Computer-Aided Reasoning for Software

CSE507

Bounded Verification

courses.cs.washington.edu/courses/cse507/18sp/

Emina Torlak
emina@cs.washington.edu

Today

Last lecture

- Full functional verification with Dafny, Boogie, and Z3

Today

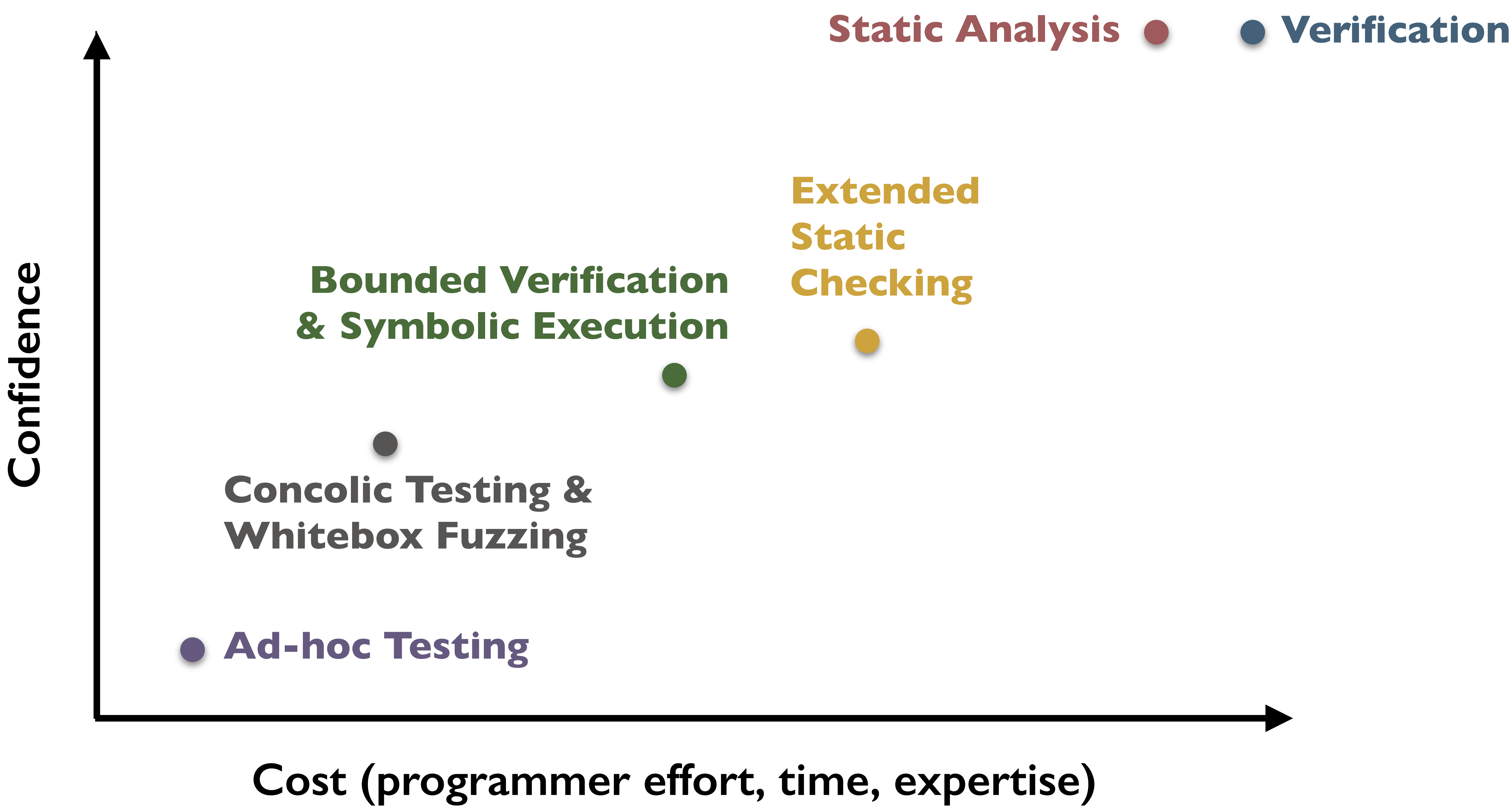
- Bounded verification with Kodkod (Forge, Miniatur, TACO)

Announcements

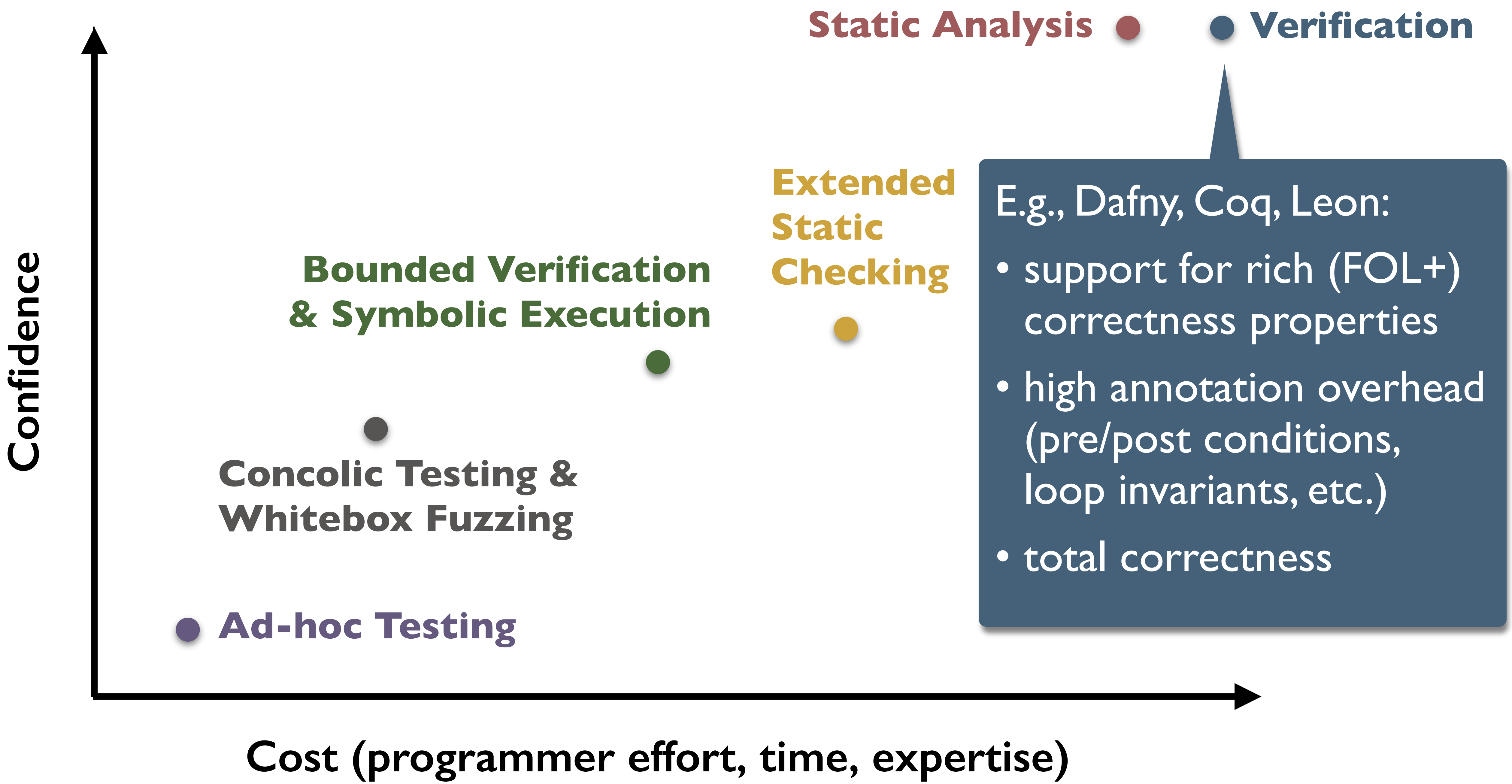
- HW2 is due.
- HW3 is out; start early.

topics

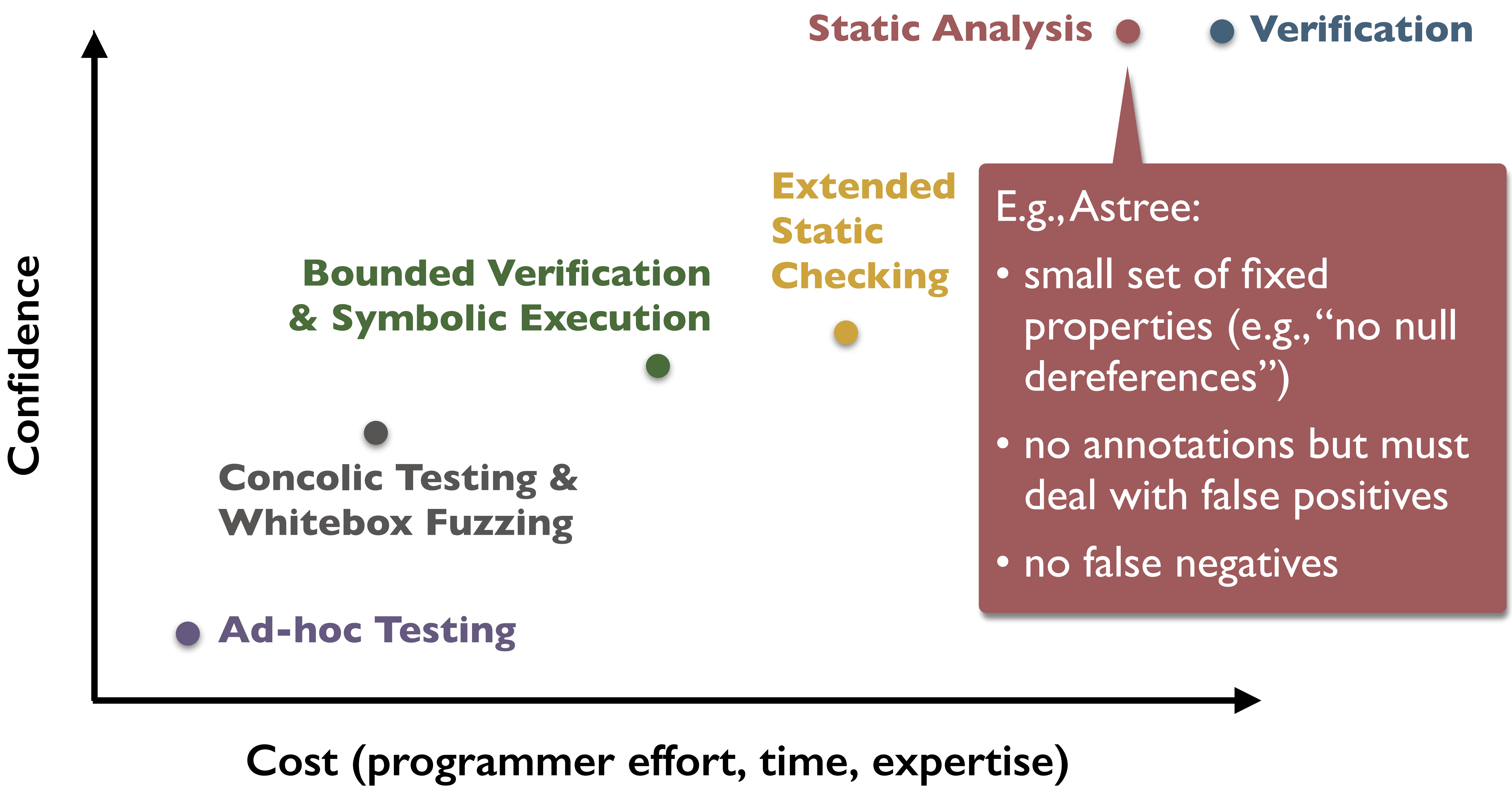
The spectrum of program verification tools



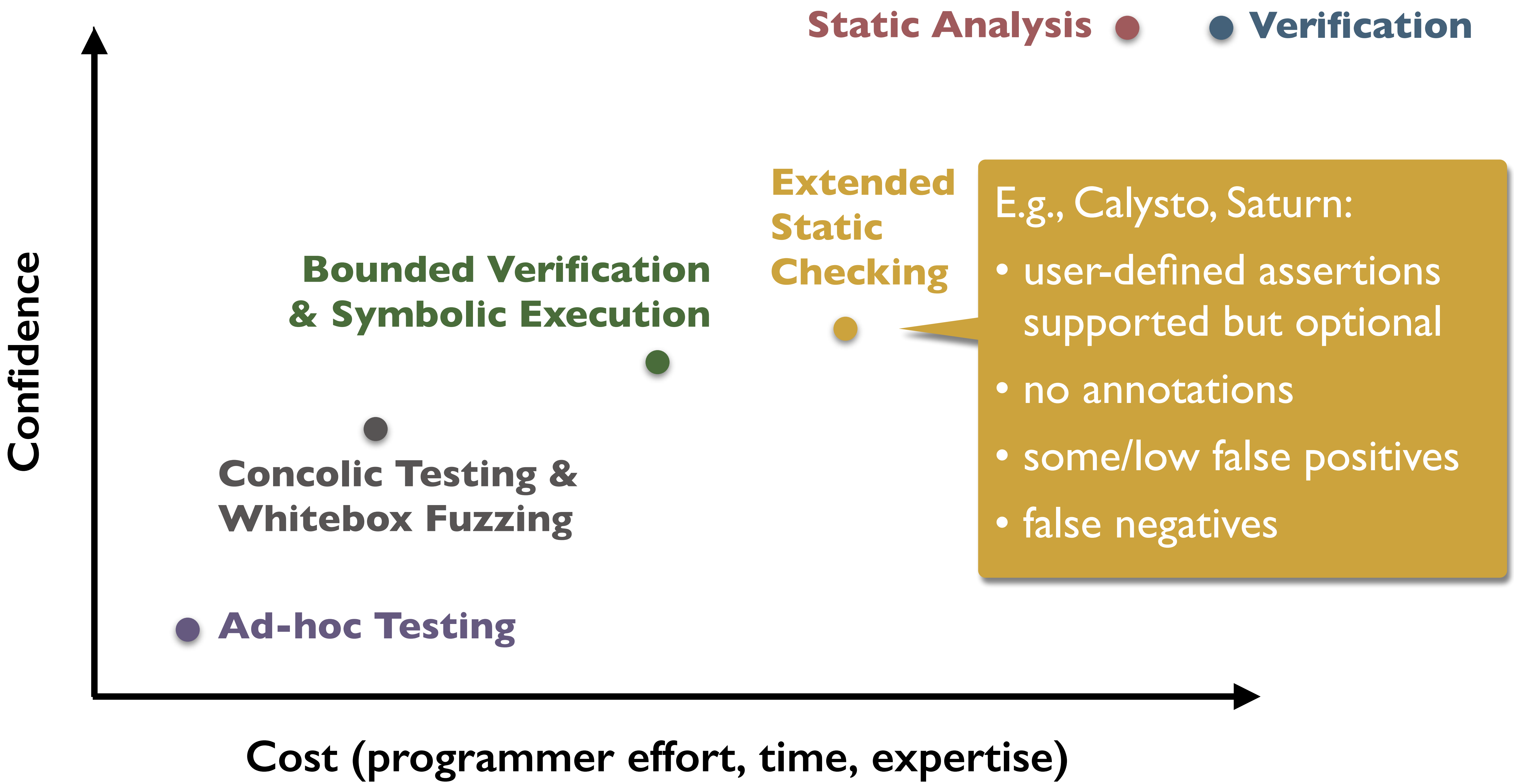
The spectrum of program verification tools



