Bounded Verification

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.
The spectrum of program verification tools

- Static Analysis
- Verification
- Bounded Verification & Symbolic Execution
- Extended Static Checking
- Concolic Testing & Whitebox Fuzzing
- Ad-hoc Testing

Confidence vs. Cost (programmer effort, time, expertise)
The spectrum of program verification tools

E.g., Dafny, Coq, Leon:
- support for rich (FOL+) correctness properties
- high annotation overhead (pre/post conditions, loop invariants, etc.)
- total correctness

Confidence

Cost (programmer effort, time, expertise)
The spectrum of program verification tools

E.g., Astree:
• small set of fixed properties (e.g., “no null dereferences”)
• no annotations but must deal with false positives
• no false negatives
The spectrum of program verification tools

Confidence

Cost (programmer effort, time, expertise)

E.g., Calysto, Saturn:
• user-defined assertions supported but optional
• no annotations
• some/low false positives
• false negatives
The spectrum of program verification tools

- CBMC, Miniatur, Forge, TACO, JPF, Klee:
  - optional user-defined harnesses, assertions, and/or FOL+ properties
  - no/low annotations
  - no/low false positives
  - false negatives

E.g., CBMC, Miniatur, Forge, TACO, JPF, Klee:
- optional user-defined harnesses, assertions, and/or FOL+ properties
- no/low annotations
- no/low false positives
- false negatives

Cost (programmer effort, time, expertise) vs. Confidence

- Bounded Verification & Symbolic Execution
- Concolic Testing & Whitebox Fuzzing
- Ad-hoc Testing

Static Analysis
Verification

Confidence
The spectrum of program verification tools

- **E.g., SAGE, Pex, CUTE, DART:**
  - test harnesses and/or user-defined assertions
  - no annotations
  - no false positives
  - false negatives

- **Bounded Verification & Symbolic Execution**
- **Concolic Testing & Whitebox Fuzzing**
- **Ad-hoc Testing**

Cost (programmer effort, time, expertise)

Confidence

Static Analysis

Verification
The spectrum of program verification tools

- Ad-hoc Testing
- Concolic Testing & Whitebox Fuzzing
- Bounded Verification & Symbolic Execution
- Extended Static Checking
- Static Analysis
- Verification

Cost (programmer effort, time, expertise)

Confidence
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

```java
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

```java
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;
}
Specifying contracts: preconditions

```java
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

```java
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 \in 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)\times null ++
    ~(old(next) \cap N\times N)
**Specifying contracts: postconditions**

```java
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 \texttt{inv}(next)
@requires \texttt{Pre(this, head, next)}
@ensures \texttt{Post(this, old(head), head, old(next), next)}

```java
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;
}
```

![Diagram of data structures](image-url)
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;
}

•\hspace{0.4cm}\textbf{Fields as binary relations}
  • head : \{ \langle \text{this}, n2 \rangle \}, next : \{ \langle n2, n1 \rangle, \ldots \}
A relational model of memory (heap)

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

```java
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;
}
A relational model of memory (heap)

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

```java
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⟩ }`

---

![Diagram of linked list](image-url)
A relational model of memory (heap)

@invariant \texttt{Inv(next)}
@requires \texttt{Pre(this, head, next)}
@ensures \texttt{Post(this, old(head), head, old(next), next)}

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

    near.next = far;
    while (far != \texttt{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⟩ \}

![Diagram of a linked list with nodes data: s1, s2, and null](image-url)
Bounded verification: step 1/4

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

```java
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)

```java
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)

