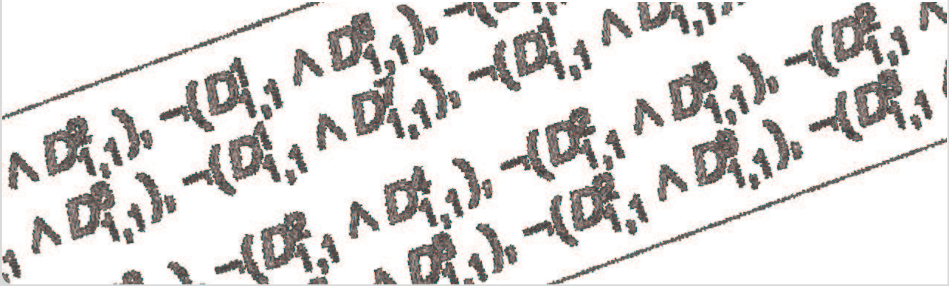
Bridging the interaction gap between logic and code

Sarah Grebing, Mattias Ulbrich

Institute for Theoretical Informatics, KIT



Motivation

Program verification is an iterative process.

- initial attempts (often) fail
- understand reason for failing is crucial

Interaction is ...

- inspection of proof state
- advancing proof state

..for each iteration

Motivation

Interaction on:

- specification
- program code
- logic/proof obligation

Switching levels is costly and not well supported

Seamless Integration

Overall Goal

Interaction concept supporting

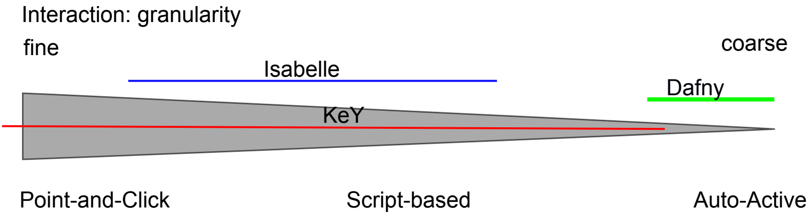
- interactive
- semi-automated
- seamless
- coupled
- fluent

proof guidance for an effective and efficient proof process.

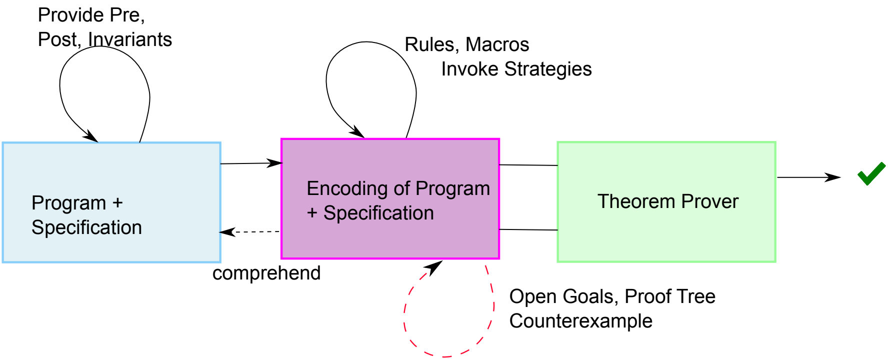
State-of-the-Art Verification Systems

Three interaction principles for guiding the proof search:

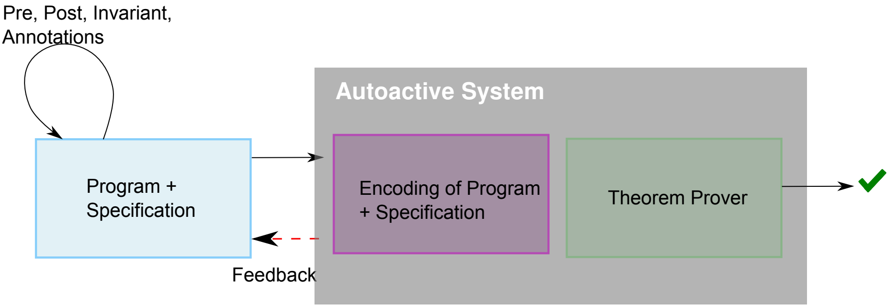
- autoactive
- point-and-click style
- script-based



Interactive Point-and-Click



Autoactive



Pre, Post, Invariant, Annotations

Program + Specification

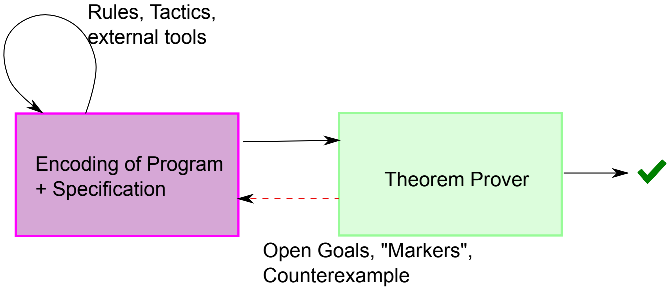
Autoactive System

Encoding of Program + Specification

Theorem Prover

Feedback

Text/Script-Based Interaction



Interactive Point-and-Click

Pros and Cons

+/- all necessary information available

+ full proof control

- interaction can be tedious

- error recovery

- two mental models have to be kept in sync

Autoactive

Pros and Cons

- + interaction on input representation → only one mental model
- + comprehensible annotations
- - missing detailed insight into logical level when proof attempt fails
- - leaky abstraction

Text/Script-Based interaction

Pros and Cons

+ problem/proof decomposition in smaller parts
+ interaction steps more coarse grained than point-and-click → more readable/comprehensible and proof plan expressible

+/- limited insight into logical level

- two mental models may have to be kept in sync

Seamless Integration

Interaction concept supporting

- interactive
- semi-automated
- seamless

proof guidance for an effective and efficient proof process.

Support:

- comprehension of failure of proof attempt
- advancing proof

Seamless Integration

Interaction concept supporting

- interactive
- semi-automated
- seamless

proof guidance for an effective and efficient proof process.

Support:

- comprehension of failure of proof attempt
  - feedback on input language
  - lightweight tools to discharge simple proof problems
  - modularize proof
- advancing proof

Seamless Integration

Interaction concept supporting

- interactive
- semi-automated
- seamless

proof guidance for an effective and efficient proof process.

