Whiley Comparison

Vivian Stewart

January 26, 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" etc. However the truth of Software Verification is in actuality much more of a field of research with very limited tools and steep learning curve. Current efforts are impressive from an academic point of view and require years of invested effort to learn the various analyses and how these pieces fit together to prove correctness of program execution.

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

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 Pierce, 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 characters 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 occurences

Returns the number of duplicate elements that hold a given value in the given array. Accumulating a count of elements in the array matching the given value. The proof requires counting which is an interesting conundrum.

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 returned the index in the given array where the first occurrence of an element matching the given value would be inserted. Or to find an insertion point into the sorted part of the array where the value, upon insertion into that part of the array and maintain sorted order.

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	✓	✓	✓	✓	✓	✓	✓

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. Deduction of bounds of different variables seems to not work in Whiley (not array bounds.) TODO: insert(), merge(), binarySearch(), qsort() The semantics of logic expressions in ACSL (Frama-C), Whiley and Dafny are based on mathematical first-order logic. 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 Degree of Completeness The previous section has taught us that writing a fairly complete specification (in fact we could still add some clauses to the specification above, as we will see in the next chapters) is not immediate, and thus that it is easy to come up with only a partial specification. Hence, it raises two frequently asked questions: how can we be sure that our specification is complete, and how complete must a specification be. The answers however do not lie in ACSL itself. For the first one, one must reason on some model of the specified application. For the second one, there is no definite answer. 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.

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.

```
inductive reachable{L} (list* root, list* node) {
  case root_reachable{L}:
    \forall list* root; reachable(root,root);
  case next_reachable{L}: // L indicates a Label -> memory state
    \forall list* root, *node;
      \valid(root) ==> reachable(root -> next, node) ==>
        reachable(root,node);
```

```
requires \valid(p) && \valid(q);
ensures *p <= *q; // pointers
behavior p_minimum:
    assumes *p < *q;
    ensures *p == \old(*p) && *q == \old(*q);
behavior q_minimum:
    assumes *p >= *q;
    ensures *p == \old(*q) && *q == \old(*p);
complete behaviors p_minimum, q_minimum;
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 the there are attempts in the specification language to mitigate the issue of reference aliasing and other such memory management quirks (modifies clause.) Unlike Whiley 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

```
// Status wyc-37:  verified [96203ms]
// wyc-36:  verified [43551ms] -54.73 percent.
function isPalindrome(int[] chars) -> (bool r)
  ensures r <==> all { x in 0..|chars|
                | chars[x] == chars[|chars| - (x + 1)] }
:
  int i = 0
  int j = |chars|

  while i < j
    where i + j == |chars| && i >= 0
    where all { k in 0..i | chars[k] == chars[|chars| - (k+1)] }
  :
    j = j - 1
    if chars[i] != chars[j]
    :
      return false
    i = i + 1

  return true
```

```
// Status:  verifies and compiles
// Result holds iff array is a palindrome :)
method isPalindrome(chars: array<int>) returns (r: bool)
  requires chars != null
  ensures r <==> forall x: int :: 0 <= x < chars.Length
                 ==> chars[x] == chars[chars.Length - (x + 1)]
{
    var i: nat := 0;
    var j: nat := chars.Length;
    //
    while i < j
      invariant i + j == chars.Length && i >= 0
      invariant forall k: int :: 0 <= k < i
                ==> chars[k] == chars[chars.Length - (k + 1)]
    {
        j := j - 1;
        if chars[i] != chars[j]
        {
            return false;
        }
        i := i + 1;
    }
    return true;
}
```

```
// Status wyc-37:  verifies wyc-36:  verifies
function firstIndexOf(int[] items, int item) -> (int r)
  // If result is positive, element at that position must match item
  ensures r >= 0 ==> items[r] == item
  // If result is positive, no element at lesser position matches item
  ensures r >= 0 ==> no { k in 0..r | items[k] == item }
  // If result is negative, no element matches item
  ensures r <  0 ==> no  { k in 0..|items| | items[k] == item }
:
  int i = 0
  while i < |items|
    // i is increasing and no element at greater position matches item
    where 0 <= i && i <= |items|
    where no { k in 0..i | items[k] == item }
  :
    if items[i] == item
    :
      return i
    i = i + 1
  // didn't find item in entire list
  return -1
```