```java
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 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);
}
```
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);
}

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

let near₀ = this.head,
    mid₀ = near₀.next,
    far₀ = mid₀.next,

    next₀ = next ++ (near₀ × far₀),
    guard = (far₀ != null),
    next₁ = next₀ ++ (mid₀ × near₀),
    near₁ = mid₀,
    mid₁ = far₀,
    far₁ = far₀.next₁,

    near₂ = if guard then near₁ else near₀,
    mid₂ = if guard then mid₁ else mid₀,
    far₂ = if guard then far₁ else far₀,
    next₂ = if guard then next₁ else next₀,
    next₃ = next₂ ++ (mid₂ × near₂)

    head₀ = head ++ (this × mid₂) |

    far₂ = null ∧ Inv(next) ∧ Pre(this, head, next) ∧
    ¬ (Inv(next₃) ∧ Post(this, head, head₀, next, next₃))
Bounded verification: step 3/4

\[
\begin{align*}
\text{this} & \subseteq \text{List} \land \text{one this} \land \\
\text{head} & \subseteq \text{List} \mapsto (\text{Node} \cup \text{null}) \land \\
\text{next} & \subseteq \text{Node} \mapsto (\text{Node} \cup \text{null}) \land \\
\text{data} & \subseteq \text{Node} \mapsto (\text{String} \cup \text{null}) \land \\
\text{let} & \text{near}_0 = \text{this} . \text{head}, \\
& \quad \text{mid}_0 = \text{near}_0 . \text{next}, \\
& \quad \text{far}_0 = \text{mid}_0 . \text{next}, \\
& \quad \text{next}_0 = \text{next} ++ (\text{near}_0 \times \text{far}_0), \\
& \quad \text{guard} = (\text{far}_0 \neq \text{null}), \\
& \quad \text{next}_1 = \text{next}_0 ++ (\text{mid}_0 \times \text{near}_0), \\
& \quad \text{near}_1 = \text{mid}_0, \\
& \quad \text{mid}_1 = \text{far}_0, \\
& \quad \text{far}_1 = \text{far}_0 . \text{next}_1, \\
& \quad \text{near}_2 = \text{if guard then} \text{near}_1 \text{ else } \text{near}_0, \\
& \quad \text{mid}_2 = \text{if guard then} \text{mid}_1 \text{ else } \text{mid}_0, \\
& \quad \text{far}_2 = \text{if guard then} \text{far}_1 \text{ else } \text{far}_0, \\
& \quad \text{next}_2 = \text{if guard then} \text{next}_1 \text{ else } \text{next}_0, \\
& \quad \text{next}_3 = \text{next}_2 ++ (\text{mid}_2 \times \text{near}_2) \\
& \quad \text{head}_0 = \text{head} ++ (\text{this} \times \text{mid}_2) \mid \\
& \quad \text{far}_2 = \text{null} \land \text{Inv(next)} \land \text{Pre(this, head, next)} \land \\
& \quad \neg (\text{Inv(next}_3) \land \text{Post(this, head, head}_0, \text{next, next}_3))
\end{align*}
\]
Bounded verification: step 3/4

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

\textbf{let} \text{near}_0 = \text{this}.\text{head}, \text{mid}_0 = \text{near}_0.\text{next}, \text{far}_0 = \text{mid}_0.\text{next}, \n\text{next}_0 = \text{next} + + (\text{near}_0 \times \text{far}_0), \text{guard} = (\text{far}_0 \neq \text{null}), \n\text{next}_1 = \text{next}_0 + + (\text{mid}_0 \times \text{near}_0), \n\text{near}_1 = \text{mid}_0, \n\text{mid}_1 = \text{far}_0, \n\text{far}_1 = \text{far}_0.\text{next}_1, \n
