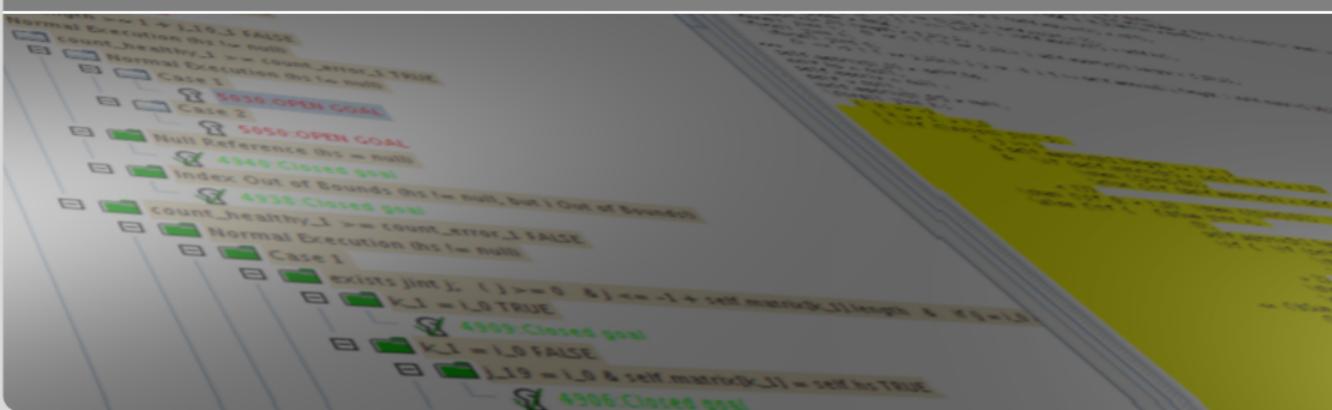


# Seamless Interactive Program Verification

Sarah Grebing, Jonas Klamroth, Mattias Ulbrich | 13. July 2019

Institute for Theoretical Informatics, KIT



The screenshot shows a graphical user interface for program verification. On the left, a tree diagram represents the state of various goals. Goals are categorized into three states: OPEN, CLOSED, and FALSE. A goal is considered FALSE if it has no children or all its children are FALSE. A goal is considered CLOSED if it has at least one child that is not FALSE. The tree structure is as follows:

- Root node: Normal Execution (hs != null) → L19 = FALSE (FALSE)
  - Child 1: count\_healthy\_1 == count\_error\_1 (OPEN)
    - Case 1: L19 = FALSE (CLOSED)
    - Case 2: L19 = TRUE (OPEN)
      - Child 1: Null Reference (hs == null) (CLOSED)
      - Child 2: 4940\_Closed goal (CLOSED)
      - Child 3: Index Out of Bounds (hs != null, but i >= length) (CLOSED)
        - Child 1: 4938\_Closed goal (CLOSED)
      - Child 4: count\_healthy\_1 == count\_error\_1 (FALSE)
        - Case 1: L19 = FALSE (CLOSED)
          - Child 1: exists jint: (j >= 0 & j <= -1 + self.matrikel.length) & L19 = L19 (OPEN)
            - Child 1: k1 = L0 (TRUE)
              - Child 1: 4909\_Closed goal (CLOSED)
            - Child 2: k1 = L0 (FALSE)
              - Child 1: L19 = L0 & self.matrikel[1] == self.hs (TRUE)
                - Child 1: 4906\_Closed goal (CLOSED)

# Motivation

Increased effectiveness of verification systems

# Motivation

More complex verification problems are tractable



Increased effectiveness of verification systems

# Motivation

User interaction now on more complex proof states



More complex verification problems are tractable



Increased effectiveness of verification systems

# Motivation

Support for proof representation and interaction for complex states is necessary



User interaction now on more complex proof states



More complex verification problems are tractable



Increased effectiveness of verification systems

User interaction is the limiting factor  
in program verification



Support for proof representation and interaction for complex states is necessary



User interaction now on more complex proof states



More complex verification problems are tractable



Increased effectiveness of verification systems

## Working Hypothesis:

User interaction is the limiting factor  
in program verification

## Working Hypothesis:

User interaction is the limiting factor  
in program verification

## Contributions

- user studies on interaction in program verification
- a new interaction concept
- prototypical implementation

## Study Topics

- proof step granularity
- time-consuming actions
- feedback mechanisms
- proof comprehension

## Study Topics

- proof step granularity
- time-consuming actions
- feedback mechanisms
- proof comprehension

## Methods

- focus group discussions: KeY and Isabelle/HOL
- semi-structured interviews with practical tasks: KeY
- qualitative content analysis and sequence models

# Result of the User Studies

*Understanding* the proof state is crucial  
and a central challenge for users

## Key Observation (Context-switch)

Difficult: Switching between elements in proof and program

## Key Observation (Context-switch)

Difficult: Switching between elements in proof and program  
⇒ Support access to and combination of information from different domains

## Key Observation (Context-switch)

Difficult: Switching between elements in proof and program  
⇒ Support access to and combination of information from different domains

## Key Observation (Orientation)

To gain orientation in the proof process typical patterns included:

- abstraction from and focusing on details
- (re)construct relation between proof state and annotated program

## Key Observation (Context-switch)

Difficult: Switching between elements in proof and program  
⇒ Support access to and combination of information from different domains

## Key Observation (Orientation)

To gain orientation in the proof process typical patterns included:

- abstraction from and focusing on details
- (re)construct relation between proof state and annotated program

⇒ Provide a global overview and a stepwise focusing on details

## Key Observation (Proof Process)

Every participant has an individual proof process.

Typical process patterns include

- using prover feedback for specification refinement
- completing specification before first verification attempt
- proof construction: single rule applications, strategies, and mixed

## Key Observation (Proof Process)

Every participant has an individual proof process.

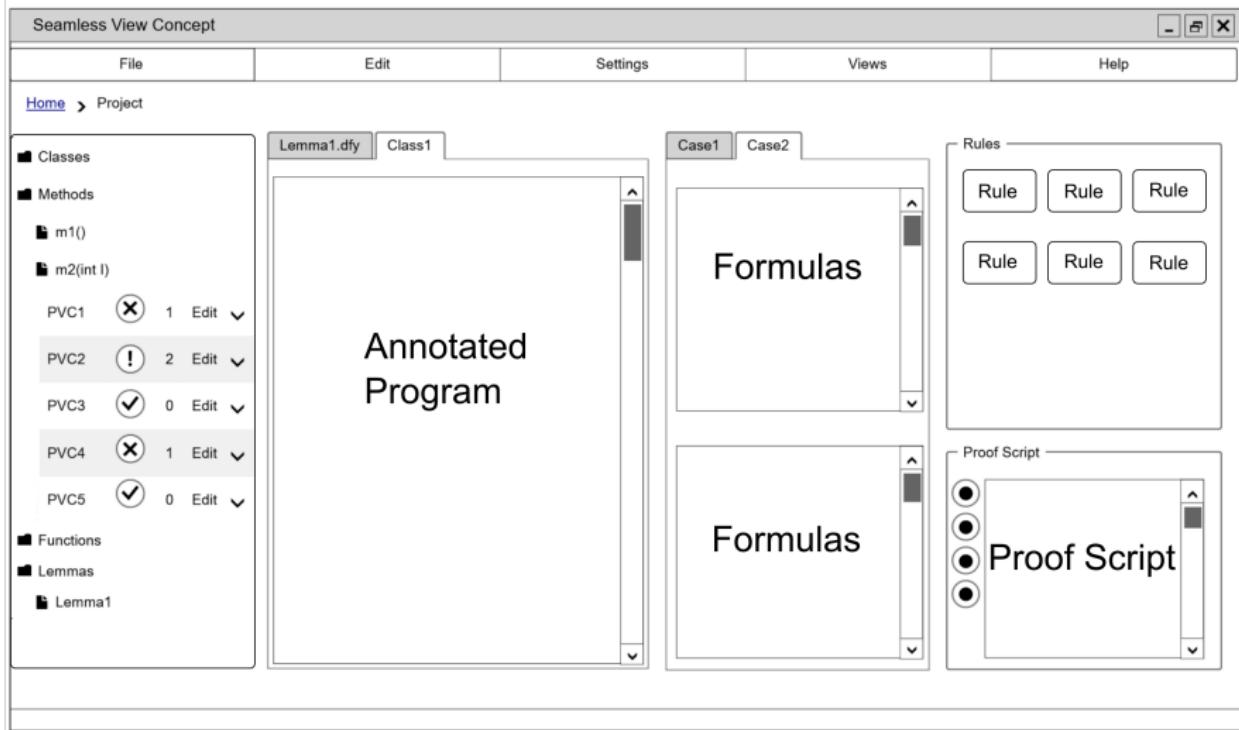
Typical process patterns include

- using prover feedback for specification refinement
- completing specification before first verification attempt
- proof construction: single rule applications, strategies, and mixed

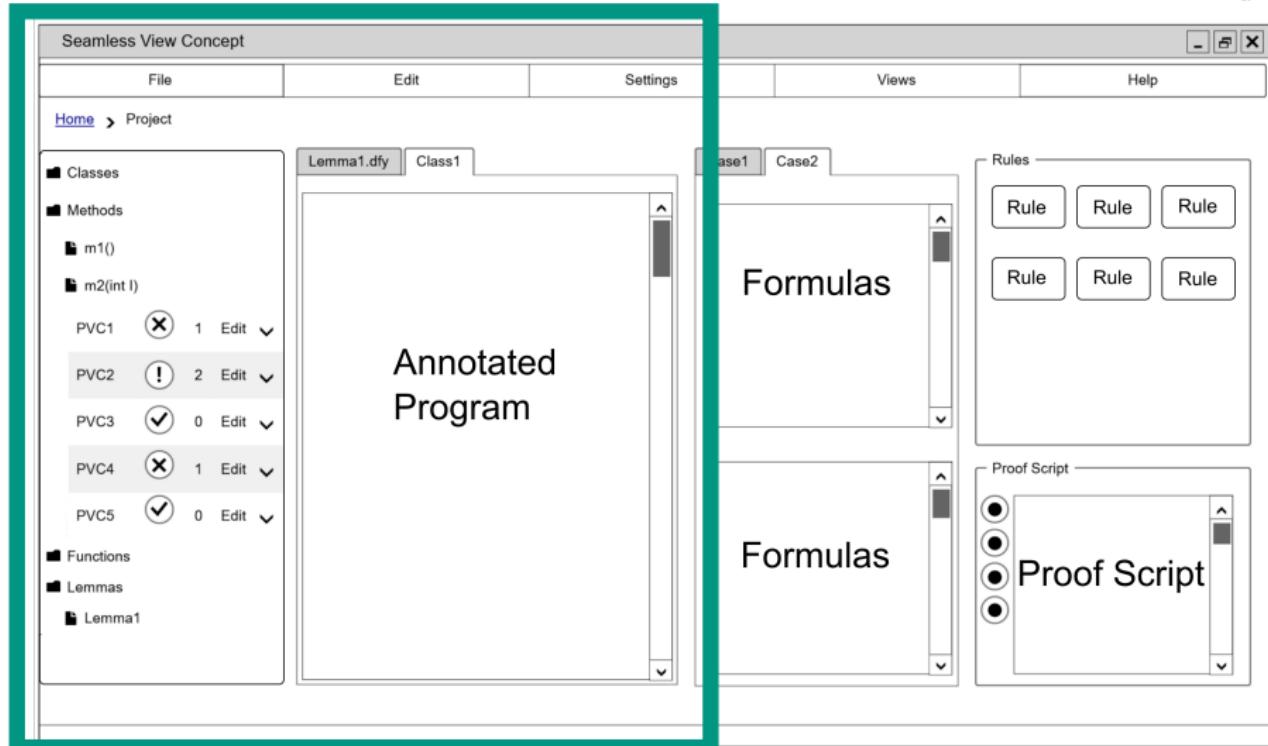
⇒ Allow many degrees of freedom and support different interaction preferences