```
// Status:  verifies
method firstIndexOf(items: array<int>, item: int) returns (r: int)
  requires items != null
  ensures r < items.Length
  // If result is positive, element at that position must match item
  ensures r >= 0 ==> items[r] == item
  // If result is positive, no element at lesser position matches item
  ensures r >= 0 ==> forall k: nat :: k < r ==> items[k] != item
  // If result is negative, no element matches item
  ensures r < 0 ==> forall k: nat :: k < items.Length
                    ==> items[k] != item
{
  var i: int := 0;
  while i < items.Length
    // i is increasing and no element at greater position matches item
    invariant 0 <= i <= items.Length
    invariant forall k: nat :: k < i ==> items[k] != item
  {
    if items[i] == item
      { return i; }
    i := i + 1;
  } // didn't find item in entire list
  return -1;
}
```

```
// Status wyc-37:  verifies [1048ms]
// wyc-36:  verifies [1429ms] -27.77 percent
function lastIndexOf(int[] items, int item) -> (int r)
// If result is positive, element at that position must match item
ensures r >= 0 ==> items[r] == item
// If result is positive, no element at greater position matches item
ensures r >= 0 ==> all { x in (r + 1)..|items| | items[x] != item }
// If result is negative, no element matches item
ensures r < 0 ==> no { x in 0..|items| | items[x] == item }
:
  int i = |items|
  while i >= 0
  // i is decreasing and no element at greater position matches item
  where 0 <= i && i <= |items|
  where no { x in i..|items| | items[x] == item }
  :
    i = i - 1
    if items[i] == item
    :
      return i
  // didn't find item in entire list
  return -1
```

```
// Status:  verifies and compiles
method lastIndexOf(items: array<int>, item: int) returns (r: int)
  requires items != null
  ensures r < items.Length
  // result is positive element at that position must match item
  ensures r >= 0 ==> items[r] == item
  // result is positive no element at greater position matches item
  ensures r >= 0 ==> forall x :: r < x < items.Length
                     ==> items[x] != item
  // If result is negative, no element matches item
  ensures r < 0 ==> forall x :: 0 <= x < items.Length
                    ==> items[x] != item
{
    r := items.Length;
    while r > 0
      // no element at greater position matches item
      invariant 0 <= r <= items.Length
      decreases r
      invariant forall x :: r <= x < items.Length ==> items[x] != item
    {
        r := r - 1;
        if items[r] == item { return; }
    }
    // didn't find item in entire list
    r := -1;
}
```

```
// Status wyc-37:  verifies.  wyc-36:  verifies.
public function maxArray(int[] items) -> (int max)
  requires |items| > 0
  ensures all { k in 0..|items| | max >= items[k] }
:
  int i = 1
  int r = items[0]

  while i < |items|
    where 0 < i && i <= |items|
    where all { k in 0..i | r >= items[k] }
  :
    if items[i] > r:
      r = items[i]
    i = i + 1

  return r
```

```
// Status:  verified
method maxArray(items: array<int>) returns (r: int)
  requires items != null
  requires items.Length > 0
  ensures forall k: nat :: 0 <= k < items.Length ==> r >= items[k]
{
  var i: int := 1;
  r := items[0];

  while i < items.Length
    invariant 0 < i <= items.Length
    invariant forall k: nat :: k < i ==> r >= items[k]
  {
    if items[i] > r
      { r := items[i]; }
    i := i + 1;
  }
}
```

```
// Status wyc-37:  wyc-36:  loop invariant
// not restored line:  16
function occurences(int[] items, int item) -> (int r)
  // some number of occurences of item
  ensures r > 0  ==> some { k in 0..|items|
                         | items[k] == item }
  // no occurences of item
  ensures r == 0 ==> all  { k in 0..|items|
                         | items[k] != item }
:
  int i = 0
  int count = 0
  //
  while i < |items|
    // i is increasing and there could be elements that match
    where 0 <= i && i <= |items|
    where count > 0  ==> some { k in 0..i
                              | items[k] == item }
    where count == 0 ==> all  { k in 0..i
                              | items[k] != item }
  :
    if items[i] == item
    :
      count = count + 1
    i = i + 1
  return count
```