\text{near}_2 = \textbf{if guard then} \text{near}_1 \textbf{else} \text{near}_0, \n\text{mid}_2 = \textbf{if guard then} \text{mid}_1 \textbf{else} \text{mid}_0, \n\text{far}_2 = \textbf{if guard then} \text{far}_1 \textbf{else} \text{far}_0, \n\text{next}_2 = \textbf{if guard then} \text{next}_1 \textbf{else} \text{next}_0, \n\text{next}_3 = \text{next}_2 + + (\text{mid}_2 \times \text{near}_2), \n\text{head}_0 = \text{head} + + (\text{this} \times \text{mid}_2) \ | 
\text{far}_2 = \text{null} \land \text{Inv (next)} \land \text{Pre (this, head, next)} \land 
\lnot (\text{Inv (next}_3) \land \text{Post (this, head, head}_0, \text{next, next}_3))
Bounded verification: step 3/4

\[\text{this} \subseteq \text{List} \land \text{one this} \land \]
\[\text{head} \subseteq \text{List} \mapsto (\text{Node} \cup \text{null}) \land \]
\[\text{next} \subseteq \text{Node} \mapsto (\text{Node} \cup \text{null}) \land \]
\[\text{data} \subseteq \text{Node} \mapsto (\text{String} \cup \text{null}) \land \]

\text{let} \quad \text{near}_0 = \text{this}.\text{head},
\quad \text{mid}_0 = \text{near}_0.\text{next},
\quad \text{far}_0 = \text{mid}_0.\text{next},
\quad \text{next}_0 = \text{next} \; \text{++} \; (\text{near}_0 \times \text{far}_0),
\quad \text{guard} = (\text{far}_0 \neq \text{null}),
\quad \text{next}_1 = \text{next}_0 \; \text{++} \; (\text{mid}_0 \times \text{near}_0),
\quad \text{near}_1 = \text{mid}_0,
\quad \text{mid}_1 = \text{far}_0,
\quad \text{far}_1 = \text{far}_0.\text{next}_1,
\quad \text{near}_2 = \text{if} \; \text{guard} \; \text{then} \; \text{near}_1 \; \text{else} \; \text{near}_0,
\quad \text{mid}_2 = \text{if} \; \text{guard} \; \text{then} \; \text{mid}_1 \; \text{else} \; \text{mid}_0,
\quad \text{far}_2 = \text{if} \; \text{guard} \; \text{then} \; \text{far}_1 \; \text{else} \; \text{far}_0,
\quad \text{next}_2 = \text{if} \; \text{guard} \; \text{then} \; \text{next}_1 \; \text{else} \; \text{next}_0,
\quad \text{next}_3 = \text{next}_2 \; \text{++} \; (\text{mid}_2 \times \text{near}_2)
\quad \text{head}_0 = \text{head} \; \text{++} \; (\text{this} \times \text{mid}_2)
\quad \text{far}_2 = \text{null} \land \text{Inv(next)} \land \text{Pre(this, head, next)} \land
\quad \neg (\text{Inv(next}_3) \land \text{Post(this, head, head}_0, \text{next, next}_3))

\{ \text{this, n0, n1, n2, s0, s1, s2, null} \}
\{ \text{null} \} \subseteq \text{null} \subseteq \{ \text{null} \}
\{ \} \subseteq \text{this} \subseteq \{ \text{this} \}
\{ \} \subseteq \text{List} \subseteq \{ \text{this} \}
\{ \} \subseteq \text{Node} \subseteq \{ \langle \text{n0} \rangle, \langle \text{n1} \rangle, \langle \text{n2} \rangle \}
\{ \} \subseteq \text{String} \subseteq \{ \langle \text{s0} \rangle, \langle \text{s1} \rangle, \langle \text{s2} \rangle \}
\{ \} \subseteq \text{head} \subseteq \{ \text{this} \times \{ \text{n0, n1, n2, null} \}