# *Interaction Concept*

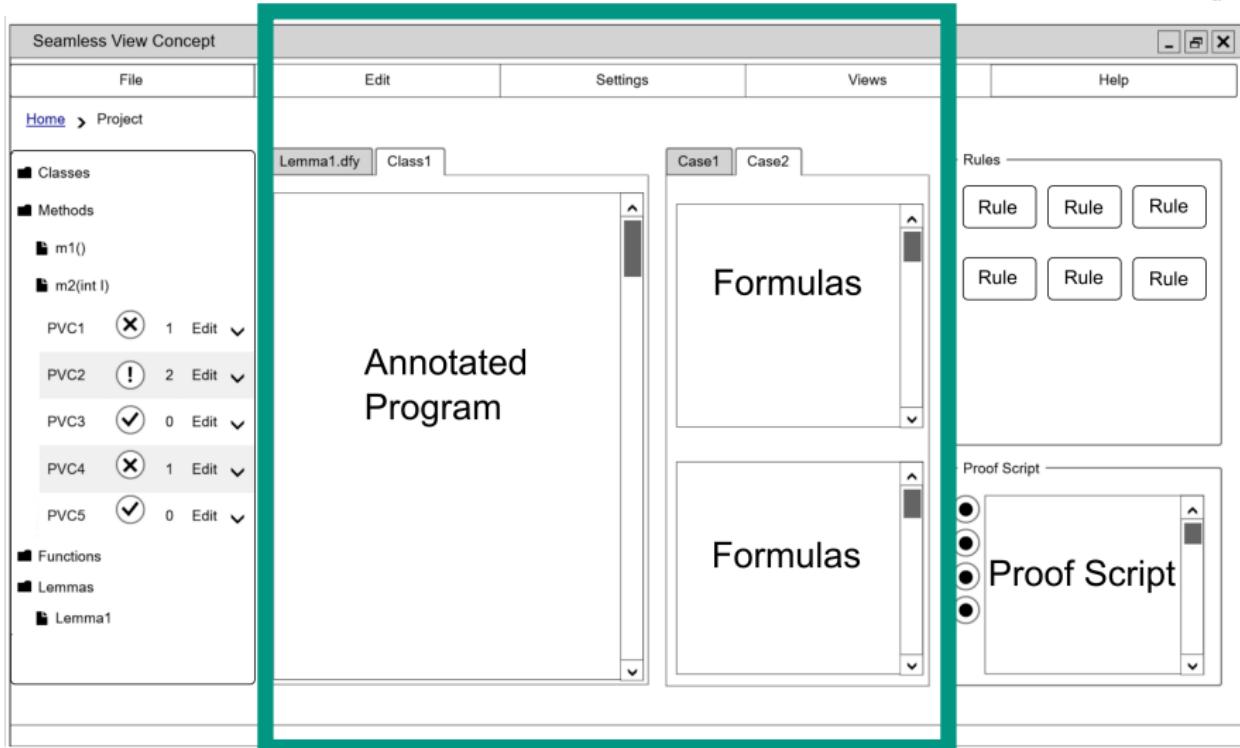
# “Seamless Interactive Program Verification”



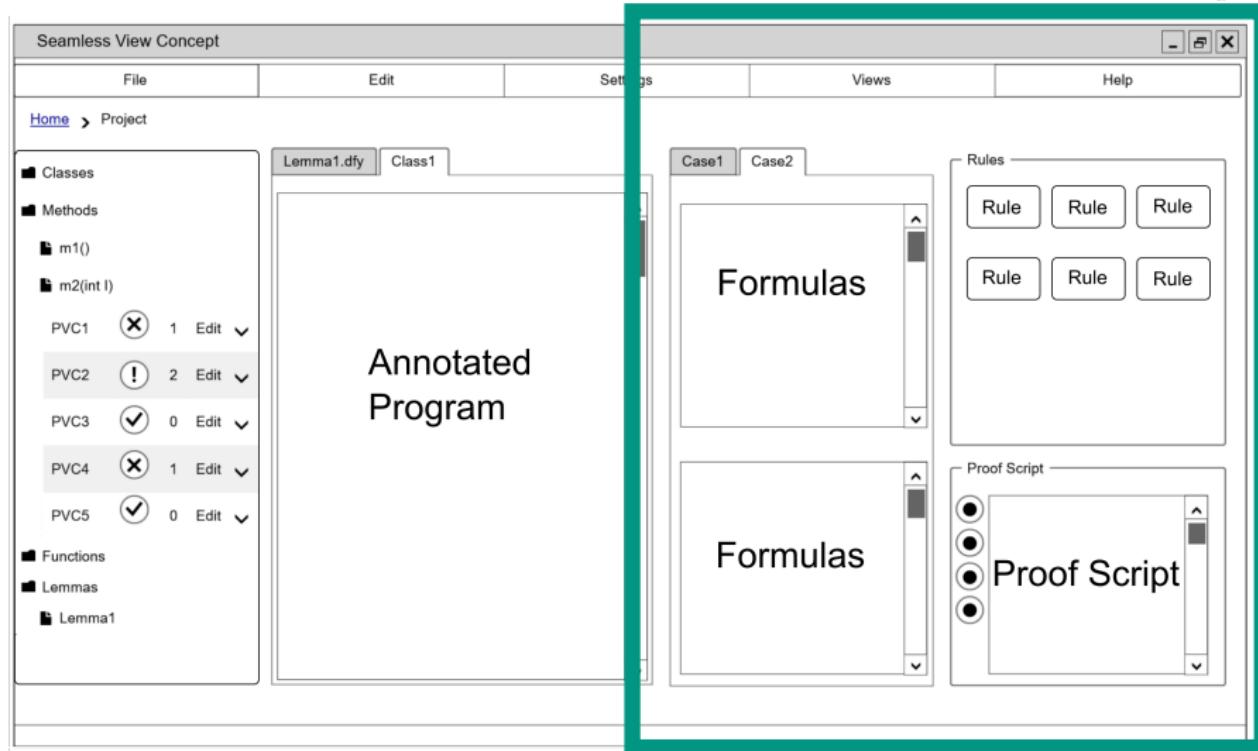
# “Seamless Interactive Program Verification”



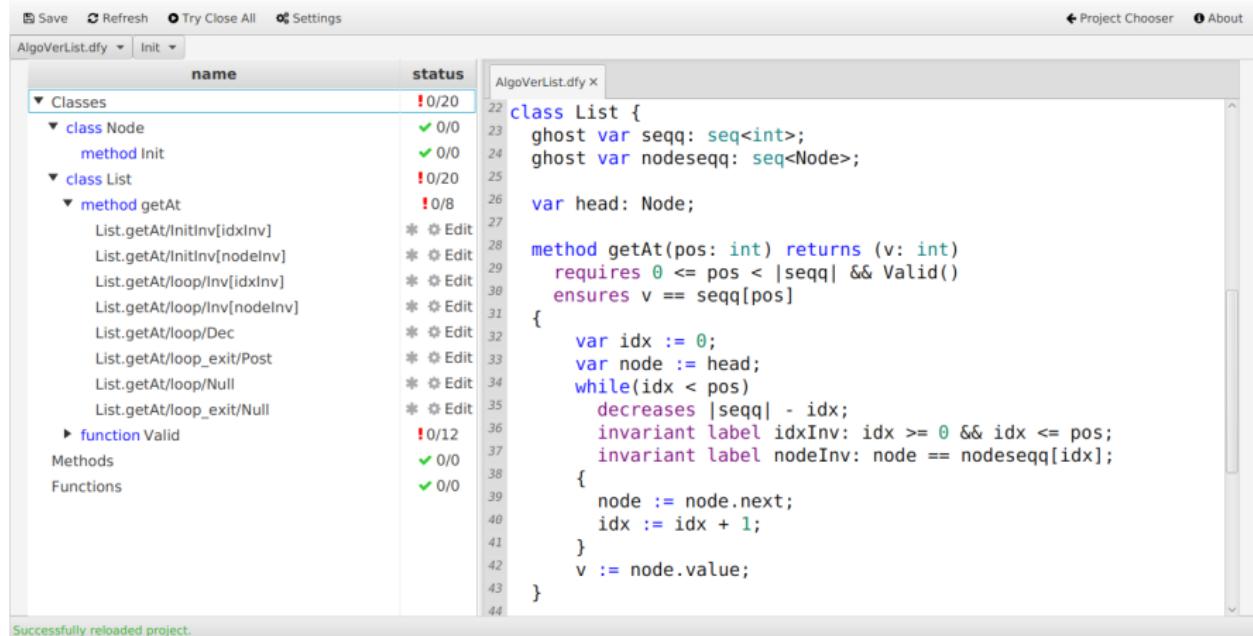
# “Seamless Interactive Program Verification”



# “Seamless Interactive Program Verification”



# Exemplary Walkthrough

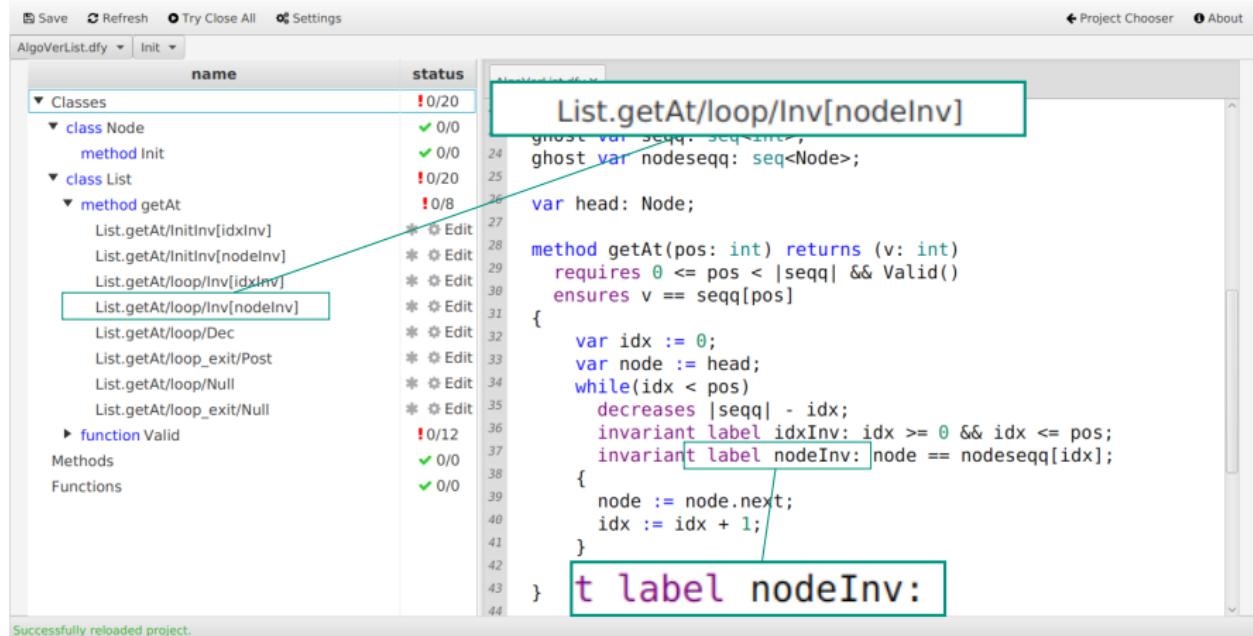


The screenshot shows a software interface for program verification. On the left, a project tree displays the file structure of 'AlgoVerList.dfy'. The tree includes sections for Classes, methods like 'Init' and 'getAt', and functions like 'Valid'. Most items show a green checkmark indicating successful verification. The right side of the interface is a code editor window displaying the CDFY code for the 'List' class. The code defines a class 'List' with ghost variables 'seqq' and 'nodeseqq' of type seq<int> and seq<Node> respectively. It contains a method 'getAt' which takes an integer 'pos' and returns an integer 'v'. The method requires  $0 \leq pos < |seqq|$  and ensures  $v = seqq[pos]$ . The implementation uses a loop to traverse the list until it reaches the specified index 'pos', returning the value at that position.

```
22 class List {
23     ghost var seqq: seq<int>;
24     ghost var nodeseqq: seq<Node>;
25
26     var head: Node;
27
28     method getAt(pos: int) returns (v: int)
29         requires 0 <= pos < |seqq| && Valid()
30         ensures v == seqq[pos]
31     {
32         var idx := 0;
33         var node := head;
34         while(idx < pos)
35             decreases |seqq| - idx;
36         invariant label idxInv: idx >= 0 && idx <= pos;
37         invariant label nodeInv: node == nodeseqq[idx];
38     }
39     node := node.next;
40     idx := idx + 1;
41 }
42 v := node.value;
43
44 }
```

Successfully reloaded project.

# Exemplary Walkthrough



The screenshot shows a software interface for program verification. On the left, a tree view displays the project structure:

- AlgoVerList.dfy
- Init
- Classes
  - class Node
    - method Init
  - class List
    - method getAt
      - List.getAt/InitInv[idxInv]
      - List.getAt/InitInv[nodeInv]
      - List.getAt/loop/idxInv
      - List.getAt/loop/inv[nodeInv]** (highlighted)
      - List.getAt/loop/Dec
      - List.getAt/loop\_exit/Post
      - List.getAt/loop/Null
      - List.getAt/loop\_exit/Null
    - function Valid
  - Methods
  - Functions

The status column indicates the number of errors (red exclamation marks) and warnings (yellow question marks). The highlighted method, `List.getAt/loop/inv[nodeInv]`, has 0 errors and 0 warnings.

The right pane contains the source code for the highlighted method:

```
ghost var seqq: seq<int>;
ghost var nodeseqq: seq<Node>;  
  
var head: Node;  
  
method getAt(pos: int) returns (v: int)
  requires 0 <= pos < |seqq| && Valid()
  ensures v == seqq[pos]
{  
  var idx := 0;
  var node := head;
  while(idx < pos)
    decreases |seqq| - idx;
    invariant label idxInv: idx >= 0 && idx <= pos;
    invariant label nodeInv: node == nodeseqq[idx];
  {  
    node := node.next;
    idx := idx + 1;
  }
}
```

A green box highlights the label `t label nodeInv:`. A blue line connects this label to the `label nodeInv:` in the code. Another blue line connects the `label idxInv:` in the code to the `label idxInv:` in the invariant.

