

Automatic Verification of Integer Array Programs

Marius Bozga¹, Peter Habermehl², Radu Iosif¹,
Filip Konečný^{3,4}, Tomáš Vojnar³

1 VERIMAG, CNRS/UJF/INPG, France

2 LIAFA, Université Paris Diderot/CNRS, France

3 Brno University of Technology, Czech Republic

4 VERIMAG, Université Joseph Fourier, France

Verification Problem

Proving Hoare triples

- $\{P\} C \{Q\}$

Verification Problem

Proving Hoare triples

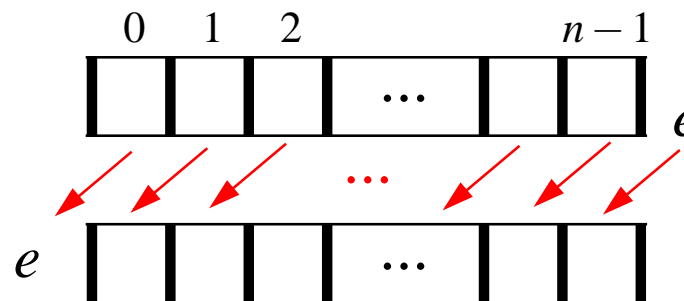
- $\{P\} C \{Q\}$

Considered programs

- integer arrays
- integer variables
- assignments
- conditional statements
- non-nested while loops

Example: Array rotation

```
1 begin
2    $e := a[0]$ ;
3    $i = 0$ ;
4   while ( $i \leq n - 2$ ) do
5      $a[i] := a[i + 1]$ ;
6      $i++$ ;
7    $a[n - 1] := e$ ;
end
```



Verification Problem

Proving Hoare triples

- $\{P\} C \{Q\}$

Considered programs

- integer arrays
- integer variables
- assignments
- conditional statements
- **non-nested** while loops

Pre/Post-condition

- SIL – a decidable logic
[Habermehl, Iosif,
Vojnar LPAR'08]

Example: Array rotation

pre-condition:

$$(n > 0) \wedge \\ \forall i. (0 \leq i \leq n - 1) \Rightarrow (a[i] = a_0[i])$$

```
1 begin
2   e := a[0];
3   i = 0;
4   while (i ≤ n - 2) do
5     a[i] := a[i + 1];
6     i ++;
7   a[n - 1] := e;
end
```

post-condition:

$$\forall i. (0 \leq i \leq n - 2) \Rightarrow (a[i] = a_0[i + 1]) \wedge \\ \forall i. (i = n - 1) \Rightarrow (a[i] = g) \\ \forall i. (i = 0) \Rightarrow (a_0[i] = g)$$

Verification Framework

Program state

- valuation of **array variables** to finite **integer sequences**

$a \mapsto [2, 3, 4, 7, 9, 10]$, $b \mapsto [1, 4, 5, 8, 9, 11]$

Verification Framework

Program state

- valuation of **array variables** to finite **integer sequences**

$a \mapsto [2, 3, 4, 7, 9, 10]$, $b \mapsto [1, 4, 5, 8, 9, 11]$

Counter automata

- counter automata = integer programs with non-deterministic assignments ($x' < 10$)
- reachability of a control state is **undecidable** in general
- **decidability** of certain subclasses: flat, reversal-bounded, ...
- tools available: ARMC, FAST, ASPIC, ...

Verification Framework

Program state

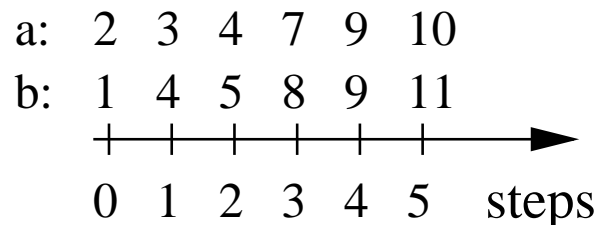
- valuation of **array variables** to finite **integer sequences**

$a \mapsto [2, 3, 4, 7, 9, 10]$, $b \mapsto [1, 4, 5, 8, 9, 11]$

Counter automata

- counter automata = integer programs with non-deterministic assignments ($x' < 10$)
- reachability of a control state is **undecidable** in general
- **decidability** of certain subclasses: flat, reversal-bounded, ...
- tools available: ARMC, FAST, ASPIC, ...