\{ \} \subseteq \text{next} \subseteq \{ \text{n0, n1, n2} \} \times \{ \text{n0, n1, n2, null} \}
\{ \} \subseteq \text{data} \subseteq \{ \text{n0, n1, n2} \} \times \{ \text{s0, s1, s2, null} \}

Finite universe of uninterpreted symbols.
Bounded verification: step 3/4

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

let near_0 = this.head, 
    mid_0 = near_0.next, 
    far_0 = mid_0.next,
next_0 = next ++ (near_0 \times far_0),
guard = (far_0 \neq \text{null}),
next_1 = next_0 ++ (mid_0 \times near_0),
near_1 = mid_0,
mid_1 = far_0,
far_1 = far_0.next_1,
near_2 = \text{if guard then} near_1 \ \text{else} \ near_0,
mid_2 = \text{if guard then} mid_1 \ \text{else} \ mid_0,
far_2 = \text{if guard then} far_1 \ \text{else} \ far_0,
next_2 = \text{if guard then} next_1 \ \text{else} \ next_0,
next_3 = next_2 ++ (mid_2 \times near_2),
head_0 = head ++ (this \times mid_2) \land 

far_2 = \text{null} \land \text{Inv}(next) \land \text{Pre}(this, head, next) \land 
\neg (\text{Inv}(next_3) \land \text{Post}(this, head, head_0, next, next_3))
Bounded verification: step 3/4

this \subseteq \text{List} \land \text{one this} \land

\text{head} \subseteq \text{List} \mapsto (\text{Node} \cup \text{null}) \land

\text{next} \subseteq \text{Node} \mapsto (\text{Node} \cup \text{null}) \land

\text{data} \subseteq \text{Node} \mapsto (\text{String} \cup \text{null}) \land

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

\text{next}_0 = \text{next} ++ (\text{near}_0 \times \text{far}_0),
\quad \text{guard} = (\text{far}_0 \neq \text{null}),
\quad \text{next}_1 = \text{next}_0 ++ (\text{mid}_0 \times \text{near}_0),
\quad \text{near}_1 = \text{mid}_0,
\quad \text{mid}_1 = \text{far}_0,
\quad \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,
\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 ++ (\text{mid}_2 \times \text{near}_2),
\quad \text{head}_0 = \text{head} ++ (\text{this} \times \text{mid}_2)
\ |
\quad \text{far}_2 = \text{null} \land \text{Inv} (\text{next}) \land \text{Pre} (\text{this}, \text{head}, \text{next}) \land
\quad \neg (\text{Inv} (\text{next}_3) \land \text{Post} (\text{this}, \text{head}, \text{head}_0, \text{next}, \text{next}_3))
Bounded verification: step 4/4

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,
    next 2 = if guard then next 1 else next 0,
    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 3 = next 2 ++ (mid 2 × near 2),
    head 0 = head ++ (this × mid 2) |

far 2 = null ∧ Inv(next) ∧ Pre(this, head, next) ∧
¬ (Inv(next3) ∧ Post(this, head, head0, next, next3))
Bounded verification: counterexample
Bounded verification: optimization

**Execution finitization** (inlining, unrolling, SSA)

**Forward VCG**

**Heap finitization** (bounds for types, fields)

**Solver**

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);
}
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.
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

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)