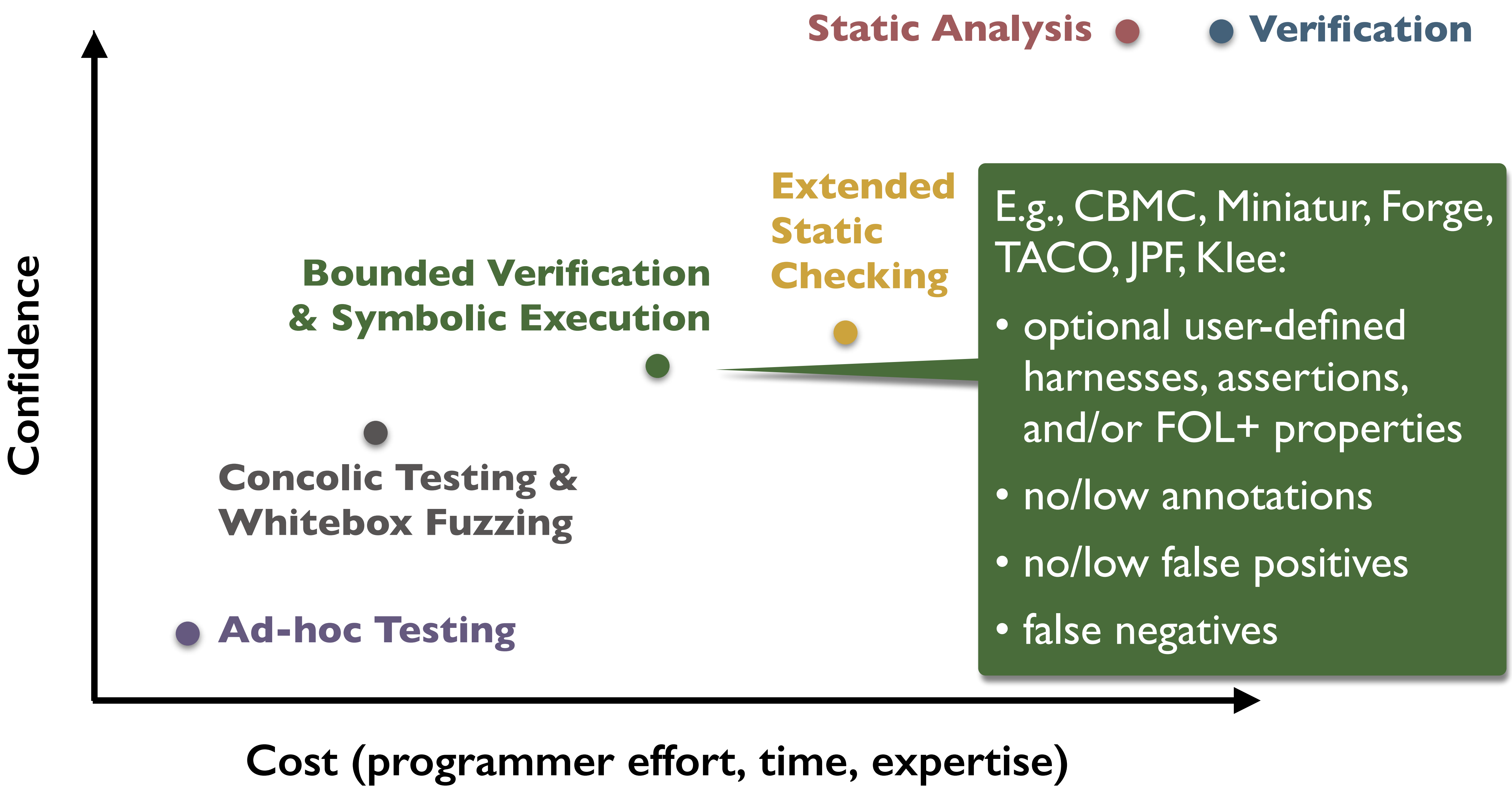
The spectrum of program verification tools



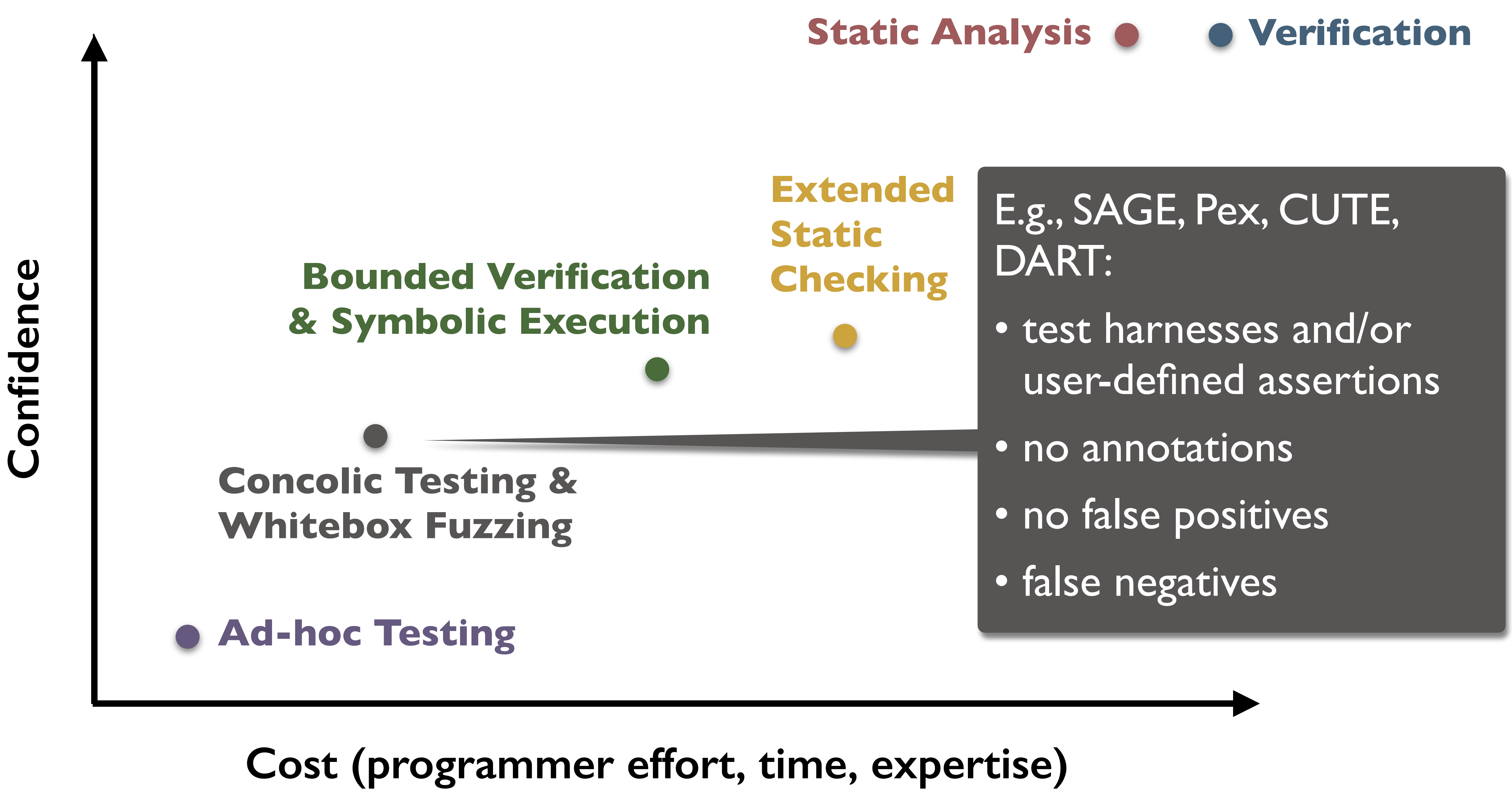
The spectrum of program verification tools



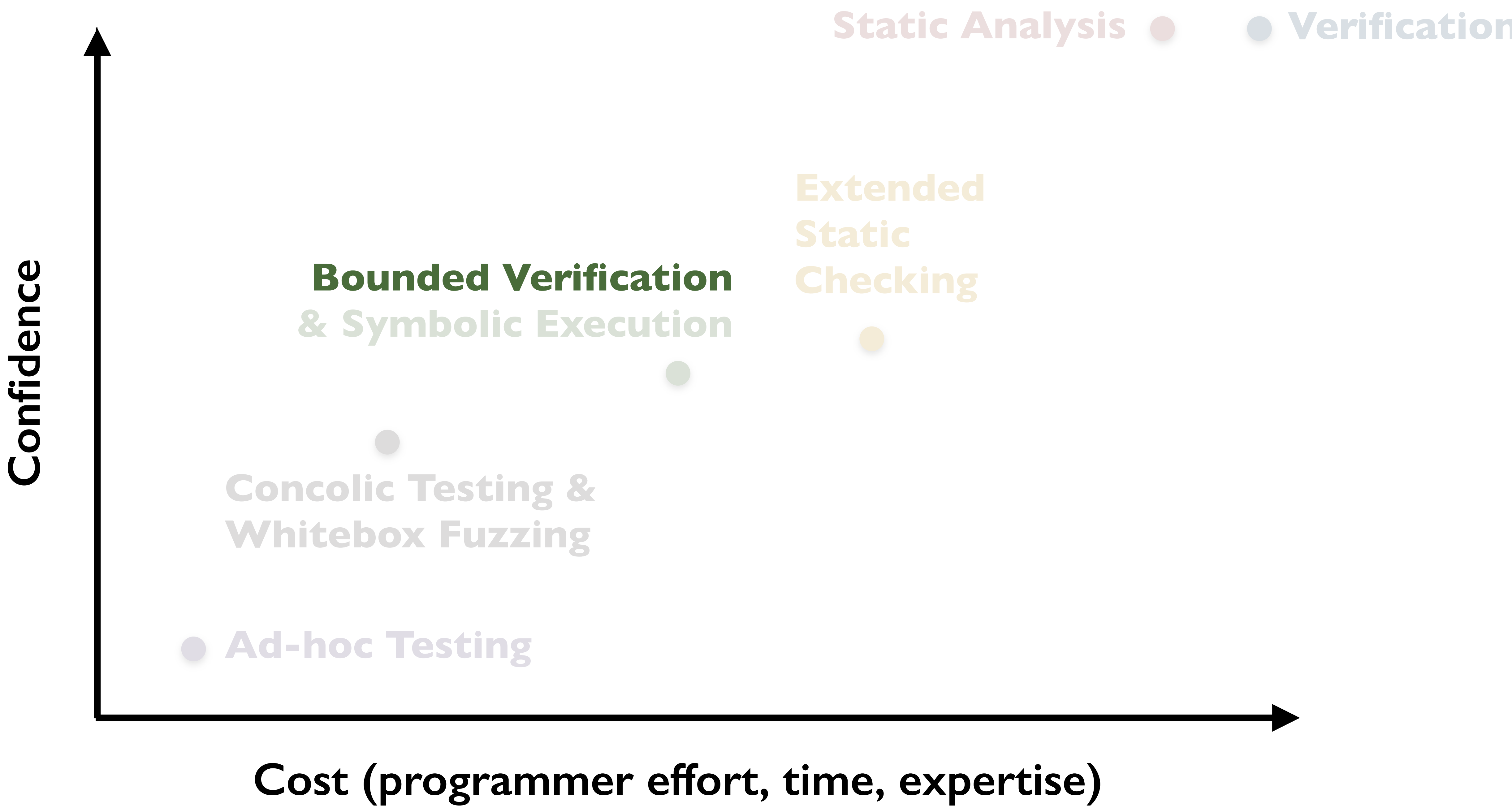
The spectrum of program verification tools



