VSComp Verification

Aisha Syed

Introduction

- Yearly competition with different variants
  - VSComp, VSTTE
- Aim is to write code for a set of challenge problems then verify properties in limited time
- Anyone can participate
- Any verification tools can be used
  - Dafny, VCC, etc.

Chosen Tools and Problems

- Used Dafny for coding and verifying
- Chose four problems:
  1. *Sum-Max*
  2. *Inversion*
  3. *Two-Duplets*
  4. *TwoWaySort*
  - Will give quick overview of first 3 problems
  - Will discuss 4th one in some detail

1. Sum-Max

- Problem:
  - Given N-element array A, computes sum of all elements, and the maximum element
- Verify:
  - Given **N >= 0 and A[i] >= 0** *for 0 <= i < N*
    - prove the post-condition that **sum <= N * Max**

2. Inversion

- Problem: Invert an injective array A on N elements in the subrange from 0 to N – 1.
- Verify:
  - Verify output array B is also injective:
    - B [A[i]] = i *(0 <= i < N)*
    - Example:
      - A = | 2 | 0 | 3 | 1 |
      - B =

0	1	2	3
1	3	0	2

3. Two-Duplets

- Problem:
  - Find two duplets in given array
  - Assume at least two unique duplets exist in array
  - Assume input array a of length N+2 with N >= 2.
  - May assume array values between 0 and N-1.
- Verify:
  - Verify post-condition:
    - Program finds 2 duplets from given array,
      - Duplet1 != Duplet2

4. TwoWay Sort

- Problem:
  - Sort boolean array
    - using two pointers moving from beginning and end of array
      - swapping of values when **TRUE** seen before **FALSE**
- Verify:
  1. *Safety*: every array access within bounds
  2. *Termination*: Prove function always terminates
  3. *Behavior*: After execution, following hold:

     (a) array is a permutation of initial contents

     (b) array is sorted in increasing order.

```
method TwoWaySort(a:array<bool>)
  requires a != null; modifies a;
  ensures compare(a, old(a)); //ensuring permutation
  ensures forall k:: forall l:: 0 <= k < l < a.Length
                                ==> !a[k] || a[l]; //sortedness
{
    i := 0
    j := a.Length - 1
    while (i <= j)
    invariant 0 <= i <= (j+1);
    invariant (i-1) <= j < a.Length;
    decreases (j+1) - i;
    decreases j - (i-1);
    invariant forall k:: 0 <= k < i ==> !a[k];
    invariant forall l:: j < l < a.Length ==> a[l];
    {
        ... //move pointers i,j and swap when TRUE before FALSE
    }
}
```

1. Safety!

```
method TwoWaySort(a:array<bool>)
  requires a != null; modifies a;
  ensures compare(a, old(a)); //ensuring permutation
  ensures forall k:: forall l:: 0 <= k < l < a.Length
                                ==> !a[k] || a[l]; //sortedness
{
    i := 0
    j := a.Length - 1
    while (i <= j)
    invariant 0 <= i <= (j+1);
    invariant (i-1) <= j < a.Length;
    decreases (j+1) - i;
    decreases j - (i-1);
    invariant forall k:: 0 <= k < i ==> !a[k];
    invariant forall l:: j < l < a.Length ==> a[l];
    {
        ... //move pointers i,j and swap values as needed
    }
}
```

2. Termination!

3. Behavior!

```
method TwoWaySort(a:array<bool>)
  requires a != null; modifies a;
  ensures compare(a, old(a)); //ensuring permutation
  ensures forall k:: forall l:: 0 <= k < l < a.Length
                                ==> !a[k] || a[l]; //sortedness
{
    i := 0
    j := a.Length - 1
    while (i <= j)
    invariant 0 <= i <= (j+1);
    invariant (i-1) <= j < a.Length;
    decreases (j+1) - i;
    decreases j - (i-1);
    invariant forall k:: 0 <= k < i ==> !a[k];
    invariant forall l:: j < l < a.Length ==> a[l];
    {
        ... //move pointers i,j and swap values as needed
    }
}
```

3. Behavior!

```
method TwoWaySort(a:array<bool>)
  requires a != null; modifies a;
  ensures compare(a, old(a)); //ensuring permutation
  ensures forall k:: forall l:: 0 <= k < l < a.Length
                                ==> !a[k] || a[l]; //sortedness
{
    i := 0
    j := a.Length - 1
    while (i <= j)
    invariant 0 <= i <= (j+1);
    invariant (i-1) <= j < a.Length;
    decreases (j+1) - i;
    decreases j - (i-1);
    invariant forall k:: 0 <= k < i ==> !a[k];
    invariant forall l:: j < l < a.Length ==> a[l];
    {
        ... //move pointers i,j and swap values as needed
    }
}
```

TRUE – FALSE X

3. Behavior!

```
method TwoWaySort(a:array<bool>)
  requires a != null; modifies a;
  ensures compare(a, old(a)); //ensuring permutation
  ensures forall k:: forall l:: 0 <= k < l < a.Length
                                ==> !a[k] || a[l]; //sortedness
{
    i := 0
    j := a.Length - 1
    while (i <= j)
    invariant 0 <= i <= (j+1);
    invariant (i-1) <= j < a.Length;
    decreases (j+1) - i;
    decreases j - (i-1);
    invariant forall k:: 0 <= k < i ==> !a[k];
    invariant forall l:: j < l < a.Length ==> a[l];
    {
        ... //move pointers i,j and swap values as needed
    }
}
```

```
predicate compare (oldArray: array<bool>,
                   newArray: array<bool>)
reads a; reads a1;
requires a != null; requires a1 != null;
{
   forall key:bool :: total(oldArray, oldArray.Length, key)
                                        ==
                      total(newArray, newArray.Length, key)
}
```

```
function total(a: array<bool>, i:int, key: bool): int
requires a != null; reads a;
{
    if i < 0
        then 0
    else if (a.Length == 0)
        then 0
    else if 0<=i<a.Length && a[i] == key
        then total(a, i-1, key) + 1
    else total(a, i-1, key)
}
```

```
method TwoWaySort(a:array<bool>)
  requires a != null; modifies a;
  ensures compare(a, old(a)); //ensuring permutation
  ensures forall k:: forall l:: 0 <= k < l < a.Length
                                ==> !a[k] || a[l]; //sortedness
{
    i := 0
    j := a.Length - 1
    while (i <= j)
    invariant 0 <= i <= (j+1);
    invariant (i-1) <= j < a.Length;
    decreases (j+1) - i;
    decreases j - (i-1);
    invariant forall k:: 0 <= k < i ==> !a[k];
    invariant forall l:: j < l < a.Length ==> a[l];
    {
        ... //move pointers i,j and swap values as needed
    }
}
```

3. Behavior!

Thank You