```
function count(items: seq<int>, item: int): nat
  decreases |items|
{
  if |items| == 0 then 0 else
    if items[0] == item
      then (1 + count( items[1..], item ))
      else count( items[1..], item )
}
// Status:  ???
method occurences(items: array<int>, item: int) returns (r: nat)
  requires items != null
  ensures r <= items.Length
  // some number of occurences of item
  ensures r > 0  ==> exists k: nat :: k < items.Length && items[k] == item
  // no occurences of item
  ensures r == 0 ==> forall k: nat :: k < items.Length ==> items[k] != item
  ensures r == count( items[..], item )
{
  var i: int := 0;
  var num: nat := 0;

  while i < items.Length
    // i is increasing and there could be elements that match
    invariant 0 <= i <= items.Length
    invariant num <= items.Length
    invariant num <= i
    invariant num > 0 ==> exists k: nat :: k < i
                          && items[k] == item
    invariant num == 0 ==> forall k: nat :: k < i
                           ==> items[k] != item
    invariant num == count( items[..i], item )
  {
    if items[i] == item
      { num := num + 1; }
    i := i + 1;
  }
  return num;
}
```

```
// Status:  verified and compiled
// wyc-37:  [1197ms] wyc-36:  [1104ms] -7.76
constant NULL is 0
// ASCII character is unsigned 8bit integer
type ASCII_char is (int n) where n >= 0 && n < 256
// C String is array of chars where at least one is NULL
type C_string is (ASCII_char[] chars)
     where some { i in 0..|chars| | chars[i] == NULL }
// Calculate length of string
function strlen(C_string str) -> (int r)
  ensures r >= 0
:
  int i = 0
  //
  while str[i] != NULL
    where i >= 0
    where all { k in 0..i | str[k] != NULL }
  :
    i = i + 1
  return i
```

```
// Status:  verifies and compiles
// Calculate length of string
method strlen(str: array<char>) returns (r: nat)
  requires str != null
  requires exists k: nat :: k < str.Length && str[k] == '\u0000'
{
  r := 0;
  while str[r] != '\u0000'
    invariant r <= str.Length
    invariant forall k: nat :: k < r ==> str[k] != '\u0000'
    decreases str.Length - r
  {
      r := r + 1;
  }
}
```

```
// Status wyc-37:  verifies [3519ms] wyc-36:  verifies [9570ms]
function linearSearch(int[] arr, int val) -> (int r)
  // arr is an ordered array
  requires all { i in 0..|arr|, j in 0..|arr|
                | i < j ==> arr[i] < arr[j] }
  // Return is between -1 and length of arr
  ensures r >= -1 && r < |arr|
  // If index returned, it is first match
  ensures r >= 0 ==> all { k in 0..r | arr[k] < val }
  // If index return, it matches val
  ensures r >= 0 ==> arr[r] == val
  // if failure, no matching element exists
  ensures r == -1 ==> no { k in 0..|arr| | arr[k] == val }:
//
  int i = 0
  //
  while i < |arr|
    where i >= 0 && i <= |arr|
    where all { k in 0..i | arr[k] < val }:
  //
    if arr[i] == val:
      return i
    else if arr[i] > val:
      return -1
    i = i + 1
  return -1
```

```
// Status:  verifies
method linearSearch(arr: array<int>, val: int) returns (r: int)
  requires arr != null
  // arr is an ordered array
  requires forall k: nat :: k < (arr.Length - 1)
           ==> arr[k] < arr[k + 1]
  // Return is between -1 and length of arr
  ensures r >= -1 && r < arr.Length
  // If index returned, it is first match
  ensures r >= 0 ==> forall k: nat :: k < r ==> arr[k] < val
  // If index return, it matches val
  ensures r >= 0 ==> arr[r] == val
  // if failure, no matching element exists
  ensures r == -1 ==> forall k: nat :: k < arr.Length
                      ==> arr[k] != val
{
  var i: int := 0;

  while i < arr.Length
    invariant 0 <= i <= arr.Length
    invariant forall k: nat :: k < i ==> arr[k] < val
  {
    if arr[i] == val
      { return i; }
    else if arr[i] > val
      { return -1; }
    i := i + 1;
  }
  return -1;
}
```

```
// Status:  verifies and compiles?...
// no, index out of bounds (negative) line 25
type nat is (int n) where n >= 0

method binarySearch( int[] items, int key ) -> (int r)
  requires |items| > 0
  requires all { j in 0..|items|, k in 0..|items|
                | j < k ==> items[j] <= items[k] }
  ensures r >= 0 ==> r < |items| && items[r] == key
  ensures r < 0  ==> all { k in 0..|items| | items[k] != key }
:
  nat low = 0
  nat high = |items|
  nat mid = 0
  //
  while low < high
    where low <= mid
    where mid < high
    where high <= |items|
    // elements outside the search range do not equal key
    where no { i in 0..low, j in high..|items|
               | items[i] != key && items[j] != key }
  :
    mid = (low + high) / 2
    if items[mid] < key:
      low = mid + 1
    else if key < items[mid]:
      high = mid
    else:
      return mid
  return -1
```

