# Verifying Array Manipulating Programs by Tiling

Supratik Chakraborty[1], Ashutosh Gupta[1], Divyesh Unadkat[1,2]

Indian Institute of Technology Bombay[1]
TCS Research[2]

Static Analysis Symposium (SAS), 2017

# Motivating Example

```
void foo(int A[], int N) {
  for (int i = 0; i < N; i++) {
    if(!(i==0 || i==N-1)) {
      if (A[i] < 5) {
        A[i+1] = A[i] + 1;
        A[i] = A[i-1];
      }
    } else {
      A[i] = 5;
    }
  }
  assert(for k in 0..N-1, A[k]>=5);
}
```

# Motivating Example

```
void foo(int A[], int N) {
  for (int i = 0; i < N; i++) {
    if(!(i==0 || i==N-1)) {
      if (A[i] < 5) {
        A[i+1] = A[i] + 1;
        A[i] = A[i-1];
      }
    } else {
      A[i] = 5;
    }
  }
  assert(for k in 0..N-1, A[k]>=5);
}
```

Goal

- Programs manipulating arrays of parametric size in loops
- Multiple indices updated and/or accessed
- Prove quantified post-conditions over arrays

# Motivating Example

```
void foo(int A[], int N) {
  for (int i = 0; i < N; i++) {
    if(!(i==0 || i==N-1)) {
      if (A[i] < 5) {
        A[i+1] = A[i] + 1;
        A[i] = A[i-1];
      }
    } else {
      A[i] = 5;
    }
  }
  assert(for k in 0..N-1, A[k]>=5);
}
```

Goal

- Programs manipulating arrays of parametric size in loops
- Multiple indices updated and/or accessed
- Prove quantified post-conditions over arrays

Classical Hoare logic-based method

- Needs a quantified loop invariant
- Difficult to compute precisely

# Motivating Example

```
void foo(int A[], int N) {
  for (int i = 0; i < N; i++) {
    if(!(i==0 || i==N-1)) {
      if (A[i] < 5) {
        A[i+1] = A[i] + 1;
        A[i] = A[i-1];
      }
    } else {
      A[i] = 5;
    }
  }
  assert(for k in 0..N-1, A[k]>=5);
}
```

Necessary loop invariant:
$\forall j.(0 \leq j < i) \implies (A[j] \geq 5)$

Goal

- Programs manipulating arrays of parametric size in loops
- Multiple indices updated and/or accessed
- Prove quantified post-conditions over arrays

Classical Hoare logic-based method

- Needs a quantified loop invariant
- Difficult to compute precisely

# Verifying Candidate Invariants to prove Post-condition

- Check validity of $\{\forall k.\phi(x) \wedge B\}\ \mathsf{L}_{\mathrm{body}}(x, x')\ \{\forall k.\phi(x')\}$
- $\exists x, x'.\forall k.\phi(x) \wedge B \wedge Enc(\mathsf{L}_{\mathrm{body}}(x, x')) \wedge \exists k.\neg\phi(x')$ must be unsat
- Check validity of $\forall k.\phi(x) \wedge \neg B \implies \forall k.\psi(x)$
- Both checks are hard as they involve a quantifier alternation

# Verifying Candidate Invariants to prove Post-condition

- Check validity of $\{\forall k.\phi(x) \wedge B\}$ $\mathsf{L}_{\mathrm{body}}(x, x')$ $\{\forall k.\phi(x')\}$
- $\exists x, x'.\forall k.\phi(x) \wedge B \wedge Enc(\mathsf{L}_{\mathrm{body}}(x, x')) \wedge \exists k.\neg\phi(x')$ must be unsat
- Check validity of $\forall k.\phi(x) \wedge \neg B \implies \forall k.\psi(x)$
- Both checks are hard as they involve a quantifier alternation

Efficient verification tools available

- Bounded model checkers such as CBMC, Corral
  - Support rich program constructs
  - Do not support quantified reasoning

# Verifying Candidate Invariants to prove Post-condition

- Check validity of $\{\forall k.\phi(x) \wedge B\}\ \mathsf{L}_{\mathrm{body}}(x, x')\ \{\forall k.\phi(x')\}$
- $\exists x, x'.\forall k.\phi(x) \wedge B \wedge Enc(\mathsf{L}_{\mathrm{body}}(x, x')) \wedge \exists k.\neg\phi(x')$ must be unsat
- Check validity of $\forall k.\phi(x) \wedge \neg B \implies \forall k.\psi(x)$
- Both checks are hard as they involve a quantifier alternation

Efficient verification tools available

- Bounded model checkers such as CBMC, Corral
  - Support rich program constructs
  - Do not support quantified reasoning
- SMT solvers such as Z3, CVC4, Yices
  - Support quantified reasoning
  - Can prove small quantified formulas with one or two of alternations
  - Scalability a concern; on-going research

# Motivating Example

```
void foo(int A[], int N) {
  for (int i = 0; i < N; i++) {
    if(!(i==0 || i==N-1)) {
      if (A[i] < 5) {
        A[i+1] = A[i] + 1;
        A[i] = A[i-1];
      }
    } else {
      A[i] = 5;
    }
  }
  assert(for k in 0..N-1, A[k]>=5);
}
```

# Motivating Example

```
void foo(int A[], int N) {
  for (int i = 0; i < N; i++) {
    if(!(i==0 || i==N-1)) {
      if (A[i] < 5) {
        A[i+1] = A[i] + 1;
        A[i] = A[i-1];
      }
    } else {
      A[i] = 5;
    }
  }
  assert(for k in 0..N-1, A[k]>=5);
}
```

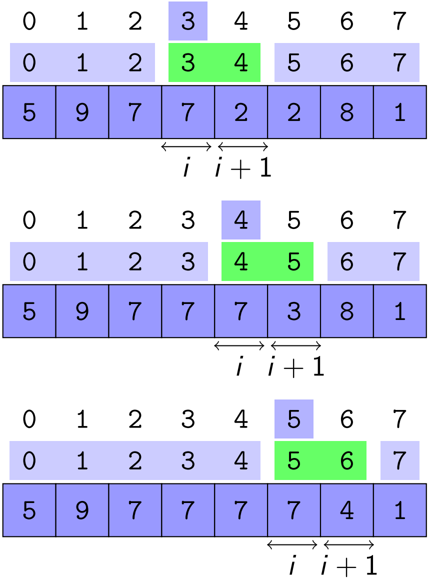
Initial array

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | Loop Counter |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | Indices |
| 5 | 9 | 7 | 1 | 9 | 2 | 8 | 1 | Cell Contents |

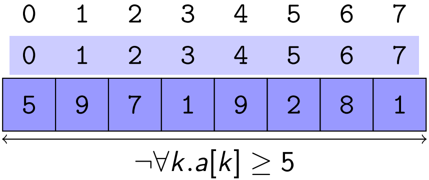
$\neg \forall k . a[k] \geq 5$

# Motivating Example

```
void foo(int A[], int N) {
  for (int i = 0; i < N; i++) {
    if(!(i==0 || i==N-1)) {
      if (A[i] < 5) {
        A[i+1] = A[i] + 1;
        A[i] = A[i-1];
      }
    } else {
      A[i] = 5;
    }
  }
  assert(for k in 0..N-1, A[k]>=5);
}
```




Initial array



# Motivating Example

```
void foo(int A[], int N) {
  for (int i = 0; i < N; i++) {
    if(!(i==0 || i==N-1)) {
      if (A[i] < 5) {
        A[i+1] = A[i] + 1;
        A[i] = A[i-1];
      }
    } else {
      A[i] = 5;
    }
  }
  assert(for k in 0..N-1, A[k]>=5);
}
```

Initial array

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 5 | 9 | 7 | 1 | 9 | 2 | 8 | 1 |

$\neg\forall k.a[k] \geq 5$

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 5 | 9 | 7 | 7 | 2 | 2 | 8 | 1 |
| | | | $i$ | | | | |

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 5 | 9 | 7 | 7 | 7 | 3 | 8 | 1 |
| | | | | $i$ | | | |

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 5 | 9 | 7 | 7 | 7 | 7 | 4 | 1 |
| | | | | | $i$ | | |

# Tiling an Array

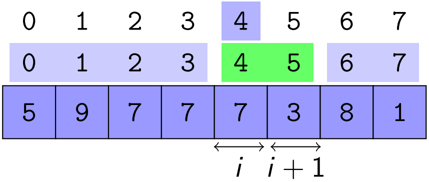
A Tile identifies the region of the array where the contribution of a generic loop iteration is localized

Tile : LoopCounter $\times$ Indices $\rightarrow \{\mathbf{tt}, \mathbf{ff}\}$ for loop L

# Tiling an Array

A Tile identifies the region of the array where the contribution of a generic loop iteration is localized

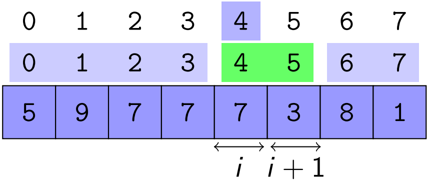
Tile : LoopCounter $\times$ Indices $\rightarrow \{\mathbf{tt}, \mathbf{ff}\}$ for loop L



# Tiling an Array

A Tile identifies the region of the array where the contribution of a generic loop iteration is localized

Tile : LoopCounter $\times$ Indices $\rightarrow \{\mathbf{tt}, \mathbf{ff}\}$ for loop L

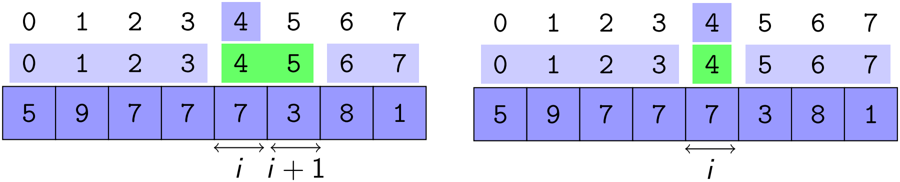



- $\text{Tile}(i, j) := i \leq j \leq i + 1$

# Tiling an Array

A Tile identifies the region of the array where the contribution of a generic loop iteration is localized