Idea of encoding of a state



Verification Framework

SIL pre-condition ϕ

```
// non-looping code
```

```
...
```

```
if() ...
```

```
else if() ...
```

```
else ...
```

```
...
```

```
while (...) {
```

```
  if() ...
```

```
  else if() ...
```

```
    ...
```

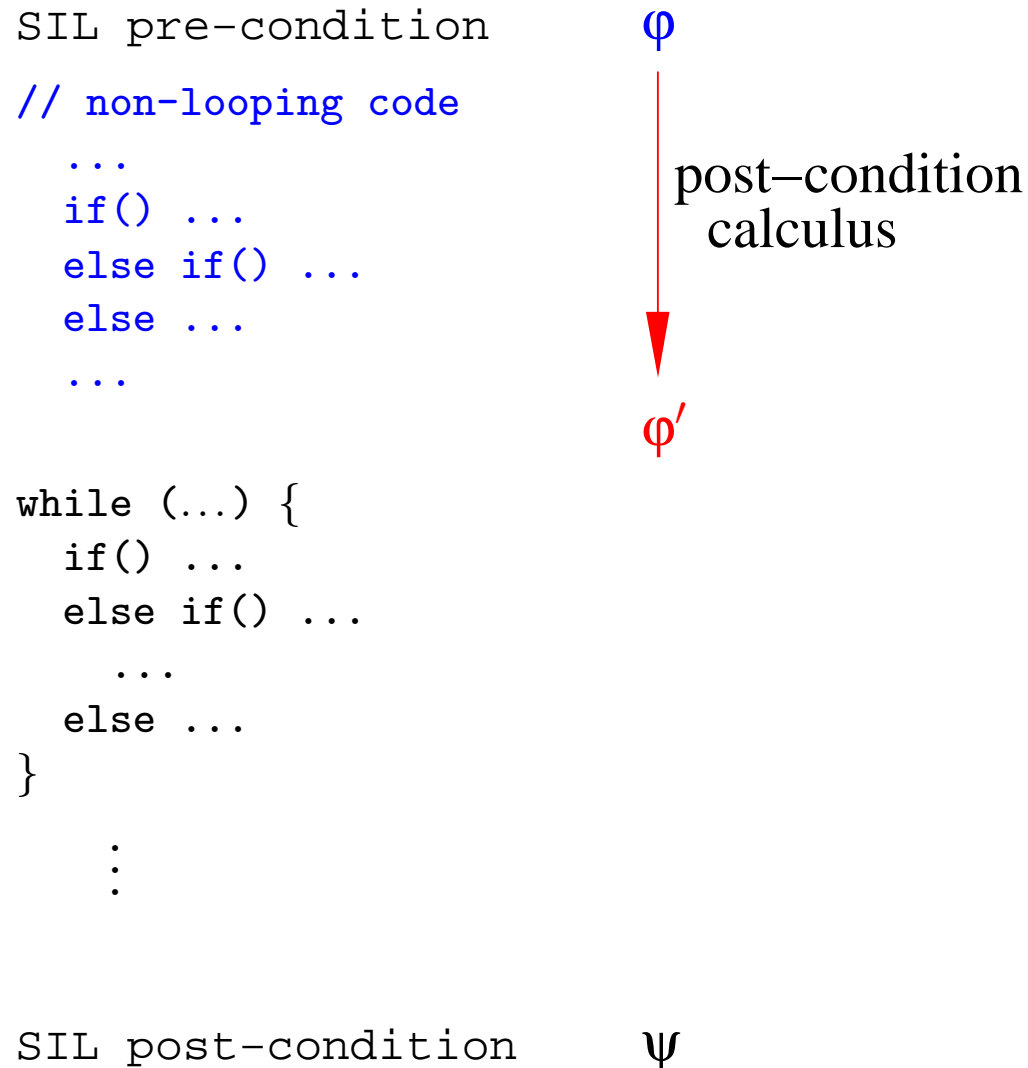
```
  else ...
```

```
}
```

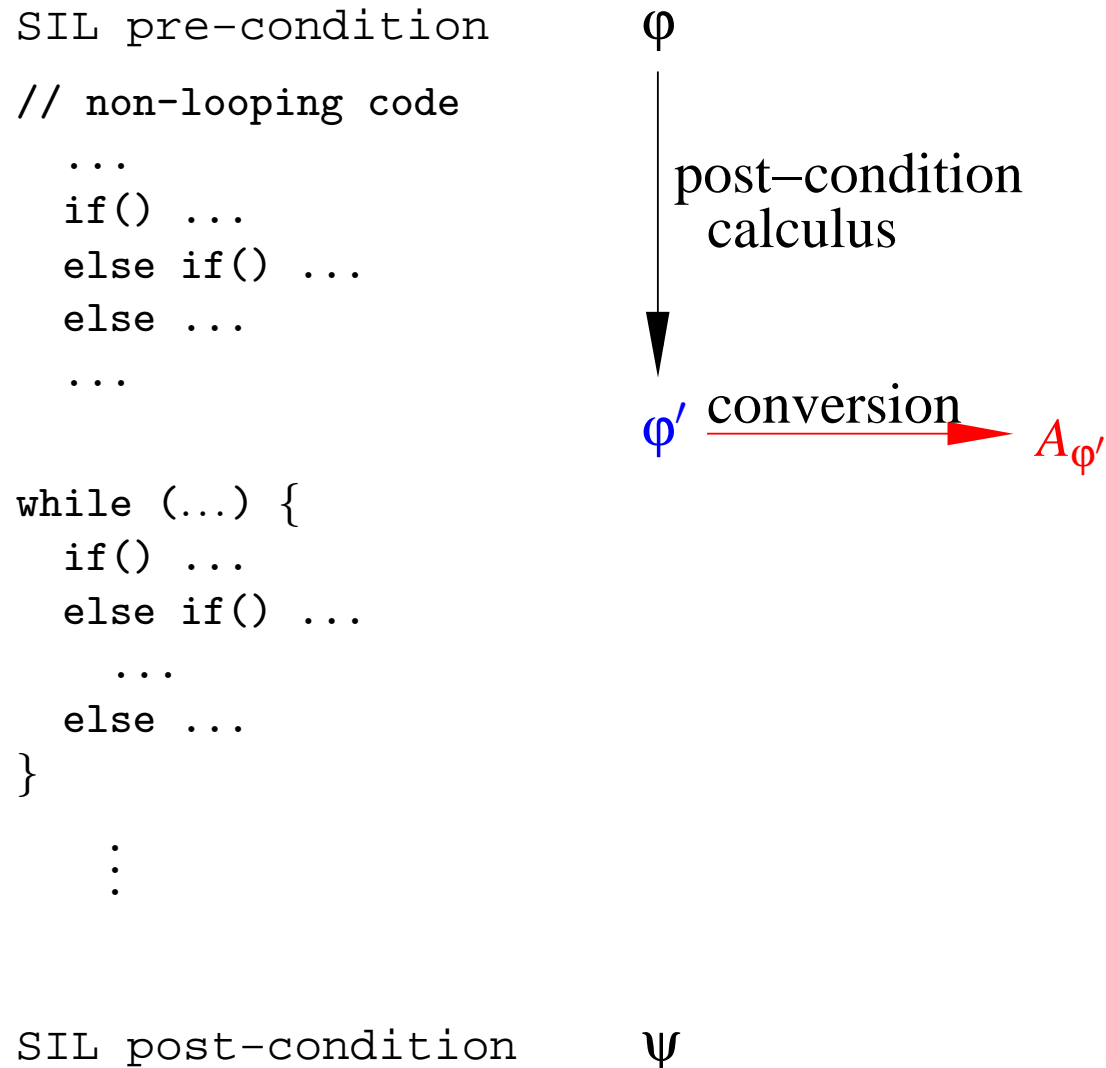
```
  :  
  :
```

SIL post-condition ψ

