array<int>, item: int) returns (r: nat)
11   requires items != null
12   ensures r <= items.Length
13   // some number of occurrences of item
14   ensures r > 0 ==> exists k: nat :: k < items.Length && items[k] == item
15   // no occurrences of item
16   ensures r == 0 ==> forall k: nat :: k < items.Length ==> items[k] != item
17   ensures r == count(items[..], item)
18 {
19   var i: int := 0;
20   var num: nat := 0;
21
22   while i < items.Length
23     // i is increasing and there could be elements that match
24     invariant 0 <= i <= items.Length
25     invariant num <= items.Length
26     invariant num <= i
27     invariant num > 0 ==> exists k: nat :: k < i
28       && items[k] == item
29     invariant num == 0 ==> forall k: nat :: k < i
30       ==> items[k] != item
31     invariant num == count(items[..i], item)
32   {
33     if items[i] == item
34       { num := num + 1; }
35     i := i + 1;
36   }
37   return num;
38 }

```

```

1 // Status: verified and compiled
2 // wyc-37: [1197ms] wyc-36: [1104ms] -7.76
3 constant NULL is 0
4 // ASCII character is unsigned 8bit integer
5 type ASCII_char is (int n) where n >= 0 && n < 256
6 // C String is array of chars where at least one is NULL
7 type C_string is (ASCII_char[] chars)
8   where some { i in 0..chars.Length | chars[i] == NULL }
9 // Calculate length of string
10 function strlen(C_string str) -> (int r)
11   ensures r >= 0
12 :
13   int i = 0
14   //
15   while str[i] != NULL
16     where i >= 0
17     where all { k in 0..i | str[k] != NULL }
18 :
19   i = i + 1
20   return i

```

```

1 // Status: verifies and compiles
2 // Calculate length of string
3 method strlen(str: array<char>) returns (r: nat)
4   requires str != null
5   requires exists k: nat :: k < str.Length && str[k] == '\u0000'
6 {
7   r := 0;
8   while str[r] != '\u0000'
9     invariant r <= str.Length
10    invariant forall k: nat :: k < r ==> str[k] != '\u0000'
11    decreases str.Length - r
12  {
13    r := r + 1;
14  }
15 }

```



```

1 // Status wyc-37: verifies [3519ms] wyc-36: verifies [9570ms]
2 function linearSearch(int[] arr, int val) -> (int r)
3 // arr is an ordered array
4   requires all { i in 0..|arr|, j in 0..|arr|
5     | i < j ==> arr[i] < arr[j] }
6 // Return is between -1 and length of arr
7 ensures r >= -1 && r < |arr|
8 // If index returned, it is first match
9 ensures r >= 0 ==> all { k in 0..r | arr[k] < val }
10 // If index return, it matches val
11 ensures r >= 0 ==> arr[r] == val
12 // if failure, no matching element exists
13 ensures r == -1 ==> no { k in 0..|arr| | arr[k] == val };
14 //
15 int i = 0
16 //
17 while i < |arr|
18   where i >= 0 && i <= |arr|
19   where all { k in 0..i | arr[k] < val };
20 //
21   if arr[i] == val:
22     return i
23   else if arr[i] > val:
24     return -1
25   i = i + 1
26 return -1

```

```

1 // Status: verifies
2 method linearSearch(arr: array<int>, val: int) returns (r: int)
3   requires arr != null
4   // arr is an ordered array
5   requires forall k: nat :: k < (arr.Length - 1)
6     ==> arr[k] < arr[k + 1]
7 // Return is between -1 and length of arr
8 ensures r >= -1 && r < arr.Length
9 // If index returned, it is first match
10 ensures r >= 0 ==> forall k: nat :: k < r ==> arr[k] < val
11 // If index return, it matches val
12 ensures r >= 0 ==> arr[r] == val
13 // if failure, no matching element exists
14 ensures r == -1 ==> forall k: nat :: k < arr.Length
15   ==> arr[k] != val
16 {
17   var i: int := 0;
18
19   while i < arr.Length
20     invariant 0 <= i <= arr.Length
21     invariant forall k: nat :: k < i ==> arr[k] < val
22   {
23     if arr[i] == val
24       { return i; }
25     else if arr[i] > val
26       { return -1; }
27     i := i + 1;
28   }
29   return -1;
30 }

```

```

1 // Status: verifies and compiles?...
2 // no, index out of bounds (negative) line 25
3 type nat is (int n) where n >= 0
4
5 method binarySearch( int[] items, int key ) -> (int r)
6   requires |items| > 0
7   requires all { j in 0..|items|, k in 0..|items|
8     | j < k ==> items[j] <= items[k] }
9   ensures r >= 0 ==> r < |items| && items[r] == key
10  ensures r < 0 ==> all { k in 0..|items| | items[k] != key }
11 :
12   nat low = 0
13   nat high = |items|
14   nat mid = 0
15   //
16   while low < high
17     where low <= mid
18     where mid < high
19     where high <= |items|
20     // elements outside the search range do not equal key
21     where no { i in 0..low, j in high..|items|
22       | items[i] != key && items[j] != key }
23 :
24   mid = (low + high) / 2
25   if items[mid] < key:
26     low = mid + 1
27   else if key < items[mid]:
28     high = mid
29   else:
30     return mid
31 return -1

```