Tile : LoopCounter $\times$ Indices $\rightarrow \{\mathbf{tt}, \mathbf{ff}\}$ for loop L

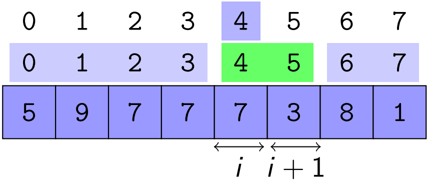



- $\text{Tile}(i, j) := i \leq j \leq i + 1$

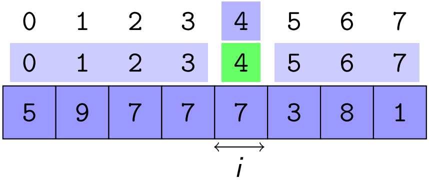
# Tiling an Array

A Tile identifies the region of the array where the contribution of a generic loop iteration is localized

Tile : LoopCounter $\times$ Indices $\rightarrow \{\mathbf{tt}, \mathbf{ff}\}$ for loop L




- $\text{Tile}(i,j) := i \leq j \leq i+1$




- $\text{Tile}(i,j) := j == i$

# Motivation for Tiling

Quantified Hoare Proof Obligation

Hard using existing tools

Quantifier Free Hoare Proof Obligation $i-1$

Quantifier Free Hoare Proof Obligation $i$

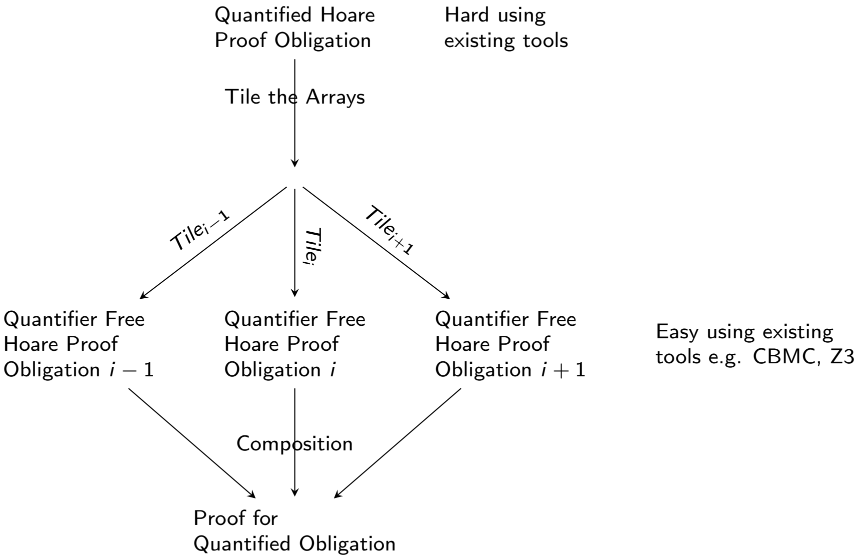
Quantifier Free Hoare Proof Obligation $i+1$

Easy using existing tools e.g. CBMC, Z3

Composition

Proof for Quantified Obligation

# Motivation for Tiling



# Verification by Tiling

Verify quantified post-conditions over arrays of parametric size

- Programs contain complex array access expressions in loops
- Use candidate quantified invariants
- Use black box back-ends such as SMT solver and BMC's

# Verification by Tiling

Verify quantified post-conditions over arrays of parametric size

- Programs contain complex array access expressions in loops
- Use candidate quantified invariants
- Use black box back-ends such as SMT solver and BMC's

*Inductive Compositional Reasoning*

- *Infer* array access patterns in loops
- *Tile* the set of indices using the inferred patterns
- *Slice* the assertion using the tile for a single iteration of the loop
  - Generates quantifier free hoare proof obligations for verification
- Compositionally *prove* universally quantified assertions on arrays
  - Composes verified quantifier free hoare proof obligations

# Heuristic Tile Generation

```
void foo(int A[], int N) {
  int j;
  for (int i = 0; i < N; i++) {
    if(!(i==0 || i==N-1)) {
      if (A[i] < 5) {
        A[i+1] = A[i] + 1;
        A[i] = A[i-1];
      }
    } else {
      A[i] = 5;
    }
    if(*) { j=i;   }
    if(*) { j=i+1; }
  }
  assert(for k in 0..N-1, A[k]>=5);
}
```

## Using array access patterns

- Store values of updated indices (say in $j$)
- Use arithmetic invariant generators to infer a relation between $i$ and $j$
- Infer $\text{Tile}(i, j) := i \leq j \leq i + 1$
- Remove overlapping indices $\text{Tile}(i, j) := j == i$

# Syntactic Restrictions on Programs

$$
\begin{array}{rcl}
\text{PB} & ::= & \text{St} \\
\text{St} & ::= & v := \text{E} \mid A[\text{E}] := \text{E} \mid \textbf{assume}(\text{BoolE}) \mid \\
 & & \textbf{if}(\text{BoolE})\ \textbf{then}\ \text{St}\ \textbf{else}\ \text{St} \mid \\
 & & \textbf{for}\ (\ell := 0; \ell < \text{E}; \ell := \ell + 1)\ \{\text{St}\} \mid \\
 & & \text{St}\ ;\ \text{St} \\
\text{E} & ::= & \text{E op E} \mid A[\text{E}] \mid v \mid \ell \mid c \\
\text{BoolE} & ::= & \text{E relop E} \mid \text{BoolE AND BoolE} \mid \\
 & & \text{NOT BoolE} \mid \text{BoolE OR BoolE}
\end{array}
$$

- No unstructured jumps
- Loop counter goes from 0 to some max value
- Assignment statements in body do not update loop counter

# Formalization

- Notation
  - $I$ denotes a sequence of array index variables
  - $\mathcal{A}$ is a set of array variables
  - Inv is a (possibly weak) loop invariant for loop L

# Formalization

- Notation
  - $I$ denotes a sequence of array index variables
  - $\mathcal{A}$ is a set of array variables
  - Inv is a (possibly weak) loop invariant for loop L
- Example Post-conditions/assertions
  - $\forall i$ between 0 and $N$, $A[i]$ is greater equal to minimum
  - $\forall i$ if $i$ is even & between 0 and $N$ then $A[i] = i$

# Formalization

- Notation
  - $I$ denotes a sequence of array index variables
  - $\mathcal{A}$ is a set of array variables
  - Inv is a (possibly weak) loop invariant for loop L
- Example Post-conditions/assertions
  - $\forall i$ between 0 and $N$, $A[i]$ is greater equal to minimum
  - $\forall i$ if $i$ is even & between 0 and $N$ then $A[i] = i$
- Formalization of Post-conditions
  - Post $\triangleq \forall I\, (\Phi(I) \implies \Psi(\mathcal{A}, I))$
  - $\Phi(I)$ - quantifier-free formula in theory of arithmetic over integers
  - $\Psi(\mathcal{A}, I)$ - quantifier-free formula in combined theory of arrays and arithmetic over integers

# Proving Assertions using Tiles

If following conditions hold on the tile, we have proven the property

T1: Covers Range relevant to property

T2: Sliced post-condition holds inductively

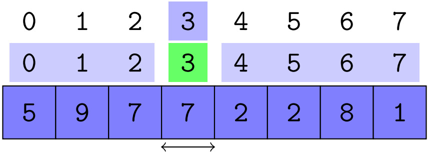
T3: Non-interference across tiles

# T1: Covers Range relevant to Property

Indices of interest must be covered by some *tile*

# T1: Covers Range relevant to Property

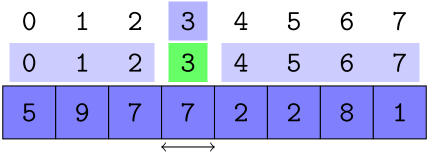
Indices of interest must be covered by some *tile*




$$\text{Tile}(i, j) := (j == i)$$

# T1: Covers Range relevant to Property

Indices of interest must be covered by some *tile*




$$\mathrm{Tile}(i, j) := (j == i)$$

- $\eta_1 \equiv \forall j\,(\Phi(j) \implies \exists i\,(\mathrm{Tile}(i, j)))$

# T1: Covers Range relevant to Property

Indices of interest must be covered by some *tile*

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 5 | 9 | 7 | 7 | 2 | 2 | 8 | 1 |

$$\mathrm{Tile}(i,j) := (j == i)$$

- $\eta_1 \equiv \forall j\,(\Phi(j) \implies \exists i\,(\mathrm{Tile}(i,j)))$
- $\eta_2 \equiv \forall i,j\,(\mathrm{Tile}(i,j) \implies \Phi(j))$

# T1: Covers Range relevant to Property

Indices of interest must be covered by some *tile*

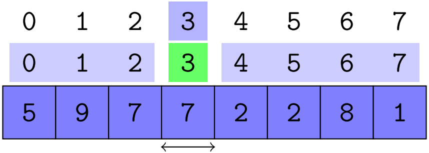
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 5 | 9 | 7 | 7 | 2 | 2 | 8 | 1 |

$$\text{Tile}(i, j) := (j == i)$$

- $\eta_1 \;\equiv\; \forall j\,(\Phi(j) \implies \exists i\,(\text{Tile}(i,j)))$
- $\eta_2 \;\equiv\; \forall i, j\,(\text{Tile}(i,j) \implies \Phi(j))$
- Validity of $\eta_1 \wedge \eta_2$ ensures T1

# T1: Covers Range relevant to Property

Indices of interest must be covered by some *tile*




$\mathrm{Tile}(i, j) := (j == i)$

- $\eta_1 \equiv \forall j\,(\Phi(j) \implies \exists i\,(\mathrm{Tile}(i, j)))$
- $\eta_2 \equiv \forall i, j\,(\mathrm{Tile}(i, j) \implies \Phi(j))$
- Validity of $\eta_1 \wedge \eta_2$ ensures T1
- Involves a quantifier alternation; can be handled by SMT solvers