Support:

- comprehension of failure of proof attempt
  - feedback on input language
  - lightweight tools to discharge simple proof problems
  - modularize proof
- advancing proof
  - proof exploration
  - fast error recovery

Decrease Costs of Iterations

Goal:

Interaction on suitable abstraction level

Decrease Costs of Iterations

Goal:

Interaction on suitable abstraction level (problem and user dependent)

Decrease Costs of Iterations

Goal:

Interaction on suitable abstraction level (problem and user dependent)

Decrease number of level-changes

- stay on input language as long as possible (input and feedback)
- coarse proof steps (proof plan)
- clear overview over overall proof state

Decrease Costs of Iterations

Goal:

Interaction on suitable abstraction level (problem and user dependent)

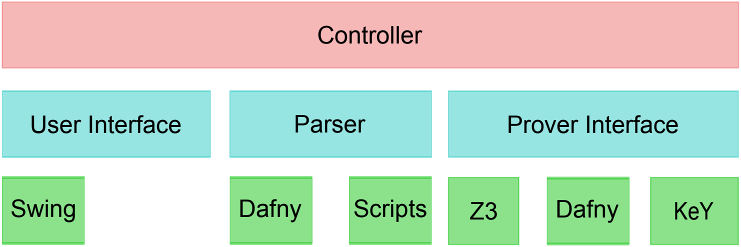
Decrease number of level-changes

- stay on input language as long as possible (input and feedback)
- coarse proof steps (proof plan)
- clear overview over overall proof state

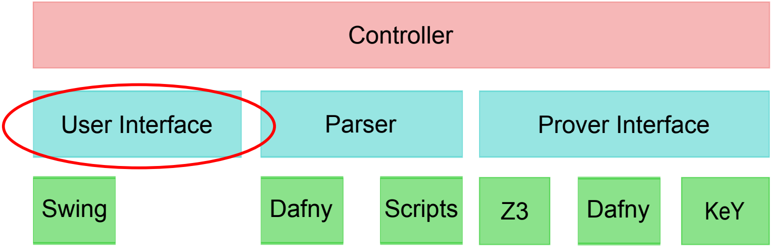
Decrease cost per level-change

- state inspection on logical level
- visible dependencies between levels
- proof exploration on logical level

Architecture



Architecture


Controller

User Interface

Parser

Prover Interface

Swing

Dafny

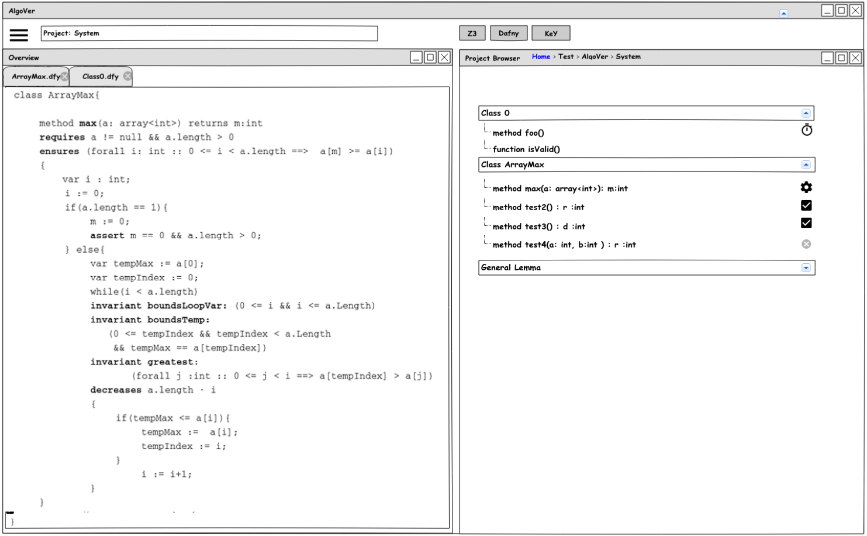
Scripts

Z3

Dafny

KeY

Example



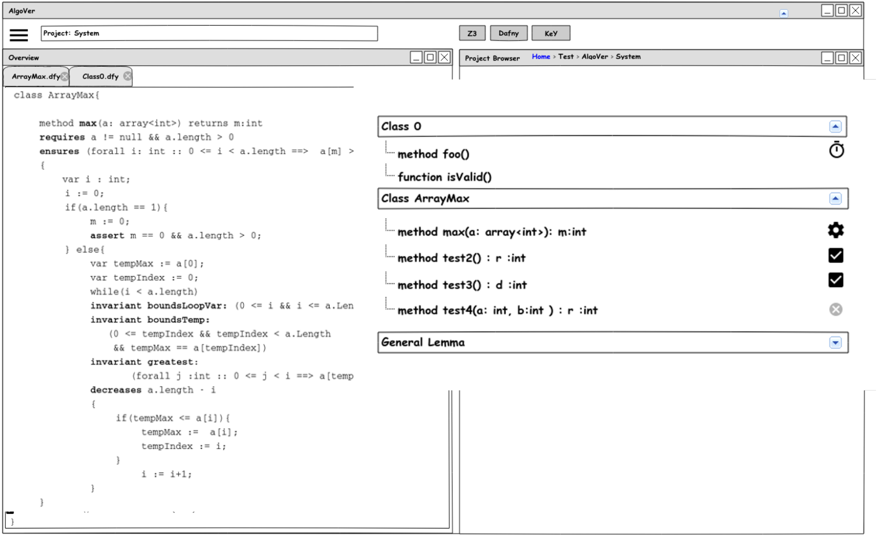
```
class ArrayMax{

    method max(a: array<int>) returns m:int
    requires a != null && a.length > 0
    ensures (forall i: int :: 0 <= i < a.length ==>  a[m] >= a[i])
    {
        var i : int;
         i := 0;
         if(a.length == 1){
             m := 0;
             assert m == 0 && a.length > 0;
         } else{
             var tempMax := a[0];
             var tempIndex := 0;
             while(i < a.length)
             invariant boundsLoopVar: (0 <= i && i <= a.Length)
             invariant boundsTemp:
                (0 <= tempIndex && tempIndex < a.Length
                 && tempMax == a[tempIndex])
             invariant greatest:
                    (forall j :int :: 0 <= j < i ==> a[tempIndex] > a[j])
             decreases a.length - i
             {
                 if(tempMax <= a[i]){
                     tempMax :=  a[i];
                     tempIndex := i;
                 }
                     i := i+1;
             }
    }
}
```

Class 0
- method foo() ⏱
- function isValid()

Class ArrayMax
- method max(a: array<int>): m:int ⚙
- method test2() : r :int ☑
- method test3() : d :int ☑
- method test4(a: int, b:int) : r :int ⊗

General Lemma

Example



```
class ArrayMax{

    method max(a: array<int>) returns m:int
    requires a != null && a.length > 0
    ensures (forall i: int :: 0 <= i < a.length ==>  a[m] >
    {
        var i : int;
         i := 0;
         if(a.length == 1){
             m := 0;
             assert m == 0 && a.length > 0;
         } else{
             var tempMax := a[0];
             var tempIndex := 0;
             while(i < a.length)
             invariant boundsLoopVar: (0 <= i && i <= a.Len
             invariant boundsTemp:
                (0 <= tempIndex && tempIndex < a.Length
                 && tempMax == a[tempIndex])
             invariant greatest:
                    (forall j :int :: 0 <= j < i ==> a[temp
             decreases a.length - i
             {
                 if(tempMax <= a[i]){
                     tempMax :=  a[i];
                     tempIndex := i;
                 }
                     i := i+1;
             }
    }
}
```