# Exemplary Walkthrough

Save Refresh Try Close All Settings Project Chooser About

AlgoVerList.dfy Init

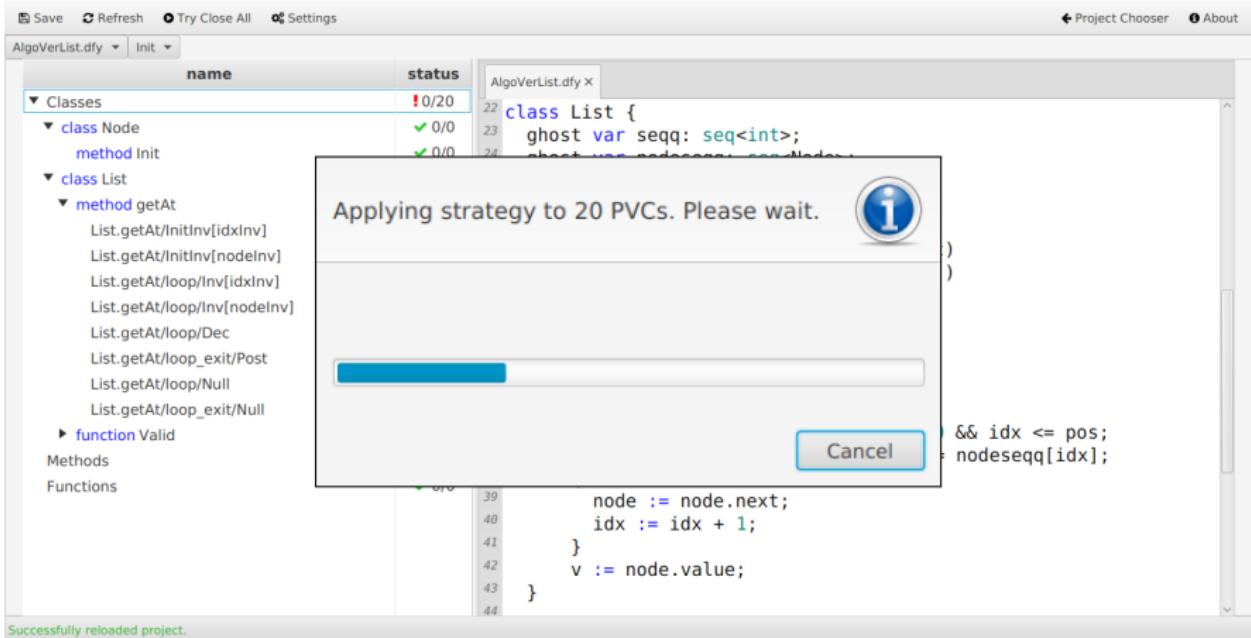
name	status
▼ Classes	! 0/20
▼ class Node	✓ 0/0
method Init	✓ 0/0
▼ class List	! 0/20
▼ method getAt	! 0/8
List.getAt/InitInv[idxInv]	* ⚡ Edit
List.getAt/InitInv[nodeInv]	* ⚡ Edit
List.getAt/loop/Inv[idxInv]	* ⚡ Edit
List.getAt/loop/Inv[nodeInv]	* ⚡ Edit
List.getAt/loop/Dec	* ⚡ Edit
List.getAt/loop_exit/Post	* ⚡ Edit
List.getAt/loop/Null	* ⚡ Edit
List.getAt/loop_exit/Null	* ⚡ Edit
► function Valid	! 0/12
Methods	✓ 0/0
Functions	✓ 0/0

AlgoVerList.dfy x

```
22 class List {
23   ghost var seqq: seq<int>;
24   ghost var nodeseqq: seq<Node>;
25
26   var head: Node;
27
28   method getAt(pos: int) returns (v: int)
29     requires 0 <= pos < |seqq| && Valid()
30     ensures v == seqq[pos]
31   {
32     var idx := 0;
33     var node := head;
34     while(idx < pos)
35       decreases |seqq| - idx;
36     invariant label idxInv: idx >= 0 && idx <= pos;
37     invariant label nodeInv: node == nodeseqq[idx];
38   {
39     node := node.next;
40     idx := idx + 1;
41   }
42   v := node.value;
43 }
```

Successfully reloaded project.

# Exemplary Walkthrough



The screenshot shows a software interface for program verification. On the left, there's a tree view of a project structure:

- AlgoVerList.dfy
- Init
- name
- status
- Classes
  - class Node (0/0)
  - method Init (0/0)
- class List (0/0)
- method getAt
  - List.getAt/InitInv[idxInv]
  - List.getAt/InitInv[nodeInv]
  - List.getAt/loop/Inv[idxInv]
  - List.getAt/loop/Inv[nodeInv]
  - List.getAt/loop/Dec
  - List.getAt/loop\_exit/Post
  - List.getAt/loop/Null
  - List.getAt/loop\_exit/Null
- function Valid

Below the tree view, there are buttons for Methods and Functions.

In the center, a modal dialog box is displayed:

Applying strategy to 20 PVCs. Please wait.

Cancel

The background code editor shows the following C-like pseudocode:

```
class List {
    ghost var seqq: seq<int>;
    ghost var nodeseqq: seq<Node>;
}

class Node {
    method int value;
    method void next();
}

method int getAt(List list, int idx) {
    if (idx <= pos) {
        node := list.nodeseqq[idx];
        idx := idx + 1;
    }
    v := node.value;
    return v;
}
```

# Exemplary Walkthrough

Save Refresh Try Close All Settings Project Chooser About

AlgoVerList.dfy Init

name	status
▼ Classes	! 0/20
▼ class Node	✓ 0/0
method Init	✓ 0/0
▼ class List	! 0/20
▼ method getAt	! 0/8
List.getAt/InitInv[idxInv]	
List.getAt/InitInv[nodeInv]	
List.getAt/loop/Inv[idxInv]	
List.getAt/loop/Inv[nodeInv]	
List.getAt/loop/Dec	
List.getAt/loop_exit/Post	
List.getAt/loop/Null	
List.getAt/loop_exit/Null	
► function Valid	! 0/12
Methods	✓ 0/0
Functions	✓ 0/0

AlgoVerList.dfy x

```
22 class List {
23   ghost var seqq: seq<int>;
24   ghost var nodeseqq: seq<Node>;
25
26   var head: Node;
27 }
```

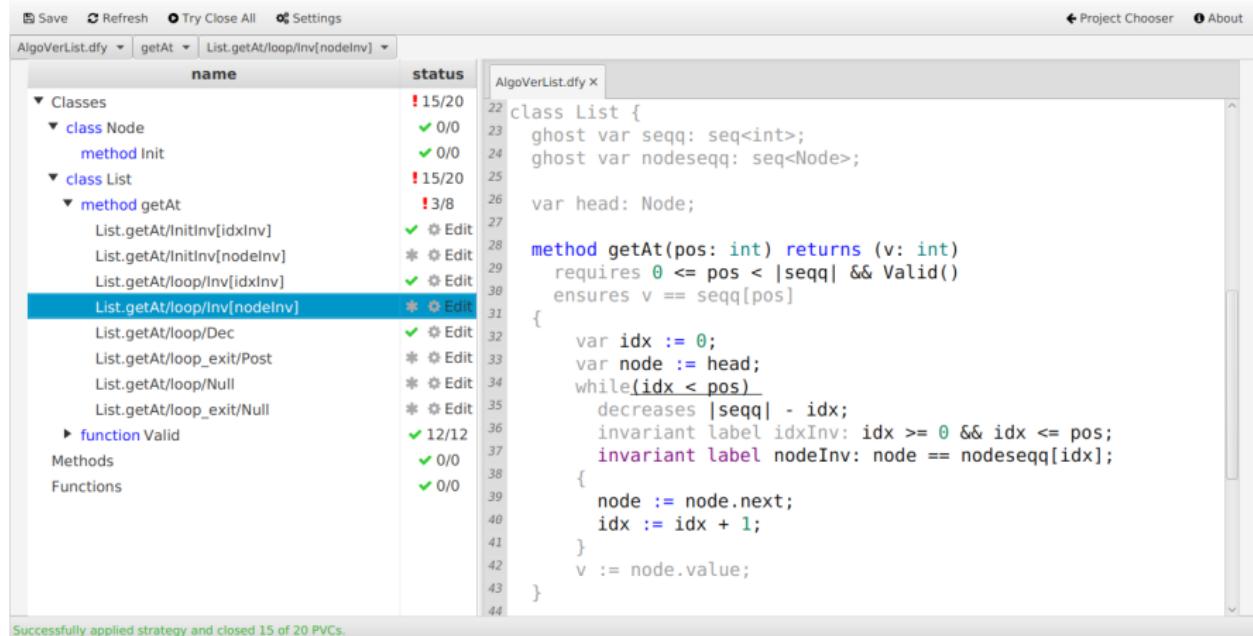
Successfully applied strategy and closed 15 of 20 PVCs.

OK

```
36   invariant label idxInv: idx >= 0 && idx <= pos;
37   invariant label nodeInv: node == nodeseqq[idx];
38   {
39     node := node.next;
40     idx := idx + 1;
41   }
42   v := node.value;
43 }
44 }
```

Successfully reloaded project.

# Exemplary Walkthrough



The screenshot shows the AlgoVerList tool interface. At the top, there are navigation buttons: Save, Refresh, Try Close All, Settings, Project Chooser, and About. Below the buttons, the file path is shown as AlgoVerList.dfy. The main area consists of two panes: a code editor on the right and a status table on the left.

**Status Table:**

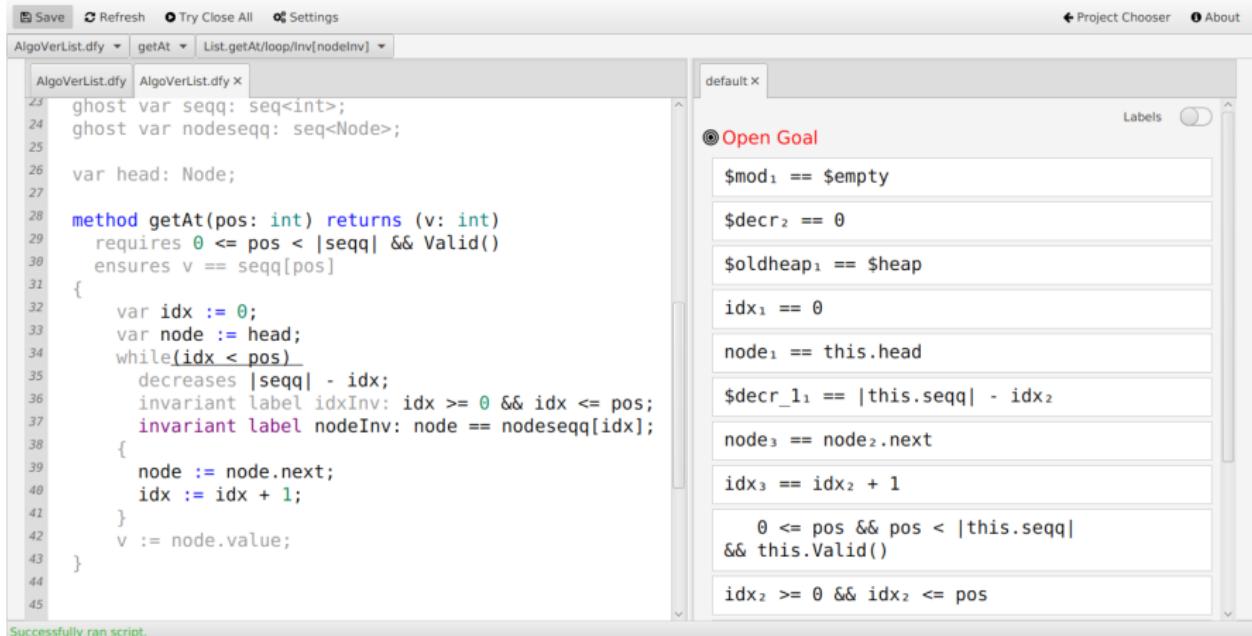
	name	status
▼ Classes		! 15/20
▼ class Node		✓ 0/0
method Init		✓ 0/0
▼ class List		! 15/20
▼ method getAt		! 3/8
List.getAt/InitInv[IdxInv]		✓ ⚡ Edit
List.getAt/InitInv[nodeInv]		* ⚡ Edit
List.getAt/loop/Inv[IdxInv]		✓ ⚡ Edit
List.getAt/loop/Inv[nodeInv]		* ⚡ Edit
List.getAt/loop/Dec		✓ ⚡ Edit
List.getAt/loop_exit/Post		* ⚡ Edit
List.getAt/loop/Null		* ⚡ Edit
List.getAt/loop_exit/Null		* ⚡ Edit
► function Valid		✓ 12/12
Methods		✓ 0/0
Functions		✓ 0/0

**Code Editor:**

```
22 class List {
23   ghost var seqq: seq<int>;
24   ghost var nodeseqq: seq<Node>;
25
26   var head: Node;
27
28   method getAt(pos: int) returns (v: int)
29     requires 0 <= pos < |seqq| && Valid()
30     ensures v == seqq[pos]
31   {
32     var idx := 0;
33     var node := head;
34     while(idx < pos)
35       decreases |seqq| - idx;
36     invariant label idxInv: idx >= 0 && idx <= pos;
37     invariant label nodeInv: node == nodeseqq[idx];
38   {
39     node := node.next;
40     idx := idx + 1;
41   }
42   v := node.value;
43 }
44 }
```

At the bottom of the status table, a message says "Successfully applied strategy and closed 15 of 20 PVCs."