# T1: Covers Range relevant to Property

- $\neg(\eta_1 \wedge \eta_2)$ must be unsat

## T1: Covers Range relevant to Property

- $\neg(\eta_1 \wedge \eta_2)$ must be unsat
- Negated smt formula is as shown below

```
(declare-fun size () Int)
(declare-fun i () Int)
(declare-fun j () Int)
(assert (or
    (and (>= j 0) (< j size)
        (forall ((i Int))
            (=> (and (>= i 0) (< i size)) (not (= j i)) )))
    (and (>= i 0) (< i size) (= j i)
        (not (and (>= j 0) (< j size))))))
(check-sat)
```

# T1: Covers Range relevant to Property

- $\neg(\eta_1 \wedge \eta_2)$ must be unsat
- Negated smt formula is as shown below

```
(declare-fun size () Int)
(declare-fun i () Int)
(declare-fun j () Int)
(assert (or
    (and (>= j 0) (< j size)
        (forall ((i Int))
            (=> (and (>= i 0) (< i size)) (not (= j i)) )))
    (and (>= i 0) (< i size) (= j i)
        (not (and (>= j 0) (< j size))))))
(check-sat)
```

- State-of-the-art solvers can prove unsatisfiability of such formulae

# T1: Covers Range relevant to Property

Indices of interest must be covered by some *tile*

```
void foo(int A[], int N) {
  for (int i = 0; i < N; i++) {
    if(!(i==0 || i==N-1)) {
      if (i%2==0 && A[i] < 5) {
        A[i+2] = A[i] + 1;
        A[i] = A[i-2];
      }
    } else {
      A[i] = 5;
    }
  }
  assert(for k in 0..N-1 && k%2==0, A[k]>=5);
}
```

# T1: Covers Range relevant to Property

Indices of interest must be covered by some *tile*

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 5 | 9 | 7 | 7 | 2 | 2 | 8 | 1 |

```
void foo(int A[], int N) {
  for (int i = 0; i < N; i++) {
    if(!(i==0 || i==N-1)) {
      if (i%2==0 && A[i] < 5) {
        A[i+2] = A[i] + 1;
        A[i] = A[i-2];
      }
    } else {
      A[i] = 5;
    }
  }
  assert(for k in 0..N-1 && k%2==0, A[k]>=5);
}
```

# T1: Covers Range relevant to Property

Indices of interest must be covered by some *tile*

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 5 | 9 | 7 | 7 | 2 | 2 | 8 | 1 |

```
void foo(int A[], int N) {
  for (int i = 0; i < N; i++) {
    if(!(i==0 || i==N-1)) {
      if (i%2==0 && A[i] < 5) {
        A[i+2] = A[i] + 1;
        A[i] = A[i-2];
      }
    } else {
      A[i] = 5;
    }
  }
  assert(for k in 0..N-1 && k%2==0, A[k]>=5);
}
```

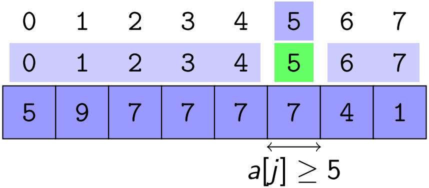
- Tiles need not cover full-range of indices
- Non-compact tiles are allowed
- Tiles cover range relevant to the property

# T2: Sliced Post-condition holds Inductively

Post-condition wrt indices in the $i^{th}$ tile holds inductively

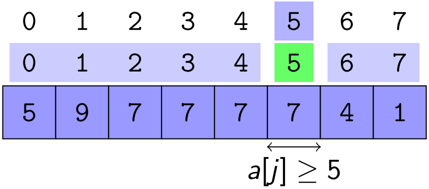
# T2: Sliced Post-condition holds Inductively

Post-condition wrt indices in the $i^{th}$ tile holds inductively



# T2: Sliced Post-condition holds Inductively

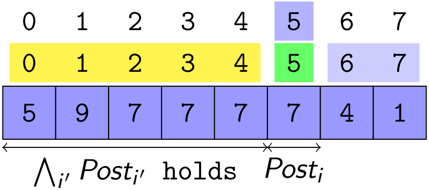
Post-condition wrt indices in the $i^{th}$ tile holds inductively




- Sliced post-condition for the $i^{th}$ tile
  $\text{Post}_i \triangleq \forall j\,(\text{Tile}(i,j) \wedge \Phi(j) \implies \Psi(\text{A},j))$

# T2: Sliced Post-condition holds Inductively

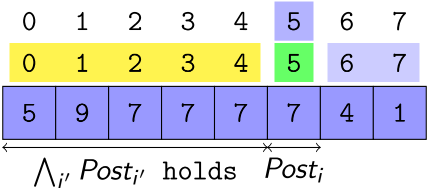
Post-condition wrt indices in the $i^{th}$ tile holds inductively




- Sliced post-condition for the $i^{th}$ tile
  $\mathrm{Post}_i \triangleq \forall j\, (\mathrm{Tile}(i, j) \wedge \Phi(j) \implies \Psi(\mathrm{A}, j))$

# T2: Sliced Post-condition holds Inductively

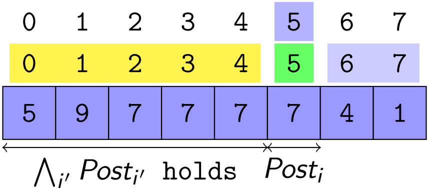
Post-condition wrt indices in the $i^{th}$ tile holds inductively




- Sliced post-condition for the $i^{th}$ tile
  $\text{Post}_i \triangleq \forall j\,(\text{Tile}(i,j) \wedge \Phi(j) \implies \Psi(\text{A}, j))$
- $\{\text{Inv} \wedge \bigwedge_{i':0\leq i'<i} \text{Post}_{i'}\}\ \text{L}_{\text{body}}\ \{\text{Inv} \wedge \text{Post}_i\}$ must be valid

# T2: Sliced Post-condition holds Inductively

Post-condition wrt indices in the $i^{th}$ tile holds inductively



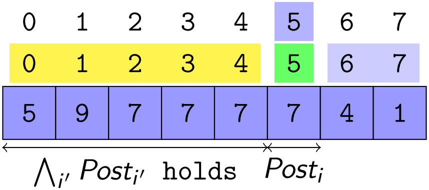

- Sliced post-condition for the $i^{th}$ tile
  $\mathrm{Post}_i \triangleq \forall j\,(\mathrm{Tile}(i,j) \wedge \Phi(j) \implies \Psi(\mathrm{A}, j))$

- $\{\mathrm{Inv} \wedge \bigwedge_{i':0 \leq i' < i} \mathrm{Post}_{i'}\}\ \mathrm{L}_{\mathrm{body}}\ \{\mathrm{Inv} \wedge \mathrm{Post}_i\}$ must be valid

- $\mathrm{Post}_i$ and $\bigwedge_{i':0 \leq i' < i} \mathrm{Post}_{i'}$ are universally quantified formula

# T2: Sliced Post-condition holds Inductively

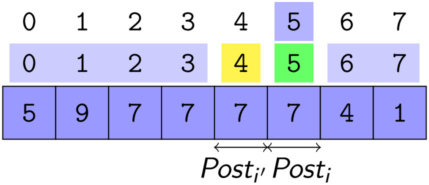
Post-condition wrt indices in the $i^{th}$ tile holds inductively




- Sliced post-condition for the $i^{th}$ tile
  $\text{Post}_i \triangleq \forall j\,(\text{Tile}(i,j) \wedge \Phi(j) \implies \Psi(\text{A}, j))$
- $\{\text{Inv} \wedge \bigwedge_{i':0\leq i'<i} \text{Post}_{i'}\}\ \text{L}_{\text{body}}\ \{\text{Inv} \wedge \text{Post}_i\}$ must be valid
- $\text{Post}_i$ and $\bigwedge_{i':0\leq i'<i} \text{Post}_{i'}$ are universally quantified formula
- Hoare logic-based reasoning tools that permit quantification have limited automation and scalability

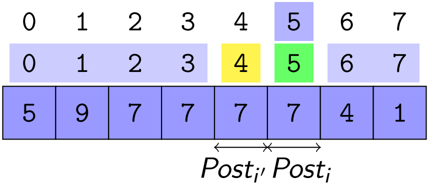
# T2*: Sliced Post-condition holds Inductively

Post-condition wrt indices in the $i^{th}$ tile holds inductively



# T2*: Sliced Post-condition holds Inductively

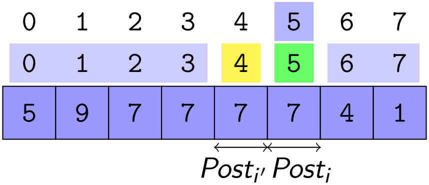
Post-condition wrt indices in the $i^{th}$ tile holds inductively




- $\mathrm{L}_{\mathrm{body}}$ is a loop free fragment of the code

# T2*: Sliced Post-condition holds Inductively

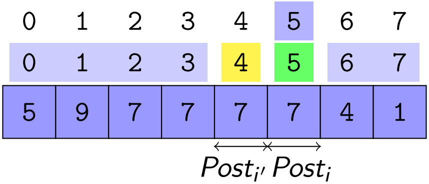
Post-condition wrt indices in the $i^{th}$ tile holds inductively




- $\mathsf{L}_{\mathrm{body}}$ is a loop free fragment of the code
- $Rd_{\mathsf{L}}(i) = \{i, i-1\}$ - Finite set of indices read in $i^{th}$ iteration