```
// Status:  verifies and compiles
predicate sorted(s: seq<int>)
{
   forall i,j :: 0 <= i < j < |s| ==> s[i] <= s[j]
}

method BinarySearch(a: array<int>, key: int) returns (index: int)

  requires a != null && sorted(a[..])
  ensures index >= 0 ==> index < a.Length && a[index] == key
  ensures index < 0  ==> forall k: nat :: k < a.Length ==> a[k] != key
{
  var low := 0;
  var high := a.Length;


  while (low < high)
    invariant 0 <= low <= high <= a.Length
    invariant forall i :: 0 <= i < a.Length
              && !(low <= i < high) ==> a[i] != key
  {
      var mid := (low + high) / 2;
      if (a[mid] < key)
        { low:= mid + 1; }
      else if (key < a[mid])
        { high := mid; }
      else
        { return mid; }

  }
  return -1;
}
```

```
// Status:  verifies wyc-37:  [1745ms]
// wyc-36:  [3987ms]  228.48 percent
public function append(int[] items, int item) -> (int[] rs)
  ensures |rs| == |items| + 1
:
  int[] result = [ 0; |items| + 1 ]
  int i = 0
  //
  while i < |items|
    where all { k in 0..i | result[k] == items[k] }
    where i >= 0
    where |result| == |items| + 1
    where i < |result|
  :
    result[i] = items[i]
    i = i + 1
  result[i] = item
  return result
```

```
// Status:  verifies and compiles
// Append a single item onto the end of the array
method append( items: array<int>, item: int ) returns (r: array<int>)
  requires items != null && items.Length > 0
  ensures r != null && r.Length == items.Length + 1
  ensures forall k: int :: 0 <= k < items.Length ==> r[k] == items[k]
{
    r := new int[items.Length + 1];
    var i: nat := 0;

    while i < items.Length
      invariant r.Length == items.Length + 1
      invariant i <= items.Length
      invariant forall k: int :: 0 <= k < i ==> r[k] == items[k]
    {
        r[i] := items[i];
        i := i + 1;
    }
    r[i] := item;
}
```

```
// Status wyc-37:  infinite loop wyc-36:  "GC overhead limit exceeded"
function remove(int[] items, int index) -> (int[] r)
  requires 0 <= index && index < |items|
  requires |items| > 0
  ensures |r| == |items| - 1
  ensures all { k in 0..index   | r[k] == items[k] }
  ensures all { k in index..|r| | r[k] == items[k + 1] }
:
  int newlen = |items| - 1
  int i = 0
  int[] result = [ 0; newlen ]
  //
  while i < index
    // items before index in result are still the same
    where 0 <= i where i <= index
    where |result| == newlen
    where all { k in 0..i | k < index ==> result[k] == items[k] }
  :
    result[i] = items[i]
    i = i + 1
  assert i == index
  while i < newlen
    // items after index in result are transposed by one place
    where index <= i where i <= newlen
    where |result| == newlen
    where all { k in 0..index | result[k] == items[k] }
    where all { k in index..i | result[k] == items[k + 1] }
  :
    result[i] = items[i + 1]
    i = i + 1
  return result
```

```
// Status:  verifies and compiles
// This function should remove the item at the given
// index from the items array, and return the resulting
// array otherwise unchanged.  The resulting array is of
// course one element shorter in length.
method remove(items: array<int>, index: nat) returns (r: array<int>)

  requires items != null && index < items.Length
  requires items.Length > 0
  ensures r != null && r.Length == items.Length - 1
  ensures forall k: nat :: k < index ==> r[k] == items[k]
  ensures forall k :: index <= k < r.Length ==> r[k] == items[k + 1]
{
    // length of the new array
    var newlen := items.Length - 1;
    r := new int[newlen];
    var i: nat := 0;

    while i < index
      // items before index in result are still the same
      invariant i <= index
      decreases index - i
      invariant r.Length == newlen
      invariant forall k: nat :: k < i
                ==> (k < index ==> r[k] == items[k])
    {
        r[i] := items[i];
        i := i + 1;
    }
    assert i == index;
    while i < newlen
      // items after index in result are transposed by one place
      invariant index <= i <= newlen
      decreases newlen - i
      invariant r.Length == newlen
      invariant forall k: nat :: k < index ==> r[k] == items[k]
      invariant forall k :: index <= k < i ==> r[k] == items[k + 1]
    {
        r[i] := items[i + 1];
        i := i + 1;
    }
}
```