The spectrum of program verification tools



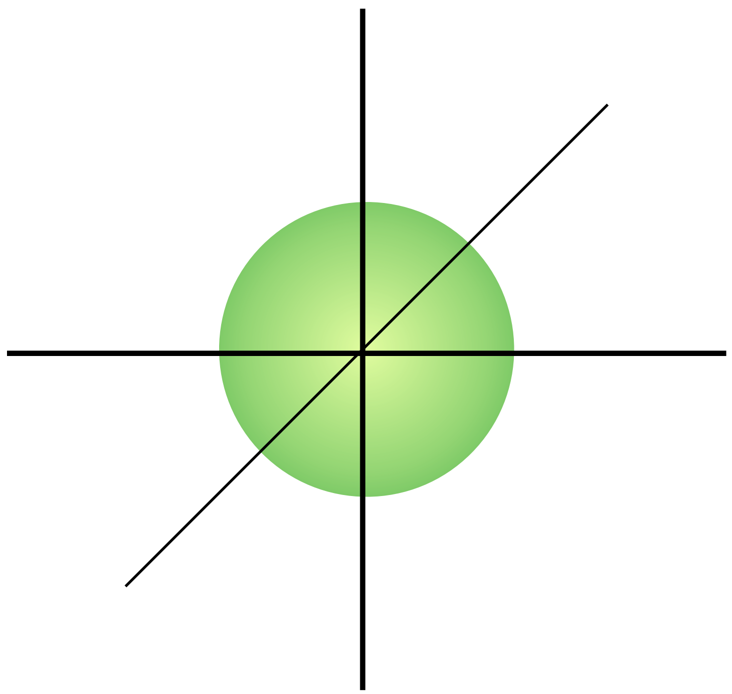
The spectrum of program verification tools



Bounded verification

Bound *everything*

- Execution length
- Bitwidth
- Heap size (number of objects per type)



Sound counterexamples but no proof

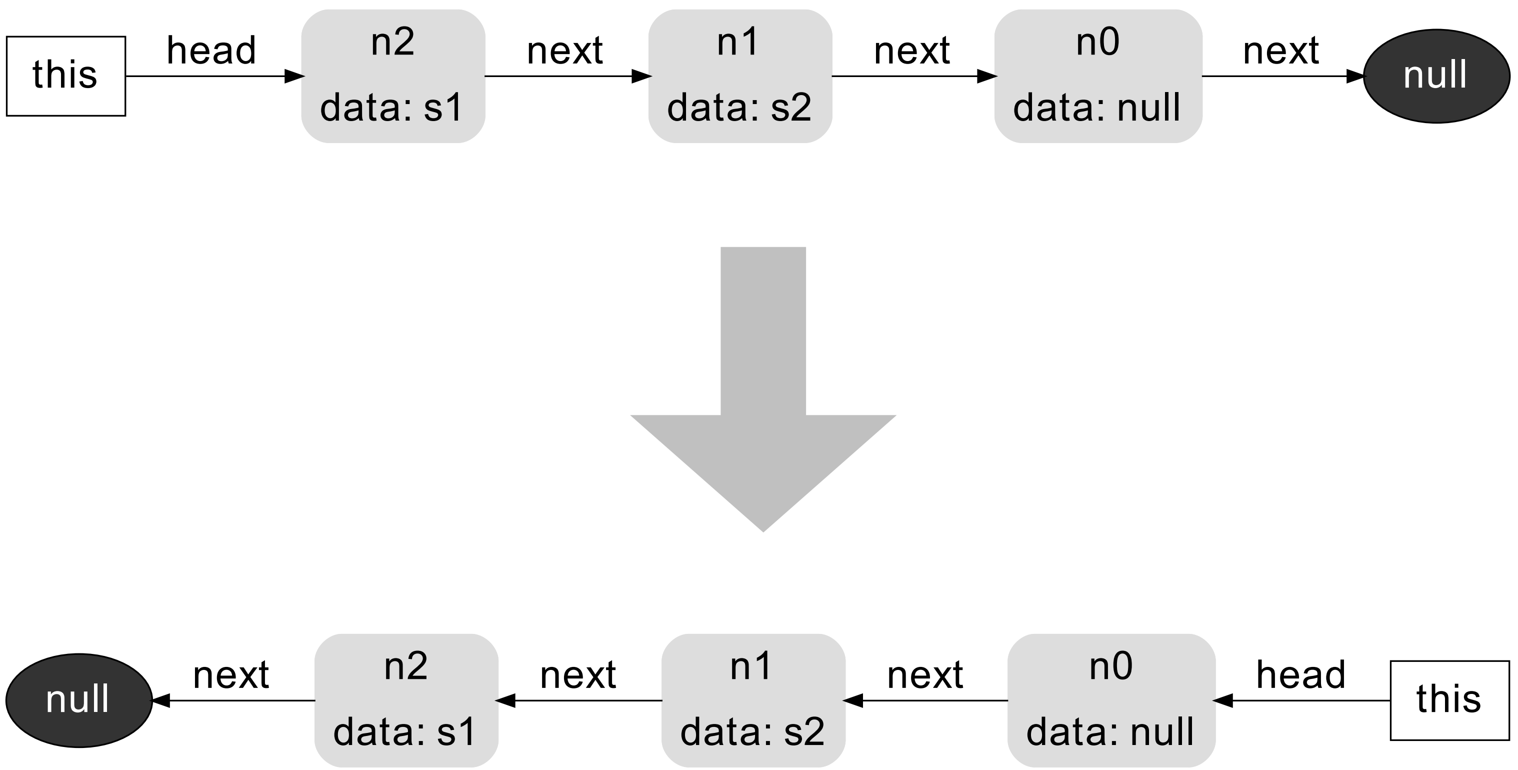
- Exhaustive search within bounded scope

Empirical "small-scope hypothesis"

- Bugs usually have small manifestations

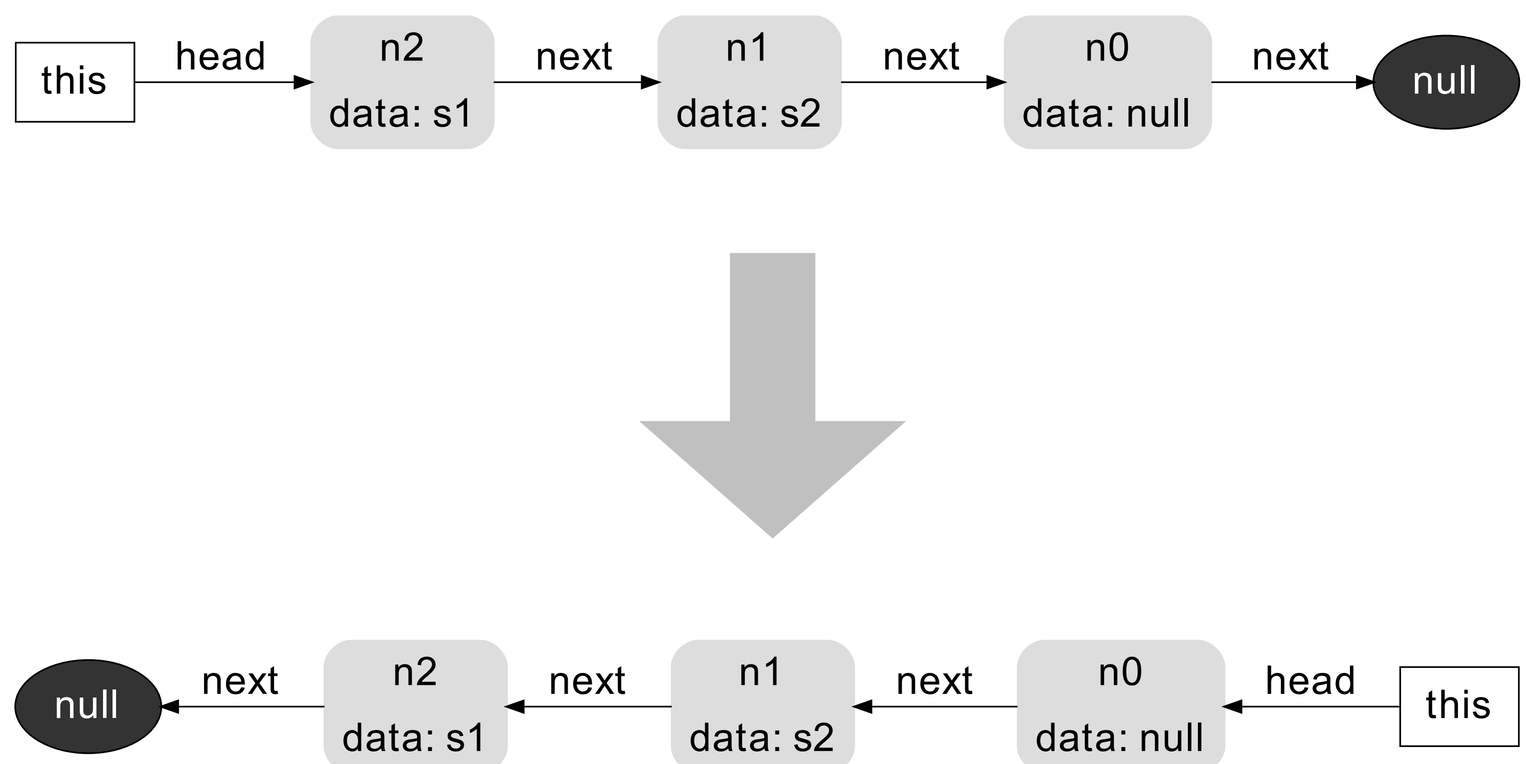
Bounded verification by example

```
class List {
  Node head;

  void reverse() {
    Node near = head;
    Node mid = near.next;
    Node far = mid.next;

    near.next = far;
    while (far != null) {
      mid.next = near;
      near = mid;
      mid = far;
      far = far.next;
    }

    mid.next = near;
    head = mid;
  }
}

class Node {
  Node next;
  String data; }
```



Bounded verification by example

```
class List {
   Node head;

   void reverse() {
      Node near = head;
      Node mid = near.next;
      Node far = mid.next;

      near.next = far;
      while (far != null) {
         mid.next = near;
         near = mid;
         mid = far;
         far = far.next;
      }

      mid.next = near;
      head = mid;
   }
}

class Node {
   Node next;
   String data; }
```

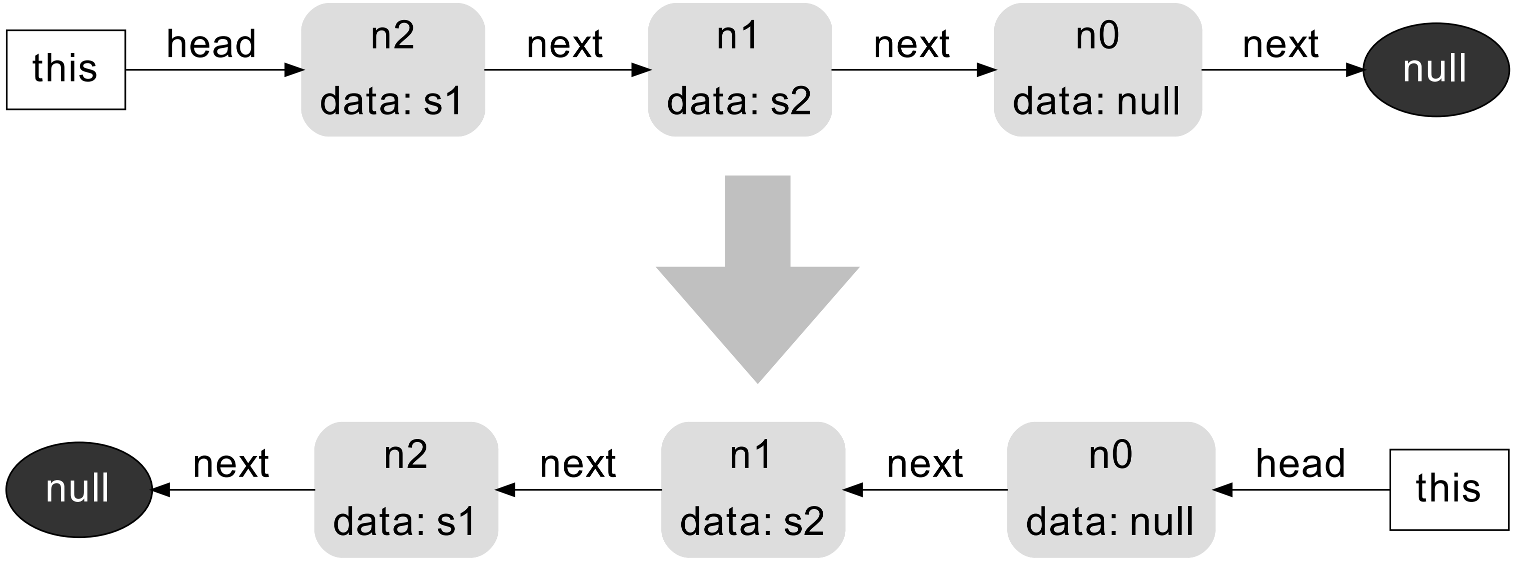


Express the property either by writing a test harness or by providing FOL + *contracts*.