# Exemplary Walkthrough

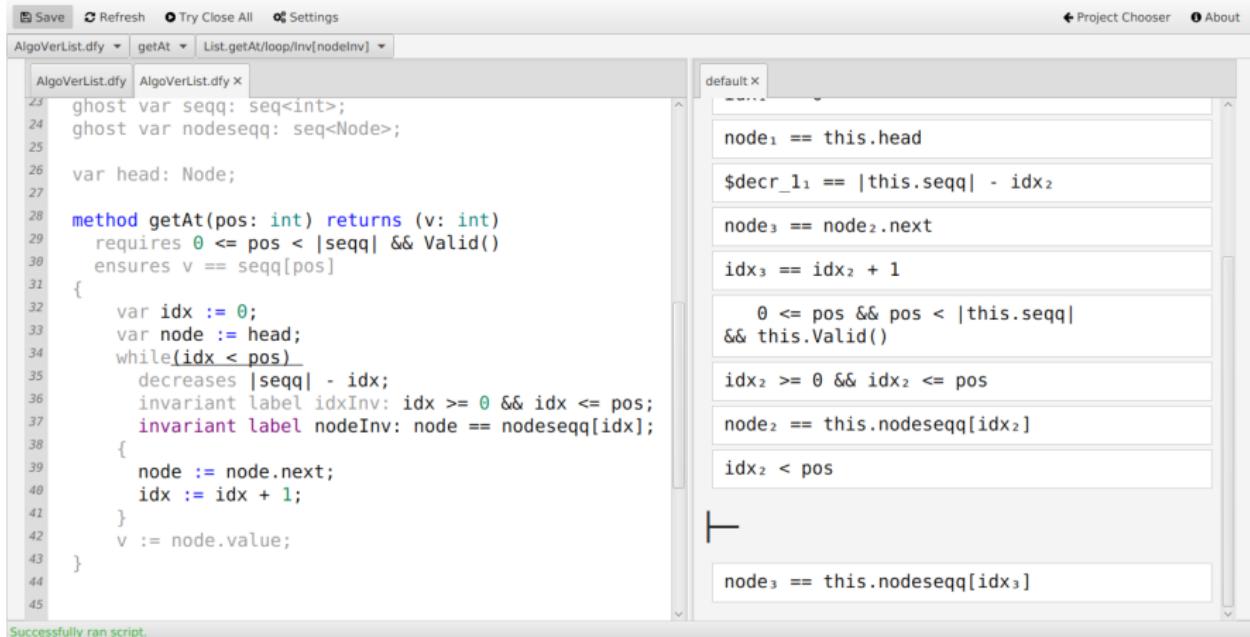


The screenshot shows a software interface for program verification. On the left, there is a code editor window titled "AlgoVerList.dfy" containing Dafny code. The code defines a class "AlgoVerList" with a method "getAt". The method takes an integer "pos" and returns an integer "v". It has three annotations: "requires" (0 <= pos < |seqq| && Valid()), "ensures" (v == seqq[pos]), and a loop invariant "invariant label idxInv: idx >= 0 && idx <= pos; invariant label nodeInv: node == nodeseqq[idx];". The code uses ghost variables "seqq" and "nodeseqq" to represent sequences of nodes. On the right, a panel titled "default X" lists several proof obligations (POs) generated by the verifier. These POs include assertions about the state variables and the loop's progress. The POs are:

- \$mod<sub>1</sub> == \$empty
- \$decr<sub>2</sub> == 0
- \$oldheap<sub>1</sub> == \$heap
- idx<sub>1</sub> == 0
- node<sub>1</sub> == this.head
- \$decr\_1 == |this.seqq| - idx<sub>2</sub>
- node<sub>3</sub> == node<sub>2</sub>.next
- idx<sub>3</sub> == idx<sub>2</sub> + 1
- 0 <= pos && pos < |this.seqq| && this.Valid()
- idx<sub>2</sub> >= 0 && idx<sub>2</sub> <= pos

At the bottom left, a message says "Successfully ran script."

# Exemplary Walkthrough



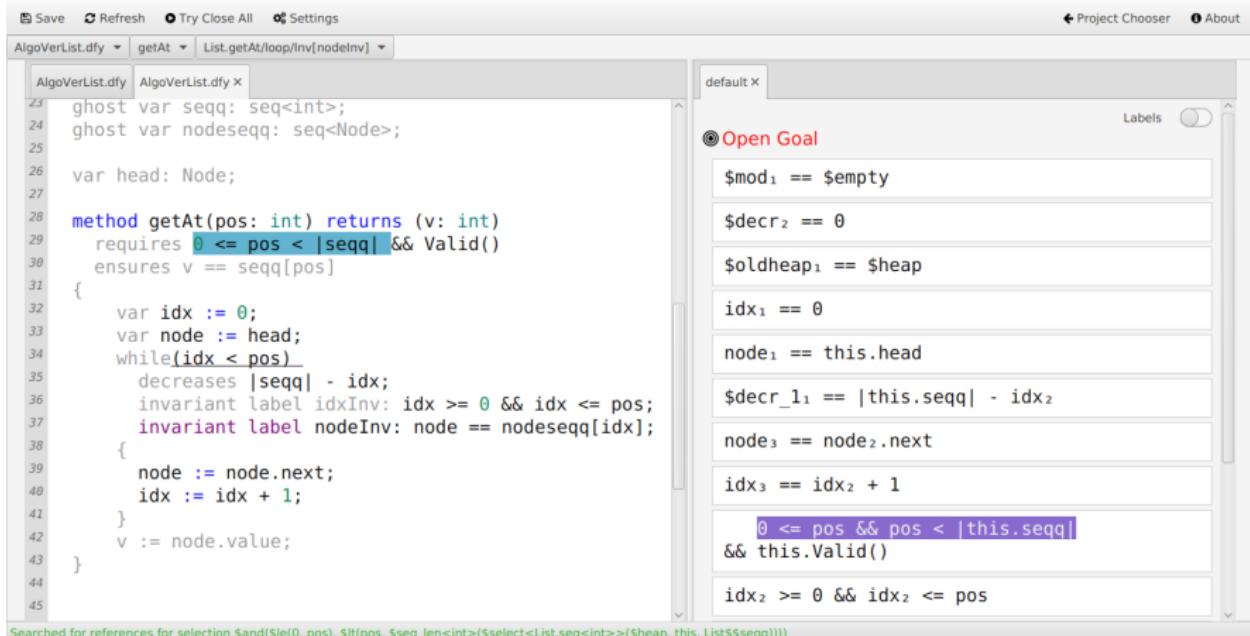
The screenshot shows a software interface for program verification. On the left, there is a code editor window titled "AlgoVerList.dfy" containing Dafny code. The code defines a ghost variable `seqq` of type `seq<int>` and a ghost variable `nodedseqq` of type `seq<Node>`. It declares a variable `head` of type `Node`. A method `getAt` is defined to return the value at index `pos`. The method requires `0 <= pos < |seqq|` and ensures `v == seqq[pos]`. The implementation uses a while loop to traverse the list until the index `idx` reaches `pos`. Inside the loop, it decreases the invariant `|seqq| - idx`. The invariant `idxInv` is `idx >= 0 && idx <= pos`, and the invariant `nodeInv` is `node == nodedseqq[idx]`. After the loop, the value `v` is assigned the value of the node at index `idx`.

On the right, a list of proof obligations is shown. These obligations include:

- `node1 == this.head`
- `$decr_l1 == |this.seqq| - idx2`
- `node3 == node2.next`
- `idx3 == idx2 + 1`
- `0 <= pos && pos < |this.seqq| && this.Valid()`
- `idx2 >= 0 && idx2 <= pos`
- `node2 == this.nodedseqq[idx2]`
- `idx2 < pos`
- `node3 == this.nodedseqq[idx3]`

At the bottom left, a message says "Successfully ran script."

# Exemplary Walkthrough



The screenshot shows the Frama-C interface with a Dafny program named `AlgoVerList.dfy`. The program defines a linked list structure and implements a `getAt` method.

```
ghost var seqq: seq<int>;
ghost var nodeseqq: seq<Node>;
var head: Node;

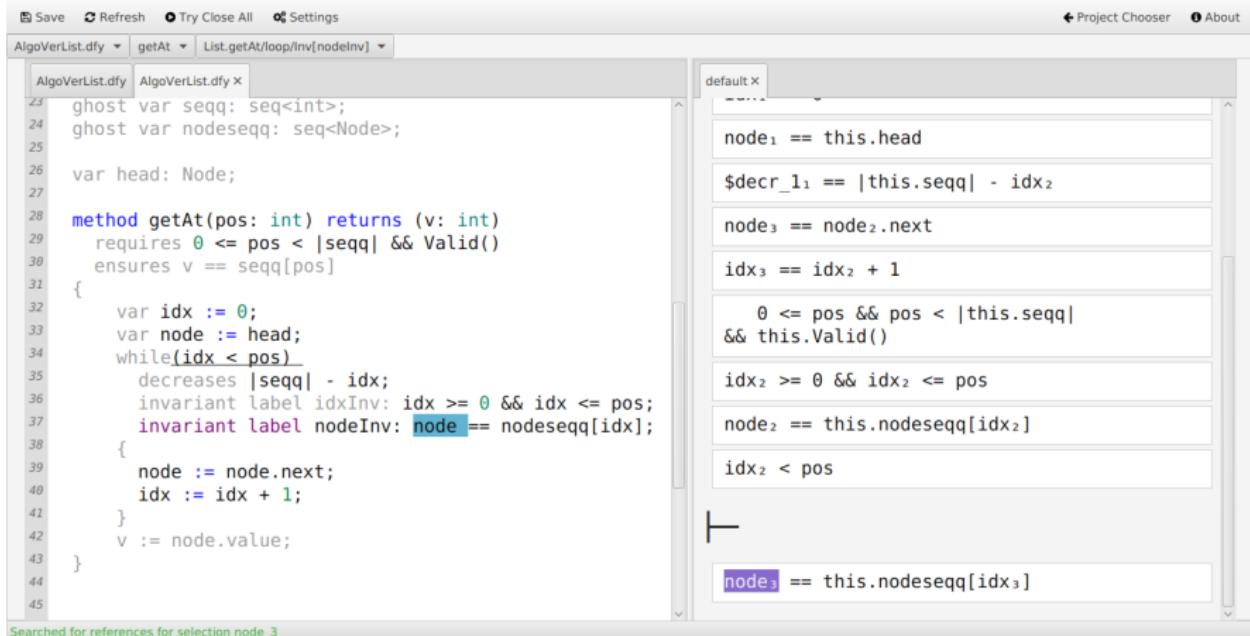
method getAt(pos: int) returns (v: int)
  requires 0 <= pos < |seqq| && Valid()
  ensures v == seqq[pos]
{
  var idx := 0;
  var node := head;
  while(idx < pos)
    decreases |seqq| - idx;
    invariant label idxInv: idx >= 0 && idx <= pos;
    invariant label nodeInv: node == nodeseqq[idx];
  {
    node := node.next;
    idx := idx + 1;
  }
  v := node.value;
}
```

The right panel displays the generated verification conditions for the `getAt` method:

- default X**
- Open Goal**
- \$mod<sub>1</sub> == \$empty
- \$decr<sub>2</sub> == 0
- \$oldheap<sub>1</sub> == \$heap
- idx<sub>1</sub> == 0
- node<sub>1</sub> == this.head
- \$decr\_1<sub>1</sub> == |this.seqq| - idx<sub>2</sub>
- node<sub>3</sub> == node<sub>2</sub>.next
- idx<sub>3</sub> == idx<sub>2</sub> + 1
- 0 <= pos && pos < |this.seqq| && this.Valid()
- idx<sub>2</sub> >= 0 && idx<sub>2</sub> <= pos

A status bar at the bottom indicates: "Searched for references for selection \$and(\$le(0, pos), \$lt(pos, \$seq\_len<int>(\$select<List,seq<int>">(\$heap, this, List\$seq))))".