# T2*: Sliced Post-condition holds Inductively

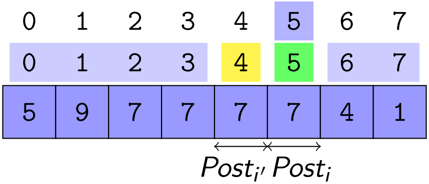
Post-condition wrt indices in the $i^{th}$ tile holds inductively




- $\mathsf{L}_{\mathrm{body}}$ is a loop free fragment of the code
- $Rd_{\mathsf{L}}(i) = \{i, i-1\}$ - Finite set of indices read in $i^{th}$ iteration
- $\zeta(i) := \bigwedge_{e_k \in Rd_{\mathsf{L}}(i)} \big( ((0 \leq i' < i) \wedge \mathsf{Tile}(i', e_k) \wedge \Phi(e_k)) \Rightarrow \Psi(\mathsf{A}, e_k) \big)$

# T2*: Sliced Post-condition holds Inductively

Post-condition wrt indices in the $i^{th}$ tile holds inductively



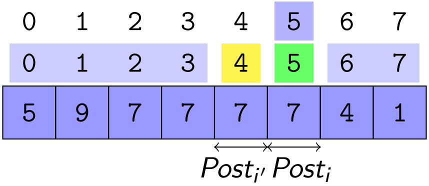

- $\mathsf{L}_{\text{body}}$ is a loop free fragment of the code
- $Rd_{\mathsf{L}}(i) = \{i, i-1\}$ - Finite set of indices read in $i^{th}$ iteration
- $\zeta(i) := \bigwedge_{e_k \in Rd_{\mathsf{L}}(i)} \big( ((0 \leq i' < i) \wedge \mathsf{Tile}(i', e_k) \wedge \Phi(e_k)) \Rightarrow \Psi(\mathsf{A}, e_k) \big)$
- $\{\mathsf{Inv} \wedge \zeta(i) \wedge \mathsf{Tile}(i, j) \wedge \Phi(j)\} \mathsf{L}_{\text{body}} \{\mathsf{Inv} \wedge \Psi(\mathsf{A}, j)\}$ must be valid

# T2*: Sliced Post-condition holds Inductively

Post-condition wrt indices in the $i^{th}$ tile holds inductively

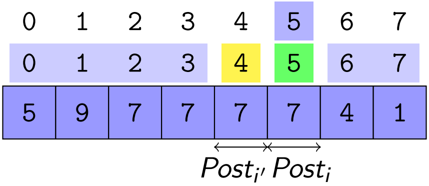
# T2*: Sliced Post-condition holds Inductively

Post-condition wrt indices in the $i^{th}$ tile holds inductively



# T2*: Sliced Post-condition holds Inductively

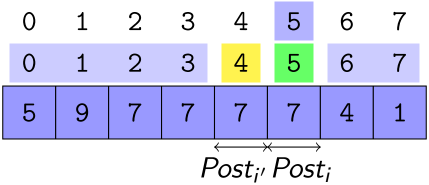
Post-condition wrt indices in the $i^{th}$ tile holds inductively




T2 - $\{\mathsf{Inv} \wedge \bigwedge_{i':0 \leq i' < i} \mathsf{Post}_{i'}\}\ \mathsf{L}_{\text{body}}\ \{\mathsf{Inv} \wedge \mathsf{Post}_i\}$

# T2*: Sliced Post-condition holds Inductively

Post-condition wrt indices in the $i^{th}$ tile holds inductively

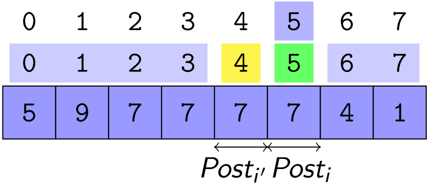



T2 - $\{\mathsf{Inv} \wedge \bigwedge_{i':0\leq i'<i} \mathsf{Post}_{i'}\}\ \mathsf{L}_{\text{body}}\ \{\mathsf{Inv} \wedge \mathsf{Post}_i\}$

T2* - $\{\mathsf{Inv} \wedge \zeta(i) \wedge \mathsf{Tile}(i,j) \wedge \Phi(j)\}\mathsf{L}_{\text{body}}\{\mathsf{Inv} \wedge \Psi(\mathsf{A},j)\}$

# T2*: Sliced Post-condition holds Inductively

Post-condition wrt indices in the $i^{th}$ tile holds inductively




T2 - $\{\mathsf{Inv} \wedge \bigwedge_{i':0\leq i'<i} \mathsf{Post}_{i'}\}\ \mathsf{L}_{\text{body}}\ \{\mathsf{Inv} \wedge \mathsf{Post}_i\}$

T2* - $\{\mathsf{Inv} \wedge \zeta(i) \wedge \mathsf{Tile}(i,j) \wedge \Phi(j)\}\mathsf{L}_{\text{body}}\{\mathsf{Inv} \wedge \Psi(\mathsf{A},j)\}$

- ⋆ T2* $\Rightarrow$ T2 and T2 $\not\Rightarrow$ T2*
- ⋆ T2* is now a quantifier free formula and can be checked using bmc

# T2*: Sliced Post-condition holds Inductively

Original Program

Transformed Program

# T2*: Sliced Post-condition holds Inductively

Original Program

```
void foo(int A[], int N) {
  for (int i = 0; i < N; i++)
  {

    if(!(i==0 || i==N-1)) {
      if (A[i] < 5) {
        A[i+1] = A[i] + 1;
        A[i] = A[i-1];
      }
    } else {
      A[i] = 5;
    }

  }
  assert(for k in 0..N-1, A[k]>=5);
}
```

Transformed Program

```
i=*;  j=*;  jp=*;
assume(0 <= i < N);
assume(j == i);
assume(jp == i-1);
assume(A[jp] >= 5);

if(!(i==0 || i==N-1)) {
  if (A[i] < 5) {
    A[i+1] = A[i] + 1;
    A[i] = A[i-1];
  }
} else {
  A[i] = 5;
}

assert(A[j] >= 5);
```

# T2*: Sliced Post-condition holds Inductively

Original Program

```
void foo(int A[], int N) {
  for (int i = 0; i < N; i++)
  {

    if(!(i==0 || i==N-1)) {
      if (A[i] < 5) {
        A[i+1] = A[i] + 1;
        A[i] = A[i-1];
      }
    } else {
      A[i] = 5;
    }

  }
  assert(for k in 0..N-1, A[k]>=5);
}
```

Transformed Program

```
i=*;  j=*;  jp=*;
assume(0 <= i < N);
assume(j == i);
assume(jp == i-1);
assume(A[jp] >= 5);

if(!(i==0 || i==N-1)) {
  if (A[i] < 5) {
    A[i+1] = A[i] + 1;
    A[i] = A[i-1];
  }
} else {
  A[i] = 5;
}

assert(A[j] >= 5);
```

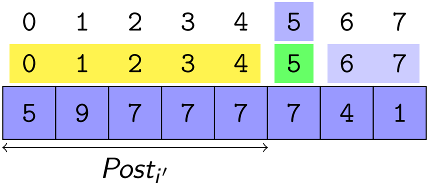
- Use CBMC to ensure T2* by checking the loop free code

# T3: Non-interference(property) across Tiles

No iteration $i > i'$ interferes with the truth of $Post_{i'}$, once established

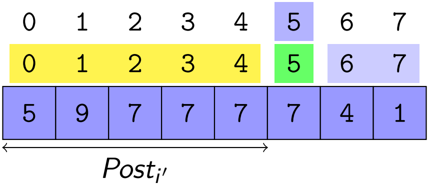
# T3: Non-interference(property) across Tiles

No iteration $i > i'$ interferes with the truth of $Post_{i'}$, once established



# T3: Non-interference(property) across Tiles

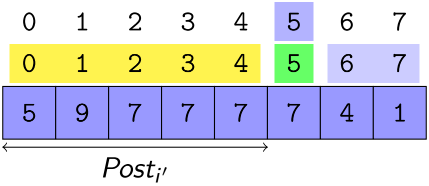
No iteration $i > i'$ interferes with the truth of $Post_{i'}$, once established




- Sliced post-condition for the $i'^{th}$ tile
  $\mathrm{Post}_{i'} \triangleq \forall j' \, (\mathrm{Tile}(i', j') \wedge \Phi(j') \implies \Psi(\mathrm{A}, j'))$

# T3: Non-interference(property) across Tiles

No iteration $i > i'$ interferes with the truth of $Post_{i'}$, once established




- Sliced post-condition for the $i'^{th}$ tile
  $\mathsf{Post}_{i'} \triangleq \forall j' \, (\mathsf{Tile}(i', j') \wedge \Phi(j') \implies \Psi(\mathsf{A}, j'))$
- $\{\mathsf{Inv} \wedge \bigwedge_{i' : 0 \leq i' < i} \mathsf{Post}_{i'}\} \mathrm{L}_{\mathrm{body}} \{\bigwedge_{i' : 0 \leq i' < i} \mathsf{Post}_{i'}\}$ must be valid

# T3: Non-interference(property) across Tiles

No iteration $i > i'$ interferes with the truth of $Post_{i'}$, once established

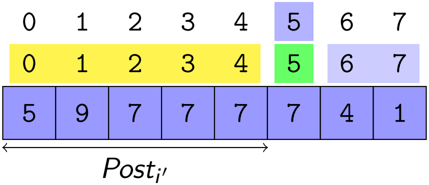
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 5 | 9 | 7 | 7 | 7 | 7 | 4 | 1 |

$\xleftrightarrow{\hspace{6em}}$ $Post_{i'}$

- Sliced post-condition for the $i'^{th}$ tile
  $\mathrm{Post}_{i'} \triangleq \forall j'\,(\mathrm{Tile}(i', j') \wedge \Phi(j') \implies \Psi(A, j'))$
- $\{\mathrm{Inv} \wedge \bigwedge_{i':0 \le i' < i} \mathrm{Post}_{i'}\}\mathrm{L}_{\mathrm{body}}\{\bigwedge_{i':0 \le i' < i} \mathrm{Post}_{i'}\}$ must be valid
- $\mathrm{Post}_i$ and $\bigwedge_{i':0 \le i' < i} \mathrm{Post}_{i'}$ are universally quantified formula

