Bounded Verification

Emina Torlak
emina@cs.washington.edu
Today
Today

Last lecture

• Full functional verification with Dafny, Boogie, and Z3
Today

Last lecture
- Full functional verification with Dafny, Boogie, and Z3

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

Last lecture
• Full functional verification with Dafny, Boogie, and Z3

Today
• Bounded verification with Kodkod (Forge, Miniatur, TACO)

Announcements
• **Homework 2** is due today at 11pm
• **Homework 3** has been released
The spectrum of program validation tools

- **Confidence**
- **Cost (programmer effort, time, expertise)**

- **Static Analysis**
- **Verification**
- **Bounded Verification & Symbolic Execution**
- **Extended Static Checking**
- **Concolic Testing & Whitebox Fuzzing**
- **Ad-hoc Testing**
The spectrum of program validation tools

E.g., Dafny, Coq, Leon:
- support for rich (FOL+) correctness properties
- high annotation overhead (pre/post conditions, invariants, etc.)
The spectrum of program validation tools

Confidence

Cost (programmer effort, time, expertise)

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

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

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

Static Analysis
- E.g., Calysto, Saturn:
  - user-defined assertions supported but optional
  - no annotations
  - some/low false positives

Verification

Cost (programmer effort, time, expertise)

Confidence
The spectrum of program validation tools

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

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

Confidence

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

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

- **Cost (programmer effort, time, expertise)**
  - Static Analysis
  - Verification

E.g., SAGE, Pex, CUTE, DART:
- test harnesses and/or user-defined assertions
- no annotations
- no false positives
The spectrum of program validation tools

- **Confidence**
- **Cost (programmer effort, time, expertise)**

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

**Questions:**

1. What are the main categories of program validation tools?
2. How does the confidence and cost relate to each other for different validation tools?
3. Can you explain the significance of **Extended Static Checking** in the context of program validation?
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
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.
Pre/post/frame conditions & data invariants

```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;
}
```

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

![Diagram of list reversal process]
Pre/post/frame conditions & data 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;
}
Pre/post/frame conditions & data 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;
}
Pre/post/frame conditions & data 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;
}
Pre/post/frame conditions & data 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;
}
A relational model of memory (heap)

```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;
}
```

@invariant \text{Inv}(next)
@requires \text{Pre}(this, head, next)
@ensures \text{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;
}
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⟩ }

@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;
}

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⟩}

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

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

```java
void reverse() {
    Node near = this.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;
}
```

Execution finitization (inlining, unrolling, SSA)
@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);
}
```

Bounded verification: step 2/4

Execution finitization (inlining, unrolling, SSA)

Forward VCG

Symbolic interpretation of the code with respect to the relational heap model.
Bounded verification: step 2/4