Specifying contracts: class invariants

```
class List {
  Node head;

  void reverse() {
    Node near = head;
    Node mid = near.next;
    Node far = mid.next;

    near.next = far;
    while (far != null) {
      mid.next = near;
      near = mid;
      mid = far;
      far = far.next;
    }

    mid.next = near;
    head = mid;
  }
}

class Node {
  Node next;
  String data;
}
```

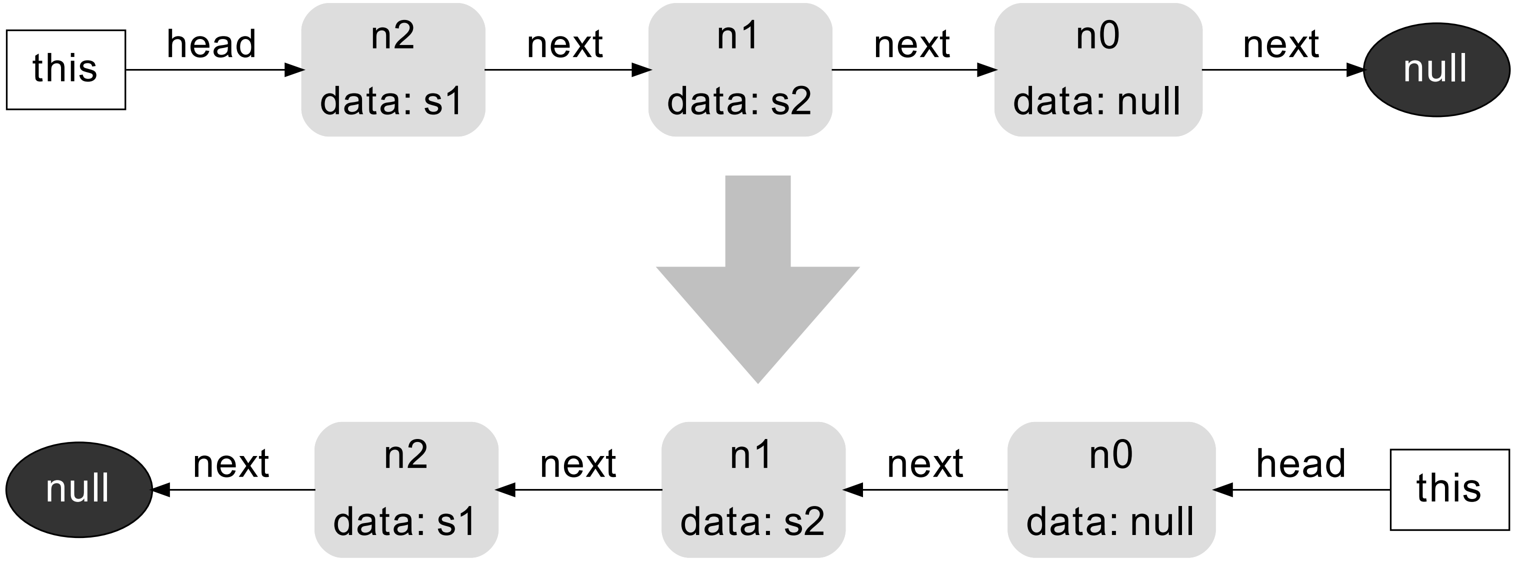
```
@invariant
  no ^next ∩ iden
```



Specifying contracts: preconditions

```
class List {
  Node head;

  void reverse() {
    Node near = head;
    Node mid = near.next;
    Node far = mid.next;

    near.next = far;
    while (far != null) {
      mid.next = near;
      near = mid;
      mid = far;
      far = far.next;
    }

    mid.next = near;
    head = mid;
  }
}

class Node {
  Node next;
  String data;
}
```

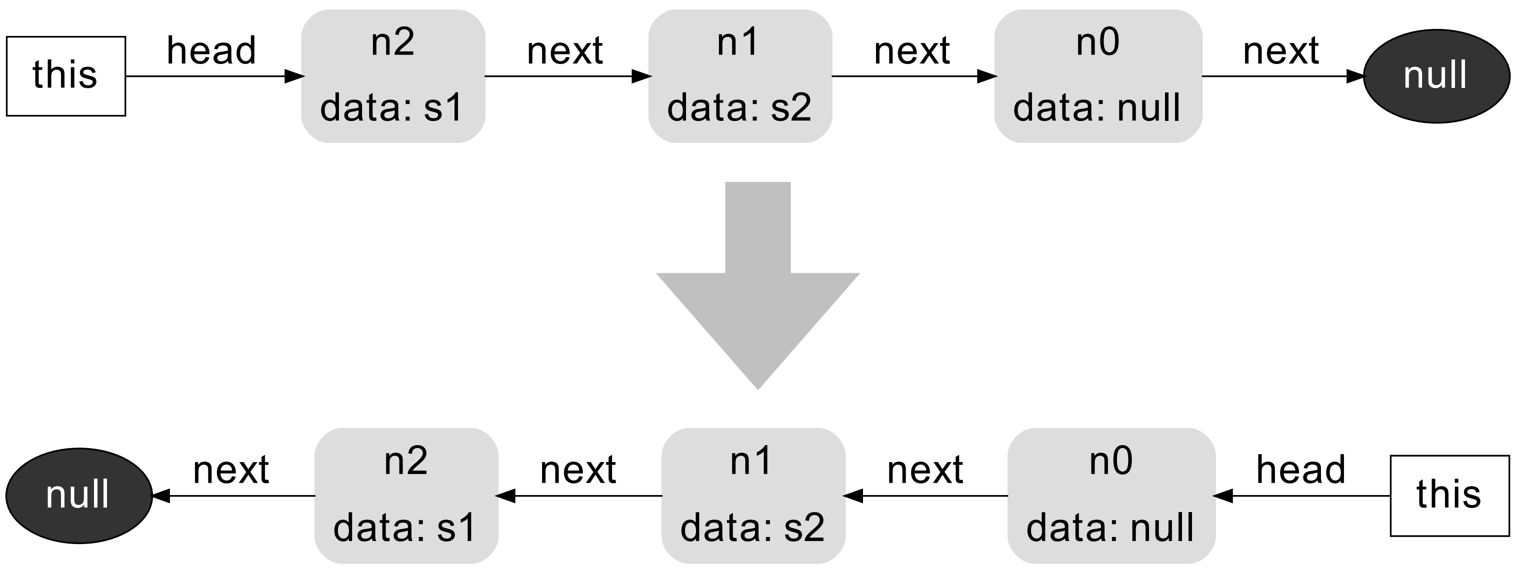
```
@invariant
  no ^next ∩ iden

@requires
  this.head != null and
  this.head.next != null
```



Specifying contracts: postconditions

```
class List {
  Node head;

  void reverse() {
    Node near = head;
    Node mid = near.next;
    Node far = mid.next;

    near.next = far;
    while (far != null) {
      mid.next = near;
      near = mid;
      mid = far;
      far = far.next;
    }

    mid.next = near;
    head = mid;
  }
}

class Node {
  Node next;
  String data;
}
```

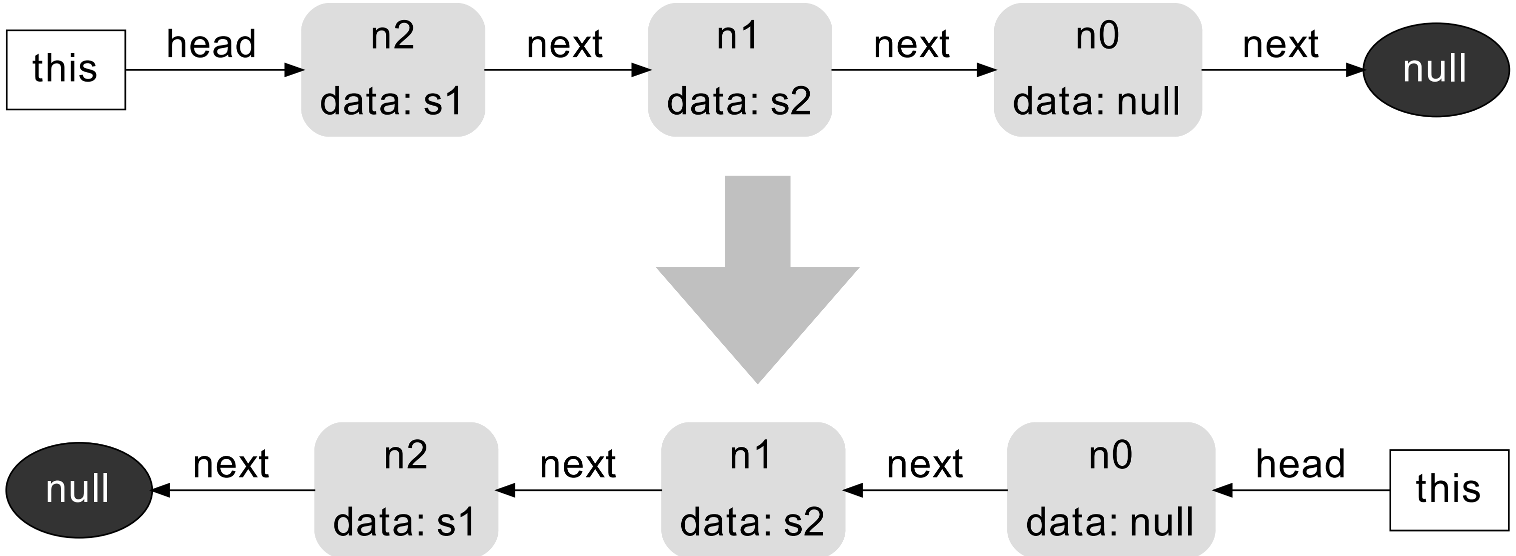
```
@invariant
  no ^next ∩ iden

@requires
  this.head != null and
  this.head.next != null

@ensures
  this.head.*next = this.old(head).*old(next) and
  let N = this.old(head).*old(next) − null |
    next = old(next) ++
           this.old(head)×null ++
           ~(old(next) ∩ N×N)
```



Specifying contracts: postconditions

```
class List {
  Node head;

  void reverse() {
    Node near = head;
    Node mid = near.next;
    Node far = mid.next;

    near.next = far;
    while (far != null) {
      mid.next = near;
      near = mid;
      mid = far;
      far = far.next;
    }

    mid.next = near;
    head = mid;
  }
}

class Node {
  Node next;
  String data;
}
```

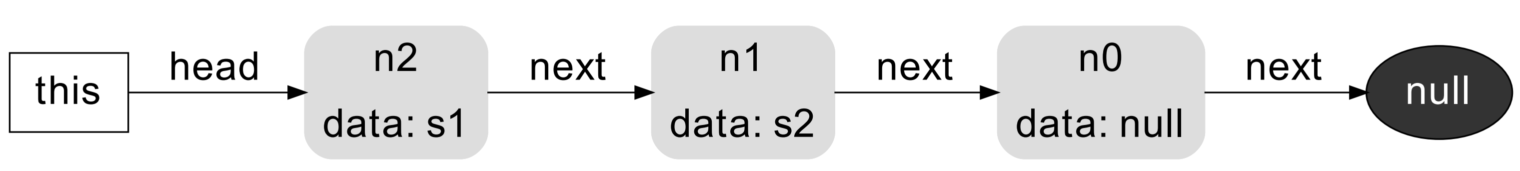
```
@invariant Inv(next)

@requires Pre(this, head, next)

@ensures Post(this, old(head), head, old(next), next)
```



A relational model of memory (heap)

@invariant Inv(next)
@requires Pre(this, head, next)
@ensures Post(this, old(head), head, old(next), next)

```
void reverse() {
  Node near = head;
  Node mid = near.next;
  Node far = mid.next;

  near.next = far;
  while (far != null) {
    mid.next = near;
    near = mid;
    mid = far;
    far = far.next;
  }

  mid.next = near;
  head = mid;
}
```



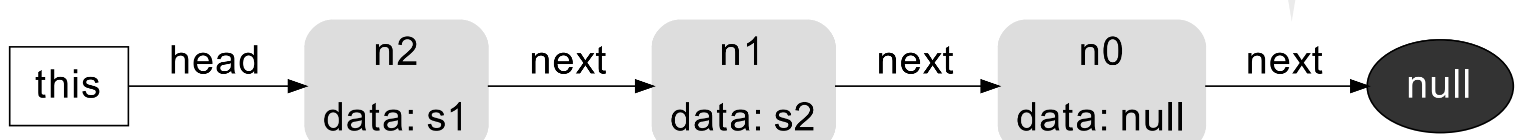
A relational model of memory (heap)

@invariant Inv(next)
@requires Pre(this, head, next)
@ensures Post(this, old(head), head, old(next), next)

```
void reverse() {
  Node near = head;
  Node mid = near.next;
  Node far = mid.next;

  near.next = far;
  while (far != null) {
    mid.next = near;
    near = mid;
    mid = far;
    far = far.next;
  }

  mid.next = near;
  head = mid;
}
```

Fields as binary relations

- head : { ⟨this, n2⟩ }, next : { ⟨n2, n1⟩, … }



A relational model of memory (heap)

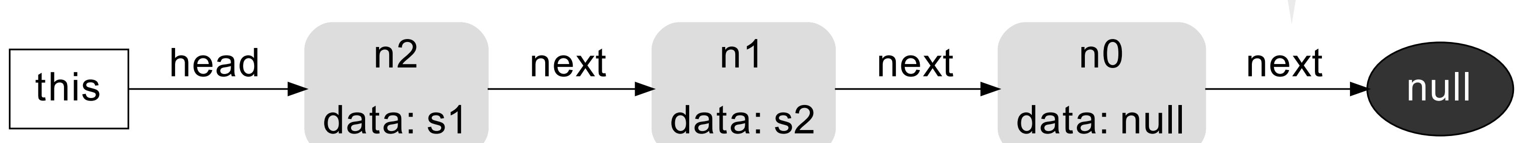
```
@invariant Inv(next)
@requires Pre(this, head, next)
@ensures Post(this, old(head), head, old(next), next)

void reverse() {
   Node near = head;
   Node mid = near.next;
   Node far = mid.next;

   near.next = far;
   while (far != null) {
      mid.next = near;
      near = mid;
      mid = far;
      far = far.next;
   }

   mid.next = near;
   head = mid;
}
```

Fields as binary relations

- head : { ⟨this, n2⟩ }, next : { ⟨n2, n1⟩, … }

Types as sets (unary relations)

- List : { ⟨this⟩ }, Node : { ⟨n0⟩, ⟨n1⟩, ⟨n2⟩ }



A relational model of memory (heap)