Class 0

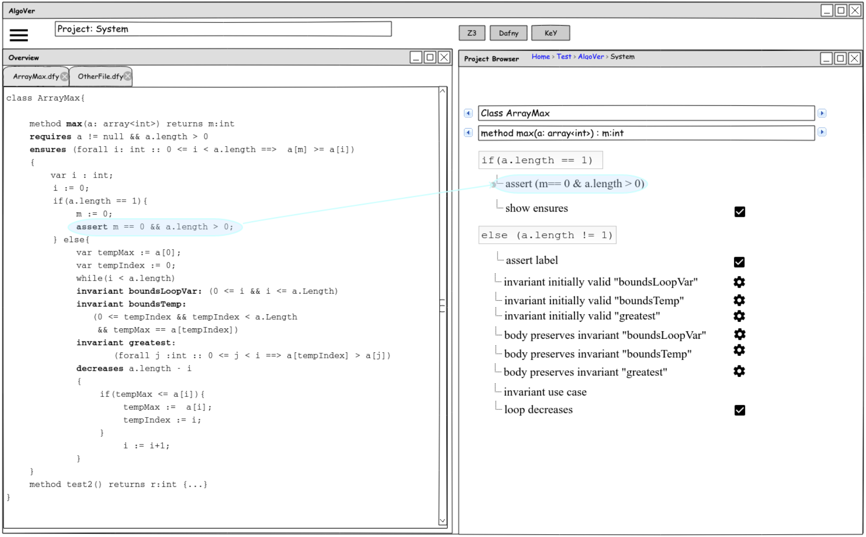
- method foo() ⏱
- function isValid()

Class ArrayMax

- method max(a: array<int>): m:int ⚙
- method test2() : r :int ☑
- method test3() : d :int ☑
- method test4(a: int, b:int) : r :int ⊗

General Lemma

Example



```
class ArrayMax{

    method max(a: array<int>) returns m:int
    requires a != null && a.length > 0
    ensures (forall i: int :: 0 <= i < a.length ==>  a[m] >= a[i])
    {
        var i : int;
        i := 0;
        if(a.length == 1){
            m := 0;
            assert m == 0 && a.length > 0;
        } else{
            var tempMax := a[0];
            var tempIndex := 0;
            while(i < a.length)
            invariant boundsLoopVar: (0 <= i && i <= a.Length)
            invariant boundsTemp:
               (0 <= tempIndex && tempIndex < a.Length
                && tempMax == a[tempIndex])
            invariant greatest:
                   (forall j :int :: 0 <= j < i ==> a[tempIndex] > a[j])
            decreases a.length - i
            {
                if(tempMax <= a[i]){
                    tempMax :=  a[i];
                    tempIndex := i;
                }
                    i := i+1;
            }
    }
    method test2() returns r:int {...}
}
```

Class ArrayMax

method max(a: array<int>) : m:int

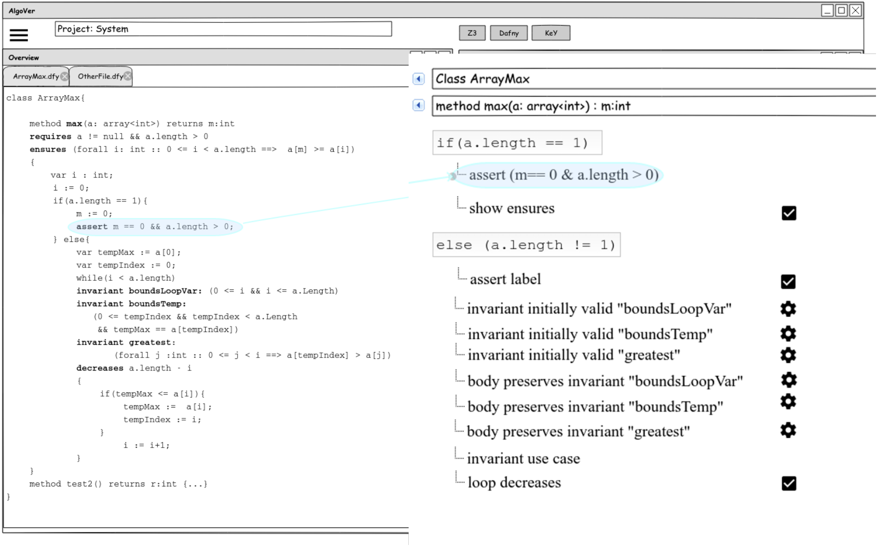
`if(a.length == 1)`

- assert (m== 0 & a.length > 0)
- show ensures ☑

`else (a.length != 1)`

- assert label ☑
- invariant initially valid "boundsLoopVar" ⚙
- invariant initially valid "boundsTemp" ⚙
- invariant initially valid "greatest" ⚙
- body preserves invariant "boundsLoopVar" ⚙
- body preserves invariant "boundsTemp" ⚙
- body preserves invariant "greatest" ⚙
- invariant use case
- loop decreases ☑

Example



```
class ArrayMax{

    method max(a: array<int>) returns m:int
    requires a != null && a.length > 0
    ensures (forall i: int :: 0 <= i < a.length ==>  a[m] >= a[i])
    {
        var i : int;
         i := 0;
         if(a.length == 1){
             m := 0;
             assert m == 0 && a.length > 0;
         } else{
             var tempMax := a[0];
             var tempIndex := 0;
             while(i < a.length)
             invariant boundsLoopVar: (0 <= i && i <= a.Length)
             invariant boundsTemp:
                (0 <= tempIndex && tempIndex < a.Length
                 && tempMax == a[tempIndex])
             invariant greatest:
                    (forall j :int :: 0 <= j < i ==> a[tempIndex] > a[j])
             decreases a.length - i
             {
                 if(tempMax <= a[i]){
                     tempMax :=  a[i];
                     tempIndex := i;
                 }
                     i := i+1;
             }
    }
    method test2() returns r:int {...}
}
```