```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);
}
```

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

\[
\begin{align*}
\text{let} & \; \text{near0} = \text{this.head}, \\
\text{mid0} & = \text{near0}.\text{next}, \\
\text{far0} & = \text{mid0}.\text{next}, \\
\text{next0} & = \text{next} ++ (\text{near0} \times \text{far0}), \\
\text{guard} & = (\text{far0} != \text{null}), \\
\text{next1} & = \text{next0} ++ (\text{mid0} \times \text{near0}), \\
\text{near1} & = \text{mid0}, \\
\text{mid1} & = \text{far0}, \\
\text{far1} & = \text{far0}.\text{next1}, \\
\text{next2} & = \text{if guard then near1 else near0}, \\
\text{mid2} & = \text{if guard then mid1 else mid0}, \\
\text{far2} & = \text{if guard then far1 else far0}, \\
\text{next2} & = \text{if guard then next1 else next0}, \\
\text{next3} & = \text{next2} ++ (\text{mid2} \times \text{near2}) \\
\text{head0} & = \text{head} ++ (\text{this} \times \text{mid2}) \land \\
\text{far2} & = \text{null} \land \text{Inv(next)} \land \text{Pre(this, head, next)} \land \\
\neg (\text{Inv(next3)} \land \text{Post(this, head, head0, next, next3)})
\end{align*}
\]
Bounded verification: step 3/4

\[
\begin{align*}
\text{this} & \subseteq \text{List} \land \text{one this} \land \\
\text{head} & \subseteq \text{List} \mapsto (\text{Node} \cup \text{null}) \land \\
\text{next} & \subseteq \text{Node} \mapsto (\text{Node} \cup \text{null}) \land \\
\text{data} & \subseteq \text{Node} \mapsto (\text{String} \cup \text{null}) \land \\
\text{let} & \text{near}_0 = \text{this.head}, \\
& \quad \text{mid}_0 = \text{near}_0.\text{next}, \\
& \quad \text{far}_0 = \text{mid}_0.\text{next}, \\
& \quad \text{next}_0 = \text{next} ++ (\text{near}_0 \times \text{far}_0), \\
& \quad \text{guard} = (\text{far}_0 \neq \text{null}), \\
& \quad \text{next}_1 = \text{next}_0 ++ (\text{mid}_0 \times \text{near}_0), \\
& \quad \text{near}_1 = \text{mid}_0, \\
& \quad \text{mid}_1 = \text{far}_0, \\
& \quad \text{far}_1 = \text{far}_0.\text{next}_1, \\
& \quad \text{near}_2 = \text{if guard then near}_1 \text{ else near}_0, \\
& \quad \text{mid}_2 = \text{if guard then mid}_1 \text{ else mid}_0, \\
& \quad \text{far}_2 = \text{if guard then far}_1 \text{ else far}_0, \\
& \quad \text{next}_2 = \text{if guard then next}_1 \text{ else next}_0, \\
& \quad \text{next}_3 = \text{next}_2 ++ (\text{mid}_2 \times \text{near}_2) \\
& \quad \text{head}_0 = \text{head} ++ (\text{this} \times \text{mid}_2) |
\end{align*}
\]

\[
\begin{align*}
\text{far}_2 & = \text{null} \land \text{Inv(next)} \land \text{Pre(this, head, next)} \land \\
& \quad \neg (\text{Inv(next}_3) \land \text{Post(this, head, head}_0, \text{next, next}_3))
\end{align*}
\]
Bounded verification: step 3/4

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

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

\[
\{ \text{this, n0, n1, n2, s0, s1, s2, null} \} \\
\{ \langle \text{null} \rangle \} \subseteq \text{null} \subseteq \{ \langle \text{null} \rangle \} \\
\{ \} \subseteq \text{this} \subseteq \{ \langle \text{this} \rangle \} \\
\{ \} \subseteq \text{List} \subseteq \{ \langle \text{this} \rangle \} \\
\{ \} \subseteq \text{Node} \subseteq \{ \langle \text{n0}, \langle n1, \langle n2 \rangle \rangle \} \\
\{ \} \subseteq \text{String} \subseteq \{ \langle s0, \langle s1, \langle s2 \rangle \rangle \} \\
\{ \} \subseteq \text{head} \subseteq \{ \text{this} \} \times \{ \text{n0, n1, n2, null} \} \\
\{ \} \subseteq \text{next} \subseteq \{ \text{n0, n1, n2} \} \times \{ \text{n0, n1, n2, null} \} \\
\{ \} \subseteq \text{data} \subseteq \{ \text{n0, n1, n2} \} \times \{ \text{s0, s1, s2, null} \}
\]
Bounded verification: step 3/4

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

Finite universe of uninterpreted symbols.

\[
\begin{align*}
\{ \text{this}, \text{n0}, \text{n1}, \text{n2}, \text{s0}, \text{s1}, \text{s2}, \text{null} \} \\
\{ \langle \text{null} \rangle \} \subseteq \text{null} \subseteq \{ \langle \text{null} \rangle \} \\
\{ \} \subseteq \text{this} \subseteq \{ \langle \text{this} \rangle \} \\
\{ \} \subseteq \text{List} \subseteq \{ \langle \text{this} \rangle \} \\
\{ \} \subseteq \text{Node} \subseteq \{ \langle \text{n0}, \langle \text{n1}, \langle \text{n2} \rangle \rangle \} \\
\{ \} \subseteq \text{String} \subseteq \{ \langle \text{s0}, \langle \text{s1}, \langle \text{s2} \rangle \rangle \} \\
\{ \} \subseteq \text{head} \subseteq \{ \text{this} \} \times \{ \text{n0}, \text{n1}, \text{n2}, \text{null} \} \\
\{ \} \subseteq \text{next} \subseteq \{ \text{n0}, \text{n1}, \text{n2} \} \times \{ \text{n0}, \text{n1}, \text{n2}, \text{null} \} \\
\{ \} \subseteq \text{data} \subseteq \{ \text{n0}, \text{n1}, \text{n2} \} \times \{ \text{s0}, \text{s1}, \text{s2}, \text{null} \}
\end{align*}
\]
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

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

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

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

Finite universe of uninterpreted symbols.

Upper bound on each relation: tuples it may contain.
Bounded verification: step 3/4

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

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

Finite universe of uninterpreted symbols.

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

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

let

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

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

next_3 = \text{next}_2 ++ (\text{mid}_2 \times \text{near}_2)
head_0 = \text{head} ++ (\text{this} \times \text{mid}_2) \mid
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, \text{next, next}_3))
Bounded verification: counterexample
Bounded verification: optimization

- Execution finitization (inlining, unrolling, SSA)
- Forward VCG
- Heap finitization (bounds for types, fields)
- Solver
Bounded verification: optimization

Execution finitization (inlining, unrolling, SSA)

Forward VCG

Heap finitization (bounds for types, fields)

Solver

Finitized program after inlining may be huge.
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.
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 valid input and the expected output, find a minimal subset of program statements that prevents the execution on the given input from reaching a valid output state.