@invariant Inv(next)
@requires Pre(this, head, next)
@ensures Post(this, old(head), head, old(next), next)

```
void reverse() {
   Node near = head;
   Node mid = near.next;
   Node far = mid.next;

   near.next = far;
   while (far != null) {
      mid.next = near;
      near = mid;
      mid = far;
      far = far.next;
   }

   mid.next = near;
   head = mid;
}
```

Fields as binary relations

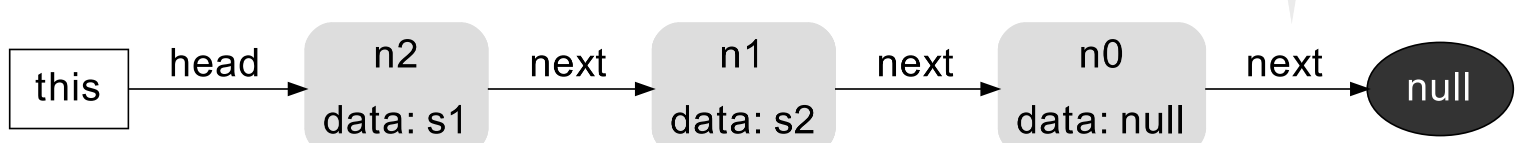
- head : { ⟨this, n2⟩ }, next : { ⟨n2, n1⟩, … }

Types as sets (unary relations)

- List : { ⟨this⟩ }, Node : { ⟨n0⟩, ⟨n1⟩, ⟨n2⟩ }

Objects as scalars (singleton sets)

- this : { ⟨this⟩ }, null : { ⟨null⟩ }



A relational model of memory (heap)

@invariant Inv(next)
@requires Pre(this, head, next)
@ensures Post(this, old(head), head, old(next), next)

```
void reverse() {
   Node near = head;
   Node mid = near.next;
   Node far = mid.next;

   near.next = far;
   while (far != null) {
      mid.next = near;
      near = mid;
      mid = far;
      far = far.next;
   }

   mid.next = near;
   head = mid;
}
```

Fields as binary relations

- head : { ⟨this, n2⟩ }, next : { ⟨n2, n1⟩, … }

Types as sets (unary relations)

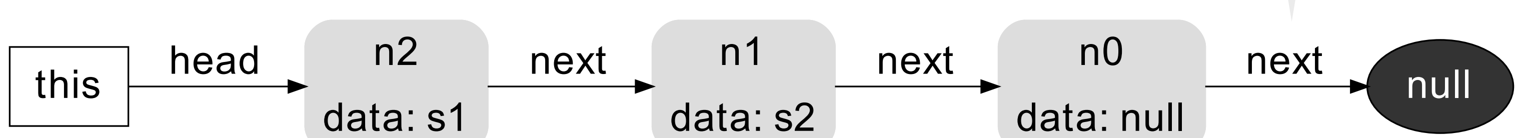
- List : { ⟨this⟩ }, Node : { ⟨n0⟩, ⟨n1⟩, ⟨n2⟩ }

Objects as scalars (singleton sets)

- this : { ⟨this⟩ }, null : { ⟨null⟩ }

Field read as relational join (.)

- this.head : { ⟨this⟩ } . { ⟨this, n2⟩ } = { ⟨n2⟩ }



A relational model of memory (heap)

@invariant Inv(next)
@requires Pre(this, head, next)
@ensures Post(this, old(head), head, old(next), next)

```
void reverse() {
   Node near = head;
   Node mid = near.next;
   Node far = mid.next;

   near.next = far;
   while (far != null) {
      mid.next = near;
      near = mid;
      mid = far;
      far = far.next;
   }

   mid.next = near;
   head = mid;
}
```

Fields as binary relations

- `head` : { ⟨this, n2⟩ }, `next` : { ⟨n2, n1⟩, … }

Types as sets (unary relations)

- `List` : { ⟨this⟩ }, Node : { ⟨n0⟩, ⟨n1⟩, ⟨n2⟩ }

Objects as scalars (singleton sets)

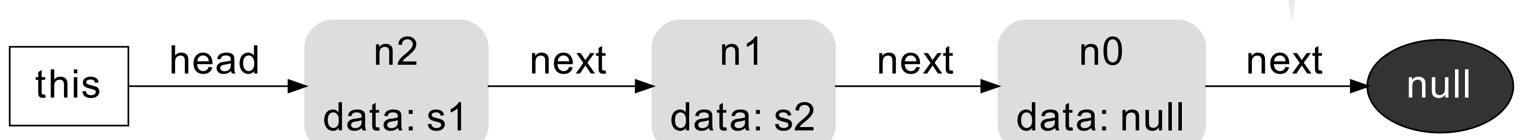
- `this` : { ⟨this⟩ }, `null` : { ⟨null⟩ }

Field read as relational join (.)

- `this.head` : { ⟨this⟩ } . { ⟨this, n2⟩ } = { ⟨n2⟩ }

Field write as relational override (++)

- `this.head = null` : head ++ (this -> null) = { ⟨this, n2⟩ } ++ { ⟨this, null⟩ } = { ⟨this, null⟩ }



Bounded verification: step 1/4

@invariant Inv(next)
@requires Pre(this, head, next)
@ensures Post(this, old(head), head, old(next), next)

```
void reverse() {
   Node near = head;
   Node mid = near.next;
   Node far = mid.next;

   near.next = far;
   while (far != null) {
      mid.next = near;
      near = mid;
      mid = far;
      far = far.next;
   }

   mid.next = near;
   head = mid;
}
```

Bounded verification: step 1/4

```
@invariant Inv(next)
@requires Pre(this, head, next)
@ensures Post(this, old(head), head, old(next), next)

void reverse() {
   Node near = head;
   Node mid = near.next;
   Node far = mid.next;

   near.next = far;
   if (far != null) {
      mid.next = near;
      near = mid;
      mid = far;
      far = far.next;
   }
   assume far == null;

   mid.next = near;
   head = mid;
}
```

Execution finitization (inlining, unrolling, SSA)

Bounded verification: step 1/4

@invariant Inv(next)
@requires Pre(this, head, next)
@ensures Post(this, old(head), head, old(next), next)

```
void reverse() {
   Node near0 = this.head;
   Node mid0 = near0.next;
   Node far0 = mid0.next;

   next0 = update(next, near0, far0);
   boolean guard = (far0 != null);
   next1 = update(next0, mid0, near0);
   near1 = mid0;
   mid1 = far0;
   far1 = far0.next1;

   near2 = phi(guard, near1, near0);
   mid2 = phi(guard, mid1, mid0);
   far2 = phi(guard, far1, far0);
   next2 = phi(guard, next1, next0);

   assume far2 == null;

   next3 = update(next2, mid2, near2);
   head0 = update(head, this, mid2);
}
```

Execution finitization (inlining, unrolling, SSA)

Bounded verification: step 2/4

@invariant Inv(next)
@requires Pre(this, head, next)
@ensures Post(this, old(head), head, old(next), next)

```
void reverse() {
   Node near_0 = this.head;
   Node mid_0 = near_0.next;
   Node far_0 = mid_0.next;

   next_0 = update(next, near_0, far_0);
   boolean guard = (far_0 != null);
   next_1 = update(next_0, mid_0, near_0);
   near_1 = mid_0;
   mid_1 = far_0;
   far_1 = far_0.next_1;

   near_2 = phi(guard, near_1, near_0);
   mid_2 = phi(guard, mid_1, mid_0);
   far_2 = phi(guard, far_1, far_0);
   next_2 = phi(guard, next_1, next_0);

   assume far_2 == null;

   next_3 = update(next_2, mid_2, near_2);
   head_0 = update(head, this, mid_2);
}
```

Execution finitization (inlining, unrolling, SSA)

Forward VCG

Symbolic interpretation of the code with respect to the relational heap model.

Bounded verification: step 2/4

```
@invariant Inv(next)
@requires Pre(this, head, next)
@ensures Post(this, old(head), head, old(next), next)

void reverse() {
  Node near0 = this.head;
  Node mid0 = near0.next;
  Node far0 = mid0.next;

  next0 = update(next, near0, far0);
  boolean guard = (far0 != null);
  next1 = update(next0, mid0, near0);
  near1 = mid0;
  mid1 = far0;
  far1 = far0.next1;

  near2 = phi(guard, near1, near0);
  mid2 = phi(guard, mid1, mid0);
  far2 = phi(guard, far1, far0);
  next2 = phi(guard, next1, next0);

  assume far2 == null;

  next3 = update(next2, mid2, near2);
  head0 = update(head, this, mid2);
}
```

$\text{this} \subseteq \text{List} \wedge \textbf{one}\ \text{this} \wedge$
$\text{head} \subseteq \text{List} \mapsto (\text{Node} \cup \text{null}) \wedge$
$\text{next} \subseteq \text{Node} \mapsto (\text{Node} \cup \text{null}) \wedge$
$\text{data} \subseteq \text{Node} \mapsto (\text{String} \cup \text{null}) \wedge$

$\textbf{let}\ \text{near}_0 = \text{this.head},$
$\text{mid}_0 = \text{near}_0.\text{next},$
$\text{far}_0 = \text{mid}_0.\text{next},$

$\text{next}_0 = \text{next} \mathbin{++} (\text{near}_0 \times \text{far}_0),$
$\text{guard} = (\text{far}_0 \mathrel{!=} \text{null}),$
$\text{next}_1 = \text{next}_0 \mathbin{++} (\text{mid}_0 \times \text{near}_0),$
$\text{near}_1 = \text{mid}_0,$
$\text{mid}_1 = \text{far}_0,$
$\text{far}_1 = \text{far}_0.\text{next}_1,$

$\text{near}_2 = \textbf{if}\ \text{guard}\ \textbf{then}\ \text{near}_1\ \textbf{else}\ \text{near}_0,$
$\text{mid}_2 = \textbf{if}\ \text{guard}\ \textbf{then}\ \text{mid}_1\ \textbf{else}\ \text{mid}_0,$
$\text{far}_2 = \textbf{if}\ \text{guard}\ \textbf{then}\ \text{far}_1\ \textbf{else}\ \text{far}_0,$
$\text{next}_2 = \textbf{if}\ \text{guard}\ \textbf{then}\ \text{next}_1\ \textbf{else}\ \text{next}_0,$
$\text{next}_3 = \text{next}_2 \mathbin{++} (\text{mid}_2 \times \text{near}_2)$
$\text{head}_0 = \text{head} \mathbin{++} (\text{this} \times \text{mid}_2)\ \mid$

$\text{far}_2 = \text{null} \wedge \text{Inv}(\text{next}) \wedge \text{Pre}(\text{this}, \text{head}, \text{next}) \wedge$
$\neg\, (\text{Inv}(\text{next}_3) \wedge \text{Post}(\text{this}, \text{head}, \text{head}_0, \text{next}, \text{next}_3))$

Bounded verification: step 3/4

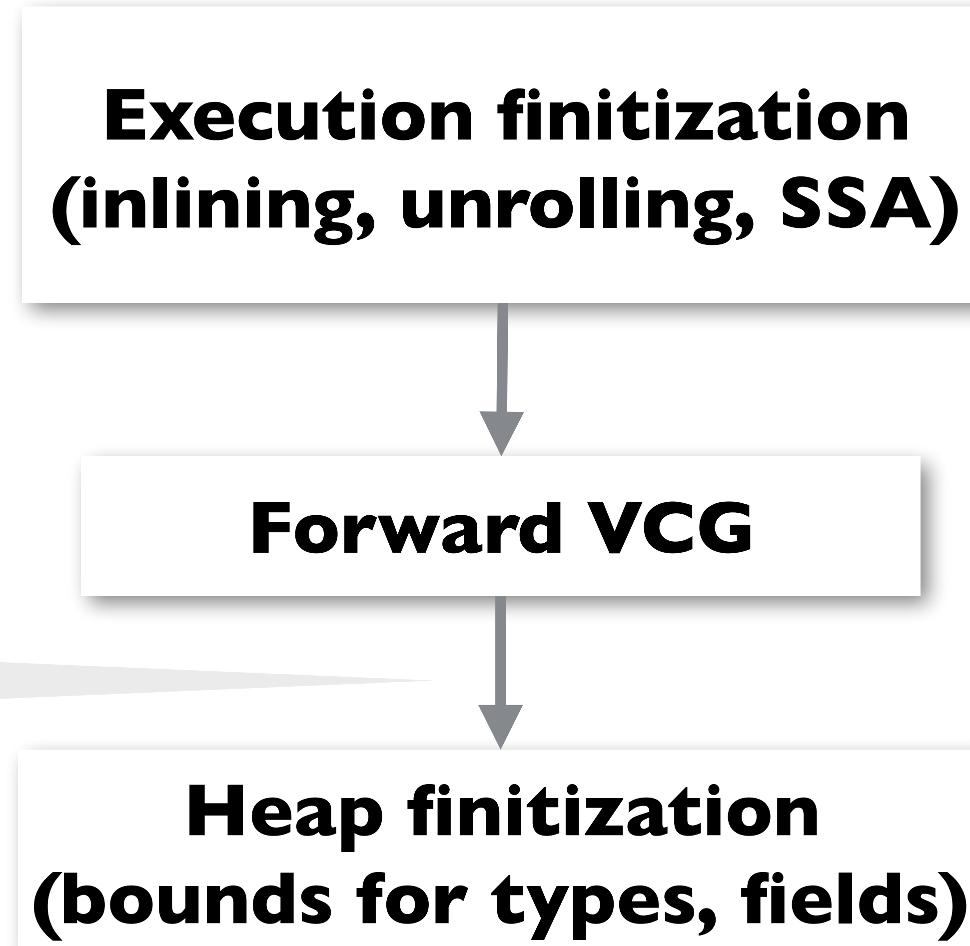
this ⊆ List ∧ **one** this ∧
head ⊆ List ↦ (Node ∪ null) ∧
next ⊆ Node ↦ (Node ∪ null) ∧
data ⊆ Node ↦ (String ∪ null) ∧

let $near_0$ = this.head,
mid_0 = $near_0$.next,
far_0 = mid_0.next,

$next_0$ = next ++ ($near_0$ × far_0),
guard = (far_0 != null),
$next_1$ = $next_0$ ++ (mid_0 × $near_0$),
$near_1$ = mid_0,
mid_1 = far_0,
far_1 = far_0.$next_1$,