Class ArrayMax

method max(a: array<int>) : m:int

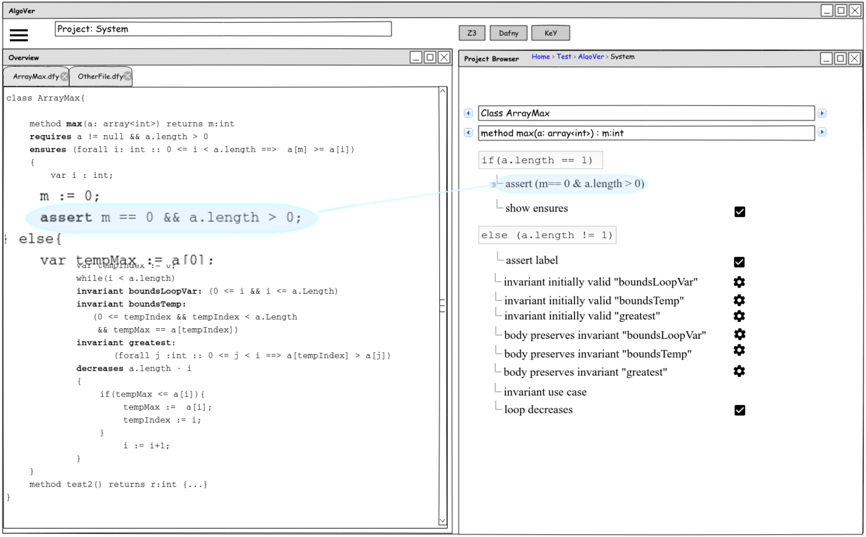
`if(a.length == 1)`

- assert (m== 0 & a.length > 0)
- show ensures ☑

`else (a.length != 1)`

- assert label ☑
- invariant initially valid "boundsLoopVar" ⚙
- invariant initially valid "boundsTemp" ⚙
- invariant initially valid "greatest" ⚙
- body preserves invariant "boundsLoopVar" ⚙
- body preserves invariant "boundsTemp" ⚙
- body preserves invariant "greatest" ⚙
- invariant use case
- loop decreases ☑

Example



```
class ArrayMax{

    method max(a: array<int>) returns m:int
    requires a != null && a.length > 0
    ensures (forall i: int :: 0 <= i < a.length ==>  a[m] >= a[i])
    {
       var i : int;
      m := 0;
      assert m == 0 && a.length > 0;
} else{
      var tempMax := a[0];
          var tempIndex := 0;
          while(i < a.length)
          invariant boundsLoopVar: (0 <= i && i <= a.Length)
          invariant boundsTemp:
             (0 <= tempIndex && tempIndex < a.Length
              && tempMax == a[tempIndex])
          invariant greatest:
                 (forall j :int :: 0 <= j < i ==> a[tempIndex] > a[j])
          decreases a.length - i
          {
              if(tempMax <= a[i]){
                  tempMax :=  a[i];
                  tempIndex := i;
              }
                  i := i+1;
          }
    }
    method test2() returns r:int {...}
}
```

Class ArrayMax

method max(a: array<int>) : m:int

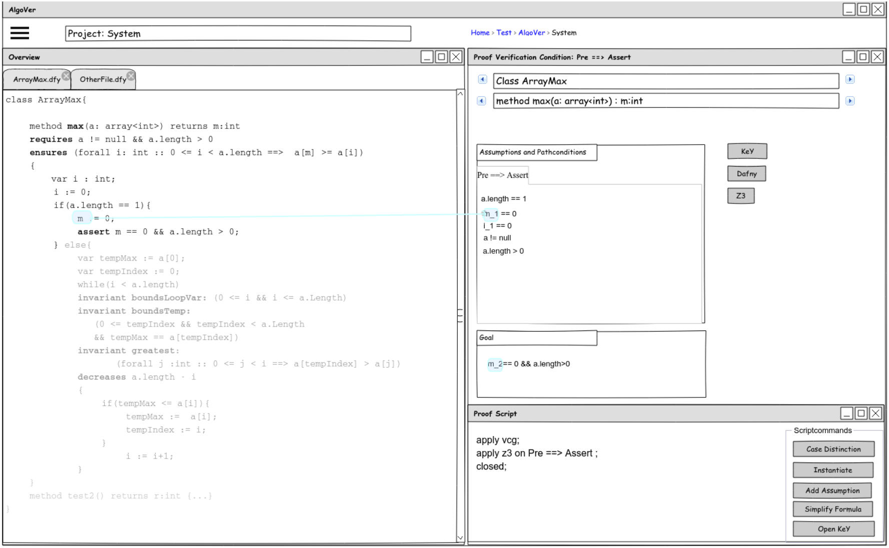
`if(a.length == 1)`

- assert (m== 0 & a.length > 0)
- show ensures ☑

`else (a.length != 1)`

- assert label ☑
- invariant initially valid "boundsLoopVar" ⚙
- invariant initially valid "boundsTemp" ⚙
- invariant initially valid "greatest" ⚙
- body preserves invariant "boundsLoopVar" ⚙
- body preserves invariant "boundsTemp" ⚙
- body preserves invariant "greatest" ⚙
- invariant use case
- loop decreases ☑

Example



```
class ArrayMax{

    method max(a: array<int>) returns m:int
    requires a != null && a.length > 0
    ensures (forall i: int :: 0 <= i < a.length ==>  a[m] >= a[i])
    {
        var i : int;
        i := 0;
        if(a.length == 1){
            m := 0;
            assert m == 0 && a.length > 0;
        } else{
            var tempMax := a[0];
            var tempIndex := 0;
            while(i < a.length)
            invariant boundsLoopVar: (0 <= i && i <= a.Length)
            invariant boundsTemp:
               (0 <= tempIndex && tempIndex < a.Length
               && tempMax == a[tempIndex])
            invariant greatest:
                   (forall j :int :: 0 <= j < i ==> a[tempIndex] > a[j])
            decreases a.length - i
            {
                if(tempMax <= a[i]){
                    tempMax :=  a[i];
                    tempIndex := i;
                }
                    i := i+1;
            }
    }
    method test2() returns r:int {...}
}
```