# T3: Non-interference(property) across Tiles

No iteration $i > i'$ interferes with the truth of $Post_{i'}$, once established



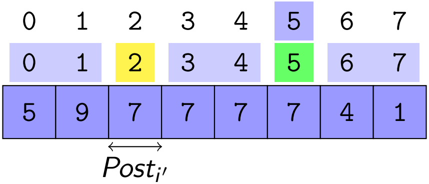

- Sliced post-condition for the $i'^{th}$ tile
  $\mathrm{Post}_{i'} \triangleq \forall j' \, (\mathrm{Tile}(i', j') \wedge \Phi(j') \implies \Psi(A, j'))$
- $\{\mathrm{Inv} \wedge \bigwedge_{i':0 \leq i' < i} \mathrm{Post}_{i'}\} \mathrm{L}_{\mathrm{body}} \{\bigwedge_{i':0 \leq i' < i} \mathrm{Post}_{i'}\}$ must be valid
- $\mathrm{Post}_i$ and $\bigwedge_{i':0 \leq i' < i} \mathrm{Post}_{i'}$ are universally quantified formula
- Hoare logic-based reasoning tools that permit quantification have limited automation and scalability

# T3*: Non-interference(property) across Tiles

No iteration $i > i'$ interferes with the truth of $Post_{i'}$, once established

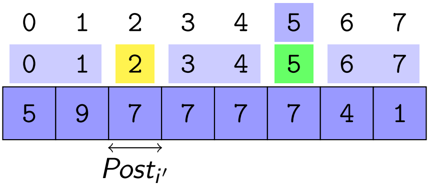
# T3*: Non-interference(property) across Tiles

No iteration $i > i'$ interferes with the truth of $Post_{i'}$, once established



# T3*: Non-interference(property) across Tiles

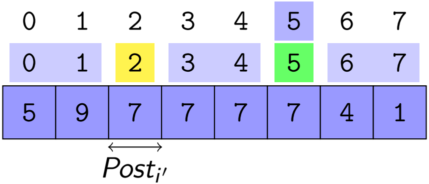
No iteration $i > i'$ interferes with the truth of $Post_{i'}$, once established




T3 - $\{\mathsf{Inv} \wedge \bigwedge_{i':0\leq i'<i} \mathsf{Post}_{i'}\}\mathsf{L}_{\mathrm{body}}\{\bigwedge_{i':0\leq i'<i} \mathsf{Post}_{i'}\}$

# T3*: Non-interference(property) across Tiles

No iteration $i > i'$ interferes with the truth of $Post_{i'}$, once established



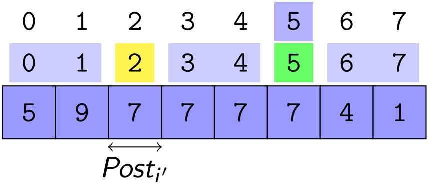

T3 - $\{\mathsf{Inv} \wedge \bigwedge_{i':0\le i'<i} \mathsf{Post}_{i'}\}\mathsf{L}_{\mathrm{body}}\{\bigwedge_{i':0\le i'<i} \mathsf{Post}_{i'}\}$

T3* - $\{\mathsf{Inv} \wedge (0 \le i' < i) \wedge \mathsf{Tile}(i', j') \wedge \Phi(j') \wedge \Psi(\mathsf{A}, j')\}\ \mathsf{L}_{\mathrm{body}}\ \{\Psi(\mathsf{A}, j')\}$

$i'$ and $j'$ be free variables here

# T3*: Non-interference(property) across Tiles

No iteration $i > i'$ interferes with the truth of $Post_{i'}$, once established




T3 - $\{\mathsf{Inv} \wedge \bigwedge_{i':0\leq i'<i} \mathsf{Post}_{i'}\}\mathsf{L}_{\mathrm{body}}\{\bigwedge_{i':0\leq i'<i} \mathsf{Post}_{i'}\}$

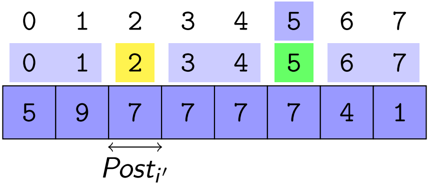
T3* - $\{\mathsf{Inv} \wedge (0 \leq i' < i) \wedge \mathsf{Tile}(i', j') \wedge \Phi(j') \wedge \Psi(\mathsf{A}, j')\}\ \mathsf{L}_{\mathrm{body}}\ \{\Psi(\mathsf{A}, j')\}$

$i'$ and $j'$ be free variables here

- $\star$ T3* $\Rightarrow$ T3 and T3 $\Rightarrow$ T3*

# T3*: Non-interference(property) across Tiles

No iteration $i > i'$ interferes with the truth of $Post_{i'}$, once established




T3 - $\{\mathsf{Inv} \wedge \bigwedge_{i':0 \leq i' < i} \mathsf{Post}_{i'}\} \mathsf{L}_{\text{body}} \{\bigwedge_{i':0 \leq i' < i} \mathsf{Post}_{i'}\}$

T3* - $\{\mathsf{Inv} \wedge (0 \leq i' < i) \wedge \mathsf{Tile}(i', j') \wedge \Phi(j') \wedge \Psi(\mathsf{A}, j')\}\ \mathsf{L}_{\text{body}}\ \{\Psi(\mathsf{A}, j')\}$

$i'$ and $j'$ be free variables here

- ⋆ T3* ⇒ T3 and T3 ⇒ T3*
- ⋆ T3* is now a quantifier free formula and can be checked using bmc

# T3*: Non-interference(property) across Tiles

Original Program

Transformed Program

# T3*: Non-interference(property) across Tiles

Original Program

```
void foo(int A[], int N) {
  for (int i = 0; i < N; i++)
  {

    if(!(i==0 || i==N-1)) {
      if (A[i] < 5) {
        A[i+1] = A[i] + 1;
        A[i] = A[i-1];
      }
    } else {
      A[i] = 5;
    }

  }
  assert(for k in 0..N-1, A[k]>=5);
}
```

Transformed Program

```
i=*;  ip=*;  jp=*;
assume(0 <= i < N);
assume(0 <= ip < i);
assume(jp == ip);
assume(A[jp] >= 5);

if(!(i==0 || i==N-1)) {
  if (A[i] < 5) {
    A[i+1] = A[i] + 1;
    A[i] = A[i-1];
  }
} else {
  A[i] = 5;
}

assert(A[jp] >= 5);
```

# T3*: Non-interference(property) across Tiles

Original Program

```
void foo(int A[], int N) {
  for (int i = 0; i < N; i++)
  {

    if(!(i==0 || i==N-1)) {
      if (A[i] < 5) {
        A[i+1] = A[i] + 1;
        A[i] = A[i-1];
      }
    } else {
      A[i] = 5;
    }

  }
  assert(for k in 0..N-1, A[k]>=5);
}
```

Transformed Program

```
i=*;  ip=*;  jp=*;
assume(0 <= i < N);
assume(0 <= ip < i);
assume(jp == ip);
assume(A[jp] >= 5);

if(!(i==0 || i==N-1)) {
  if (A[i] < 5) {
    A[i+1] = A[i] + 1;
    A[i] = A[i-1];
  }
} else {
  A[i] = 5;
}

assert(A[jp] >= 5);
```

- Use CBMC to ensure T3* by checking the loop free code

# Inductive Compositional Reasoning

- Inductive Reasoning

T2 Sliced post-condition holds for each iteration

# Inductive Compositional Reasoning

- Inductive Reasoning

  T2 Sliced post-condition holds for each iteration

- Compositional Reasoning

  T3 Truth of sliced post-condition once established is not altered subsequently

  T1 Tiles cover the entire range of array indices of interest

# Inductive Compositional Reasoning

- Inductive Reasoning

  T2 Sliced post-condition holds for each iteration

- Compositional Reasoning

  T3 Truth of sliced post-condition once established is not altered subsequently

  T1 Tiles cover the entire range of array indices of interest

**Theorem**

*Suppose* $\mathrm{Tile} : \mathrm{LoopCounter} \times \mathrm{Indices} \rightarrow \{\mathbf{tt}, \mathbf{ff}\}$ *satisfies T1, T2 and T3. If* $\mathrm{Pre} \Rightarrow \mathrm{Inv}$ *holds and the loop* $\mathrm{L}$ *iterates at least once, then the Hoare triple* $\{\mathrm{Pre}\}\ \mathrm{L}\ \{\mathrm{Post}\}$ *holds.*

# Proof.

The proof proceeds by induction on values of LoopCounter (say $i$).