```java
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

```java
@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);
}
```

Start with the encoding for bounded verification.

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

let near0 = this.head,
  mid0 = near0.next,
  far0 = mid0.next,

  next0 = next ++ (near0 × far0),
  guard = (far0 != null),
  next1 = next0 ++ (mid0 × near0),
  near1 = mid0,
  mid1 = far0,
  far1 = far0.next1,

  near2 = if guard then near1 else near0,
  mid2 = if guard then mid1 else mid0,
  far2 = if guard then far1 else far0,
  next2 = if guard then next1 else next0,
  next3 = next2 ++ (mid2 × near2)

  head0 = head ++ (this × mid2) |

  far2 = null ∧ Inv(next) ∧ Pre(this, head, next) ∧
  ¬ (Inv(next3) ∧ Post(this, head, head0, next, next3))
```
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.next;

    near2 = phi(guard, far1, near0);
    mid2 = phi(guard, mid0, far0);
    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.

this \subseteq \text{List} \land \textbf{one this} \land
head \subseteq \text{List} \leftrightarrow (\text{Node} \cup \text{null}) \land
next \subseteq \text{Node} \leftrightarrow (\text{Node} \cup \text{null}) \land
data \subseteq \text{Node} \leftrightarrow (\text{String} \cup \text{null}) \land

near0 = this.head \land
mid0 = near0.next \land
far0 = mid0.next \land

next0 = next ++ (near0 \times far0) \land
next1 = next0 ++ (mid0 \times near0) \land
near1 = mid0 \land
mid1 = far0 \land
far1 = far0.next1 \land

\textbf{let} guard = (far0 != null),
near2 = \textbf{if} guard \textbf{then} near1 \textbf{else} near0,
mid2 = \textbf{if} guard \textbf{then} mid1 \textbf{else} mid0,
far2 = \textbf{if} guard \textbf{then} far1 \textbf{else} far0,
next2 = \textbf{if} guard \textbf{then} next1 \textbf{else} next0 |

next3 = next2 ++ (mid2 \times near2) \land
head0 = head ++ (this \times mid2) \land
far2 = null \land \text{Inv(next)} \land \text{Pre(this, head, next)} \land
\text{Inv(next3)} \land \text{Post(this, head, head0, next, next3)
Fault localization: bounds

\[ \text{this} \subseteq \text{List} \land \text{one} \text{ this} \land \]
\[ \text{head} \subseteq \text{List} \mapsto (\text{Node} \cup \text{null}) \land \]
\[ \text{next} \subseteq \text{Node} \mapsto (\text{Node} \cup \text{null}) \land \]
\[ \text{data} \subseteq \text{Node} \mapsto (\text{String} \cup \text{null}) \land \]

\[ \text{near}_0 = \text{this.head} \land \]
\[ \text{mid}_0 = \text{near}_0.\text{next} \land \]
\[ \text{far}_0 = \text{mid}_0.\text{next} \land \]
\[ \text{next}_0 = \text{next} ++ (\text{near}_0 \times \text{far}_0) \land \]
\[ \text{next}_1 = \text{next}_0 ++ (\text{mid}_0 \times \text{near}_0) \land \]
\[ \text{near}_1 = \text{mid}_0 \land \]
\[ \text{mid}_1 = \text{far}_0 \land \]
\[ \text{far}_1 = \text{far}_0.\text{next}_1 \land \]

let \( \text{guard} = (\text{far}_0 \neq \text{null}), \)

\[ \text{near}_2 = \text{if} \text{guard} \text{ then} \text{near}_1 \text{ else} \text{near}_0, \]
\[ \text{mid}_2 = \text{if} \text{guard} \text{ then} \text{mid}_1 \text{ else} \text{mid}_0, \]
\[ \text{far}_2 = \text{if} \text{guard} \text{ then} \text{far}_1 \text{ else} \text{far}_0, \]
\[ \text{next}_2 = \text{if} \text{guard} \text{ then} \text{next}_1 \text{ else} \text{next}_0 | \]
\[ \text{next}_3 = \text{next}_2 ++ (\text{mid}_2 \times \text{near}_2) \land \]
\[ \text{head}_0 = \text{head} ++ (\text{this} \times \text{mid}_2) \land \]
\[ \text{far}_2 = \text{null} \land \text{Inv} (\text{next}) \land \text{Pre} (\text{this, head, next}) \land \]
\[ \text{Inv} (\text{next}_3) \land \text{Post} (\text{this, head, head}_0, \text{next, next}_3) \]

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

