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

Confidence

Cost (programmer effort, time, expertise)

Bounded Verification & Symbolic Execution

Concolic Testing & Whitebox Fuzzing

Ad-hoc Testing

Static Analysis

Verification

Extended Static Checking
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
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

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

Confidence

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

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
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

The spectrum includes:
- **Concolic Testing & Whitebox Fuzzing**
- **Bounded Verification & Symbolic Execution**
- **Extended Static Checking**
- **Static Analysis**
- **Verification**

Cost (programmer effort, time, expertise)

Confidence
The spectrum of program verification tools

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

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

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: 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

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;
}
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: 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;
}
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;
}
```

![Diagram of memory heap with nodes n0, n1, n2, and head, showing data flow and null termination]
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 \texttt{Inv(next)}
@requires \texttt{Pre(this, head, next)}
@ensures \texttt{Post(this, old(head), head, old(next), next)}

\begin{verbatim}
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;
}
\end{verbatim}

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)

```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⟩}
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⟩}

[Diagram showing the memory layout and the reverse method]
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⟩ }

![Diagram of memory heap with nodes and data]
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;
}
```
@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;
    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);
}
@invariant \texttt{Inv(next)}
@requires \texttt{Pre(this, head, next)}
@ensures \texttt{Post(this, old(head), head, old(next), next)}

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

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

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

    assume \texttt{far2 == null};

    next3 = update(next2, mid2, near2);
    head0 = update(head, \texttt{this}, mid2);
}
```
Bounded verification: step 2/4

```plaintext
@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);
}
```

```plaintext
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

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

\[
\begin{align*}
\text{let} \ n & = \text{this.head}, \\
\text{mid} & = n.\text{next}, \\
\text{far} & = \text{mid}.\text{next}, \\
\text{next} & = \text{next} ++ (n \times \text{far}), \\
\text{guard} & = (\text{far} \neq \text{null}), \\
\text{next} & = \text{next} ++ (\text{mid} \times n), \\
\text{near} & = \text{mid}, \\
\text{mid} & = \text{far}, \\
\text{far} & = \text{far}.\text{next}, \\
\text{near} & = \text{if guard then near else near}, \\
\text{mid} & = \text{if guard then mid else mid}, \\
\text{far} & = \text{if guard then far else far}, \\
\text{next} & = \text{if guard then next else next}, \\
\text{next} & = \text{next} ++ (\text{mid} \times \text{near}), \\
\text{head} & = \text{head} ++ (\text{this} \times \text{mid}) | \\
\text{far} & = \text{null} \land \text{Inv(next)} \land \text{Pre(this, head, next)} \land \\
\neg (\text{Inv(next)} \land \text{Post(this, head, head0, next, next3)})
\end{align*}
\]
Bounded verification: step 3/4

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

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 \land \text{Inv(next)} \land \text{Pre(this, head, next)} \land 
\neg (\text{Inv(next3)} \land \text{Post(this, head, head0, next, next3)})
Bounded verification: step 3/4

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

let near\(_0\) = this\_head,
mid\(_0\) = near\(_0\).\_next,
far\(_0\) = mid\(_0\).\_next,

next\(_0\) = next ++ (near\(_0\) × far\(_0\)),
guard = (far\(_0\) \neq \text{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 \land \text{Inv(next)} \land \text{Pre(this, head, next)} \land 
\neg (\text{Inv(next3)} \land \text{Post(this, head, head0, next, next3)})

Finite universe of uninterpreted symbols.

\{ \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 \} \}
\{ \} \subseteq \text{String} \subseteq \{ \langle \text{s0}, \langle \text{s1}, \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} \}
Bounded verification: step 3/4

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))

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

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

\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} \ \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), \\
    \text{head}_0 = \text{head} ++ (\text{this} \times \text{mid}_2), \\
    \text{far}_2 = \text{null} \land \text{Inv} (\text{next}) \land \text{Pre}(\text{this}, \text{head}, \text{next}) \land \\
    \neg (\text{Inv} (\text{next}_3) \land \text{Post}(\text{this, head, head}_0, \text{next, next}_3))
\]

Finite universe of uninterpreted symbols:

\[
\begin{align*}
\{ \text{this}, n0, n1, n2, s0, s1, s2, \text{null} \} \\
\{ \langle \text{null} \rangle \} \subseteq \text{null} \subseteq \{ \langle \text{null} \rangle \} \\
\langle \text{null} \rangle \subseteq \{ \text{this} \} \\
\langle \text{this} \rangle \subseteq \{ \text{this} \} \times \{ n0, n1, n2, \text{null} \} \\
\langle n0, n1, n2 \rangle \subseteq \{ \text{this} \} \times \{ n0, n1, n2, \text{null} \} \\
\langle s0, s1, s2 \rangle \subseteq \{ \text{null} \} \times \{ s0, s1, s2, \text{null} \} \\
\end{align*}
\]

Lower bound on each relation: tuples it \textit{must} contain.

Upper bound on each relation: tuples it \textit{may} contain.
Bounded verification: step 4/4

\[
\begin{align*}
\text{let } & \text{near} = 0, \text{mid} = 0 \text{.next}, \text{far} = 0 \text{.next}, \text{next} = 0 \text{.next} + (\text{near} \times \text{far}) , \text{guard} = (\text{far} \neq \text{null}) , \\
& \text{next} = \text{next} + (\text{mid} \times \text{near}) , \\
& \text{head} = \text{head} + (\text{this} \times \text{mid}) , \\
& \text{far} = \text{null} \land \text{Inv(\text{next})} \land \text{Pre(\text{this}, \text{head}, \text{next})} \land \\
& \neg (\text{Inv(\text{next}_3)} \land \text{Post(\text{this}, \text{head}, \text{head}_0, \text{next}, \text{next}_3))}
\end{align*}
\]
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)