# Exemplary Walkthrough



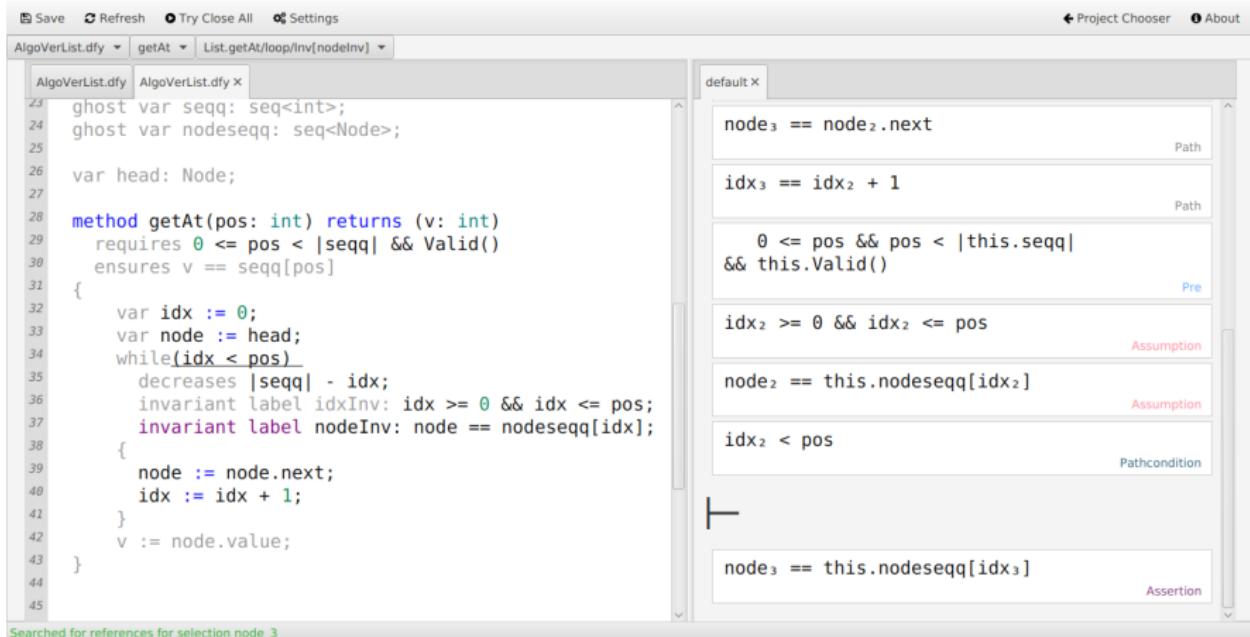
The screenshot shows a software interface for program verification. On the left, there is a code editor window titled "AlgoVerList.dfy" containing Dafny code. The code defines a class "AlgoVerList" with a method "getAt". The method takes an integer "pos" and returns an integer "v". It has preconditions that "0 <= pos < |seqq|" and "Valid()", and a postcondition that "v == seqq[pos]". The implementation uses a while loop to iterate from index 0 to "pos", updating a variable "node" to the next node in the sequence and an index "idx" to the next position. The loop invariant is that "idx >= 0 && idx <= pos" and "node == nodeseqq[idx]". After the loop, the value at index "pos" is assigned to "v".

On the right, a sidebar displays a list of generated proof obligations. These obligations include:

- default X
- node<sub>1</sub> == this.head
- \$decr\_l1 == |this.seqq| - idx<sub>2</sub>
- node<sub>3</sub> == node<sub>2</sub>.next
- idx<sub>3</sub> == idx<sub>2</sub> + 1
- 0 <= pos && pos < |this.seqq| && this.Valid()
- idx<sub>2</sub> >= 0 && idx<sub>2</sub> <= pos
- node<sub>2</sub> == this.nodeseqq[idx<sub>2</sub>]
- idx<sub>2</sub> < pos
- node<sub>3</sub> == this.nodeseqq[idx<sub>3</sub>]

A status bar at the bottom indicates: "Searched for references for selection node\_3".

# Exemplary Walkthrough



The screenshot shows a software interface for program verification. On the left, there is a code editor window titled "AlgoVerList.dfy" containing Dafny code. The code defines a ghost variable `seqq` of type `seq<int>`, another ghost variable `nodedseqq` of type `seq<Node>`, and a mutable variable `head` of type `Node`. It includes a method `getAt` that takes an integer `pos` and returns an integer `v`. The method's preconditions are `0 <= pos < |seqq| && Valid()` and its postcondition is `v == seqq[pos]`. The implementation uses a while loop to iterate from index 0 to `pos`, updating `node` to `node.next` and `idx` to `idx + 1`. Finally, it returns `node.value`. Invariant labels `idxInv` and `nodeInv` are used to ensure the loop's progress and the node's value respectively.

On the right, a sidebar displays several verification conditions (VCs) for the `getAt` method:

- `node3 == node2.next` (Path)
- `idx3 == idx2 + 1` (Path)
- `0 <= pos && pos < |this.seqq| && this.Valid()` (Pre)
- `idx2 >= 0 && idx2 <= pos` (Assumption)
- `node2 == this.nodedseqq[idx2]` (Assumption)
- `idx2 < pos` (Pathcondition)
- `node3 == this.nodedseqq[idx3]` (Assertion)

A search bar at the bottom of the sidebar indicates: "Searched for references for selection node\_3".

# Exemplary Walkthrough

Save Refresh Try Close All Settings

AlgoVerList.dfy getAt List.getValueAt/loop/Inv[nodeInv] ▾

default X Labels

Open Goal

```
$mod1 == $empty
$decr2 == 0
$oldheap1 == $heap
idx1 == 0
node1 == this.head
$decr_1 == |this.seqq| - idx2
node3 == node2.next
idx3 == idx2 + 1
  0 <= pos && pos < |this.seqq|
  && this.Valid()
idx2 >= 0 && idx2 <= pos
node2 == this.nodeseqq[idx2]
idx2 < pos
```

Successfully ran script.

1 0 1

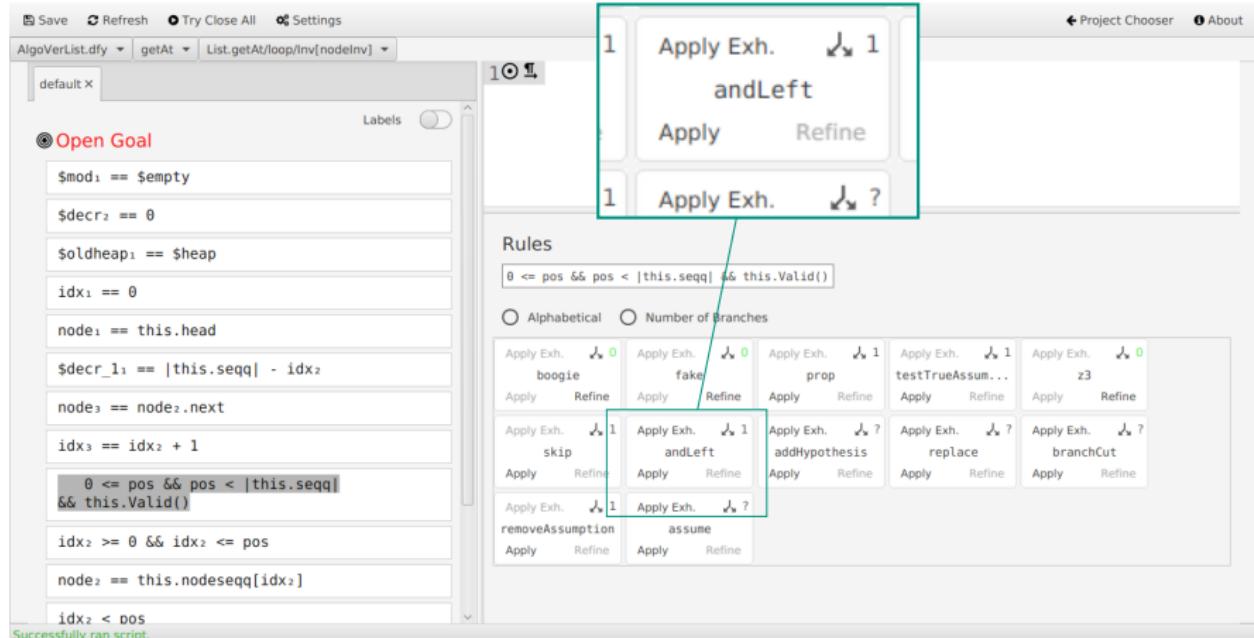
### Rules

0 <= pos && pos < |this.seqq| && this.Valid()

Alphabetical Number of Branches

Apply Exh. boogie 0	Apply Exh. fake 0	Apply Exh. prop 1	Apply Exh. testTrueAssum... 1	Apply Exh. z3 0
Apply Refine	Apply Refine	Apply Refine	Apply Refine	Apply Refine
Apply Exh. skip 1	Apply Exh. andLeft 1	Apply Exh. addHypothesis ?	Apply Exh. replace ?	Apply Exh. branchCut ?
Apply Refine	Apply Refine	Apply Refine	Apply Refine	Apply Refine
Apply Exh. removeAssumption 1	Apply Exh. assume ?			
Apply Refine	Apply Refine			

# Exemplary Walkthrough



The screenshot shows the AlgoVerList tool interface. On the left, the "Open Goal" section displays the following program state:

```
$mod1 == $empty
$decr2 == 0
$oldheap1 == $heap
idx1 == 0
node1 == this.head
$decr_1 == |this.seqq| - idx2
node3 == node2.next
idx3 == idx2 + 1
  0 <= pos && pos < |this.seqq|
  && this.Valid()
idx2 >= 0 && idx2 <= pos
node2 == this.nodeseqq[idx2]
idx2 < pos
Successfully ran script.
```

In the center, a modal dialog titled "Rules" lists various exhaustiveness rules:

Rule	Count	Details
Apply Exh. boogie	0	Apply Refine
Apply Exh. fake	0	Apply Refine
Apply Exh. prop	1	Apply Refine
Apply Exh. testTrueAssum...	1	Apply Refine
Apply Exh. z3	0	Apply Refine
Apply Exh. skip	1	Apply Refine
Apply Exh. andLeft	1	Apply Refine
Apply Exh. addHypothesis	?	Apply Refine
Apply Exh. replace	?	Apply Refine
Apply Exh. branchCut	?	Apply Refine
Apply Exh. removeAssumption	1	Apply Refine
Apply Exh. assume	?	Apply Refine

The "andLeft" rule is highlighted with a green box and has a question mark icon next to its count, indicating it is the selected rule.

# Exemplary Walkthrough

Save Refresh Try Close All Settings

AlgoVerList.dfy getAt List.getValueAt/loop/Inv[nodeInv] ▾

default X

```
idx1 == 0
node1 == this.head
$decr_11 == |this.seqq| - idx2
node3 == node2.next
idx3 == idx2 + 1
0 <= pos && pos < |this.seqq|
idx2 >= 0 && idx2 <= pos
node2 == this.nodeseqq[idx2]
idx2 < pos
this.Valid()
node3 == this.nodeseqq[idx3]
```

1 1.

Rules

```
0 <= pos && pos < |this.seqq| && this.Valid()
```

Alphabetical  Number of Branches

Apply Exh. boogie 0	Apply Exh. fake 0	Apply Exh. prop 1	Apply Exh. testTrueAssum... 1
Refine	Refine	Refine	Refine
Apply z3	Apply skip	Apply andLeft	Apply addHypothesis
Refine	Refine	Refine	Refine
Apply Exh. replace ?	Apply Exh. branchCut ?	Apply Exh. removeAssumption 1	Apply Exh. assume ?
Refine	Refine	Refine	Refine

Successfully ran script.

# Exemplary Walkthrough

Save Refresh Try Close All Settings

AlgoVerList.dfy getAt List.getValueAt/loop/Inv[nodeInv] default X Labels

Open Goal

```
$mod1 == $empty
$decr2 == 0
$oldheap1 == $heap
idx1 == 0
node1 == this.head
$decr_11 == |this.seqq| - idx2
node3 == node2.next
idx3 == idx2 + 1
0 <= pos && pos < |this.seqq|
idx2 >= 0 && idx2 <= pos
node2 == this.nodeseqq[idx2]
idx2 < pos
this.Valid!
```

Successfully ran script.