***Given:***

$$\text{Pre} \Rightarrow \text{Inv} \tag{1}$$

***Base Case:***

Prove $\{\text{Pre}\}\ \ \text{L}_{\text{body}}\ \ \{\text{Post}_i\}$ holds, where $i = 0$

$$\{\text{Inv}\}\ \ \text{L}_{\text{body}}\ \ \{\text{Inv} \wedge \text{Post}_i\} \qquad (\because T2) \tag{2}$$

$$\{\text{Pre}\}\ \ \text{L}_{\text{body}}\ \ \{\text{Inv} \wedge \text{Post}_i\} \qquad (\because \textit{From } (1) \ \&\ (2)) \tag{3}$$

$$\{\text{Pre}\}\ \ \text{L}_{\text{body}}\ \ \{\text{Post}_i\} \qquad (\because \text{Inv} \wedge \{\text{Post}_i\} \Rightarrow \{\text{Post}_i\}) \tag{4}$$

***Induction Hypothesis:***

$$\{\text{Pre}\}\ \ (\text{L}_{\text{body}})^{i'}\ \ \{\bigwedge_{i':0 \le i' < i} \text{Post}_{i'}\} \qquad (\because T3) \tag{5}$$

# Proof. Cont...

***Induction:***

Assuming hypothesis, prove $\{\text{Pre}\}\ \ (\text{L}_{\text{body}})^{i'}\ \ \{\bigwedge_{i':0\leq i'\leq i} \text{Post}_{i'}\}$ holds.

$$\{\text{Inv} \wedge \bigwedge_{i':0\leq i'<i} \text{Post}_{i'}\}\ \ \text{L}_{\text{body}}\ \ \{\text{Inv} \wedge \text{Post}_i\} \qquad (\because T2) \quad (6)$$

$$\{\text{Inv} \wedge \bigwedge_{i':0\leq i'<i} \text{Post}_{i'}\}\ \ \text{L}_{\text{body}}\ \ \{\text{Post}_i\} \qquad (\because \text{Inv} \wedge \text{Post}_i \Rightarrow \text{Post}_i) \quad (7)$$

At the end of the $i^{th}$ iteration of the loop L the following Hoare triple holds:

$$\{\text{Pre}\}\ \ (\text{L}_{\text{body}})^{i'}\ \ \{\bigwedge_{i':0\leq i'\leq i} \text{Post}_{i'}\} \qquad (\because \textit{From } (5) \ \&\ (7)) \quad (8)$$

$$\bigwedge_i \text{Post}_i \equiv \text{Post} \qquad (\because T1) \quad (9)$$

$$\{\text{Pre}\}\ \ \text{L}\ \ \{\text{Post}\} \qquad (\because \textit{From } (8) \ \&\ (9)) \qquad \square$$

# Sequentially Composed Loops

```
void copynswap(int N)
{
 int i, tmp;
 int a[], b[], acopy[];

 for (i = 0; i < N; i++) {
   acopy[i] = a[i];
 }

 for (i = 0; i < N; i++) {
   tmp = a[i];
   a[i] = b[i];
   b[i] = tmp;
 }

 for (i = 0; i < N; i++) {
   assert(b[i] == acopy[i]);
 }
}
```

# Sequentially Composed Loops

```
void copynswap(int N)
{
 int i, tmp;
 int a[], b[], acopy[];

 for (i = 0; i < N; i++) {
   acopy[i] = a[i];
 }

 for (i = 0; i < N; i++) {
   tmp = a[i];
   a[i] = b[i];
   b[i] = tmp;
 }

 for (i = 0; i < N; i++) {
   assert(b[i] == acopy[i]);
 }
}
```

## Mid-conditions

- Invariants between sequentially composed loops
- Hard to generate precise invariants
- Identify *candidate* mid-conditions using annotation assistants

# Sequentially Composed Loops

```
void copynswap(int N)
{
 int i, tmp;
 int a[], b[], acopy[];

 for (i = 0; i < N; i++) {
   acopy[i] = a[i];
 }

 for (i = 0; i < N; i++) {
   tmp = a[i];
   a[i] = b[i];
   b[i] = tmp;
 }

 for (i = 0; i < N; i++) {
   assert(b[i] == acopy[i]);
 }
}
```

## Mid-conditions

- Invariants between sequentially composed loops
- Hard to generate precise invariants
- Identify *candidate* mid-conditions using annotation assistants

## *Candidate* mid-conditions

- $\forall i (a[i] = acopy[i])$
- $\forall i (a[i] \neq b[i])$

# Sequentially Composed Loops

```
void copynswap(int N)
{
 int i, tmp;
 int a[], b[], acopy[];

 for (i = 0; i < N; i++) {
   acopy[i] = a[i];
 }

 for (i = 0; i < N; i++) {
   tmp = a[i];
   a[i] = b[i];
   b[i] = tmp;
 }

 for (i = 0; i < N; i++) {
   assert(b[i] == acopy[i]);
 }
}
```

## Mid-conditions

- Invariants between sequentially composed loops
- Hard to generate precise invariants
- Identify *candidate* mid-conditions using annotation assistants
- *Prove* them using Tiling

## *Candidate* mid-conditions

- $\forall i(a[i] = acopy[i])$
- $\forall i(a[i] \neq b[i])$

## Proved mid-conditions

- $\forall i(a[i] = acopy[i])$

# Nested Loops

```
void nested(int N)
{
  int i, j, VAL=2, arr[];

  assume(N % 5 == 0);
  for(i = 1; i <= N/5; i++)
  {
    for(j = 1; j <= 5; j++)
    {
      if(j >= VAL)
        arr[i*5 - j] = j;
      else
        arr[i*5 - j] = 0;
    }
  }

  assert(for i in 0..N-1 (arr[i]>=VAL || arr[i]==0));
}
```

# Nested Loops

```
void nested(int N)
{
  int i, j, VAL=2, arr[];

  assume(N % 5 == 0);
  for(i = 1; i <= N/5; i++)
  {
    for(j = 1; j <= 5; j++)
    {
      if(j >= VAL)
        arr[i*5 - j] = j;
      else
        arr[i*5 - j] = 0;
    }
  }

  assert(for i in 0..N-1 (arr[i]>=VAL || arr[i]==0));
}
```

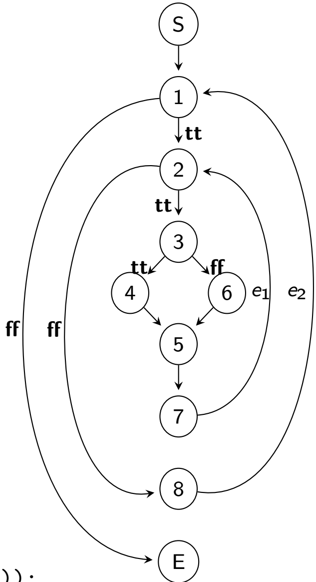



Figure: CFG

# CFG, Cutpoints and Segments

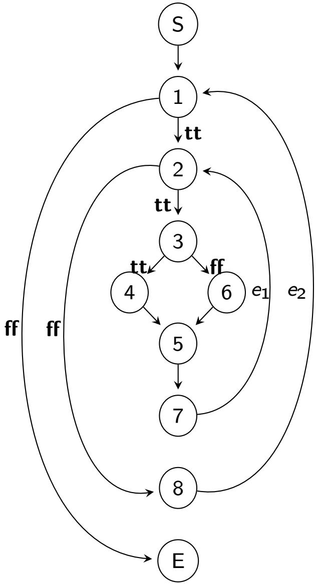



Figure: CFG

# CFG, Cutpoints and Segments

**Back-edge** - Edge from a node within the body of a loop to the node representing the corresponding loop head

- Edges $e_1$ and $e_2$

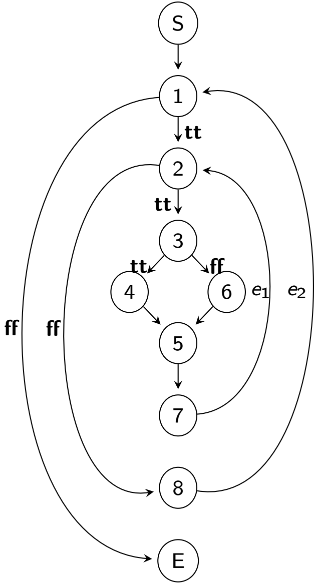



Figure: CFG

# CFG, Cutpoints and Segments

**Back-edge** - Edge from a node within the body of a loop to the node representing the corresponding loop head

- Edges $e_1$ and $e_2$

**Cut-Points** - Target nodes of back-edges and Start and End nodes

- Nodes 1, 2, $S$, $E$

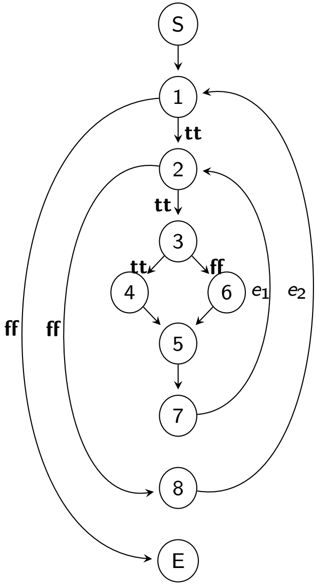



Figure: CFG

# CFG, Cutpoints and Segments

**Back-edge** - Edge from a node within the body of a loop to the node representing the corresponding loop head

- Edges $e_1$ and $e_2$

**Cut-Points** - Target nodes of back-edges and Start and End nodes

- Nodes 1, 2, $S$, $E$

**Segments** - Acyclic sub-graph of a CFG

- starts from a cut-point and ends at another cut-point
- does not pass through any other cut-point in between

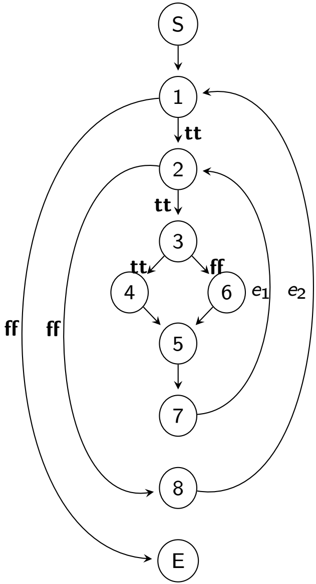



Figure: CFG

# CFG, Cutpoints and Segments

**Back-edge** - Edge from a node within the body of a loop to the node representing the corresponding loop head

- Edges $e_1$ and $e_2$

**Cut-Points** - Target nodes of back-edges and Start and End nodes

- Nodes 1, 2, $S$, $E$

**Segments** - Acyclic sub-graph of a CFG

- starts from a cut-point and ends at another cut-point
- does not pass through any other cut-point in between

$S \mapsto 1$, $1 \mapsto 2$,
$2 \mapsto 3 \mapsto \{4, 6\} \mapsto 5 \mapsto 7 \mapsto 2$,
$2 \mapsto 8 \mapsto 1$, $1 \mapsto E$ are segments