```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);
}
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 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);
}
From bounded verification to fault localization

```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);
}
```

Given a buggy program and a valid input and the expected output, find a minimal subset of program statements that prevents the execution on the given input from reaching a valid output state.

Introduce additional “indicator” relations into the encoding.

The resulting formula, together with the input partial model, is unsatisfiable.
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: 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.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);
}

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

```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.next;

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

    assume far2 == null;

    next3 = update(next2, mid2, near2);
    head0 = update(head, this, mid2);
    this \subseteq List \land one this \land
    head \subseteq List \leftrightarrow (Node \cup null) \land
    next \subseteq Node \leftrightarrow (Node \cup null) \land
    data \subseteq Node \leftrightarrow (String \cup 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.next \land

    let guard = (far0 != null),
    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 \times near2) \land
    head0 = head ++ (this \times mid2) \land
    far2 = null \land Inv(next) \land Pre(this, head, next) \land
    Inv(next) \land Post(this, head, head0, next, next3)
```
Fault localization: bounds

\[
\begin{align*}
\text{this} \subseteq & \ \text{List} \land \text{one this} \land \\
\text{head} \subseteq & \ \text{List} \mapsto (\text{Node} \cup \text{null}) \land \\
\text{next} \subseteq & \ \text{Node} \mapsto (\text{Node} \cup \text{null}) \land \\
\text{data} \subseteq & \ \text{Node} \mapsto (\text{String} \cup \text{null}) \land \\
\text{near}_0 = & \ \text{this.head} \land \\
\text{mid}_0 = & \ \text{near}_0.\text{next} \land \\
\text{far}_0 = & \ \text{mid}_0.\text{next} \land \\
\text{next}_0 = & \ \text{next} \mathbin{\text{++}} (\text{near}_0 \times \text{far}_0) \land \\
\text{next}_1 = & \ \text{next}_0 \mathbin{\text{++}} (\text{mid}_0 \times \text{near}_0) \land \\
\text{near}_1 = & \ \text{mid}_0 \land \\
\text{mid}_1 = & \ \text{far}_0 \land \\
\text{far}_1 = & \ \text{far}_0.\text{next}_1 \land \\
\end{align*}
\]

\[
\begin{align*}
\text{let guard} = & \ (\text{far}_0 \neq \text{null}), \\
\text{near}_2 = & \ \text{if guard then near}_1 \text{ else near}_0, \\
\text{mid}_2 = & \ \text{if guard then mid}_1 \text{ else mid}_0, \\
\text{far}_2 = & \ \text{if guard then far}_1 \text{ else far}_0, \\
\text{next}_2 = & \ \text{if guard then next}_1 \text{ else next}_0 \mid \\
\text{next}_3 = & \ \text{next}_2 \mathbin{\text{++}} (\text{mid}_2 \times \text{near}_2) \land \\
\text{head}_0 = & \ \text{head} \mathbin{\text{++}} (\text{this} \times \text{mid}_2) \land \\
\text{far}_2 = & \ \text{null} \land \text{Inv(next)} \land \text{Pre(this, head, next)} \land \\
\text{Inv(next)} \land & \ \text{Post(this, head, head}_0, \text{next, next}_3) \\
\end{align*}
\]