```

1 // Status: verifies and compiles
2 predicate sorted(s: seq<int>)
3 {
4   forall i,j :: 0 <= i < j < |s| ==> s[i] <= s[j]
5 }
6
7 method BinarySearch(a: array<int>, key: int) returns (index: int)
8
9   requires a != null && sorted(a[..])
10  ensures index >= 0 ==> index < a.Length && a[index] == key
11  ensures index < 0 ==> forall k: nat :: k < a.Length ==> a[k] != key
12 {
13   var low := 0;
14   var high := a.Length;
15
16   while (low < high)
17     invariant 0 <= low <= high <= a.Length
18     invariant forall i :: 0 <= i < a.Length
19       && !(low <= i < high) ==> a[i] != key
20   {
21     var mid := (low + high) / 2;
22     if (a[mid] < key)
23       { low := mid + 1; }
24     else if (key < a[mid])
25       { high := mid; }
26     else
27       { return mid; }
28   }
29 }
30 return -1;
31 }
32

```

```

1 // Status: verifies wyc-37: [1745ms]
2 // wyc-36: [3987ms] 228.48 percent
3 public function append(int[] items, int item) -> (int[] rs)
4   ensures |rs| == |items| + 1
5 :
6   int[] result = [ 0; |items| + 1 ]
7   int i = 0
8   //
9   while i < |items|
10     where all { k in 0..i | result[k] == items[k] }
11     where i >= 0
12     where |result| == |items| + 1
13     where i < |result|
14 :
15     result[i] = items[i]
16     i = i + 1
17 result[i] = item
18 return result

```

```

1 // Status: verifies and compiles
2 // Append a single item onto the end of the array
3 method append( items: array<int>, item: int ) returns (r: array<int>)
4   requires items != null && items.Length > 0
5   ensures r != null && r.Length == items.Length + 1
6   ensures forall k: int :: 0 <= k < items.Length ==> r[k] == items[k]
7 {
8   r := new int[items.Length + 1];
9   var i: nat := 0;
10
11   while i < items.Length
12     invariant r.Length == items.Length + 1
13     invariant i <= items.Length
14     invariant forall k: int :: 0 <= k < i ==> r[k] == items[k]
15   {
16     r[i] := items[i];
17     i := i + 1;
18   }
19   r[i] := item;
20 }

```

```

1 // Status wyc-37: infinite loop wyc-36: "GC overhead limit exceeded"
2 function remove(int[] items, int index) -> (int[] r)
3   requires 0 <= index && index < |items|
4   requires |items| > 0
5   ensures |r| == |items| - 1
6   ensures all { k in 0..index | r[k] == items[k] }
7   ensures all { k in index..|r| | r[k] == items[k + 1] }
8 :
9   int newlen = |items| - 1
10  int i = 0
11  int[] result = [ 0; newlen ]
12  //
13  while i < index
14    // items before index in result are still the same
15    where 0 <= i where i <= index
16    where |result| == newlen
17    where all { k in 0..i | k < index ==> result[k] == items[k] }
18  :
19    result[i] = items[i]
20    i = i + 1
21  assert i == index
22  while i < newlen
23    // items after index in result are transposed by one place
24    where index <= i where i <= newlen
25    where |result| == newlen
26    where all { k in 0..index | result[k] == items[k] }
27    where all { k in index..i | result[k] == items[k + 1] }
28  :
29    result[i] = items[i + 1]
30    i = i + 1
31  return result

```

```

1 // Status: verifies and compiles
2 // This function should remove the item at the given
3 // index from the items array, and return the resulting
4 // array otherwise unchanged. The resulting array is of
5 // course one element shorter in length.
6 method remove(items: array<int>, index: nat) returns (r: array<int>)
7 :
8   requires items != null && index < items.Length
9   requires items.Length > 0
10  ensures r != null && r.Length == items.Length - 1
11  ensures forall k: nat :: k < index ==> r[k] == items[k]
12  ensures forall k: nat :: index <= k < r.Length ==> r[k] == items[k + 1]
13 {
14   // length of the new array
15   var newlen := items.Length - 1;
16   r := new int[newlen];
17   var i: nat := 0;
18
19   while i < index
20     // items before index in result are still the same
21     invariant i <= index
22     decreases index - i
23     invariant r.Length == newlen
24     invariant forall k: nat :: k < i
25       ==> (k < index ==> r[k] == items[k])
26   {
27     r[i] := items[i];
28     i := i + 1;
29   }
30  assert i == index;
31  while i < newlen
32    // items after index in result are transposed by one place
33    invariant index <= i <= newlen
34    decreases newlen - i
35    invariant r.Length == newlen
36    invariant forall k: nat :: k < index ==> r[k] == items[k]
37    invariant forall k: nat :: index <= k < i ==> r[k] == items[k + 1]
38  {
39    r[i] := items[i + 1];
40    i := i + 1;
41  }
42 }

```