```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);
}
```
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 \text{Pre}(this, head, next)
@ensures \text{Post}(this, old(head), head, old(next), next)

```java
def 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.
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.
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 \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

let near0 = this.head,
    mid0 = near0.next,
    far0 = mid0.next,
    next0 = next ++ (near0 \times far0),
    guard = (far0 != null),
    next1 = next0 ++ (mid0 \times near0),
    near1 = mid0,
    mid1 = far0,
    far1 = far0.next1,
    far2 = 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)
head0 = head ++ (this \times mid2) \mid

far2 = null \land \text{Inv(next)} \land \text{Pre(this, head, next)} \land
\neg (\text{Inv(next3)} \land \text{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);
    mid2 = phi(guard, mid1, far1);
    far2 = phi(guard, near2, 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.

```java
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)
```
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} + + (\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 \\

\text{let} \quad \text{guard} = (\text{far}_0 \neq \text{null}), \\
\text{near}_2 & = \text{if} \quad \text{guard} \quad \text{then} \quad \text{near}_1 \quad \text{else} \quad \text{near}_0, \\
\text{mid}_2 & = \text{if} \quad \text{guard} \quad \text{then} \quad \text{mid}_1 \quad \text{else} \quad \text{mid}_0, \\
\text{far}_2 & = \text{if} \quad \text{guard} \quad \text{then} \quad \text{far}_1 \quad \text{else} \quad \text{far}_0, \\
\text{next}_2 & = \text{if} \quad \text{guard} \quad \text{then} \quad \text{next}_1 \quad \text{else} \quad \text{next}_0 |
\end{align*}
\]

\[
\begin{align*}
\text{next}_3 & = \text{next}_2 + + (\text{mid}_2 \times \text{near}_2) \land \\
\text{head}_0 & = \text{head} + + (\text{this} \times \text{mid}_2) \land \\
\text{far}_2 & = \text{null} \land \text{Inv(next)} \land \text{Pre(this, head, next)} \land \\
\text{Inv(next}_3) \land \text{Post(this, head, head}_0, \text{next, next}_3)
\end{align*}
\]

\[
\begin{align*}
\{ \text{this, n0, n1, n2, s0, s1, s2, null} \}
\end{align*}
\]