\[
\begin{align*}
\text{Input expressed as a partial model.} \\
\{ \text{this, n0, n1, n2, s0, s1, s2, null} \} \\
\text{null} = \{ \text{<null>} \} \\
\text{this} = \{ \text{<this>} \} \\
\text{List} = \{ \text{<this>} \} \\
\text{Node} = \{ \text{<n0>, <n1>, <n2>} \} \\
\text{String} = \{ \text{<s1>, <s2>} \} \\
\text{head} = \{ \text{<this, n2>} \} \\
\text{next} = \{ \text{<n2, n1>, <n1, n0>, <n0, null>} \} \\
\text{data} = \{ \text{<n2, s1>, <n1, s2>, <n0, null>} \} \\
\end{align*}
\]
Fault localization: bounds

\[\text{this} \subseteq \text{List} \land \text{one this} \land\]
\[\text{head} \subseteq \text{List} \mapsto (\text{Node} \cup \text{null}) \land\]
\[\text{next} \subseteq \text{Node} \mapsto (\text{Node} \cup \text{null}) \land\]
\[\text{data} \subseteq \text{Node} \mapsto (\text{String} \cup \text{null}) \land\]
\[\text{near}_0 = \text{this.head} \land\]
\[\text{mid}_0 = \text{near}_0.\text{next} \land\]
\[\text{far}_0 = \text{mid}_0.\text{next} \land\]
\[\text{next}_0 = \text{next} ++ (\text{near}_0 \times \text{far}_0) \land\]
\[\text{next}_1 = \text{next}_0 ++ (\text{mid}_0 \times \text{near}_0) \land\]
\[\text{near}_1 = \text{mid}_0 \land\]
\[\text{mid}_1 = \text{far}_0 \land\]
\[\text{far}_1 = \text{far}_0.\text{next}_1 \land\]

\textbf{let} guard = (\text{far}_0 \ne \text{null}),
\[\text{near}_2 = \text{if guard then near}_1 \text{ else near}_0,\]
\[\text{mid}_2 = \text{if guard then mid}_1 \text{ else mid}_0,\]
\[\text{far}_2 = \text{if guard then far}_1 \text{ else far}_0,\]
\[\text{next}_2 = \text{if guard then next}_1 \text{ else next}_0 |\]
\[\text{next}_3 = \text{next}_2 ++ (\text{mid}_2 \times \text{near}_2) \land\]
\[\text{head}_0 = \text{head} ++ (\text{this} \times \text{mid}_2) \land\]
\[\text{far}_2 = \text{null} \land \text{Inv(next)} \land \text{Pre(this, head, next)} \land\]
\[\text{Inv(next}_3) \land \text{Post(this, head, head}_0, \text{next, next}_3)\]
Fault localization: minimal unsat core

this ⊆ List ∧ one this ∧
head ⊆ List → (Node ∪ null) ∧
next ⊆ Node → (Node ∪ null) ∧
data ⊆ Node → (String ∪ null) ∧
near₀ = this.head ∧
mid₀ = near₀.next ∧
far₀ = mid₀.next ∧

next₀ = next ++ (near₀ × far₀) ∧
next₁ = next₀ ++ (mid₀ × near₀) ∧
near₁ = mid₀ ∧
mid₁ = far₀ ∧
far₁ = far₀.next₁ ∧

let guard = (far₀ ≠ null),
    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)

{ 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> }

{ } ⊆ head₀ ⊆ { this } × { n₀, n₁, n₂, null }
{ } ⊆ next₀ ⊆ { n₀, n₁, n₂ } × { n₀, n₁, n₂, null }
{ } ⊆ next₁ ⊆ { n₀, n₁, n₂ } × { n₀, n₁, n₂, null }
{ } ⊆ next₃ ⊆ { n₀, n₁, n₂ } × { n₀, n₁, n₂, null }
{ } ⊆ near₀ ⊆ { n₀, n₁, n₂, null }
{ } ⊆ near₁ ⊆ { n₀, n₁, n₂, null }
{ } ⊆ mid₀ ⊆ { n₀, n₁, n₂, null }
{ } ⊆ mid₁ ⊆ { n₀, n₁, n₂, null }
{ } ⊆ far₀ ⊆ { n₀, n₁, n₂, null }
{ } ⊆ far₁ ⊆ { 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)

```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);
}
```
Summary

Today

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

Next lecture

• Symbolic execution and concolic testing