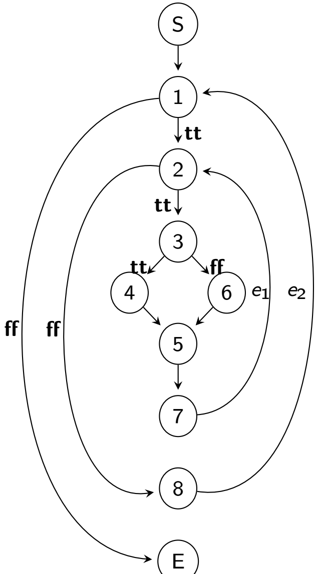



Figure: CFG

# Segment-based Verification

Analysis now applies to each segment in the topological order

- $c_1$ and $c_2$ be two cut-points
- Segment $s$ starts at $c_1$, ends at $c_2$
- $Inv_{c_1}$ and $Inv_{c_2}$ are candidate invariants at $c_1$ and $c_2$
- Generate tiles for each segment
- Check T1, T2*, T3* for each segment $s$
- Invariants for the nested loop example
  - $Inv_1 := \forall k(0 \leq k < 5 * i) \implies (arr[k] >= VAL || arr[k] == 0)$
  - $Inv_2 := \forall k(5*i-j \leq k < 5*i) \implies (arr[k] >= VAL || arr[k] == 0)$

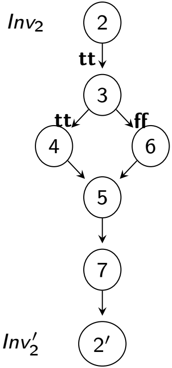


Figure: Verifying a Segment

# Tiler Tool Diagram

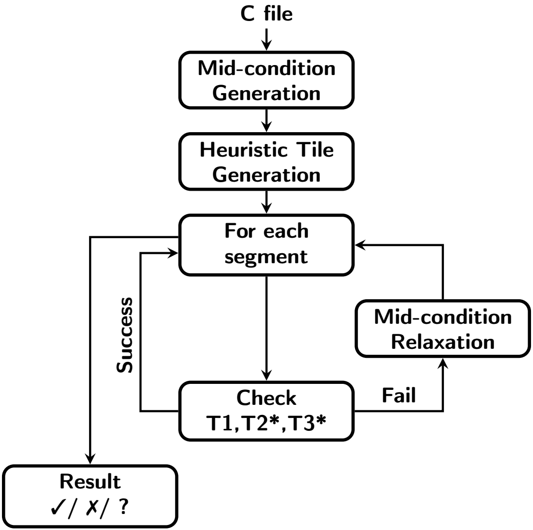



Figure: Tiler Tool Diagram

# Tiler Implementation

- Built on top of LLVM/CLANG infrastructure in C++
- Mid-condition generation
  - Daikon learns candidate scalar invariants from concrete traces
  - Lift these to quantified invariants
- Heuristic tile generation
  - Determine indices in terms of loop counters
  - Get a closed form expression in terms of index expressions
  - Remove possible overlaps
- Checking conditions T1, T2 and T3
  - Z3 for checking the validity of T1
  - CBMC for checking the validity of T2 and T3

# Tiler Benchmarking

- 60 benchmarks from industry and academia
- Performance compared with tools
  - SMACK+Corral - Bounded model checker
  - Booster - Acceleration based verification for arrays
  - Vaphor - Distinguished cell abstraction for arrays
- Memory limit - 1GB
- Time limit - 900s

# Tiler in Action

| Benchmark | #L | **Tiler** | S+C | Booster | Vaphor |
|---|---|---|---|---|---|
| cpynrev.c | 2 | ✓3.8 | † | ✓3.1 | ✓5.4 |
| cpynswp.c | 2 | ✓4.2 | † | ✓12.4 | ✓1.38 |
| cpynswp2.c | 3 | ✓10.2 | † | ✓198 | ✓7.2* |
| maxinarr.c | 1 | ✓0.51 | † | ✓0.01 | ✓0.11 |
| mininarr.c | 1 | ✓0.53 | † | ✓0.02 | ✓0.13 |
| poly1.c | 1 | TO | † | ✓15.7 | TO |
| poly2.c | 2 | **?** 6.44 | † | **?** 19.5 | TO |
| tcpy.c | 1 | **?** 0.65 | † | TO | ✓25.1 |
| rew.c | 1 | ✓0.48 | † | ✓0.01 | TO |
| skipped.c | 1 | ✓1.24 | † | TO | TO |
| rewrev.c | 1 | ✓0.39 | † | TO | TO |
| pr4.c | 1 | ✓0.68 | † | TO | TO |
| pr5.c | 1 | ✓1.32 | † | TO | TO |
| pnr4.c | 1 | ✓0.86 | † | TO | TO |
| pnr5.c | 1 | ✓1.98 | † | TO | TO |
| mbpr4.c | 4 | ✓12.75 | † | TO | TO |
| mbpr5.c | 5 | ✓18.08 | † | TO | TO |
| nr4.c | 1-1 | ✓2.43* | † | TO | TO |
| nr5.c | 1-1 | ✓2.90* | † | TO | TO |
| copy9u.c | 9 | ✗0.16 | ✗4.48 | ✗0.44 | ✗30.8 |
| skippedu.c | 1 | ✗0.81 | ✗2.94 | ✗0.02 | TO |

# Tiler in Action

| Benchmark | #L | **Tiler** | S+C | Booster | Vaphor |
|---|---|---|---|---|---|
| cpynrev.c | 2 | ✓3.8 | † | ✓3.1 | ✓5.4 |
| cpynswp.c | 2 | ✓4.2 | † | ✓12.4 | ✓1.38 |
| cpynswp2.c | 3 | ✓10.2 | † | ✓198 | ✓7.2* |
| maxinarr.c | 1 | ✓0.51 | † | ✓0.01 | ✓0.11 |
| mininarr.c | 1 | ✓0.53 | † | ✓0.02 | ✓0.13 |
| poly1.c | 1 | TO | † | ✓15.7 | TO |
| poly2.c | 2 | ? 6.44 | † | ? 19.5 | TO |
| tcpy.c | 1 | ? 0.65 | † | TO | ✓25.1 |
| rew.c | 1 | ✓0.48 | † | ✓0.01 | TO |
| skipped.c | 1 | ✓1.24 | † | TO | TO |
| rewrev.c | 1 | ✓0.39 | † | TO | TO |
| pr4.c | 1 | ✓0.68 | † | TO | TO |
| pr5.c | 1 | ✓1.32 | † | TO | TO |
| pnr4.c | 1 | ✓0.86 | † | TO | TO |
| pnr5.c | 1 | ✓1.98 | † | TO | TO |
| mbpr4.c | 4 | ✓12.75 | † | TO | TO |
| mbpr5.c | 5 | ✓18.08 | † | TO | TO |
| nr4.c | 1-1 | ✓2.43* | † | TO | TO |
| nr5.c | 1-1 | ✓2.90* | † | TO | TO |
| copy9u.c | 9 | ✗0.16 | ✗4.48 | ✗0.44 | ✗30.8 |
| skippedu.c | 1 | ✗0.81 | ✗2.94 | ✗0.02 | TO |

# Tiler in Action

| Benchmark | #L | **Tiler** | S+C | Booster | Vaphor |
|---|---|---|---|---|---|
| cpynrev.c | 2 | ✓3.8 | † | ✓3.1 | ✓5.4 |
| cpynswp.c | 2 | ✓4.2 | † | ✓12.4 | ✓1.38 |
| cpynswp2.c | 3 | ✓10.2 | † | ✓198 | ✓7.2* |
| maxinarr.c | 1 | ✓0.51 | † | ✓0.01 | ✓0.11 |
| mininarr.c | 1 | ✓0.53 | † | ✓0.02 | ✓0.13 |
| poly1.c | 1 | TO | † | ✓15.7 | TO |
| poly2.c | 2 | ? 6.44 | † | ? 19.5 | TO |
| tcpy.c | 1 | ? 0.65 | † | TO | ✓25.1 |
| rew.c | 1 | ✓0.48 | † | ✓0.01 | TO |
| skipped.c | 1 | ✓1.24 | † | TO | TO |
| rewrev.c | 1 | ✓0.39 | † | TO | TO |
| pr4.c | 1 | ✓0.68 | † | TO | TO |
| pr5.c | 1 | ✓1.32 | † | TO | TO |
| pnr4.c | 1 | ✓0.86 | † | TO | TO |
| pnr5.c | 1 | ✓1.98 | † | TO | TO |
| mbpr4.c | 4 | ✓12.75 | † | TO | TO |
| mbpr5.c | 5 | ✓18.08 | † | TO | TO |
| nr4.c | 1-1 | ✓2.43* | † | TO | TO |
| nr5.c | 1-1 | ✓2.90* | † | TO | TO |
| copy9u.c | 9 | ✗0.16 | ✗4.48 | ✗0.44 | ✗30.8 |
| skippedu.c | 1 | ✗0.81 | ✗2.94 | ✗0.02 | TO |