$near_2$ = **if** guard **then** $near_1$ **else** $near_0$,
mid_2 = **if** guard **then** mid_1 **else** mid_0,
far_2 = **if** guard **then** far_1 **else** far_0,
$next_2$ = **if** guard **then** $next_1$ **else** $next_0$,
$next_3$ = $next_2$ ++ (mid_2 × $near_2$)
$head_0$ = head ++ (this × mid_2) **|**

far_2 = null ∧ Inv(next) ∧ Pre(this, head, next) ∧
¬ (Inv($next_3$) ∧ Post(this, head, $head_0$, next, $next_3$))



Bounded verification: step 3/4

$this \subseteq List \wedge$ **one** $this \wedge$
$head \subseteq List \mapsto (Node \cup null) \wedge$
$next \subseteq Node \mapsto (Node \cup null) \wedge$
$data \subseteq Node \mapsto (String \cup null) \wedge$

let $near_0 = this.head,$
$mid_0 = near_0.next,$
$far_0 = mid_0.next,$

$next_0 = next \mathbin{++} (near_0 \times far_0),$
$guard = (far_0 \mathrel{!=} null),$
$next_1 = next_0 \mathbin{++} (mid_0 \times near_0),$
$near_1 = mid_0,$
$mid_1 = far_0,$
$far_1 = far_0.next_1,$

$near_2 =$ **if** guard **then** $near_1$ **else** $near_0,$
$mid_2 =$ **if** guard **then** mid_1 **else** $mid_0,$
$far_2 =$ **if** guard **then** far_1 **else** $far_0,$
$next_2 =$ **if** guard **then** $next_1$ **else** $next_0,$
$next_3 = next_2 \mathbin{++} (mid_2 \times near_2)$
$head_0 = head \mathbin{++} (this \times mid_2)$ **|**

$far_2 = null \wedge Inv(next) \wedge Pre(this, head, next) \wedge$
$\neg (Inv(next_3) \wedge Post(this, head, head_0, next, next_3))$

$\{$ **this**, **n0**, **n1**, **n2**, **s0**, **s1**, **s2**, **null** $\}$

$\{ \langle \mathbf{null} \rangle \} \subseteq null \subseteq \{ \langle \mathbf{null} \rangle \}$

$\{\} \subseteq this \subseteq \{ \langle \mathbf{this} \rangle \}$
$\{\} \subseteq List \subseteq \{ \langle \mathbf{this} \rangle \}$
$\{\} \subseteq Node \subseteq \{ \langle \mathbf{n0} \rangle, \langle \mathbf{n1} \rangle, \langle \mathbf{n2} \rangle \}$
$\{\} \subseteq String \subseteq \{ \langle \mathbf{s0} \rangle, \langle \mathbf{s1} \rangle, \langle \mathbf{s2} \rangle \}$

$\{\} \subseteq head \subseteq \{ \mathbf{this} \} \times \{ \mathbf{n0}, \mathbf{n1}, \mathbf{n2}, \mathbf{null} \}$
$\{\} \subseteq next \subseteq \{ \mathbf{n0}, \mathbf{n1}, \mathbf{n2} \} \times \{ \mathbf{n0}, \mathbf{n1}, \mathbf{n2}, \mathbf{null} \}$
$\{\} \subseteq data \subseteq \{ \mathbf{n0}, \mathbf{n1}, \mathbf{n2} \} \times \{ \mathbf{s0}, \mathbf{s1}, \mathbf{s2}, \mathbf{null} \}$

Bounded verification: step 3/4

this ⊆ List ∧ **one** this ∧
head ⊆ List ↦ (Node ∪ null) ∧
next ⊆ Node ↦ (Node ∪ null) ∧
data ⊆ Node ↦ (String ∪ null) ∧

let $near_0$ = this.head,
mid_0 = $near_0$.next,
far_0 = mid_0.next,

$next_0$ = next ++ ($near_0$ × far_0),
guard = (far_0 != null),
$next_1$ = $next_0$ ++ (mid_0 × $near_0$),
$near_1$ = mid_0,
mid_1 = far_0,
far_1 = far_0.$next_1$,

$near_2$ = **if** guard **then** $near_1$ **else** $near_0$,
mid_2 = **if** guard **then** mid_1 **else** mid_0,
far_2 = **if** guard **then** far_1 **else** far_0,
$next_2$ = **if** guard **then** $next_1$ **else** $next_0$,
$next_3$ = $next_2$ ++ (mid_2 × $near_2$)
$head_0$ = head ++ (this × mid_2) |

far_2 = null ∧ Inv(next) ∧ Pre(this, head, next) ∧
¬ (Inv($next_3$) ∧ Post(this, head, $head_0$, next, $next_3$))

Finite **universe** of uninterpreted symbols.

{ **this**, **n0**, **n1**, **n2**, **s0**, **s1**, **s2**, **null** }

{ ⟨**null**⟩ } ⊆ null ⊆ { ⟨**null**⟩ }

{} ⊆ this ⊆ { ⟨ **this** ⟩ }
{} ⊆ List ⊆ { ⟨ **this** ⟩ }
{} ⊆ Node ⊆ { ⟨**n0**⟩, ⟨**n1**⟩, ⟨**n2**⟩ }
{} ⊆ String ⊆ { ⟨**s0**⟩, ⟨**s1**⟩, ⟨**s2**⟩ }

{} ⊆ head ⊆ { **this** } × { **n0**, **n1**, **n2**, **null** }
{} ⊆ next ⊆ { **n0**, **n1**, **n2** } × { **n0**, **n1**, **n2**, **null** }
{} ⊆ data ⊆ { **n0**, **n1**, **n2** } × { **s0**, **s1**, **s2**, **null** }

Bounded verification: step 3/4

this ⊆ List ∧ **one** this ∧
head ⊆ List ↦ (Node ∪ null) ∧
next ⊆ Node ↦ (Node ∪ null) ∧
data ⊆ Node ↦ (String ∪ null) ∧

let $near_0$ = this.head,
mid_0 = $near_0$.next,
far_0 = mid_0.next,

$next_0$ = next ++ ($near_0$ × far_0),
guard = (far_0 != null),
$next_1$ = $next_0$ ++ (mid_0 × $near_0$),
$near_1$ = mid_0,
mid_1 = far_0,
far_1 = far_0.$next_1$,

$near_2$ = **if** guard **then** $near_1$ **else** $near_0$,
mid_2 = **if** guard **then** mid_1 **else** mid_0,
far_2 = **if** guard **then** far_1 **else** far_0,
$next_2$ = **if** guard **then** $next_1$ **else** $next_0$,
$next_3$ = $next_2$ ++ (mid_2 × $near_2$)
$head_0$ = head ++ (this × mid_2) **|**

far_2 = null ∧ Inv(next) ∧ Pre(this, head, next) ∧
¬ (Inv($next_3$) ∧ Post(this, head, $head_0$, next, $next_3$))

Finite **universe** of uninterpreted symbols.

{ **this**, **n0**, **n1**, **n2**, **s0**, **s1**, **s2**, **null** }

{ ⟨**null**⟩ } ⊆ null ⊆ { ⟨**null**⟩ }

{} ⊆ this ⊆ { ⟨ **this** ⟩ }
{} ⊆ List ⊆ { ⟨ **this** ⟩ }
{} ⊆ Node ⊆ { ⟨**n0**⟩, ⟨**n1**⟩, ⟨**n2**⟩ }
{} ⊆ String ⊆ { ⟨**s0**⟩, ⟨**s1**⟩, ⟨**s2**⟩ }

{} ⊆ head ⊆ { **this** } × { **n0**, **n1**, **n2**, **null** }
{} ⊆ next ⊆ { **n0**, **n1**, **n2** } × { **n0**, **n1**, **n2**, **null** }
{} ⊆ data ⊆ { **n0**, **n1**, **n2** } × { **s0**, **s1**, **s2**, **null** }

Upper bound on each relation: tuples it *may* contain.

Bounded verification: step 3/4

this ⊆ List ∧ **one** this ∧
head ⊆ List ↦ (Node ∪ null) ∧
next ⊆ Node ↦ (Node ∪ null) ∧
data ⊆ Node ↦ (String ∪ null) ∧

let $near_0$ = this.head,
mid_0 = $near_0$.next,
far_0 = mid_0.next,

$next_0$ = next ++ ($near_0$ × far_0),
guard = (far_0 != null),
$next_1$ = $next_0$ ++ (mid_0 × $near_0$),
$near_1$ = mid_0,
mid_1 = far_0,
far_1 = far_0.$next_1$,

$near_2$ = **if** guard **then** $near_1$ **else** $near_0$,
mid_2 = **if** guard **then** mid_1 **else** mid_0,
far_2 = **if** guard **then** far_1 **else** far_0,
$next_2$ = **if** guard **then** $next_1$ **else** $next_0$,
$next_3$ = $next_2$ ++ (mid_2 × $near_2$)
$head_0$ = head ++ (this × mid_2) **|**

far_2 = null ∧ Inv(next) ∧ Pre(this, head, next) ∧
¬ (Inv($next_3$) ∧ Post(this, head, $head_0$, next, $next_3$))

Finite **universe** of uninterpreted symbols.

{ **this**, **n0**, **n1**, **n2**, **s0**, **s1**, **s2**, **null** }

{ ⟨**null**⟩ } ⊆ null ⊆ { ⟨**null**⟩ }

{} ⊆ this ⊆ { ⟨ **this** ⟩ }
{} ⊆ List ⊆ { ⟨ **this** ⟩ }
{} ⊆ Node ⊆ { ⟨**n0**⟩, ⟨**n1**⟩, ⟨**n2**⟩ }
{} ⊆ String ⊆ { ⟨**s0**⟩, ⟨**s1**⟩, ⟨**s2**⟩ }

{} ⊆ head ⊆ { **this** } × { **n0**, **n1**, **n2**, **null** }
{} ⊆ next ⊆ { **n0**, **n1**, **n2** } × { **n0**, **n1**, **n2**, **null** }
{} ⊆ data ⊆ { **n0**, **n1**, **n2** } × { **s0**, **s1**, **s2**, **null** }

Lower bound on each relation: tuples it *must* contain.

Upper bound on each relation: tuples it *may* contain.

Bounded verification: step 4/4

this ⊆ List ∧ **one** this ∧
head ⊆ List ↦ (Node ∪ null) ∧
next ⊆ Node ↦ (Node ∪ null) ∧
data ⊆ Node ↦ (String ∪ null) ∧

let ne
mi
far
ne
gu
ne
ne
mi
far
ne
mi
far
ne
$next_3$ = $next_2$ ++ (mid_2 × $near_2$)
$head_0$ = head ++ (this × mid_2) |

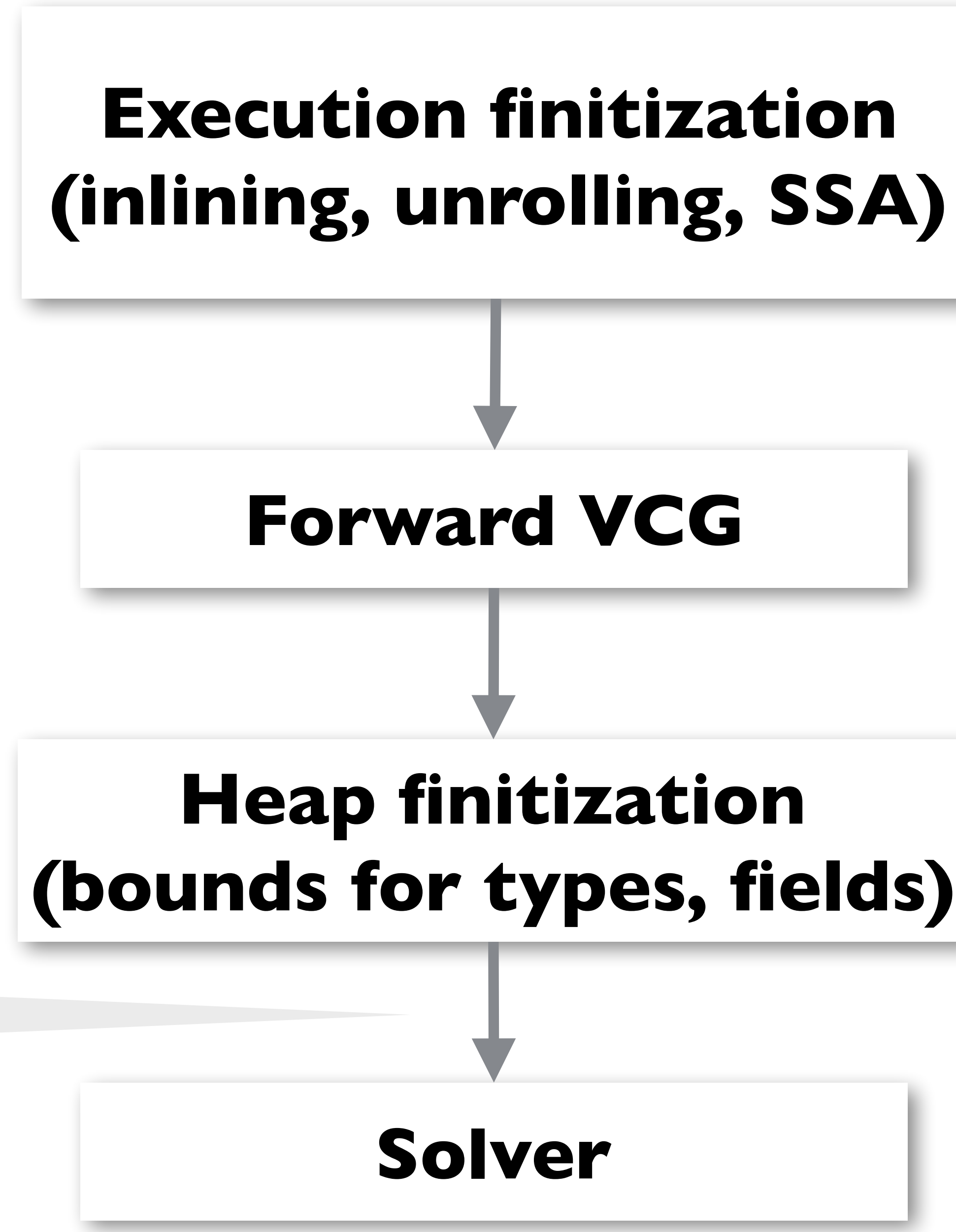
far_2 = null ∧ Inv(next) ∧ Pre(this, head, next) ∧
¬ (Inv($next_3$) ∧ Post(this, head, $head_0$, next, $next_3$))