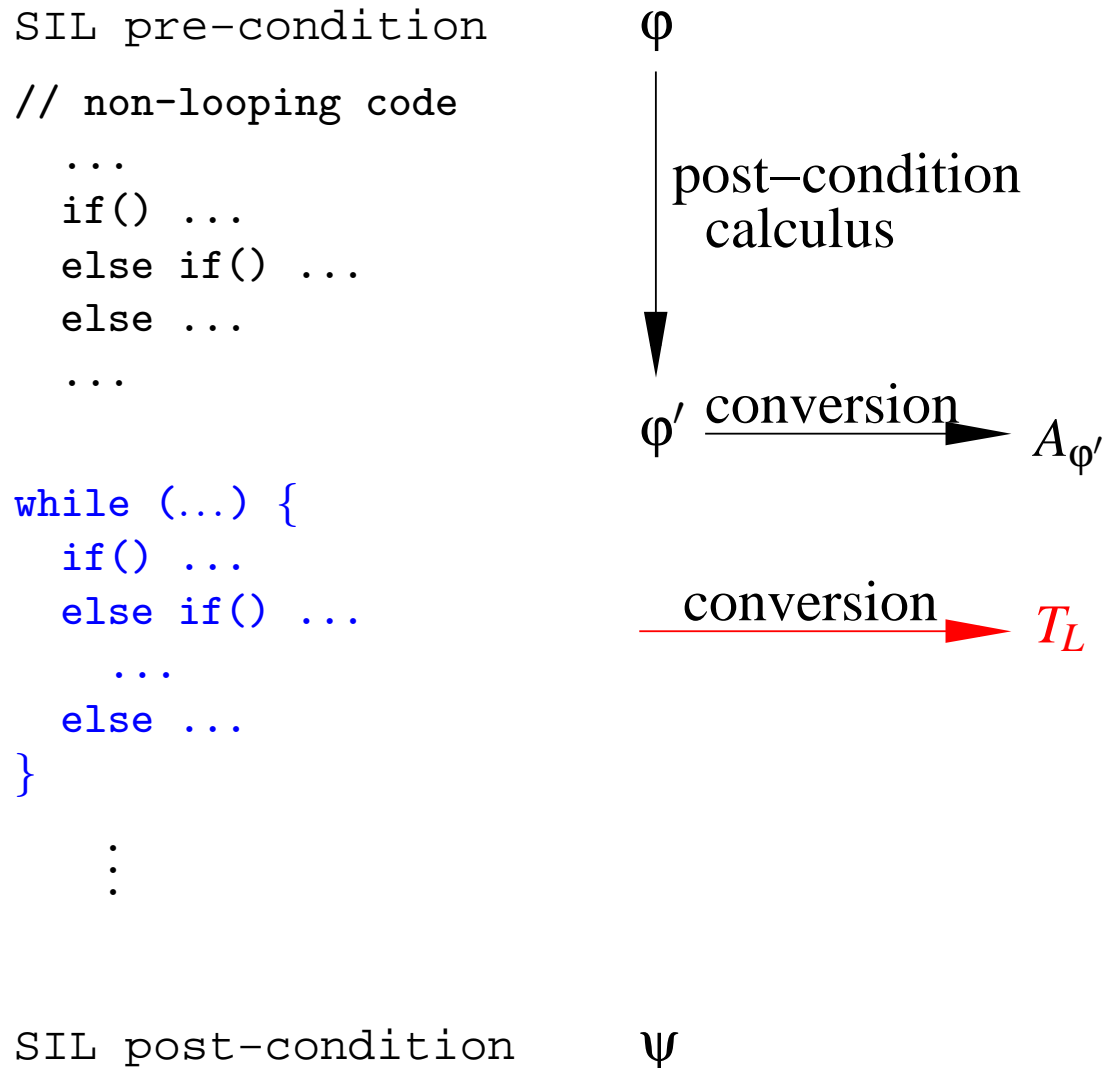
Verification Framework



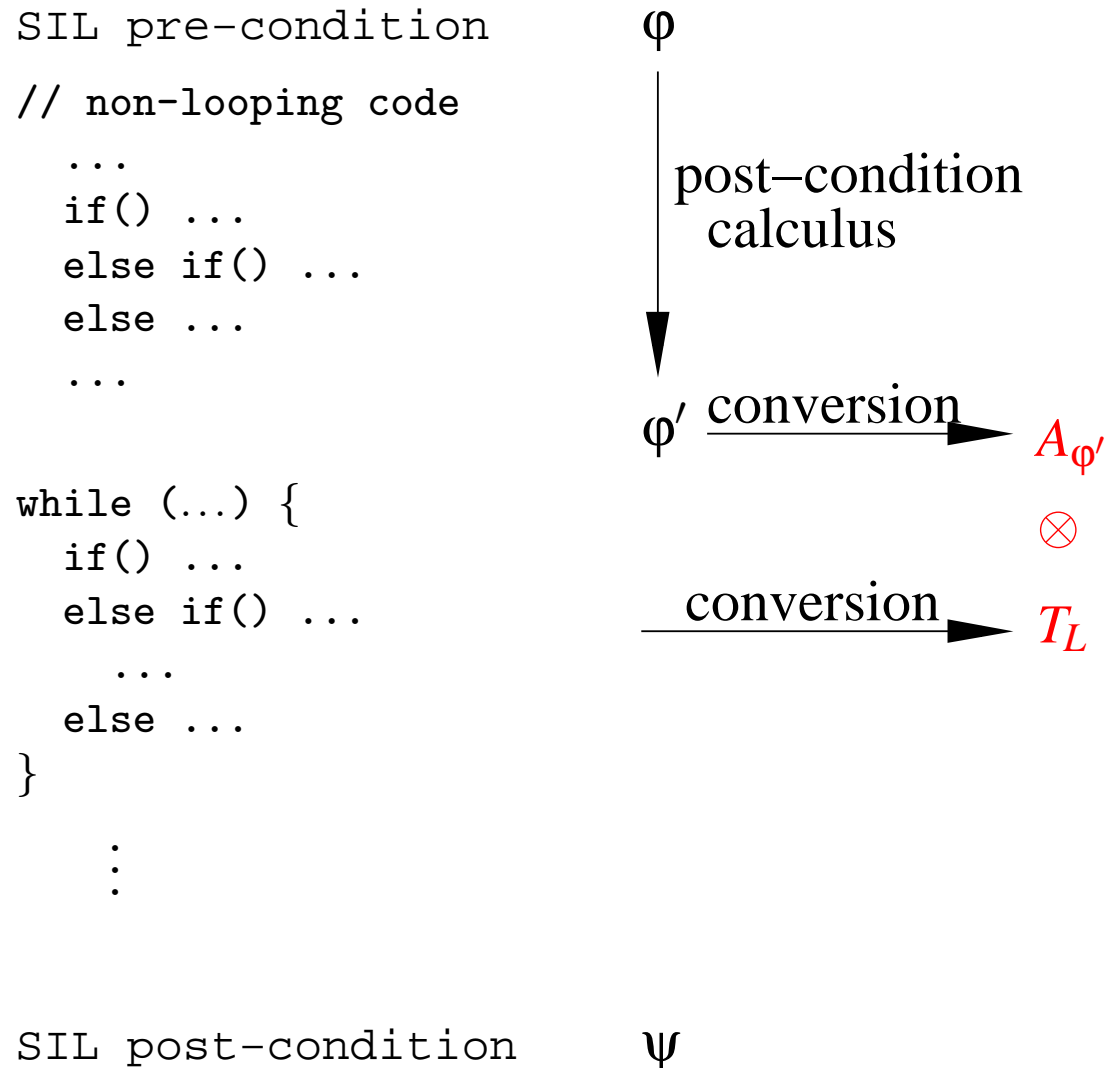
Verification Framework



Verification Framework

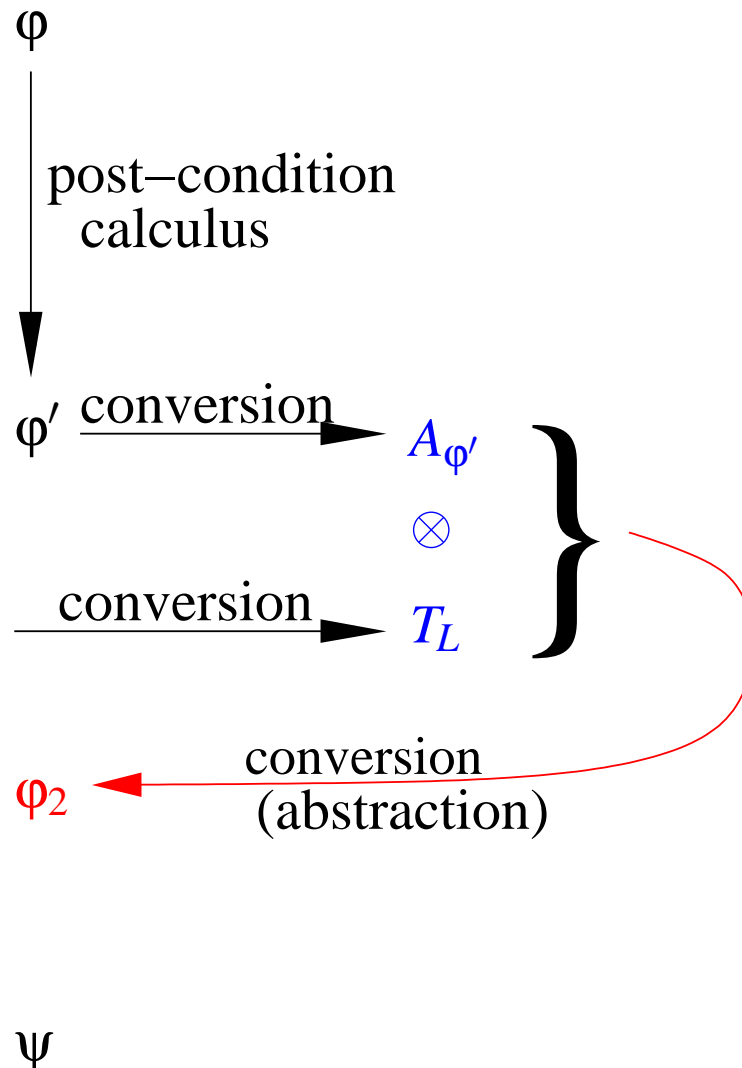


Verification Framework



Verification Framework