# Tiler in Action

| Benchmark | #L | **Tiler** | S+C | Booster | Vaphor |
|---|---|---|---|---|---|
| cpynrev.c | 2 | ✓3.8 | † | ✓3.1 | ✓5.4 |
| cpynswp.c | 2 | ✓4.2 | † | ✓12.4 | ✓1.38 |
| cpynswp2.c | 3 | ✓10.2 | † | ✓198 | ✓7.2* |
| maxinarr.c | 1 | ✓0.51 | † | ✓0.01 | ✓0.11 |
| mininarr.c | 1 | ✓0.53 | † | ✓0.02 | ✓0.13 |
| poly1.c | 1 | TO | † | ✓15.7 | TO |
| poly2.c | 2 | ? 6.44 | † | ? 19.5 | TO |
| tcpy.c | 1 | ? 0.65 | † | TO | ✓25.1 |
| rew.c | 1 | ✓0.48 | † | ✓0.01 | TO |
| skipped.c | 1 | ✓1.24 | † | TO | TO |
| rewrev.c | 1 | ✓0.39 | † | TO | TO |
| pr4.c | 1 | ✓0.68 | † | TO | TO |
| pr5.c | 1 | ✓1.32 | † | TO | TO |
| pnr4.c | 1 | ✓0.86 | † | TO | TO |
| pnr5.c | 1 | ✓1.98 | † | TO | TO |
| mbpr4.c | 4 | ✓12.75 | † | TO | TO |
| mbpr5.c | 5 | ✓18.08 | † | TO | TO |
| nr4.c | 1-1 | ✓2.43* | † | TO | TO |
| nr5.c | 1-1 | ✓2.90* | † | TO | TO |
| copy9u.c | 9 | ✗0.16 | ✗4.48 | ✗0.44 | ✗30.8 |
| skippedu.c | 1 | ✗0.81 | ✗2.94 | ✗0.02 | TO |

# Tiler in Action

| Benchmark | #L | **Tiler** | S+C | Booster | Vaphor |
|---|---|---|---|---|---|
| cpynrev.c | 2 | ✓3.8 | † | ✓3.1 | ✓5.4 |
| cpynswp.c | 2 | ✓4.2 | † | ✓12.4 | ✓1.38 |
| cpynswp2.c | 3 | ✓10.2 | † | ✓198 | ✓7.2* |
| maxinarr.c | 1 | ✓0.51 | † | ✓0.01 | ✓0.11 |
| mininarr.c | 1 | ✓0.53 | † | ✓0.02 | ✓0.13 |
| poly1.c | 1 | TO | † | ✓15.7 | TO |
| poly2.c | 2 | ? 6.44 | † | ? 19.5 | TO |
| tcpy.c | 1 | ? 0.65 | † | TO | ✓25.1 |
| rew.c | 1 | ✓0.48 | † | ✓0.01 | TO |
| skipped.c | 1 | ✓1.24 | † | TO | TO |
| rewrev.c | 1 | ✓0.39 | † | TO | TO |
| pr4.c | 1 | ✓0.68 | † | TO | TO |
| pr5.c | 1 | ✓1.32 | † | TO | TO |
| pnr4.c | 1 | ✓0.86 | † | TO | TO |
| pnr5.c | 1 | ✓1.98 | † | TO | TO |
| mbpr4.c | 4 | ✓12.75 | † | TO | TO |
| mbpr5.c | 5 | ✓18.08 | † | TO | TO |
| nr4.c | 1-1 | ✓2.43* | † | TO | TO |
| nr5.c | 1-1 | ✓2.90* | † | TO | TO |
| copy9u.c | 9 | ✗0.16 | ✗4.48 | ✗0.44 | ✗30.8 |
| skippedu.c | 1 | ✗0.81 | ✗2.94 | ✗0.02 | TO |

# Battery Voltage Regulator

```
void BVR(int N, int MIN)
{
  int i, volArray[N];

  if(N % 4 != 0) { return; }

  for(i = 1; i <= N/4; i++)
  {
    if(1 >= MIN)
      volArray[i*4-1] = 1;
    else
      volArray[i*4-1] = 0;
    if(3 >= MIN)
      volArray[i*4-2] = 3;
    else
      volArray[i*4-2] = 0;
    if(7 >= MIN)
      volArray[i*4-3] = 7;
    else
      volArray[i*4-3] = 0;
    if(5 >= MIN)
      volArray[i*4-4] = 5;
    else
      volArray[i*4-4] = 0;
  }

  for(i = 0; i < N; i++)
  {
    assert(volArray[i] >= MIN ||
      volArray[i] == 0);
  }
}
```

# Battery Voltage Regulator

```
void BVR(int N, int MIN)
{
  int i, volArray[N];

  if(N % 4 != 0) { return; }

  for(i = 1; i <= N/4; i++)
  {
    if(1 >= MIN)
      volArray[i*4-1] = 1;
    else
      volArray[i*4-1] = 0;
    if(3 >= MIN)
      volArray[i*4-2] = 3;
    else
      volArray[i*4-2] = 0;
    if(7 >= MIN)
      volArray[i*4-3] = 7;
    else
      volArray[i*4-3] = 0;
    if(5 >= MIN)
      volArray[i*4-4] = 5;
    else
      volArray[i*4-4] = 0;
  }

  for(i = 0; i < N; i++)
  {
    assert(volArray[i] >= MIN ||
      volArray[i] == 0);
  }
}
```

$$\text{Tile}(i, j) := 4 * i - 4 \leq j < 4 * i$$

# Skipped Indices

```
void skip(int N)
{
  int i;
  int a[N];

  if(N % 2 != 0)
  {
    return;
  }

  assume(N % 2 == 0);
  for(i = 1; i <= N/2; i++ )
  {
    if( a[2*i-2] > 2*i-2 )
    {
      a[2*i-2] = 2*i-2;
    }

    if( a[2*i-1] > 2*i-1 )
    {
      a[2*i-1] = 2*i-1;
    }
  }

  for(i = 0; i < N; i++)
  {
    assert(a[i] <= i);
  }
  return;
}
```

# Skipped Indices

```
void skip(int N)
{
  int i;
  int a[N];

  if(N % 2 != 0)
  {
    return;
  }

  assume(N % 2 == 0);
  for(i = 1; i <= N/2; i++ )
  {
    if( a[2*i-2] > 2*i-2 )
    {
      a[2*i-2] = 2*i-2;
    }

    if( a[2*i-1] > 2*i-1 )
    {
      a[2*i-1] = 2*i-1;
    }
  }

  for(i = 0; i < N; i++)
  {
    assert(a[i] <= i);
  }
  return;
}
```

$\mathrm{Tile}(i,j) := 2*i-2 \leq j < 2*i$

# Array Reversal

```
void revcopynswap(int N)
{
  int i;
  int tmp;
  int a[N];
  int b[N];
  int rev_copy[N];

  for(i = 0; i < N; i++)
  {
    rev_copy[N-i-1] = a[i];
  }
  for(i = 0; i < N; i++)
  {
    tmp = a[i];
    a[i] = b[i];
    b[i] = tmp;
  }

  for(i = 0; i < N; i++)
  {
    assert(b[i] == rev_copy[N-i-1]);
  }
}
```

# Array Reversal

```
void revcopynswap(int N)
{
  int i;
  int tmp;
  int a[N];
  int b[N];
  int rev_copy[N];

  for(i = 0; i < N; i++)
  {
    rev_copy[N-i-1] = a[i];
  }

  for(i = 0; i < N; i++)
  {
    tmp = a[i];
    a[i] = b[i];
    b[i] = tmp;
  }

  for(i = 0; i < N; i++)
  {
    assert(b[i] == rev_copy[N-i-1]);
  }
}
```

Loop 1 - $\text{Tile}(i, j) := j == N - i - 1$

Loop 2 - $\text{Tile}(i, j) := j == i$

# Tiles in Benchmarks

- **Reverse** the contents of the array
  - $\text{Tile}(i, j) := j == N - i - 1$
- A **bunch** of indices updated in a loop
  - $\text{Tile}(i, j) := 2 * i - 2 \leq j < 2 * i$
  - $\text{Tile}(i, j) := 3 * i - 3 \leq j < 3 * i$
  - $\text{Tile}(i, j) := 4 * i - 4 \leq j < 4 * i$
- **Adjacent** indices to the counter
  - $\text{Tile}(i, j) := j == i - 1$
  - $\text{Tile}(i, j) := j == i + 1$
- Most **common** tile in array processing loops
  - $\text{Tile}(i, j) := j == i$

# Tiler Limitations

```
void tcpy(int N)
{
  int i, a[N], reverse[N];

  if(N % 2 != 0)
  { return; }

  assume(N % 2 == 0);
  for (i = 0; i < N/2; i++)
  {
     reverse[i] = a[N-i-1];
     reverse[N-i-1] = a[i];
  }

  for(i = 0; i < N; i++)
  {
     assert(a[i] = reverse[N-i-1]);
  }
}
```

```
void poly2(int N)
{
  int i, a[N];

  for(i=0; i<N; i++)
  {
    a[i] = i*i + 2;
  }

  for(i=0; i<N; i++)
  {
    a[i] = a[i] - 2;
  }

  for(i=0; i<N; i++)
  {
    assert(a[i] == i*i);
  }
}
```

# Related Work

- Abstract Interpretation based
  - Jiangchao Liu and Xavier Rival (2015). "Abstraction of Arrays Based on Non Contiguous Partitions". In: *VMCAI'15*, pp. 282–299
  - Patrick Cousot, Radhia Cousot, and Francesco Logozzo (2011). "A parametric segmentation functor for fully automatic and scalable array content analysis". In: *POPL'11*, pp. 105–118
  - Sumit Gulwani, Bill McCloskey, and Ashish Tiwari (2008). "Lifting abstract interpreters to quantified logical domains". In: *POPL'08*, pp. 235–246
- Abstraction based
  - David Monniaux and Laure Gonnord (2016). "Cell Morphing: From Array Programs to Array-Free Horn Clauses". In: *SAS'16*, pp. 361–382
  - Francesco Alberti, Silvio Ghilardi, and Natasha Sharygina (2014). "Booster: An Acceleration-Based Verification Framework for Array Programs". In: *ATVA'14*, pp. 18–23
- Without explicit partitioning
  - Isil Dillig, Thomas Dillig, and Alex Aiken (2010). "Fluid Updates: Beyond Strong vs. Weak Updates". In: *ESOP'10*, pp. 246–266

# Conclusion and Future Work

- Presented a novel verification technique that
  - proves universally quantified assertions over arrays
  - decomposes reasoning about arrays using *tiles*
  - is property driven, compositional and efficient
- Future directions
  - Automated synthesis of *tiles*
  - Combining the strengths of Booster, Vaphor and Tiler
  - Integration of other candidate invariant generators

*Thank you*