{ **this**, **n0**, **n1**, **n2**, **s0**, **s1**, **s2**, **null** }

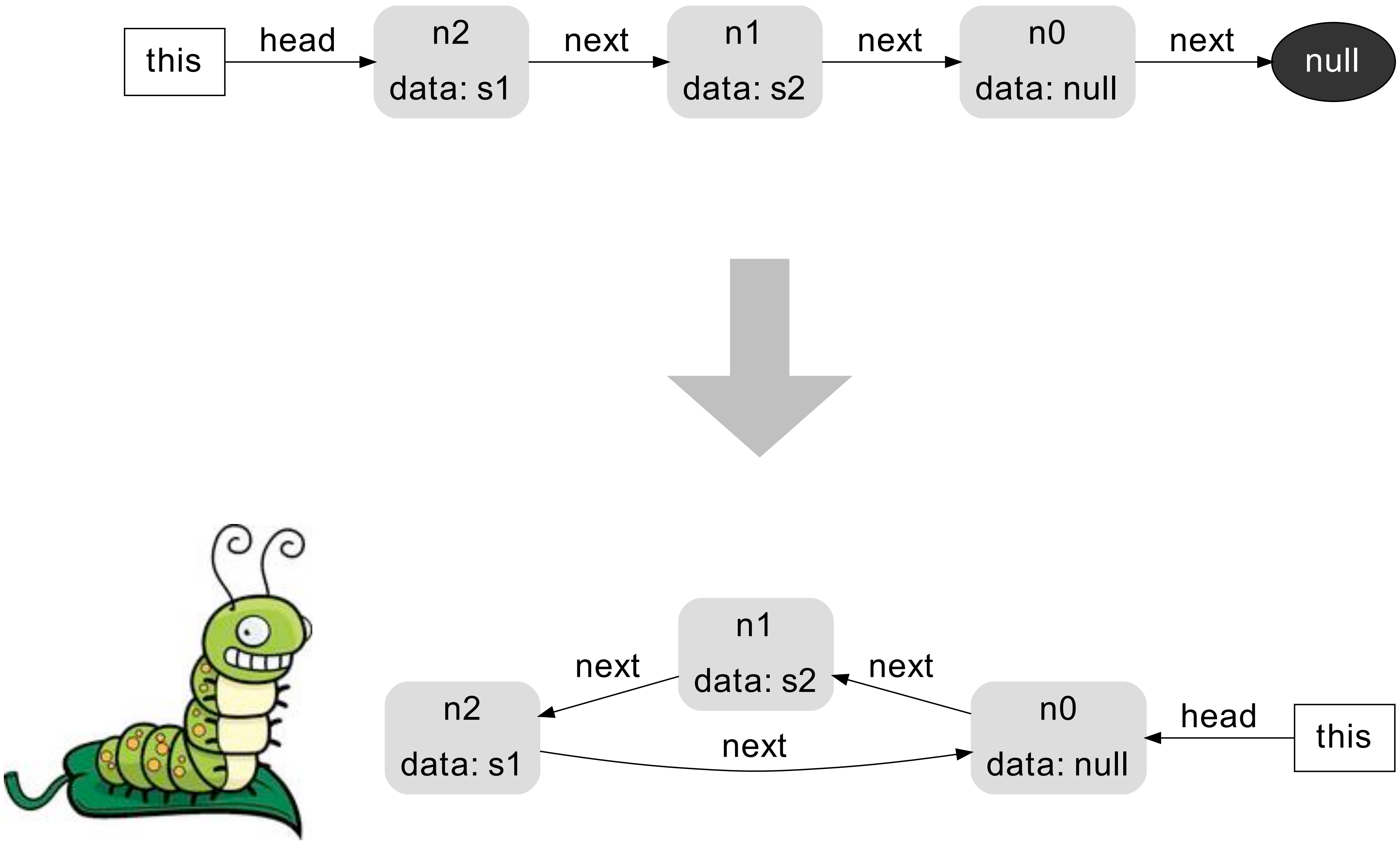
{ ⟨**null**⟩ } ⊆ null ⊆ { ⟨**null**⟩ }

{} ⊆ this ⊆ { ⟨ **this** ⟩ }
{} ⊆ List ⊆ { ⟨ **this** ⟩ }
{} ⊆ Node ⊆ { ⟨**n0**⟩, ⟨**n1**⟩, ⟨**n2**⟩ }
{} ⊆ String ⊆ { ⟨**s0**⟩, ⟨**s1**⟩, ⟨**s2**⟩ }

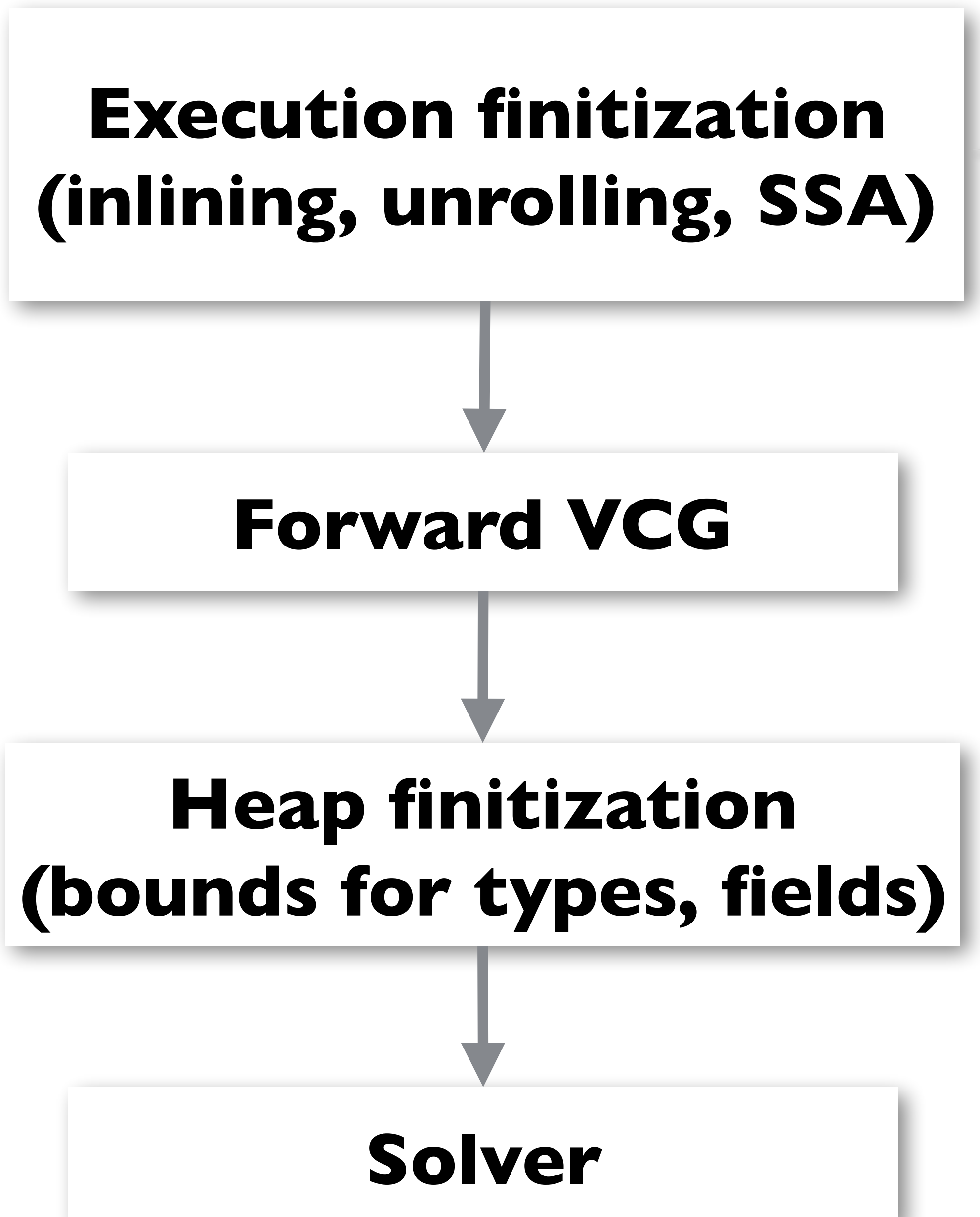
{} ⊆ head ⊆ { **this** } × { **n0**, **n1**, **n2**, **null** }
{} ⊆ next ⊆ { **n0**, **n1**, **n2** } × { **n0**, **n1**, **n2**, **null** }
{} ⊆ data ⊆ { **n0**, **n1**, **n2** } × { **s0**, **s1**, **s2**, **null** }



Bounded verification: counterexample



Bounded verification: optimization



Finitized program after inlining may be huge.

Full inlining is rarely needed to check partial correctness.

Optimization: Counterexample-Guided Abstraction Refinement with Unsatisfiable Cores [Taghdiri, 2004]

From bounded verification to fault localization

```
@invariant Inv(next)
@requires Pre(this, head, next)
@ensures Post(this, old(head), head, old(next), next)

void reverse() {
   Node near0 = this.head;
   Node mid0 = near0.next;
   Node far0 = mid0.next;

   next0 = update(next, near0, far0);
   boolean guard = (far0 != null);
   next1 = update(next0, mid0, near0);
   near1 = mid0;
   mid1 = far0;
   far1 = far0.next1;

   near2 = phi(guard, near1, near0);
   mid2 = phi(guard, mid1, mid0);
   far2 = phi(guard, far1, far0);
   next2 = phi(guard, next1, next0);

   assume far2 == null;

   next3 = update(next2, mid2, near2);
   head0 = update(head, this, mid2);
}
```

From bounded verification to fault localization

@invariant Inv(next)
@requires Pre(this, head, next)
@ensures Post(this, old(head), head, old(next), next)

```
void reverse() {
   Node near₀ = this.head;
   Node mid₀ = near₀.next;
   Node far₀ = mid₀.next;

   next₀ = update(next, near₀, far₀);
   boolean guard = (far₀ != null);
   next₁ = update(next₀, mid₀, near₀);
   near₁ = mid₀;
   mid₁ = far₀;
   far₁ = far₀.next₁;

   near₂ = phi(guard, near₁, near₀);
   mid₂ = phi(guard, mid₁, mid₀);
   far₂ = phi(guard, far₁, far₀);
   next₂ = phi(guard, next₁, next₀);

   assume far₂ == null;

   next₃ = update(next₂, mid₂, near₂);
   head₀ = update(head, this, mid₂);
}
```

Given a buggy program and a failure-triggering input, find a minimal subset of program statements that prevents the execution on the given input from producing a correct output.

From bounded verification to fault localization

```
@invariant Inv(next)
@requires Pre(this, head, next)
@ensures Post(this, old(head), head, old(next), next)

void reverse() {
   Node near0 = this.head;
   Node mid0 = near0.next;
   Node far0 = mid0.next;

   next0 = update(next, near0, far0);
   boolean guard = (far0 != null);
   next1 = update(next0, mid0, near0);
   near1 = mid0;
   mid1 = far0;
   far1 = far0.next1;

   near2 = phi(guard, near1, near0);
   mid2 = phi(guard, mid1, mid0);
   far2 = phi(guard, far1, far0);
   next2 = phi(guard, next1, next0);

   assume far2 == null;

   next3 = update(next2, mid2, near2);
   head0 = update(head, this, mid2);
}
```

Given a buggy program and a failure-triggering input, find a minimal subset of program statements that prevents the execution on the given input from producing a correct output.

Introduce additional "indicator" relations into the encoding.

From bounded verification to fault localization

```
@invariant Inv(next)
@requires Pre(this, head, next)
@ensures Post(this, old(head), head, old(next), next)

void reverse() {
  Node near0 = this.head;
  Node mid0 = near0.next;
  Node far0 = mid0.next;

  next0 = update(next, near0, far0);
  boolean guard = (far0 != null);
  next1 = update(next0, mid0, near0);
  near1 = mid0;
  mid1 = far0;
  far1 = far0.next1;

  near2 = phi(guard, near1, near0);
  mid2 = phi(guard, mid1, mid0);
  far2 = phi(guard, far1, far0);
  next2 = phi(guard, next1, next0);

  assume far2 == null;

  next3 = update(next2, mid2, near2);
  head0 = update(head, this, mid2);
}
```

Given a buggy program and a failure-triggering input, find a minimal subset of program statements that prevents the execution on the given input from producing a correct output.

Introduce additional "indicator" relations into the encoding.

The resulting formula, together with the input partial model, is unsatisfiable.