```
SIL pre-condition  
// non-looping code  
...  
if() ...  
else if() ...  
else ...  
...  
  
while (...) {  
  if() ...  
  else if() ...  
  ...  
  else ...  
}  
:  
  
SIL post-condition
```



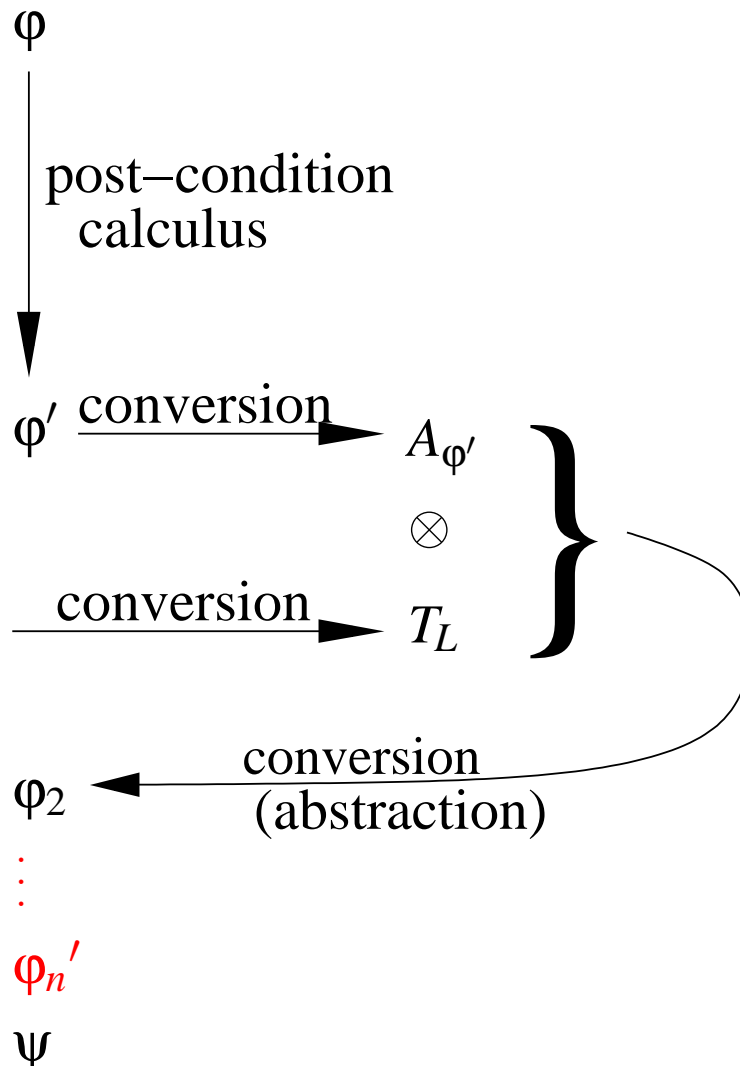
Verification Framework

```

SIL pre-condition
// non-looping code
...
if() ...
else if() ...
else ...
...

while (...) {
  if() ...
  else if() ...
  ...
  else ...
}
...

SIL post-condition
  