```

1 // Status wyc-37: null pointer.
2 // wyc-36: loop inv. not restored: line 33
3 function copy(int[] src, int sStart, int[] dest, int dStart, int len)
4   -> (int[] r)
5 :
6   // starting points in both arrays cannot be negative
7   requires sStart >= 0 && dStart >= 0 && len > 0
8   // Source array must contain enough elements to be copied
9   requires |src| >= sStart + len
10  // Destination array must have enough space for copied elements
11  requires |dest| >= dStart + len
12  // Result is same size as dest
13  ensures |r| == |dest|
14  // All elements before copied region are same
15  ensures all { k in 0..dStart | r[k] == dest[k] }
16  // All elements in copied region match src
17  ensures all { k in 0..len | r[dStart + k] == src[sStart + k] }
18  // All elements above copied region are same
19  ensures all { k in dStart + len..|dest| | r[k] == dest[k] }
20 :
21  int i = 0
22  int[] odest = dest
23  assert all { k in 0..|dest| | dest[k] == odest[k] }
24  //
25  while i < len
26    where 0 <= i where i <= len
27    where |dest| == |odest|
28    // all items are still the same before dStart index
29    where all { k in 0..dStart | dest[k] == odest[k] }
30    // all items after dStart index are still the same
31    where all { k in (dStart + len)..|dest| | dest[k] == odest[k] }
32    // inbetween items are copied from src
33    where all { k in sStart..sStart + i, j in dStart..dStart + i
34      | src[k] == dest[j] }
35  :
36    dest[dStart + i] = src[sStart + i]
37    i = i + 1
38  return dest

```

```

1 // Status: verifies and compiles
2 method copy(src: array<int>, sStart: nat
3   , dest: array<int>, dStart: nat, len: nat)
4   returns (r: array<int>)
5 :
6   // both arrays cannot be null
7   requires dest != null && src != null;
8   // Source array must contain enough elements to be copied
9   requires src.Length >= sStart + len;
10  // Destination array must have enough space for copied elements
11  requires dest.Length >= dStart + len;
12  // Result is same size as dest
13  ensures r != null;
14  ensures r.Length == dest.Length;
15  // All elements before copied region are same
16  ensures r[..dStart] == dest[..dStart];
17  // All elements above copied region are same
18  ensures r[dStart + len..] == dest[dStart + len..];
19  // All elements in copied region match src
20  ensures forall k: nat :: k < len
21    ==> r[dStart + k] == src[sStart + k];
22 {
23   if len == 0 { return dest; }
24   var i: nat := 0;
25   r := new int[dest.Length];
26
27   while (i < r.Length)
28     invariant i <= r.Length
29     invariant forall k: nat :: k < i ==> r[k] == dest[k]
30   {
31     r[i] := dest[i];
32     i := i + 1;
33   }
34  assert r[..] == dest[..];
35  i := 0;
36  while (i < len)
37    invariant i <= len
38    invariant r[..dStart] == dest[..dStart]
39    invariant r[dStart..dStart + i] == src[sStart..sStart + i]
40    invariant r[dStart + len..] == dest[dStart + len..]
41  {
42    r[dStart + i] := src[sStart + i];
43    i := i + 1;
44  }
45 }

```

```

1 // Status wyc-37: null pointer
2 // wyc-36: line 37: loop inv. !restored
3
4 function displace( int[] arr, int start, int len) -> (int[] r)
5   requires len > 0
6   requires start + len <= |arr|
7   ensures |r| == |arr|
8   ensures all { k in 0..start | r[k] == arr[k] }
9   ensures all { k in (start + len)..|arr| | r[k] == arr[k] }
10  ensures all { k in (start + 1)..start + len
11    | r[k] == arr[k - 1] }
12  ensures r[start] == arr[start + len - 1]
13 :
14   int i = start
15   int[] res = arr
16   res[start] = arr[start + len - 1]
17
18   while i < |arr|
19     where |res| == |arr|
20     where (start + len) <= i && i <= |arr|
21     where res[start] == arr[start + len - 1]
22     where all { k in 0..start | res[k] == arr[k] }
23     where all { k in (start + len)..i | res[k] == arr[k] }
24     where all { k in start..start + len - 1
25       | res[k + 1] == arr[k] }
26 :
27   res[i] = arr[i]
28   i = i + 1
29
30   return res

```