Pre ==> Assert

```
a.length == 1
m_1 == 0
i_1 == 0
a != null
a.length > 0
```

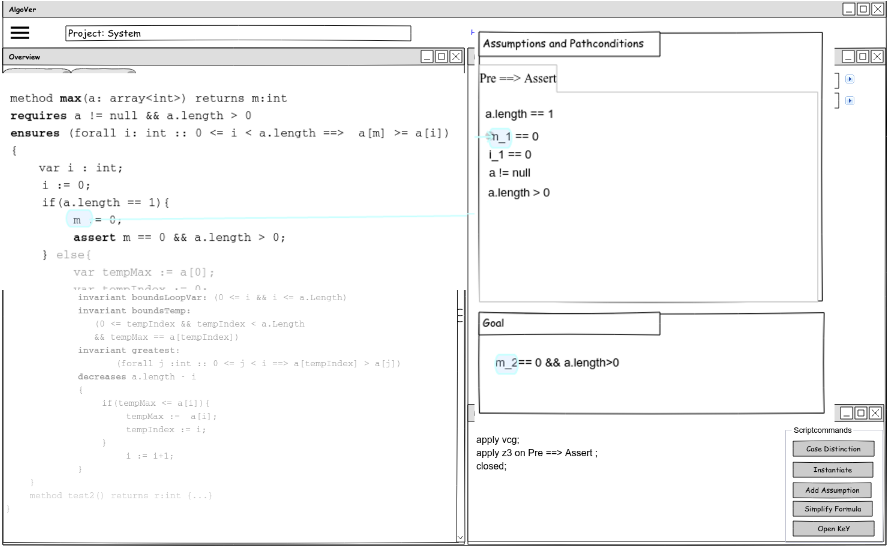
Goal

```
m_2== 0 && a.length>0
```

Proof Script

```
apply vcg;
apply z3 on Pre ==> Assert ;
closed;
```

Example



```
method max(a: array<int>) returns m:int
requires a != null && a.length > 0
ensures (forall i: int :: 0 <= i < a.length ==>  a[m] >= a[i])
{
    var i : int;
     i := 0;
     if(a.length == 1){
         m  := 0;
         assert m == 0 && a.length > 0;
     } else{
         var tempMax := a[0];
         var tempIndex := 0;
          invariant boundsLoopVar: (0 <= i && i <= a.Length)
          invariant boundsTemp:
             (0 <= tempIndex && tempIndex < a.Length
             && tempMax == a[tempIndex])
          invariant greatest:
                 (forall j :int :: 0 <= j < i ==> a[tempIndex] > a[j])
          decreases a.length - i
          {
              if(tempMax <= a[i]){
                  tempMax :=  a[i];
                  tempIndex := i;
              }
                  i := i+1;
          }
     }
     method test2() returns r:int {...}
}
```

a.length == 1

m_1 == 0

i_1 == 0

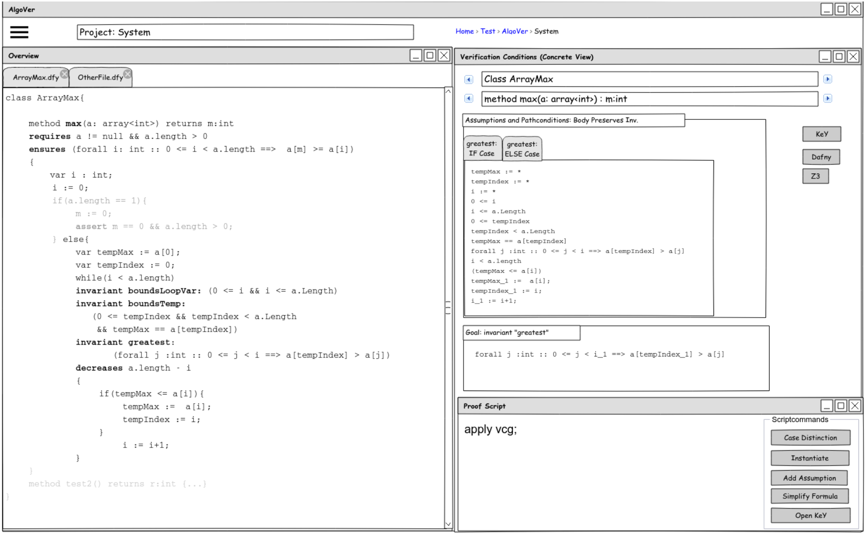
a != null

a.length > 0

m_2== 0 && a.length>0

```
apply vcg;
apply z3 on Pre ==> Assert ;
closed;
```

Example



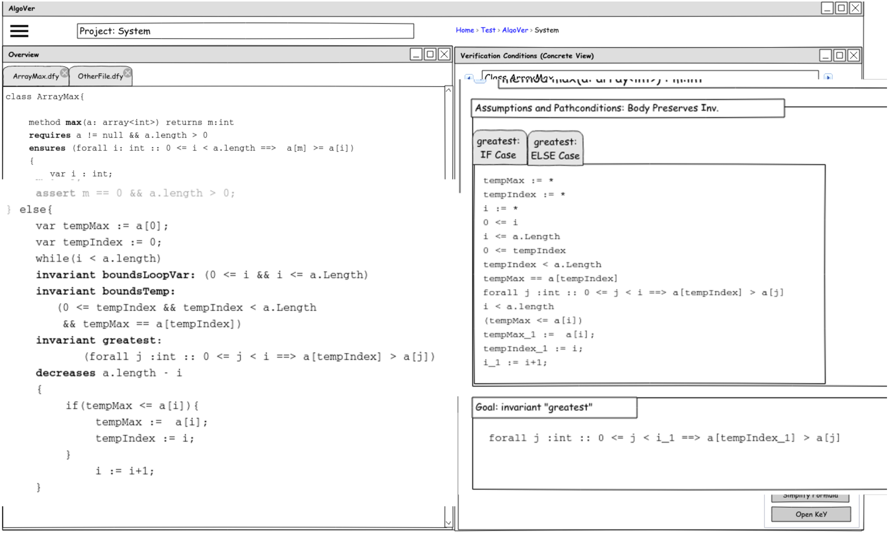
```
class ArrayMax{

    method max(a: array<int>) returns m:int
    requires a != null && a.length > 0
    ensures (forall i: int :: 0 <= i < a.length ==>  a[m] >= a[i])
    {
        var i : int;
        i := 0;
        if(a.length == 1){
            m := 0;
            assert m == 0 && a.length > 0;
        } else{
            var tempMax := a[0];
            var tempIndex := 0;
            while(i < a.length)
            invariant boundsLoopVar: (0 <= i && i <= a.Length)
            invariant boundsTemp:
                (0 <= tempIndex && tempIndex < a.Length
                 && tempMax == a[tempIndex])
            invariant greatest:
                    (forall j :int :: 0 <= j < i ==> a[tempIndex] > a[j])
            decreases a.length - i
            {
                if(tempMax <= a[i]){
                    tempMax :=  a[i];
                    tempIndex := i;
                }
                    i := i+1;
            }
    }
    method test2() returns r:int {...}
}
```

```
tempMax := *
tempIndex := *
i := *
0 <= i
i <= a.Length
0 <= tempIndex
tempIndex < a.Length
tempMax == a[tempIndex]
forall j :int :: 0 <= j < i ==> a[tempIndex] > a[j]
i < a.length
(tempMax <= a[i])
tempMax_1 :=  a[i];
tempIndex_1 := i;
i_1 := i+1;
```

```
forall j :int :: 0 <= j < i_1 ==> a[tempIndex_1] > a[j]
```

```
apply vcg;
```