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

\[
\begin{align*}
\langle \rangle & \subseteq \text{head}_0 \subseteq \{ \text{this} \} \times \{ \text{n0, n1, n2, null} \} \\
\langle \rangle & \subseteq \text{next}_0 \subseteq \{ \text{n0, n1, n2} \} \times \{ \text{n0, n1, n2, null} \} \\
\langle \rangle & \subseteq \text{next}_1 \subseteq \{ \text{n0, n1, n2} \} \times \{ \text{n0, n1, n2, null} \} \\
\langle \rangle & \subseteq \text{next}_3 \subseteq \{ \text{n0, n1, n2} \} \times \{ \text{n0, n1, n2, null} \} \\
\langle \rangle & \subseteq \text{near}_0 \subseteq \{ \text{n0, n1, n2, null} \} \\
\langle \rangle & \subseteq \text{near}_1 \subseteq \{ \text{n0, n1, n2, null} \} \\
\langle \rangle & \subseteq \text{mid}_0 \subseteq \{ \text{n0, n1, n2, null} \} \\
\langle \rangle & \subseteq \text{mid}_1 \subseteq \{ \text{n0, n1, n2, null} \} \\
\langle \rangle & \subseteq \text{far}_0 \subseteq \{ \text{n0, n1, n2, null} \} \\
\langle \rangle & \subseteq \text{far}_1 \subseteq \{ \text{n0, n1, n2, null} \}
\end{align*}
\]

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

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

\[ \text{near}_0 = \text{this}.\text{head} \land \]
\[ \text{mid}_0 = \text{near}_0.\text{next} \land \]
\[ \text{far}_0 = \text{mid}_0.\text{next} \land \]

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

\textbf{let} \quad \text{guard} = (\text{far}_0 \neq \text{null}),
\[ \text{near}_2 = \text{if} \ \text{guard} \ \text{then} \ \text{near}_1 \ \text{else} \ \text{near}_0, \]
\[ \text{mid}_2 = \text{if} \ \text{guard} \ \text{then} \ \text{mid}_1 \ \text{else} \ \text{mid}_0, \]
\[ \text{far}_2 = \text{if} \ \text{guard} \ \text{then} \ \text{far}_1 \ \text{else} \ \text{far}_0, \]
\[ \text{next}_2 = \text{if} \ \text{guard} \ \text{then} \ \text{next}_1 \ \text{else} \ \text{next}_0 \mid \]

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

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

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

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

\[ \{ \} \subseteq \text{head}_0 \subseteq \{ \text{this} \} \times \{ \text{n0}, \text{n1}, \text{n2}, \text{null} \} \]
\[ \{ \} \subseteq \text{next}_0 \subseteq \{ \text{n0}, \text{n1}, \text{n2} \} \times \{ \text{n0}, \text{n1}, \text{n2}, \text{null} \} \]
\[ \{ \} \subseteq \text{next}_1 \subseteq \{ \text{n0}, \text{n1}, \text{n2} \} \times \{ \text{n0}, \text{n1}, \text{n2}, \text{null} \} \]
\[ \{ \} \subseteq \text{next}_3 \subseteq \{ \text{n0}, \text{n1}, \text{n2} \} \times \{ \text{n0}, \text{n1}, \text{n2}, \text{null} \} \]

\[ \{ \} \subseteq \text{near}_0 \subseteq \{ \text{n0}, \text{n1}, \text{n2}, \text{null} \} \]
\[ \{ \} \subseteq \text{near}_1 \subseteq \{ \text{n0}, \text{n1}, \text{n2}, \text{null} \} \]
\[ \{ \} \subseteq \text{mid}_0 \subseteq \{ \text{n0}, \text{n1}, \text{n2}, \text{null} \} \]
\[ \{ \} \subseteq \text{mid}_1 \subseteq \{ \text{n0}, \text{n1}, \text{n2}, \text{null} \} \]
\[ \{ \} \subseteq \text{far}_0 \subseteq \{ \text{n0}, \text{n1}, \text{n2}, \text{null} \} \]
\[ \{ \} \subseteq \text{far}_1 \subseteq \{ \text{n0}, \text{n1}, \text{n2}, \text{null} \} \]
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);
}
```
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