```

1 // Status: verifies and compiles
2 // rotates a region of the array by one place forward
3 predicate rotated(arr1:seq<int>, arr2:seq<int>)
4   requires |arr1| == |arr2|
5 {
6   (foldall i :: 1 <= i < |arr1| ==> arr2[i] == arr1[i - 1]) &&
7   ( |arr1| > 0 ==> arr2[0] == arr1[ |arr1| - 1 ] )
8 }
9 method displace(arr: array<int>, start: nat, len: nat)
10  returns (r: array<int>)
11  requires arr != null
12  requires len > 0
13  requires start + len <= arr.Length
14  ensures r != null && r.Length == arr.Length
15  ensures arr[..start] == r[..start]
16  ensures arr[(start + len)..] == r[(start + len)..]
17  ensures rotated(arr[start .. start+len], r[start .. start+len])
18 {
19   var i: nat := 0;
20   r := new int[arr.Length];
21   while i < start
22     invariant i <= start
23     invariant forall k: nat :: k < i ==> r[k] == arr[k]
24   {
25     r[i] := arr[i];
26     i := i + 1;
27   }
28   assert arr[..start] == r[..start];
29   r[start] := arr[start+len-1];
30   assert r[start] == arr[start+len-1];
31
32   i := start+len;
33   while i < start+len
34     invariant start < i <= start+len
35     invariant arr[..start] == r[..start]
36     invariant r[start] == arr[start+len-1]
37     invariant forall k: nat :: start < k < i ==> r[k] == arr[k-1]
38   {
39     r[i] := arr[i-1];
40     i := i + 1;
41   }
42   assert rotated(arr[start .. start+len], r[start .. start+len]);
43
44   i := start+len;
45   while i < arr.Length
46     invariant start+len <= i <= arr.Length
47     invariant arr[..start] == r[..start]
48     invariant rotated(arr[start .. start+len], r[start .. start+len])
49     invariant forall k: nat :: start+len <= k < i ==> r[k] == arr[k]
50   {
51     r[i] := arr[i];
52     i := i + 1;
53   }
54 }

```

```

1 // Status wyc-37: null pointer
2 // wyc-36: loop inv. doesn't hold on entry line: 34
3 function insert(int[] items, int item, int index) -> (int[] r)
4   requires 0 <= index && index < |items|
5   requires |items| > 0
6   ensures |r| == |items| + 1
7   ensures all { k in 0..index | items[k] == r[k] }
8   ensures r[index] == item
9   ensures all { j in index..|r|, k in index..|items|
10     | j == k + 1 ==> items[k] == r[j] }
11 :
12 // length of the new array
13 int newlen = |items| + 1
14 int[] result = [ 0; newlen ]
15 int i = 0
16 //
17 while i < |items|
18   // items before index in result are still the same
19   where 0 <= i && i <= |items|
20   where |result| == newlen
21   where i <= index ==> all { k in 0..i | items[k] == result[k] }
22   where i > index ==> result[index] == item &&
23     all { k in 0..index | items[k] == result[k] }
24   where i > index ==> all { j in index..i, k in index..i
25     | j == k + 1 ==> items[k] == result[j] }
26 :
27   if i < index:
28     result[i] = items[i]
29   else if i == index:
30     result[i] = item
31   else:
32     result[i + 1] = items[i]
33     i = i + 1
34 return result

```

```

1 // Status: verifies and compiles
2 method insert(items: array<int>, item: int, index: nat)
3   returns (r: array<int>)
4   requires items != null && index < items.Length
5   requires items.Length > 0
6   ensures r != null && r.Length == items.Length + 1
7   ensures forall k: nat :: k < index ==> r[k] == items[k]
8   ensures r[index] == item
9   ensures forall k :: index < k < r.Length ==> r[k] == items[k - 1]
10 {
11 // length of the new array
12 var newlen := items.Length + 1;
13 r := new int[newlen];
14 var i: nat := 0;
15
16 while i < index
17   // items before index in result are still the same
18   invariant i <= index
19   decreases index - i
20   invariant r.Length == newlen
21   invariant forall k: nat :: k < i ==> (k < index ==> r[k] == items[k])
22 {
23   r[i] := items[i];
24   i := i + 1;
25 }
26 assert i == index;
27 r[i] := item;
28 i := i + 1;
29 while i < newlen
30   // items after index in result are transposed by one place
31   invariant index < i <= newlen
32   decreases newlen - i
33   invariant r.Length == newlen
34   invariant forall k: nat :: k < index ==> r[k] == items[k]
35   invariant r[index] == item
36   invariant forall k :: index < k < i ==> r[k] == items[k - 1]
37 {
38   r[i] := items[i - 1];
39   i := i + 1;
40 }
41 }

```