Example

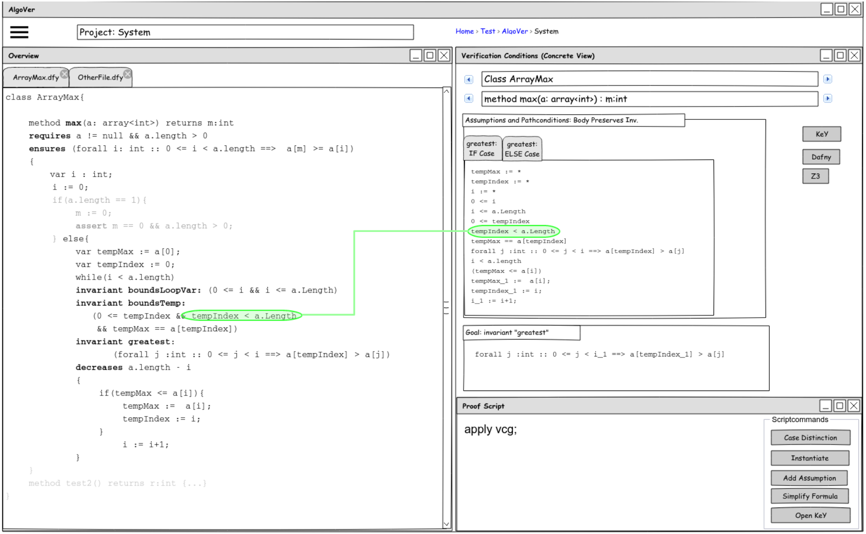

AlgoVer
Project: System
Home > Test > AlgoVer > System
Overview
ArrayMax.dfy
OtherFile.dfy
Verification Conditions (Concrete View)
Class ArrayMax
Assumptions and Pathconditions: Body Preserves Inv.
greatest: IF Case
greatest: ELSE Case
Goal: invariant "greatest"
Simplify Formula
Open KeY

```
class ArrayMax{

    method max(a: array<int>) returns m:int
    requires a != null && a.length > 0
    ensures (forall i: int :: 0 <= i < a.length ==>  a[m] >= a[i])
    {
        var i : int;
     assert m == 0 && a.length > 0;
} else{
     var tempMax := a[0];
     var tempIndex := 0;
     while(i < a.length)
     invariant boundsLoopVar: (0 <= i && i <= a.Length)
     invariant boundsTemp:
        (0 <= tempIndex && tempIndex < a.Length
         && tempMax == a[tempIndex])
     invariant greatest:
            (forall j :int :: 0 <= j < i ==> a[tempIndex] > a[j])
     decreases a.length - i
     {
         if(tempMax <= a[i]){
             tempMax :=  a[i];
             tempIndex := i;
         }
             i := i+1;
     }
```

```
tempMax := *
tempIndex := *
i := *
0 <= i
i <= a.Length
0 <= tempIndex
tempIndex < a.Length
tempMax == a[tempIndex]
forall j :int :: 0 <= j < i ==> a[tempIndex] > a[j]
i < a.length
(tempMax <= a[i])
tempMax_1 :=  a[i];
tempIndex_1 := i;
i_1 := i+1;
```

```
forall j :int :: 0 <= j < i_1 ==> a[tempIndex_1] > a[j]
```

Example



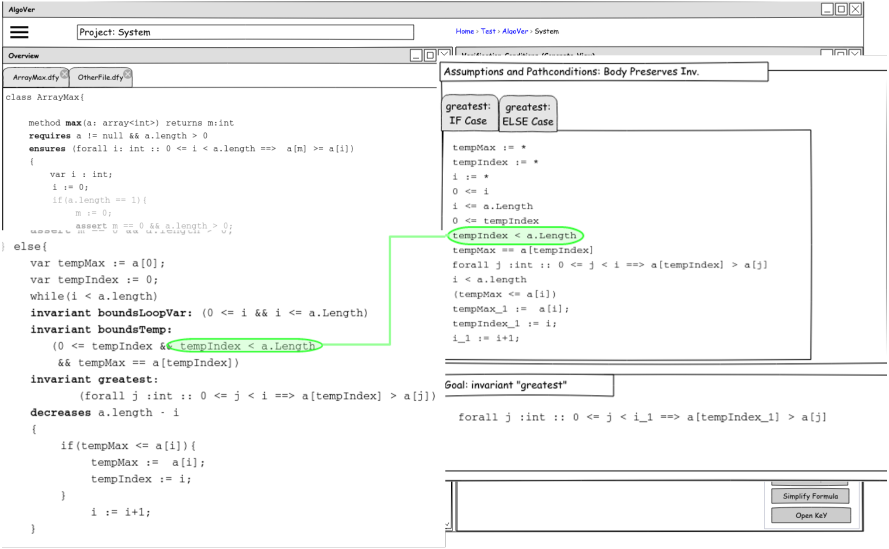
```
class ArrayMax{

    method max(a: array<int>) returns m:int
    requires a != null && a.length > 0
    ensures (forall i: int :: 0 <= i < a.length ==>  a[m] >= a[i])
    {
        var i : int;
        i := 0;
        if(a.length == 1){
            m := 0;
            assert m == 0 && a.length > 0;
        } else{
            var tempMax := a[0];
            var tempIndex := 0;
            while(i < a.length)
            invariant boundsLoopVar: (0 <= i && i <= a.Length)
            invariant boundsTemp:
               (0 <= tempIndex && tempIndex < a.Length
                && tempMax == a[tempIndex])
            invariant greatest:
                  (forall j :int :: 0 <= j < i ==> a[tempIndex] > a[j])
            decreases a.length - i
            {
                if(tempMax <= a[i]){
                    tempMax :=  a[i];
                    tempIndex := i;
                }
                    i := i+1;
            }
    }
    method test2() returns r:int {...}
}
```

```
tempMax := *
tempIndex := *
i := *
0 <= i
i <= a.Length
0 <= tempIndex
tempIndex < a.Length
tempMax == a[tempIndex]
forall j :int :: 0 <= j < i ==> a[tempIndex] > a[j]
i < a.length
(tempMax <= a[i])
tempMax_1 :=  a[i];
tempIndex_1 := i;
i_1 := i+1;
```

```
forall j :int :: 0 <= j < i_1 ==> a[tempIndex_1] > a[j]
```

```
apply vcg;
```

Example



```
class ArrayMax{

    method max(a: array<int>) returns m:int
    requires a != null && a.length > 0
    ensures (forall i: int :: 0 <= i < a.length ==>  a[m] >= a[i])
    {
        var i : int;
         i := 0;
         if(a.length == 1){
             m := 0;
             assert m == 0 && a.length > 0;
} else{
    var tempMax := a[0];
    var tempIndex := 0;
    while(i < a.length)
    invariant boundsLoopVar: (0 <= i && i <= a.Length)
    invariant boundsTemp:
        (0 <= tempIndex && tempIndex < a.Length
         && tempMax == a[tempIndex])
    invariant greatest:
            (forall j :int :: 0 <= j < i ==> a[tempIndex] > a[j])
    decreases a.length - i
    {
        if(tempMax <= a[i]){
            tempMax :=  a[i];
            tempIndex := i;
        }
            i := i+1;
    }
```

Assumptions and Pathconditions: Body Preserves Inv.