```
// Status wyc-37:  null pointer.
// wyc-36:  loop inv.  not restored:  line 33
function copy(int[] src, int sStart, int[] dest, int dStart, int len)
         -> (int[] r)
  // starting points in both arrays cannot be negative
  requires sStart >= 0 && dStart >= 0 && len > 0
  // Source array must contain enough elements to be copied
  requires |src| >= sStart + len
  // Destination array must have enough space for copied elements
  requires |dest| >= dStart + len
  // Result is same size as dest
  ensures |r| == |dest|
  // All elements before copied region are same
  ensures all { k in 0..dStart | r[k] == dest[k] }
  // All elements in copied region match src
  ensures all { k in 0..len | r[dStart + k] == src[sStart + k] }
  // All elements above copied region are same
  ensures all { k in dStart + len..|dest| | r[k] == dest[k] }
:
  int i = 0
  int[] odest = dest
  assert all { k in 0..|dest| | dest[k] == odest[k] }
  //
  while i < len
    where 0 <= i where i <= len
    where |dest| == |odest|
    // all items are still the same before dStart index
    where all { k in 0..dStart | dest[k] == odest[k] }
    // all items after dStart index are still the same
    where all { k in (dStart + len)..|dest| | dest[k] == odest[k] }
    // inbetween items are copied from src
    where all { k in sStart..sStart + i, j in dStart..dStart + i
                  | src[k] == dest[j] }
  :
    dest[dStart + i] = src[sStart + i]
    i = i + 1
  return dest
```

```
// Status:  verifies and compiles
method copy( src: array<int>, sStart: nat
     , dest: array<int>, dStart: nat, len: nat)
    returns (r: array<int>)
  // both arrays cannot be null
  requires dest != null && src != null;
  // Source array must contain enough elements to be copied
  requires src.Length >= sStart + len;
  // Destination array must have enough space for copied elements
  requires dest.Length >= dStart + len;
  // Result is same size as dest
  ensures r != null;
  ensures r.Length == dest.Length;
  // All elements before copied region are same
  ensures r[..dStart] == dest[..dStart];
  // All elements above copied region are same
  ensures r[dStart + len..] == dest[dStart + len..];
  // All elements in copied region match src
  ensures forall k: nat :: k < len
          ==> r[dStart + k] == src[sStart + k];

{
    if len == 0 { return dest; }
    var i: nat := 0;
    r := new int[dest.Length];

    while (i < r.Length)
      invariant i <= r.Length
      invariant forall k: nat :: k < i ==> r[k] == dest[k]
    {
        r[i] := dest[i];
        i := i + 1;
    }
    assert r[..] == dest[..];
    i := 0;
    while (i < len)
      invariant i <= len
      invariant r[..dStart] == dest[..dStart]
      invariant r[dStart..dStart + i] == src[sStart..sStart + i]
      invariant r[dStart + len..] == dest[dStart + len..]
    {
        r[dStart + i] := src[sStart + i];
        i := i + 1;
    }
}
```

```
// Status wyc-37:  null pointer
// wyc-36:  line 37:  loop inv.  !restored

function displace( int[] arr, int start, int len) -> (int[] r)
  requires len > 0
  requires start + len <= |arr|
  ensures |r| == |arr|
  ensures all { k in 0..start | r[k] == arr[k] }
  ensures all { k in (start + len)..|arr| | r[k] == arr[k] }
  ensures all { k in (start + 1)..start + len
              | r[k] == arr[k - 1] }
  ensures r[start] == arr[start + len - 1]
:
  int i = start
  int[] res = arr
  res[start] = arr[start + len - 1]

  while i < |arr|
    where |res| == |arr|
    where (start + len) <= i && i <= |arr|
    where res[start] == arr[start + len - 1]
    where all { k in 0..start | res[k] == arr[k] }
    where all { k in (start + len)..i | res[k] == arr[k] }
    where all { k in start..start + len - 1
              | res[k + 1] == arr[k] }
  :
    res[i] = arr[i]
    i = i + 1