From bounded verification to fault localization

```
@invariant Inv(next)
@requires Pre(this, head, next)
@ensures Post(this, old(head), head, old(next), next)

void reverse() {
  Node near₀ = this.head;
  Node mid₀ = near₀.next;
  Node far₀ = mid₀.next;

  next₀ = update(next, near₀, far₀);
  boolean guard = (far₀ != null);
  next₁ = update(next₀, mid₀, near₀);
  near₁ = mid₀;
  mid₁ = far₀;
  far₁ = far₀.next₁;

  near₂ = phi(guard, near₁, near₀);
  mid₂ = phi(guard, mid₁, mid₀);
  far₂ = phi(guard, far₁, far₀);
  next₂ = phi(guard, next₁, next₀);

  assume far₂ == null;

  next₃ = update(next₂, mid₂, near₂);
  head₀ = update(head, this, mid₂);
}
```

Given a buggy program and a failure-triggering input, find a minimal subset of program statements that prevents the execution on the given input from producing a correct output.

Introduce additional "indicator" relations into the encoding.

The resulting formula, together with the input partial model, is unsatisfiable.

A minimal unsatisfiable core of this formula represents an irreducible cause of the program's failure to meet the specification.

Fault localization: encoding

```
@invariant Inv(next)
@requires Pre(this, head, next)
@ensures Post(this, old(head), head, old(next), next)

void reverse() {
  Node near0 = this.head;
  Node mid0 = n
  Node far0 = m

  next0 = updat
  boolean guar
  next1 = update(next0, mid0, near0);
  near1 = mid0;
  mid1 = far0;
  far1 = far0.next1;

  near2 = phi(guard, near1, near0);
  mid2 = phi(guard, mid1, mid0);
  far2 = phi(guard, far1, far0);
  next2 = phi(guard, next1, next0);

  assume far2 == null;

  next3 = update(next2, mid2, near2);
  head0 = update(head, this, mid2);
}
```

Start with the encoding for bounded verification.

$this \subseteq List \wedge \textbf{one}\ this \wedge$

$head \subseteq List \mapsto (Node \cup null) \wedge$

$next \subseteq Node \mapsto (Node \cup null) \wedge$

$data \subseteq Node \mapsto (String \cup null) \wedge$

$\textbf{let}\ near_0 = this.head,$
$mid_0 = near_0.next,$
$far_0 = mid_0.next,$

$next_0 = next \mathbin{++} (near_0 \times far_0),$
$guard = (far_0 \mathrel{!=} null),$
$next_1 = next_0 \mathbin{++} (mid_0 \times near_0),$
$near_1 = mid_0,$
$mid_1 = far_0,$
$far_1 = far_0.next_1,$

$near_2 = \textbf{if}\ guard\ \textbf{then}\ near_1\ \textbf{else}\ near_0,$
$mid_2 = \textbf{if}\ guard\ \textbf{then}\ mid_1\ \textbf{else}\ mid_0,$
$far_2 = \textbf{if}\ guard\ \textbf{then}\ far_1\ \textbf{else}\ far_0,$
$next_2 = \textbf{if}\ guard\ \textbf{then}\ next_1\ \textbf{else}\ next_0,$
$next_3 = next_2 \mathbin{++} (mid_2 \times near_2)$
$head_0 = head \mathbin{++} (this \times mid_2)\ \mid$

$far_2 = null \wedge Inv(next) \wedge Pre(this, head, next) \wedge$
$\neg (Inv(next_3) \wedge Post(this, head, head_0, next, next_3))$

Fault localization: encoding

```
@invariant Inv(next)
@requires Pre(this, head, next)
@ensures Post(this, old(head), head, old(next), next)

void reverse() {
  Node near0 = this.head;
  Node mid0 = near0.next;
  Node far0 = mid0.next;

  next0 = update(next, near0, far0);
  boolean guard = (far0 != null);
  next1 = update(next0, mid0, near0);
  near1 = mid0;
  mid1 = far0;
  far1 = far0.n

  near2 = phi(g
  mid2 = phi(gu
  far2 = phi(guard, far1, far0);
  next2 = phi(guard, next1, next0);

  assume far2 == null;

  next3 = update(next2, mid2, near2);
  head0 = update(head, this, mid2);
}
```

Introduce fresh relations for source-level expressions.

$\text{this} \subseteq \text{List} \wedge \textbf{one}\ \text{this} \wedge$
$\text{head} \subseteq \text{List} \mapsto (\text{Node} \cup \text{null}) \wedge$
$\text{next} \subseteq \text{Node} \mapsto (\text{Node} \cup \text{null}) \wedge$
$\text{data} \subseteq \text{Node} \mapsto (\text{String} \cup \text{null}) \wedge$

$\text{near}_0 = \text{this.head} \wedge$
$\text{mid}_0 = \text{near}_0\text{.next} \wedge$
$\text{far}_0 = \text{mid}_0\text{.next} \wedge$

$\text{next}_0 = \text{next} \mathbin{++} (\text{near}_0 \times \text{far}_0) \wedge$
$\text{next}_1 = \text{next}_0 \mathbin{++} (\text{mid}_0 \times \text{near}_0) \wedge$
$\text{near}_1 = \text{mid}_0 \wedge$
$\text{mid}_1 = \text{far}_0 \wedge$
$\text{far}_1 = \text{far}_0\text{.next}_1 \wedge$

$\textbf{let}\ \text{guard} = (\text{far}_0\ \text{!=}\ \text{null}),$
$\quad \text{near}_2 = \textbf{if}\ \text{guard}\ \textbf{then}\ \text{near}_1\ \textbf{else}\ \text{near}_0,$
$\quad \text{mid}_2 = \textbf{if}\ \text{guard}\ \textbf{then}\ \text{mid}_1\ \textbf{else}\ \text{mid}_0,$
$\quad \text{far}_2 = \textbf{if}\ \text{guard}\ \textbf{then}\ \text{far}_1\ \textbf{else}\ \text{far}_0,$
$\quad \text{next}_2 = \textbf{if}\ \text{guard}\ \textbf{then}\ \text{next}_1\ \textbf{else}\ \text{next}_0\ |$

$\quad \text{next}_3 = \text{next}_2 \mathbin{++} (\text{mid}_2 \times \text{near}_2) \wedge$
$\quad \text{head}_0 = \text{head} \mathbin{++} (\text{this} \times \text{mid}_2) \wedge$
$\quad \text{far}_2 = \text{null} \wedge \text{Inv}(\text{next}) \wedge \text{Pre}(\text{this}, \text{head}, \text{next}) \wedge$
$\quad \text{Inv}(\text{next}_3) \wedge \text{Post}(\text{this}, \text{head}, \text{head}_0, \text{next}, \text{next}_3)$

Fault localization: bounds

$this \subseteq List \wedge$ **one** $this \wedge$
$head \subseteq List \mapsto (Node \cup null) \wedge$
$next \subseteq Node \mapsto (Node \cup null) \wedge$
$data \subseteq Node \mapsto (String \cup null) \wedge$

$near_0 = this.head \wedge$
$mid_0 = near_0.next \wedge$
$far_0 = mid_0.next \wedge$

$next_0 = next \mathbin{++} (near_0 \times far_0) \wedge$
$next_1 = next_0 \mathbin{++} (mid_0 \times near_0) \wedge$
$near_1 = mid_0 \wedge$
$mid_1 = far_0 \wedge$
$far_1 = far_0.next_1 \wedge$

let guard = (far_0 != null),
  $near_2$ = **if** guard **then** $near_1$ **else** $near_0$,
  mid_2 = **if** guard **then** mid_1 **else** mid_0,
  far_2 = **if** guard **then** far_1 **else** far_0,
  $next_2$ = **if** guard **then** $next_1$ **else** $next_0$ |

  $next_3 = next_2 \mathbin{++} (mid_2 \times near_2) \wedge$
  $head_0 = head \mathbin{++} (this \times mid_2) \wedge$
  $far_2 = null \wedge Inv(next) \wedge Pre(this, head, next) \wedge$
  $Inv(next_3) \wedge Post(this, head, head_0, next, next_3)$

Input expressed as a partial model.

{ **this**, **n0**, **n1**, **n2**, **s0**, **s1**, **s2**, **null** }

null = { <**null**> }
this = { <**this**> }
List = { <**this**> }
Node = { <**n0**>, <**n1**>, <**n2**> }
String = { <**s1**>, <**s2**> }

head = { <**this**, **n2**> }
next = { <**n2**, **n1**>, <**n1**, **n0**>, <**n0**, **null**> }
data = { <**n2**, **s1**>, <**n1**, **s2**>, <**n0**, **null**> }

$\{\} \subseteq head_0 \subseteq \{ this \} \times \{ n0, n1, n2, null \}$
$\{\} \subseteq next_0 \subseteq \{ n0, n1, n2 \} \times \{ n0, n1, n2, null \}$
$\{\} \subseteq next_1 \subseteq \{ n0, n1, n2 \} \times \{ n0, n1, n2, null \}$
$\{\} \subseteq next_3 \subseteq \{ n0, n1, n2 \} \times \{ n0, n1, n2, null \}$
$\{\} \subseteq near_0 \subseteq \{ n0, n1, n2, null \}$
$\{\} \subseteq near_1 \subseteq \{ n0, n1, n2, null \}$
$\{\} \subseteq mid_0 \subseteq \{ n0, n1, n2, null \}$
$\{\} \subseteq mid_1 \subseteq \{ n0, n1, n2, null \}$
$\{\} \subseteq far_0 \subseteq \{ n0, n1, n2, null \}$
$\{\} \subseteq far_1 \subseteq \{ n0, n1, n2, null \}$

Fault localization: minimal unsat core

$this \subseteq List \wedge$ **one** $this \wedge$
$head \subseteq List \mapsto (Node \cup null) \wedge$
$next \subseteq Node \mapsto (Node \cup null) \wedge$
$data \subseteq Node \mapsto (String \cup null) \wedge$

$near_0 = this.head \wedge$
$mid_0 = near_0.next \wedge$
$far_0 = mid_0.next \wedge$

$next_0 = next \mathbin{++} (near_0 \times far_0) \wedge$
$next_1 = next_0 \mathbin{++} (mid_0 \times near_0) \wedge$
$near_1 = mid_0 \wedge$
$mid_1 = far_0 \wedge$
$far_1 = far_0.next_1 \wedge$

let guard = (far_0 != null),
  $near_2$ = **if** guard **then** $near_1$ **else** $near_0$,
  mid_2 = **if** guard **then** mid_1 **else** mid_0,
  far_2 = **if** guard **then** far_1 **else** far_0,
  $next_2$ = **if** guard **then** $next_1$ **else** $next_0$ |

  $next_3 = next_2 \mathbin{++} (mid_2 \times near_2) \wedge$
  $head_0 = head \mathbin{++} (this \times mid_2) \wedge$
  $far_2 = null \wedge Inv(next) \wedge Pre(this, head, next) \wedge$
  $Inv(next_3) \wedge Post(this, head, head_0, next, next_3)$

{ **this**, **n0**, **n1**, **n2**, **s0**, **s1**, **s2**, **null** }

null = { <**null**> }
this = { <**this**> }
List = { <**this**> }
Node = { <**n0**>, <**n1**>, <**n2**> }
String = { <**s1**>, <**s2**> }

head = { <**this**, **n2**> }
next = { <**n2**, **n1**>, <**n1**, **n0**>, <**n0**, **null**> }
data = { <**n2**, **s1**>, <**n1**, **s2**>, <**n0**, **null**> }

$\{\} \subseteq head_0 \subseteq$ { **this** } × { **n0**, **n1**, **n2**, **null** }
$\{\} \subseteq next_0 \subseteq$ { **n0**, **n1**, **n2** } × { **n0**, **n1**, **n2**, **null** }
$\{\} \subseteq next_1 \subseteq$ { **n0**, **n1**, **n2** } × { **n0**, **n1**, **n2**, **null** }
$\{\} \subseteq next_3 \subseteq$ { **n0**, **n1**, **n2** } × { **n0**, **n1**, **n2**, **null** }
$\{\} \subseteq near_0 \subseteq$ { **n0**, **n1**, **n2**, **null** }
$\{\} \subseteq near_1 \subseteq$ { **n0**, **n1**, **n2**, **null** }
$\{\} \subseteq mid_0 \subseteq$ { **n0**, **n1**, **n2**, **null** }
$\{\} \subseteq mid_1 \subseteq$ { **n0**, **n1**, **n2**, **null** }
$\{\} \subseteq far_0 \subseteq$ { **n0**, **n1**, **n2**, **null** }
$\{\} \subseteq far_1 \subseteq$ { **n0**, **n1**, **n2**, **null** }

Fault localization: minimal unsat core

```
@invariant Inv(next)
@requires Pre(this, head, next)
@ensures Post(this, old(head), head, old(next), next)

void reverse() {
   Node near0 = this.head;
   Node mid0 = near0.next;
   Node far0 = mid0.next;

   next0 = update(next, near0, far0);
   boolean guard = (far0 != null);
   next1 = update(next0, mid0, near0);
   near1 = mid0;
   mid1 = far0;
   far1 = far0.next1;

   near2 = phi(guard, near1, near0);
   mid2 = phi(guard, mid1, mid0);
   far2 = phi(guard, far1, far0);
   next2 = phi(guard, next1, next0);

   assume far2 == null;

   next3 = update(next2, mid2, near2);
   head0 = update(head, this, mid2);
}
```

Fault localization: minimal unsat core

```
@invariant Inv(next)
@requires Pre(this, head, next)
@ensures Post(this, old(head), head, old(next), next)

void reverse() {
   Node near = head;
   Node mid = near.next;
   Node far = mid.next;

   near.next = far;
   while (far != null) {
      mid.next = near;
      near = mid;
      mid = far;
      far = far.next;
   }

   mid.next = near;
   head = mid;
}
```

Summary

Today

- Bounded verification
  - A relational model of the heap
  - CEGAR with unsat cores
  - Fault localization

Next lecture

- Symbolic execution and concolic testing