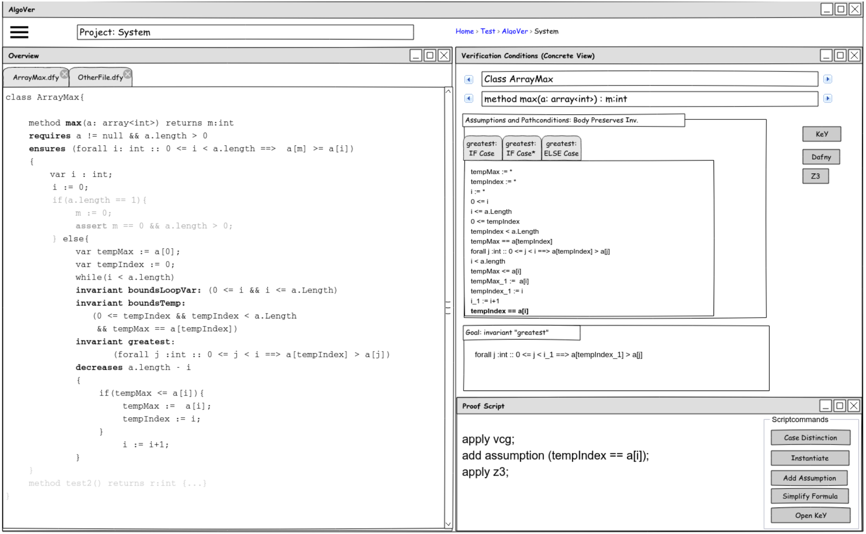
greatest: IF Case | greatest: ELSE Case

```
tempMax := *
tempIndex := *
i := *
0 <= i
i <= a.Length
0 <= tempIndex
tempIndex < a.Length
tempMax == a[tempIndex]
forall j :int :: 0 <= j < i ==> a[tempIndex] > a[j]
i < a.length
(tempMax <= a[i])
tempMax_1 :=  a[i];
tempIndex_1 := i;
i_1 := i+1;
```

Goal: invariant "greatest"

```
forall j :int :: 0 <= j < i_1 ==> a[tempIndex_1] > a[j]
```

Example



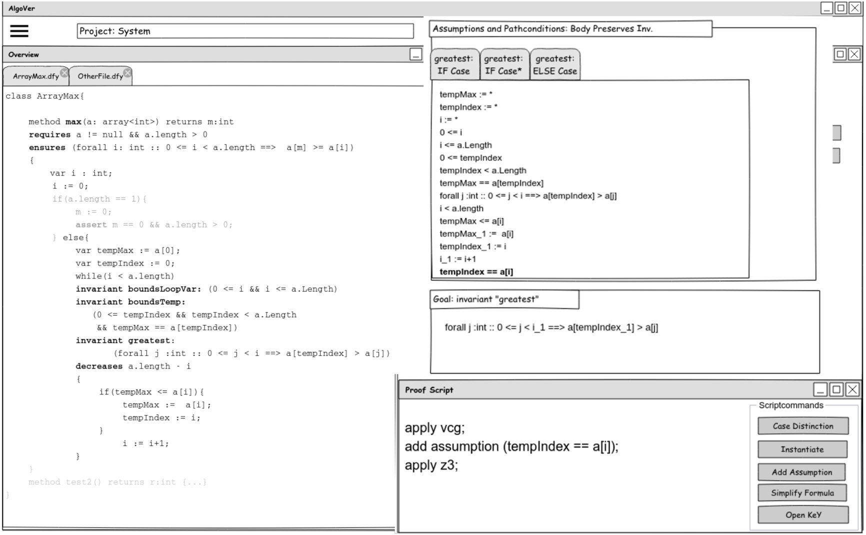
```
class ArrayMax{

    method max(a: array<int>) returns m:int
    requires a != null && a.length > 0
    ensures (forall i: int :: 0 <= i < a.length ==>  a[m] >= a[i])
    {
        var i : int;
         i := 0;
         if(a.length == 1){
             m := 0;
             assert m == 0 && a.length > 0;
         } else{
             var tempMax := a[0];
             var tempIndex := 0;
             while(i < a.length)
             invariant boundsLoopVar: (0 <= i && i <= a.Length)
             invariant boundsTemp:
                (0 <= tempIndex && tempIndex < a.Length
                 && tempMax == a[tempIndex])
             invariant greatest:
                    (forall j :int :: 0 <= j < i ==> a[tempIndex] > a[j])
             decreases a.length - i
             {
                 if(tempMax <= a[i]){
                     tempMax :=  a[i];
                     tempIndex := i;
                 }
                     i := i+1;
             }
    }
    method test2() returns r:int {...}
}
```

```
tempMax := *
tempIndex := *
i := *
0 <= i
i <= a.Length
0 <= tempIndex
tempIndex < a.Length
tempMax == a[tempIndex]
forall j :int :: 0 <= j < i ==> a[tempIndex] > a[j]
i < a.length
tempMax <= a[i]
tempMax_1 :=  a[i]
tempIndex_1 := i
i_1 := i+1
tempIndex == a[i]
```

```
forall j :int :: 0 <= j < i_1 ==> a[tempIndex_1] > a[j]
```

```
apply vcg;
add assumption (tempIndex == a[i]);
apply z3;
```

Example



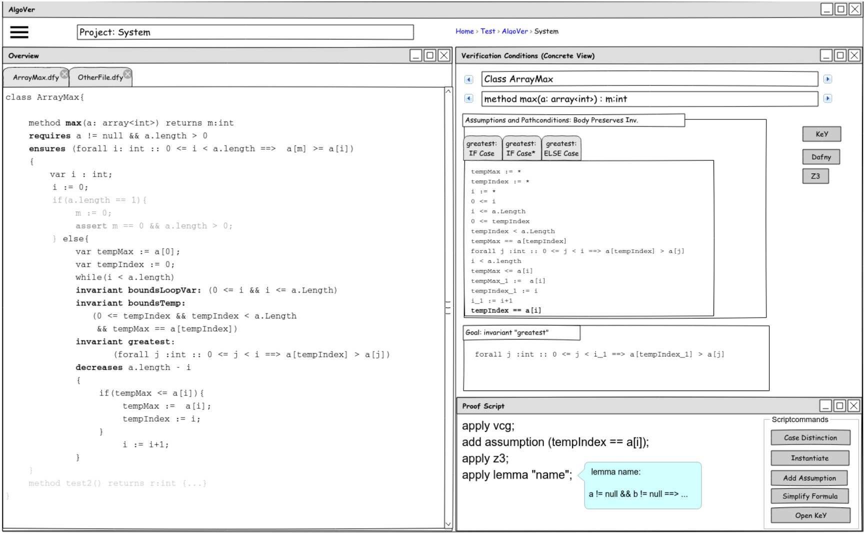
```
class ArrayMax{

    method max(a: array<int>) returns m:int
    requires a != null && a.length > 0
    ensures (forall i: int :: 0 <= i < a.length ==>  a[m] >= a[i])
    {
        var i : int;
        i := 0;
        if(a.length == 1){
            m := 0;
            assert m == 0 && a.length > 0;
        } else{
            var tempMax := a[0];
            var tempIndex := 0;
            while(i < a.length)
            invariant boundsLoopVar: (0 <= i && i <= a.Length)
            invariant boundsTemp:
                (0 <= tempIndex && tempIndex < a.Length
                 && tempMax == a[tempIndex])
            invariant greatest:
                    (forall j :int :: 0 <= j < i ==> a[tempIndex] > a[j])
            decreases a.length - i
            {
                if(tempMax <= a[i]){
                    tempMax :=  a[i];
                    tempIndex := i;
                }
                    i := i+1;
            }
    }
    method test2() returns r:int {...}
}
```