10 andLeft on='... ((?match: \_&&\_ &&\_) ... |-';
20 1

Rules

0 <= pos && pos < |this.seqq|

Alphabetical  Number of Branches

Apply Exh. boogie 0	Apply Exh. fake 0	Apply Exh. prop 1	Apply Exh. testTrueAssum... 1	Apply Exh. z3 0
Apply Refine	Apply Refine	Apply Refine	Apply Refine	Apply Refine
Apply Exh. skip 1	Apply Exh. andLeft 1	Apply Exh. addHypothesis ?	Apply Exh. replace ?	Apply Exh. branchCut ?
Apply Refine	Apply Refine	Apply Refine	Apply Refine	Apply Refine
Apply Exh. removeAssumption 1	Apply Exh. assume ?			
Apply Refine	Apply Refine			

# Exemplary Walkthrough

Save Refresh Try Close All Settings

AlgoVerList.dfy getAt List.getAt/loop/Inv[nodeInv] ▾

```
default X
nodez == this.nodeseqq[idxz]

idxz < pos

    |this.seqq| == |this.nodeseqq|
    && |this.seqq| >= 1
    && this.head != null
    && (forall n:int :: (
        n >= 0
        && n < |this.nodeseqq|
        ==> this.nodeseqq[n] != null))
    && (forall i:int :: (
        i >= 0 && i < |this.seqq|
        ==> this.seqq[i]
        ==> this.nodeseqq[i].value))
    && (forall k:int :: (
        k >= 0 && k < |this.nodeseqq| - 1
        ==> this.nodeseqq[k].next != null))
    && this.nodeseqq[0] == this.head
    && this.nodeseqq[|this.nodeseqq| - 1].next == null

(forall j:int :: j >= 0 && j < |this.nodeseqq| - 1
==> this.nodeseqq[j].next
==> this.nodeseqq[(j + 1)])
```

```
nodez == this.nodeseqq[idxz]
```

1○ andLeft on='...((?match: \_ && \_ && \_)) ...'
2○ expand on='...((?match: this.Valid())) ...'
3○ andLeft on='...((?match: \_ && \_ && \_ && \_ &&
4○ 1

< >

### Rules

```
(forall j:int :: j >= 0 && j < |this.nodeseqq| - 1
==> this.nodeseqq[j].next
==> this.nodeseqq[(j + 1)])
```

Alphabetical  Number of Branches

Apply Exh. boogie	Apply Exh. fake	Apply Exh. prop	Apply Exh. testTrueAssum...
Apply Refine	Apply Refine	Apply Refine	Apply Refine
Apply Exh. z3	Apply Exh. skip	Apply Exh. addHypothesis	Apply Exh. replace
Apply Refine	Apply Refine	Apply Refine	Apply Refine
Apply Exh. inst	Apply Exh. branchCut	Apply Exh. removeAssumption	Apply Exh. assume
Apply Refine	Apply Refine	Apply Refine	Apply Refine

Successfully ran script.

# Exemplary Walkthrough

Save Refresh Try Close All Settings

AlgoVerList.dfy | getAt | List.getAt/loop/inv[nodeInv] |

default X  
nodez == this.nodeseqq[10x2]  
idxz < pos

```
|this.seqq| == |this.nodeseqq|
  && |this.seqq| >= 1
  && this.head != null
  && (forall n:int :: ( n >= 0
    && n < |this.nodeseqq|
    ==> this.nodeseqq[n] != null))
  && (forall i:int :: ( i >= 0 && i < |this.seqq|
    ==> this.seqq[i]
    ==> this.nodeseqq[i].value))
  && (forall k:int :: ( k >= 0 && k < |this.nodeseqq| - 1
    ==> this.nodeseqq[k].next != null))
  && this.nodeseqq[0] == this.head
  && this.nodeseqq[(|this.nodeseqq| - 1)].next == null

(forall j:int :: j >= 0 && j < |this.nodeseqq| - 1
==> this.nodeseqq[j].next
==> this.nodeseqq[(j + 1)])
```

nodez == this.nodeseqq[idxz]

andLeft on='...((?match: \_ && \_ && \_)) ...'  
expand on='...((?match: this.Valid())) ...'  
andLeft on='...((?match: \_ && \_ && \_ && \_ && \_)) ...'

Rules

```
(forall j:int :: j >= 0 && j < |this.nodeseqq| - 1
==> this.nodeseqq[j].next
==> this.nodeseqq[(j + 1)])
```

Alphabetical Number of Branches

Apply Exh. boogie 0	Apply Exh. fake 0	Apply Exh. prop 1	Apply Exh. testTrueAssum... 1
Apply Refine	Apply Refine	Apply Refine	Apply Refine
Apply Exh. z3 0	Apply Exh. skip 1	Apply Exh. addHypothesis ?	Apply Exh. replace ?
Apply Refine	Apply Refine	Apply Refine	Apply Refine
Apply Exh. inst ?	Apply Exh. branchCut ?	Apply Exh. removeAssumption 1	Apply Exh. assume ?
Apply Refine	Apply Refine	Apply Refine	Apply Refine

Successfully ran script.

# Exemplary Walkthrough

Save Refresh Try Close All Settings Project Chooser About

AlgoVerList.dfy getAt List.getAt/loop/inv[nodeInv] ▾

default X nodez == this.nodeseqq[10x2]  
idxz < pos

|this.seqq| == |this.nodeseqq|  
&& |this.seqq| >= 1  
&& this.head != null  
&& (forall n:int :: (  
    && (forall i:int :: ( i >  
        ==>  
        ==>  
        && (forall k:int :: ( k >= 0  
            ==> this.n  
            && this.nodeseqq[0] == this.head  
            && this.nodeseqq[(|this.nodeseqq| - 1  
  
(forall j:int :: j >= 0 && j < |this.nodeseqq| - 1  
    ==> this.nodeseqq[j].next  
        ==> this.nodeseqq[|(j + 1)|])  
  
nodez == this.nodeseqq[idxz]

1O andLeft on='...((?match: \_ && \_ && \_)) ...'  
2O expand on='...((?match: this.Valid())) ...'  
3O andLeft on='...((?match: \_ && \_ && \_ && \_ && \_)) ...'  
4O 

Please insert the requested parameters to apply the rule.  
with idx\_2  
on (forall j:int :: j >= 0 && j < |  
Apply Cancel

of Branches

Apply Exh. z3	Apply Exh. skip	Apply Exh. addHypothesis	Apply Exh. testTrueAssum...
Apply Refine	Apply Refine	Apply Refine	Apply Refine
Apply Exh. inst	Apply Exh. branchCut	Apply Exh. removeAssumption	Apply Exh. assume
Apply Refine	Apply Refine	Apply Refine	Apply Refine

Successfully ran script.

# Exemplary Walkthrough

Save Refresh Try Close All Settings

Project Chooser About

AlgoVerList.dfy getAt List.getAt/loop/Inv[nodeInv] ▾

default X

```
10x2 <= 0 && idx2 <= pos
node2 == this.nodeseqq[idx2]

idx2 < pos

        |this.seqq| == |this.nodeseqq|
        && |this.seqq| >= 1
        && this.head != null
        && (forall n:int :: (           n >= 0
                                && n < |this.nodeseqq|
                                ==> this.nodeseqq[n] != null))
        && (forall i:int :: (           i >= 0 && i < |this.seqq|
                                ==> this.seqq[i]
                                ==> this.nodeseqq[i].value))
        && (forall k:int :: (           k >= 0 && k < |this.nodeseqq| - 1
                                ==> this.nodeseqq[k].next != null))
        && this.nodeseqq[0] == this.head
        && this.nodeseqq[|this.nodeseqq| - 1].next == null

    idx2 >= 0 && idx2 < |this.nodeseqq| - 1
==> this.nodeseqq[idx2].next == this.nodeseqq[|idx2 + 1|]
```

node2 == this.nodeseqq[idx2]

1O andLeft on='...((?match: \_ && \_ && \_)) ...'
2O expand on='...((?match: this.Valid())) ...'
3O andLeft on='...((?match: \_ && \_ && \_ && \_ &
4O inst with='idx2';
5O 1

Rules

idx2 >= 0 && idx2 < |this.nodeseqq| - 1  
==> this.nodeseqq[idx2].next == this.nodeseqq[|idx2 + 1|]

Alphabetical  Number of Branches

Apply Exh. 0 boogie Apply Refine	Apply Exh. 0 fake Apply Refine	Apply Exh. 1 prop Apply Refine	Apply Exh. 1 testTrueAssum... Apply Refine
Apply Exh. 0 z3 Apply Refine	Apply Exh. 1 skip Apply Refine	Apply Exh. 2 implLeft Apply Refine	Apply Exh. 2 modusTollens Apply Refine
Apply Exh. 7 addHypothesis Apply Refine	Apply Exh. ? replace Apply Refine	Apply Exh. ? branchCut Apply Refine	Apply Exh. 1 removeAssumption Apply Refine

# Exemplary Walkthrough

Save Refresh Try Close All Settings

Project Chooser About

AlgoVerList.dfy getAt List.getAt/loop/Inv[nodeInv] ▾

```
default X
  idxz <= 0 && idxz <= pos
  nodez == this.nodeseqq[idxz]

  idxz < pos

    |this.seqq| == |this.nodeseqq|
    && |this.seqq| >= 1
    && this.head != null
    && (forall n:int :: ( n >= 0
      && n < |this.nodeseqq|
      ==> this.nodeseqq[n] != null))
    && (forall i:int :: ( i >= 0 && i < |this.seqq|
      ==> this.seqq[i]
      ==> this.nodeseqq[i].value))
    && (forall k:int :: ( k >= 0 && k < |this.nodeseqq| - 1
      ==> this.nodeseqq[k].next != null))
    && this.nodeseqq[0] == this.head
    && this.nodeseqq[|this.nodeseqq| - 1].next == null

    idxz >= 0 && idxz < |this.nodeseqq| - 1
    ==> this.nodeseqq[idxz].next == this.nodeseqq[|idxz + 1|]

  nodez == this.nodeseqq[idxz]
```

1O andLeft on='...((?match: \_ && \_ && \_)) ...'  
2O expand on='...((?match: this.Valid())) ...'  
3O andLeft on='...((?match: \_ && \_ && \_ && \_ &&  
4O inst with='idxz';  
5O 



## Rules

idxz == 0 && idxz < |this.nodeseqq| - 1  
==> this.nodeseqq[idxz].next == this.nodeseqq[|idxz + 1|]

Alphabetical  Number of Branches

Apply Exh.  0 boogie Apply Refine	Apply Exh.  0 fake Apply Refine	Apply Exh.  1 prop Apply Refine	Apply Exh.  1 testTrueAssum... Apply Refine
Apply Exh.  0 z3 Apply Refine	Apply Exh.  1 skip Apply Refine	Apply Exh.  2 implLeft Apply Refine	Apply Exh.  2 modusTollens Apply Refine
Apply Exh.  7 addHypothesis Apply Refine	Apply Exh.  ? replace Apply Refine	Apply Exh.  ? branchCut Apply Refine	Apply Exh.  1 removeAssumption Apply Refine

# Exemplary Walkthrough

Save Refresh Try Close All Settings

AlgoVerList.dfy getAt List.getValueAt/loop/Inv[nodeInv] ▾

default X Labels

✓ Closed Goal

```
$mod1 == $empty
$decr2 == 0
$oldheap1 == $heap
idx1 == 0
node1 == this.head
$decr_11 == |this.seqq| - idx2
node3 == node2.next
idx3 == idx2 + 1
0 <= pos && pos < |this.seqq|
idx2 >= 0 && idx2 <= pos
node2 == this.nodeseqq[idx2]
idx2 < pos
```

1O andLeft on='...((?match: \_ && \_ && \_)) ...'
2O expand on='...((?match: this.Valid())) ...'
3O andLeft on='...((?match: \_ && \_ && \_ && \_ && \_)) ...'
4O inst with='idx2';
5O boogie ;
6O 

Rules

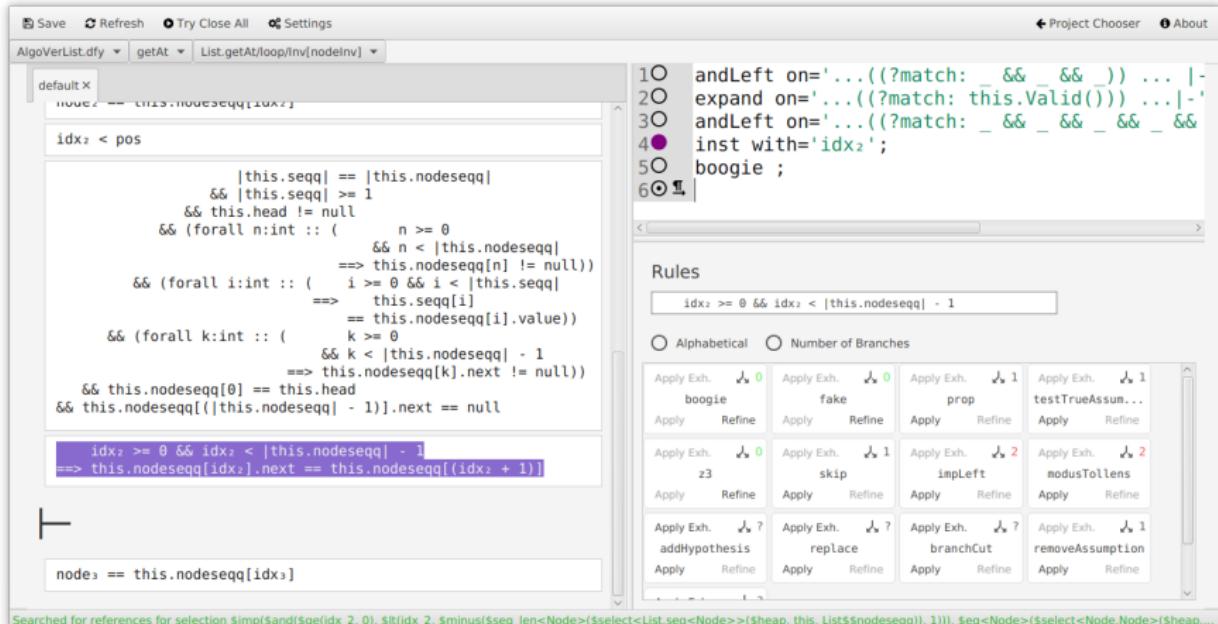
```
$mod1 == $empty
```

Alphabetical  Number of Branches

Apply Exh. boogie 0 boogie Apply Refine	Apply Exh. fake 0 fake Apply Refine	Apply Exh. prop 1 prop Apply Refine	Apply Exh. testTrueAssum... 1 testTrueAssum... Apply Refine
Apply Exh. z3 0 z3 Apply Refine	Apply Exh. skip 1 skip Apply Refine	Apply Exh. addHypothesis ? addHypothesis Apply Refine	Apply Exh. replace ? replace Apply Refine
Apply Exh. branchCut ? branchCut Apply Refine	Apply Exh. removeAssumption 1 removeAssumption Apply Refine	Apply Exh. assume ? assume Apply Refine	

Successfully applied rule boogie.