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

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

\} \subseteq \text{head}_0 \subseteq \{ \text{this} \} \times \{ \text{n0, n1, n2, null} \}
\} \subseteq \text{next}_0 \subseteq \{ \text{n0, n1, n2} \} \times \{ \text{n0, n1, n2, null} \}
\} \subseteq \text{next}_1 \subseteq \{ \text{n0, n1, n2} \} \times \{ \text{n0, n1, n2, null} \}
\} \subseteq \text{next}_3 \subseteq \{ \text{n0, n1, n2} \} \times \{ \text{n0, n1, n2, null} \}
\} \subseteq \text{near}_0 \subseteq \{ \text{n0, n1, n2, null} \}
\} \subseteq \text{near}_1 \subseteq \{ \text{n0, n1, n2, null} \}
\} \subseteq \text{mid}_0 \subseteq \{ \text{n0, n1, n2, null} \}
\} \subseteq \text{mid}_1 \subseteq \{ \text{n0, n1, n2, null} \}
\} \subseteq \text{far}_0 \subseteq \{ \text{n0, n1, n2, null} \}
\} \subseteq \text{far}_1 \subseteq \{ \text{n0, n1, n2, null} \}

Input expressed as a partial model.
Fault localization: minimal unsat core

\[
\begin{align*}
\text{this} & \subseteq \text{List} \land \text{one this} \land \\
\text{head} & \subseteq \text{List} \mapsto (\text{Node} \cup \text{null}) \land \\
\text{next} & \subseteq \text{Node} \mapsto (\text{Node} \cup \text{null}) \land \\
\text{data} & \subseteq \text{Node} \mapsto (\text{String} \cup \text{null}) \land \\
\text{near} & = 0 \land \\
\text{mid} & = \text{near} \land \\
\text{far} & = \text{mid} \land \\
\text{next} & = \text{next} + (\text{near} \times \text{far}) \land \\
\text{next} & = \text{next} + (\text{mid} \times \text{near}) \land \\
\text{near} & = \text{mid} \land \\
\text{mid} & = \text{far} \land \\
\text{far} & = \text{far} \land \\
\text{let} & \text{guard} = (\text{far} != \text{null}), \\
\text{near} & = \text{if guard then near else near}, \\
\text{mid} & = \text{if guard then mid else mid}, \\
\text{far} & = \text{if guard then far else far}, \\
\text{next} & = \text{if guard then next else next} \mid \\
\text{next} & = \text{next} + (\text{mid} \times \text{near}) \land \\
\text{head} & = \text{head} + (\text{this} \times \text{mid}) \land \\
\text{far} & = \text{null} \land \text{Inv(next)} \land \text{Pre(this, head, next)} \land \\
\text{Inv(next)} \land \text{Post(this, head, head, next, next)}
\end{align*}
\]

\[
\begin{align*}
\text{this} & = \{ \text{this}, \text{n0}, \text{n1}, \text{n2}, \text{s0}, \text{s1}, \text{s2}, \text{null} \} \\
\text{null} & = \{ \text{null} \} \\
\text{this} & = \{ \text{this} \} \\
\text{List} & = \{ \text{this} \} \\
\text{Node} & = \{ \text{n0}, \text{n1}, \text{n2} \} \\
\text{String} & = \{ \text{s1}, \text{s2} \} \\
\text{head} & = \{ \text{this}, \text{n2} \} \\
\text{next} & = \{ \text{n2}, \text{s1}, <\text{n1}, \text{s2}, <\text{n0}, \text{null} \} \\
\text{data} & = \{ \text{n2}, \text{s1}, <\text{n1}, <\text{s2}, <\text{n0}, \text{null} \} \\
\text{near} & = \text{mid} \land \\
\text{mid} & = \text{far} \land \\
\text{far} & = \text{far} \land \\
\text{let} & \text{guard} = (\text{far} \neq \text{null}), \\
\text{near} & = \text{if guard then near else near}, \\
\text{mid} & = \text{if guard then mid else mid}, \\
\text{far} & = \text{if guard then far else far}, \\
\text{next} & = \text{if guard then next else next} \mid \\
\text{next} & = \text{next} + (\text{mid} \times \text{near}) \land \\
\text{head} & = \text{head} + (\text{this} \times \text{mid}) \land \\
\text{far} & = \text{null} \land \text{Inv(next)} \land \text{Pre(this, head, next)} \land \\
\text{Inv(next)} \land \text{Post(this, head, head, next, next)}
\end{align*}
\]
Fault localization: minimal unsat core

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

```java
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)

```java
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