  return res
```

```
// Status:  verifies and compiles
// rotates a region of the array by one place forward
predicate rotated(arr1:seq<int>, arr2:seq<int>)
  requires |arr1| == |arr2|
{
  (foldall i :: 1 <= i < |arr1| ==> arr2[i] == arr1[i - 1]) &&
  ( |arr1| > 0 ==> arr2[0] == arr1[ |arr1| - 1 ] )
}
method displace(arr: array<int>, start: nat, len: nat)
  returns (r: array<int>)
  requires arr != null
  requires len > 0
  requires start + len <= arr.Length
  ensures r != null && r.Length == arr.Length
  ensures arr[..start] == r[..start]
  ensures arr[(start + len)..] == r[(start + len)..]
  ensures rotated(arr[start .. start+len], r[start .. start+len])
{
  var i: nat := 0;
  r := new int[arr.Length];
  while i < start
    invariant i <= start
    invariant forall k: nat :: k < i ==> r[k] == arr[k]
  {
    r[i] := arr[i];
    i := i + 1;
  }
  assert arr[..start] == r[..start];
  r[start] := arr[start+len-1];
  assert r[start] == arr[start+len-1];

  i := start+1;
  while i < start+len
    invariant start < i <= start+len
    invariant arr[..start] == r[..start]
    invariant r[start] == arr[start+len-1]
    invariant forall k: nat :: start < k < i ==> r[k] == arr[k-1]
  {
    r[i] := arr[i-1];
    i := i + 1;
  }
  assert rotated(arr[start .. start+len], r[start .. start+len]);

  i := start+len;
  while i < arr.Length
    invariant start+len <= i <= arr.Length
    invariant arr[..start] == r[..start]
    invariant rotated(arr[start .. start+len], r[start .. start+len])
    invariant forall k: nat :: start+len <= k < i ==> r[k] == arr[k]
  {
    r[i] := arr[i];
    i := i + 1;
  }
}
```

```
// Status wyc-37:  null pointer
// wyc-36:  loop inv.  doesn't hold on entry line:  34
function insert(int[] items, int item, int index) -> (int[] r)
  requires 0 <= index && index < |items|
  requires |items| > 0
  ensures |r| == |items| + 1
  ensures all { k in 0..index | items[k] == r[k] }
  ensures r[index] == item
  ensures all { j in index..|r|, k in index..|items|
              | j == k + 1 ==> items[k] == r[j] }
:
  // length of the new array
  int newlen = |items| + 1
  int[] result = [ 0; newlen ]
  int i = 0
  //
  while i < |items|
    // items before index in result are still the same
    where 0 <= i && i <= |items|
    where |result| == newlen
    where i <= index ==> all { k in 0..i | items[k] == result[k] }
    where i > index ==> result[index] == item &&
                        all { k in 0..index | items[k] == result[k] }
    where i > index ==> all { j in index..i, k in index..i
                            | j == k + 1 ==> items[k] == result[j] }
  :
    if i < index:
      result[i] = items[i]
    else if i == index:
      result[i] = item
    else:
      result[i + 1] = items[i]
    i = i + 1
  return result
```

```
// Status:  verifies and compiles
method insert(items: array<int>, item: int, index: nat)
  returns (r: array<int>)
  requires items != null && index < items.Length
  requires items.Length > 0
  ensures r != null && r.Length == items.Length + 1
  ensures forall k: nat :: k < index ==> r[k] == items[k]
  ensures r[index] == item
  ensures forall k :: index < k < r.Length ==> r[k] == items[k - 1]
{
    // length of the new array
    var newlen := items.Length + 1;
    r := new int[newlen];
    var i: nat := 0;

    while i < index
      // items before index in result are still the same
      invariant i <= index
      decreases index - i
      invariant r.Length == newlen
      invariant forall k: nat :: k < i ==> (k < index ==> r[k] == items[k])
    {
        r[i] := items[i];
        i := i + 1;
    }
    assert i == index;
    r[i] := item;
    i := i + 1;
    while i < newlen
      // items after index in result are transposed by one place
      invariant index < i <= newlen
      decreases newlen - i
      invariant r.Length == newlen
      invariant forall k: nat :: k < index ==> r[k] == items[k]
      invariant r[index] == item
      invariant forall k :: index < k < i ==> r[k] == items[k - 1]
    {
        r[i] := items[i - 1];
        i := i + 1;
    }
}
```