# Relation in the Proof



The screenshot shows a proof state in the AlgoVerList tool. The left pane displays a sequence of logical steps, starting with a default assumption and progressing through several conditional statements involving variables like `node2`, `idx2`, and `node3`. The right pane lists a set of available rules, each with an apply button and a refine button.

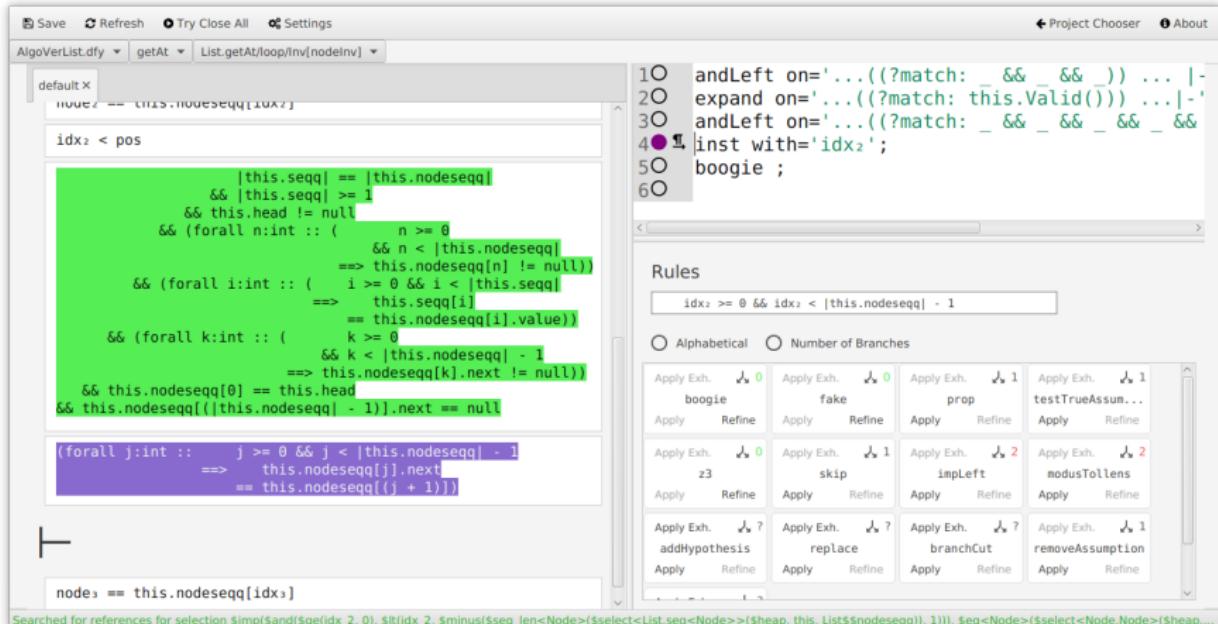
Left pane (Proof State):

```
default X
node2 == this.nodeseqq[1..x]
idx2 < pos
    |this.seqq| == |this.nodeseqq|
    && |this.seqq| >= 1
    && this.head != null
    && (forall n:int :: ( n >= 0
                           && n < |this.nodeseqq|
                           ==> this.nodeseqq[n] != null))
    && (forall i:int :: ( i >= 0 && i < |this.seqq|
                           ==> this.seqq[i]
                           == this.nodeseqq[i].value))
    && (forall k:int :: ( k >= 0
                           && k < |this.nodeseqq| - 1
                           ==> this.nodeseqq[k].next != null))
    && this.nodeseqq[0] == this.head
    && this.nodeseqq[|this.nodeseqq| - 1].next == null
    idx2 >= 0 && idx2 < |this.nodeseqq| - 1
    ==> this.nodeseqq[idx2].next == this.nodeseqq[(idx2 + 1)]
node3 == this.nodeseqq[idx3]
```

Right pane (Rules):

Rule Name	Count	Action
andLeft	0	Apply Exh. Refine
expand	0	Apply Exh. Refine
andLeft on	1	Apply Exh. Refine
inst with='idx2'	1	Apply Exh. Refine
boogie	1	Apply Exh. Refine
z3	0	Apply Exh. Refine
skip	1	Apply Exh. Refine
implLeft	2	Apply Exh. Refine
modusTollens	2	Apply Exh. Refine
addHypothesis	?	Apply Exh. Refine
replace	?	Apply Exh. Refine
branchCut	?	Apply Exh. Refine
removeAssumption	1	Apply Exh. Refine

# Relation in the Proof



The screenshot shows a proof assistant interface with the following components:

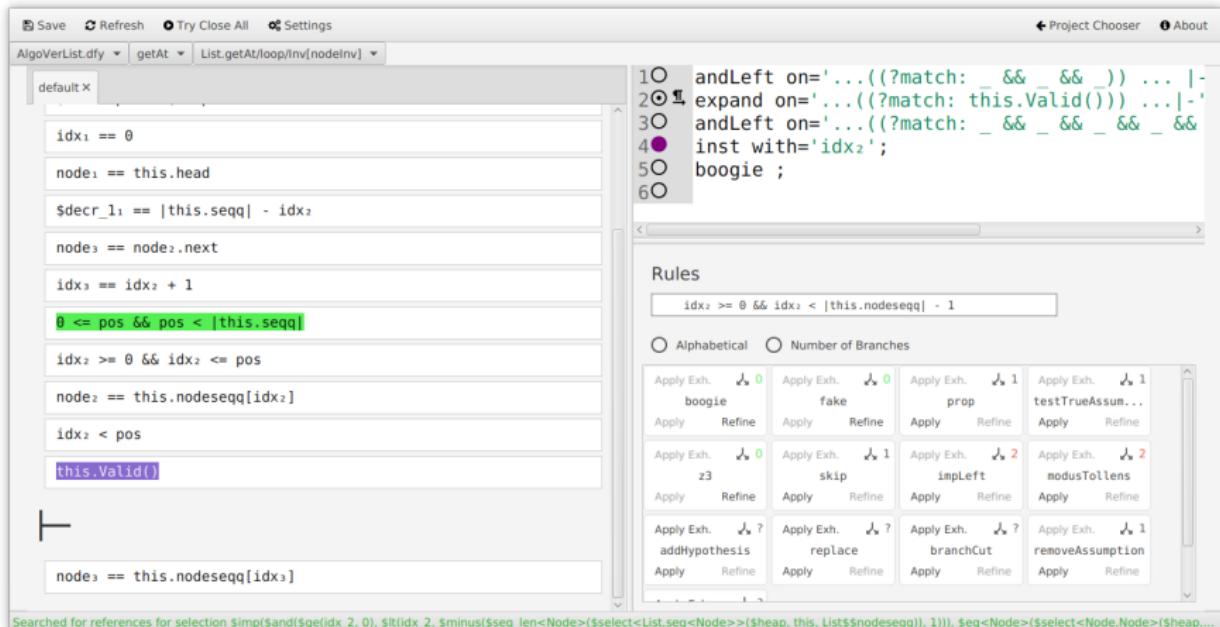
- Top Bar:** Save, Refresh, Try Close All, Settings, Project Chooser, About.
- Toolbar:** AlgoVerList.dfy, getAt, List.getAt/loop/inv[nodeInv].
- Left Panel (Proof State):** A code editor window showing a proof script. The current selection is highlighted in green:

```
default X
  nodez == this.nodeseqq[10xz]
  idxz < pos
    [this.seqq] == [this.nodeseqq]
      && |this.seqq| >= 1
      && this.head != null
      && (forall n:int :: ( n >= 0
        && n < |this.nodeseqq|
        ==> this.nodeseqq[n] != null))
      && (forall i:int :: ( i >= 0 && i < |this.seqq|
        ==> this.seqq[i]
        == this.nodeseqq[i].value))
      && (forall k:int :: ( k >= 0
        && k < |this.nodeseqq| - 1
        ==> this.nodeseqq[k].next != null))
      && this.nodeseqq[0] == this.head
      && this.nodeseqq[|this.nodeseqq| - 1].next == null

    (forall j:int :: ( j >= 0 && j < |this.nodeseqq| - 1
      ==> this.nodeseqq[j].next
      == this.nodeseqq[(j + 1)])
```
- Right Panel (Rule List):** A list of proof rules categorized under "Rules".

	idxz >= 0 && idxz <  this.nodeseqq  - 1						
<input type="radio"/> Alphabetical	<input type="radio"/> Number of Branches						
Apply Exh.	boogie	Apply Exh.	fake	Apply Exh.	prop	Apply Exh.	testTrueAssum...
Apply	Refine	Apply	Refine	Apply	Refine	Apply	Refine
Apply Exh.	z3	Apply Exh.	skip	Apply Exh.	implLeft	Apply Exh.	modusTollens
Apply	Refine	Apply	Refine	Apply	Refine	Apply	Refine
Apply Exh.	addHypothesis	Apply Exh.	replace	Apply Exh.	branchCut	Apply Exh.	removeAssumption
Apply	Refine	Apply	Refine	Apply	Refine	Apply	Refine
- Bottom Status Bar:** Searched for references for selection \$imp(\$and(\$ge(idx\_2, 0), \$lt(idx\_2, \$minus(\$seq\_len<Node>(\$select<List,seq<Node>">(\$heap, this, List\$nodeseqq)), 1))), \$eq<Node>(\$select<Node,Node>(\$heap,...

# Relation in the Proof



The screenshot shows a software interface for program verification. At the top, there are menu items: Save, Refresh, Try Close All, Settings, Project Chooser, and About. Below the menu is a toolbar with buttons for Save, Refresh, Try Close All, and Settings. The main area has tabs: AlgoVerList.dfy, getAt, and List.getAt/loop/Inv[nodeInv]. The left pane displays a sequence of code or proof steps:

```
default X
idx1 == 0
node1 == this.head
$decr_1: == |this.seqq| - idx1
node2 == node1.next
idx2 == idx1 + 1
0 <= pos && pos < |this.seqq|
idx2 >= 0 && idx2 <= pos
node2 == this.nodeseqq[idx2]
idx2 < pos
this.Valid()
node3 == this.nodeseqq[idx3]
```

The right pane shows a list of rules:

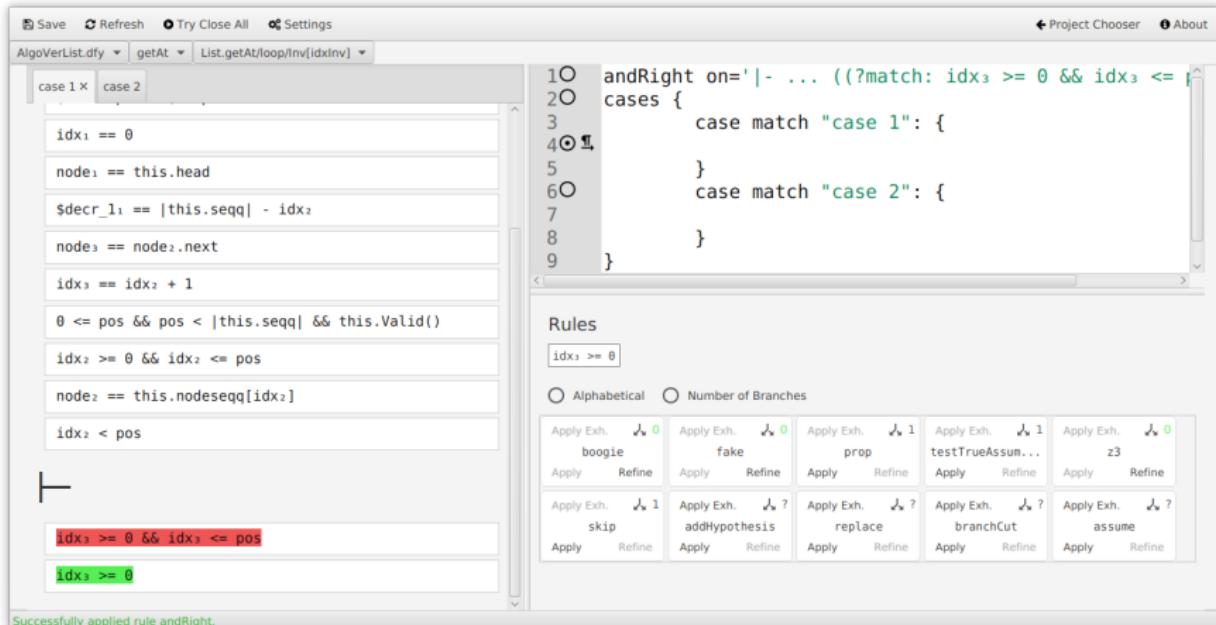
Rule	Condition
andLeft on='...((?match: _ && _ && _)) ...  -'	idx2 >= 0 && idx2 <  this.nodeseqq  - 1
expand on='...((?match: this.Valid())) ...  -'	
andLeft on='...((?match: _ && _ && _ && _ && _ && _)) ...  -'	
inst with='idx2'; boogie ;	
boogie ;	