tempMax := *
tempIndex := *
i := *
0 <= i
i <= a.Length
0 <= tempIndex
tempIndex < a.Length
tempMax == a[tempIndex]
forall j :int :: 0 <= j < i ==> a[tempIndex] > a[j]
i < a.length
tempMax <= a[i]
tempMax_1 := a[i]
tempIndex_1 := i
i_1 := i+1
tempIndex == a[i]

forall j :int :: 0 <= j < i_1 ==> a[tempIndex_1] > a[j]

apply vcg;
add assumption (tempIndex == a[i]);
apply z3;

Example



```
class ArrayMax{

    method max(a: array<int>) returns m:int
    requires a != null && a.length > 0
    ensures (forall i: int :: 0 <= i < a.length ==>  a[m] >= a[i])
    {
        var i : int;
         i := 0;
         if(a.length == 1){
             m := 0;
             assert m == 0 && a.length > 0;
         } else{
             var tempMax := a[0];
             var tempIndex := 0;
             while(i < a.length)
             invariant boundsLoopVar: (0 <= i && i <= a.Length)
             invariant boundsTemp:
                (0 <= tempIndex && tempIndex < a.Length
                 && tempMax == a[tempIndex])
             invariant greatest:
                    (forall j :int :: 0 <= j < i ==> a[tempIndex] > a[j])
             decreases a.length - i
             {
                 if(tempMax <= a[i]){
                     tempMax :=  a[i];
                     tempIndex := i;
                 }
                     i := i+1;
             }
    }
    method test2() returns r:int {...}
}
```

```
tempMax := *
tempIndex := *
i := *
0 <= i
i <= a.Length
0 <= tempIndex
tempIndex < a.Length
tempMax == a[tempIndex]
forall j :int :: 0 <= j < i ==> a[tempIndex] > a[j]
i < a.length
tempMax <= a[i]
tempMax_1 :=  a[i]
tempIndex_1 := i
i_1 := i+1
tempIndex == a[i]
```

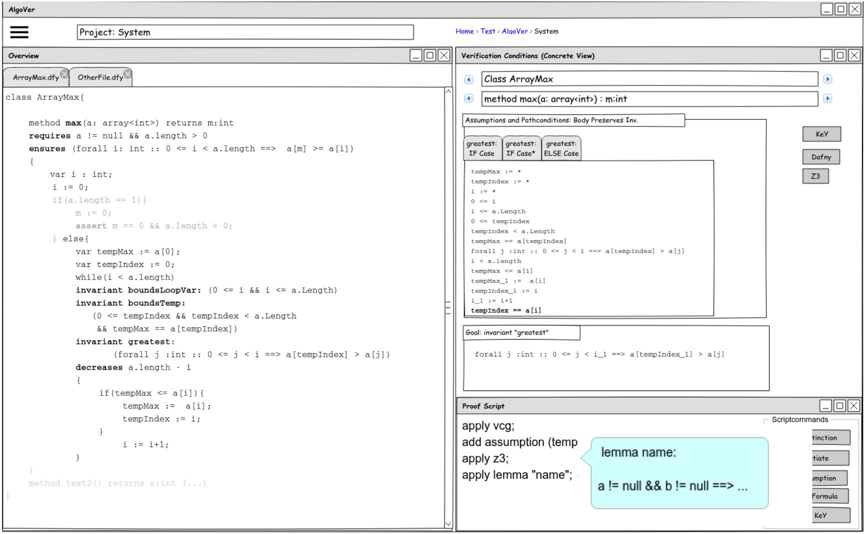
```
forall j :int :: 0 <= j < i_1 ==> a[tempIndex_1] > a[j]
```

```
apply vcg;
add assumption (tempIndex == a[i]);
apply z3;
apply lemma "name";
```

lemma name:

a != null && b != null ==> ...

Example



```
class ArrayMax{

    method max(a: array<int>) returns m:int
    requires a != null && a.length > 0
    ensures (forall i: int :: 0 <= i < a.length ==>  a[m] >= a[i])
    {
        var i : int;
        i := 0;
        if(a.length == 1){
            m := 0;
            assert m == 0 && a.length > 0;
        } else{
            var tempMax := a[0];
            var tempIndex := 0;
            while(i < a.length)
            invariant boundsLoopVar: (0 <= i && i <= a.Length)
            invariant boundsTemp:
               (0 <= tempIndex && tempIndex < a.Length
                && tempMax == a[tempIndex])
            invariant greatest:
                   (forall j :int :: 0 <= j < i ==> a[tempIndex] > a[j])
            decreases a.length - i
            {
                if(tempMax <= a[i]){
                    tempMax :=  a[i];
                    tempIndex := i;
                }
                    i := i+1;
            }
    }
    method test2() returns r:int {...}
}
```

```
tempMax := *
tempIndex := *
i := *
0 <= i
i <= a.Length
0 <= tempIndex
tempIndex < a.Length
tempMax == a[tempIndex]
forall j :int :: 0 <= j < i ==> a[tempIndex] > a[j]
i < a.length
tempMax <= a[i]
tempMax_1 :=  a[i]
tempIndex_1 := i
i_1 := i+1
tempIndex == a[i]
```

```
forall j :int :: 0 <= j < i_1 ==> a[tempIndex_1] > a[j]
```

```
apply vcg;
add assumption (temp
apply z3;
apply lemma "name";
```

lemma name:

a != null && b != null ==> ...

Summary

Bridging interaction gap between code and logic by

- fluent transition between levels
- different coupled proof views
- interaction on all levels
- seamless integration of sophisticated methods

Future Work

- implement concept
- evaluate concept with users

Discussion

In your experience:

What is the bottleneck for interaction?