Below the rules, there is a section titled "Rules" with a search bar containing the condition `idx2 >= 0 && idx2 < |this.nodeseqq| - 1`. There are two radio buttons: "Alphabetical" and "Number of Branches". A scrollable list of rules is shown:

Rule	Condition
Apply Exh. boogie	idx2 >= 0
Apply Refine	
Apply Exh. fake	idx2 >= 0
Apply Refine	
Apply Exh. prop	idx2 >= 0
Apply Refine	
Apply Exh. testTrueAssum...	idx2 >= 0
Apply Refine	
Apply Exh. z3	idx2 >= 0
Apply Refine	
Apply Exh. skip	idx2 >= 0
Apply Refine	
Apply Exh. impleft	idx2 >= 0
Apply Refine	
Apply Exh. modulusTollens	idx2 >= 0
Apply Refine	
Apply Exh. addHypothesis	idx2 >= 0
Apply Refine	
Apply Exh. replace	idx2 >= 0
Apply Refine	
Apply Exh. branchCut	idx2 >= 0
Apply Refine	
Apply Exh. removeAssumption	idx2 >= 0
Apply Refine	

At the bottom, a status message says: "Searched for references for selection \$imp(\$and(\$ge(idx\_2, 0), \$lt(idx\_2, \$minus(\$seqq\_len<Node>)(\$select<List,seq<Node>">(\$heap, this, List\$\$nodeseqq)), 1))), \$eq<Node>(\$select<Node,Node>">(\$heap, this, List\$\$nodeseqq), node2))".

# Branching in a Proof



The screenshot shows the AlgoVer tool interface with the following details:

- Toolbar:** Save, Refresh, Try Close All, Settings, Project Chooser, About.
- File:** AlgoVerList.dfy, getAt, List.getAt/loop/Inv[idxInv].
- Left Panel:** A tree view showing a proof state with nodes labeled case 1 X and case 2. The code under case 1 X is:

```
case 1 X
  case 2
    idx1 == 0
    node1 == this.head
    $decr_1 == |this.seqq| - idx1
    node2 == node1.next
    idx2 == idx1 + 1
    0 <= pos && pos < |this.seqq| && this.Valid()
    idx2 >= 0 && idx2 <= pos
    node2 == this.nodeseqq[idx2]
    idx2 < pos
```
- Right Panel:** A code editor showing:

```
1O andRight on='|- ... ((?match: idx3 >= 0 && idx3 <= |' ...
2O cases {
3
4O ①
5
6O
7
8
9}
```
- Rules Section:** A grid of branching rules:

idx3 >= 0	
<input type="radio"/> Alphabetical	
Apply Exh. boogie	Apply Exh. fake
Apply Refine	Apply Refine
Apply Exh. skip	Apply Exh. addHypothesis
Apply Refine	Apply Refine
Apply Exh. replace	Apply Exh. branchCut
Apply Refine	Apply Refine
Apply Exh. assume	Apply Exh. ?
Apply Refine	Apply Refine
- Status Bar:** Successfully applied rule andRight.

# Further Features and WIP

- simple lemmas can be encoded using Dafny syntax allowing to add rewrite rules
- call and callsite information is available

Calls:

Function f1 (line13)

Parameter Name	Type	Argument on call
x	int	3

Decreases Clause:  
 $f2(x, 2)$

Function f1 (line16)

Parameter Name	Type	Argument on call
x	int	4

Decreases Clause:  
 $f2(x, 2)$

Function f1 (line18)

Parameter Name	Type	Argument on call
x	int	5

Decreases Clause:  
 $f2(x, 2)$

[Close](#)

# Related Work

## Provide different views

e.g., KIV, Why3, KeY, KeYmaeraX, Coq, Isabelle,...

## Combination of automation and interaction

Why3 (hierarchical structuring of proof task and usage of different solvers)

## Interaction styles

- auto-active verification: VCC, Dafny, OpenJML
- direct manipulation: KeY, KeYmaeraX, KIV
- text-based: Coq, Isabelle
- combination direct manipulation + text-based: KeYmaeraX

## Verification IDEs

e.g., Why3, DafnyIDE

## Our Contributions

- qualitative user studies on interaction in interactive verification systems
- a new user interaction concept
  - combination of different interaction styles
  - application of usability principles  
(e.g., visual clarity, substitutivity, anticipation, consistency, ...)
  - integration of supporting features
- an implementation of the concept in the research prototype DIVE  
(<https://github.com/mattulbrich/dive>)

## Conclusion

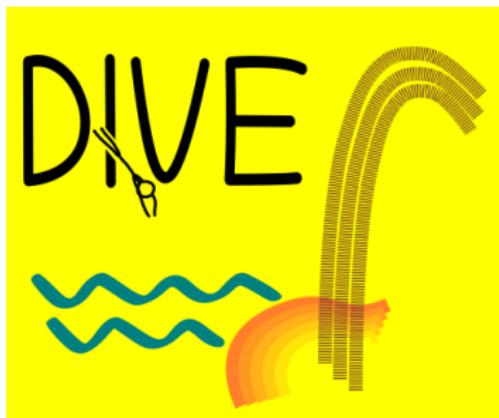
- proof state comprehension is one of the central issues
- there is not one single interaction strategy
- interaction styles can be combined for an improved user support

## Future Work

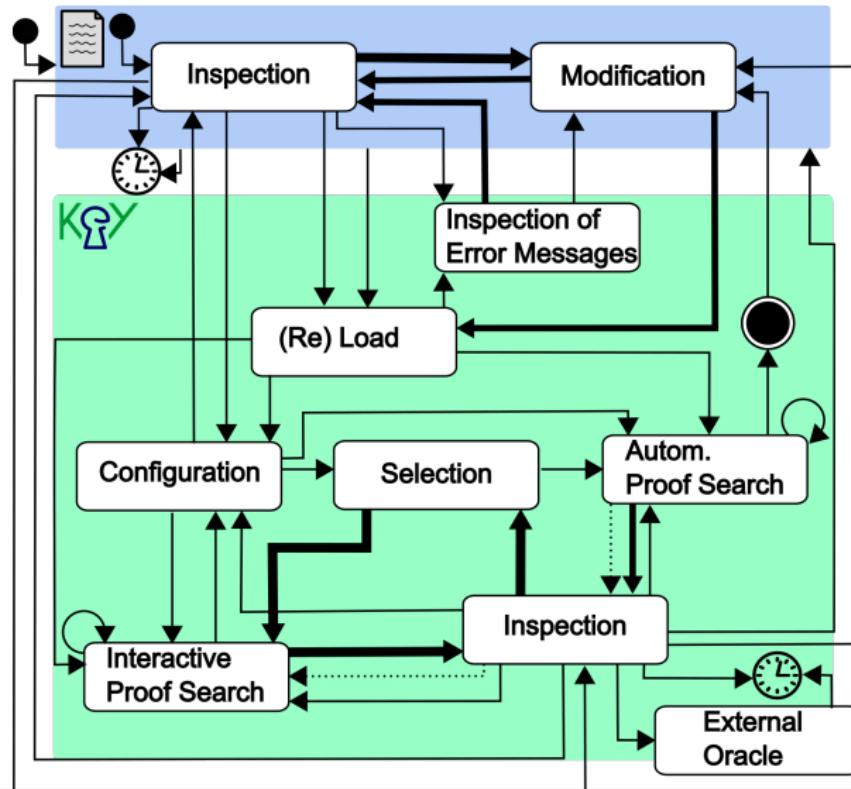
- encode user interactions back to annotations
- include counter examples in the DIVE user interface
- larger case studies and quantitative user studies

Thank you for your attention!

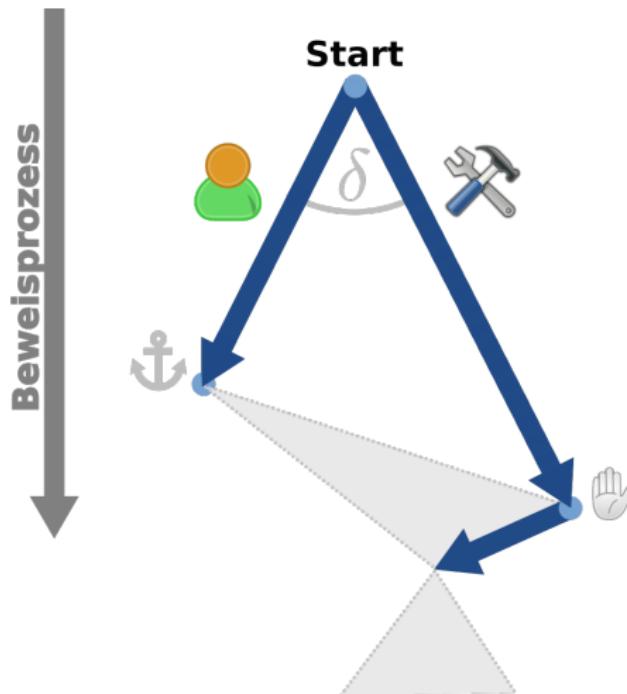
<https://github.com/mattulbrich/dive>



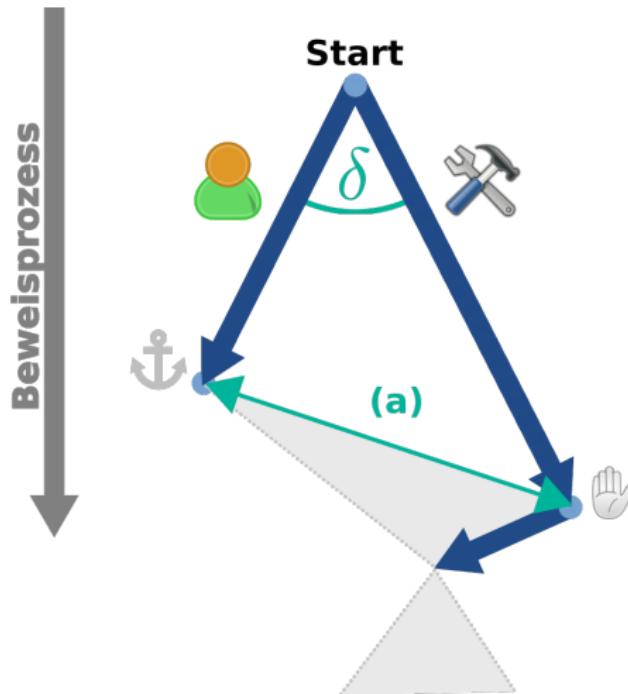
# Consolidated Sequence Model



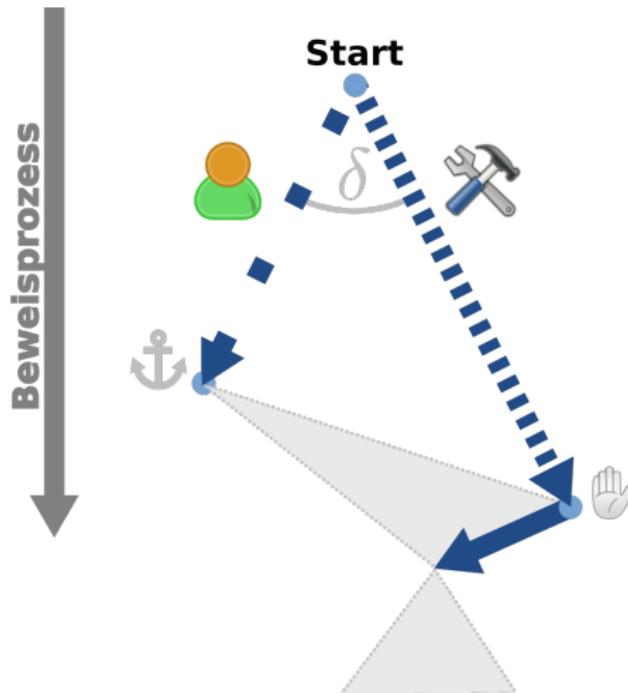
# Model of Process



# Model of Process



# Model of Process



# Model of Process