```



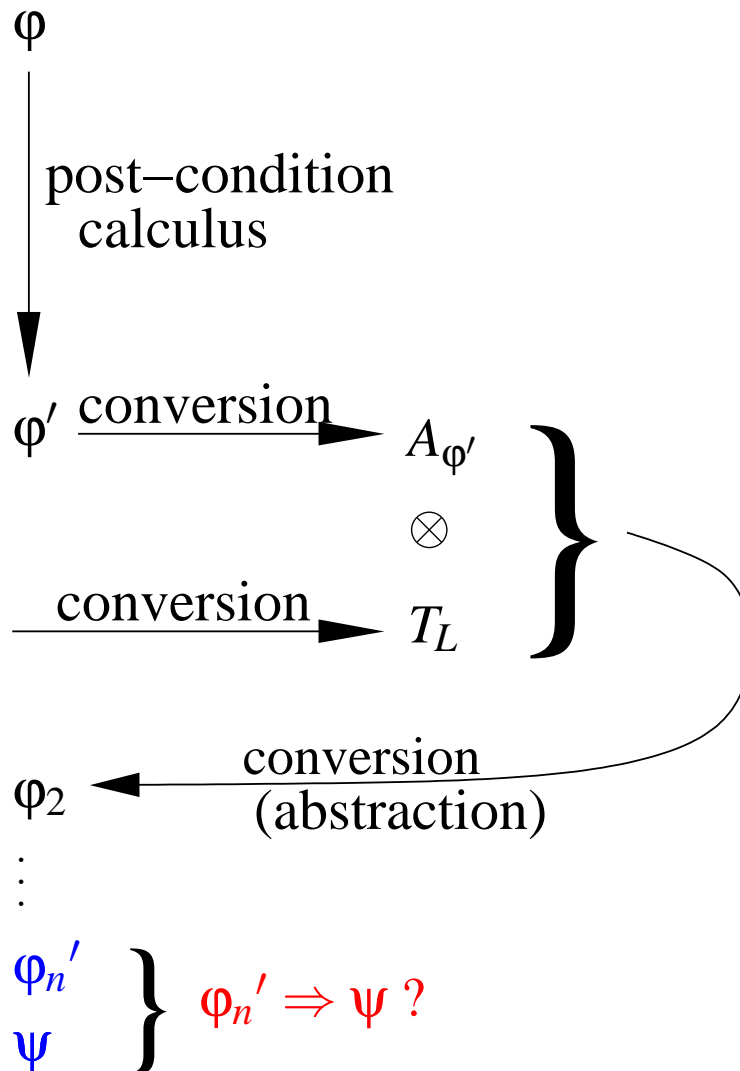
Verification Framework

```

SIL pre-condition
// non-looping code
...
if() ...
else if() ...
else ...
...

while (...) {
  if() ...
  else if() ...
  ...
  else ...
}
...

SIL post-condition
  
```



SIL – Single Index Logic

$$n > 0 \quad \wedge \quad \forall i. (0 \leq i \leq n - 2) \rightarrow a[i] = a_0[i + 1]$$

SIL formulae – boolean combinations of:

- Presburger constraints on bound variables and
- array properties comparing array elements within a constant sized window.

SIL – Single Index Logic

$$n > 0 \quad \wedge \quad \forall i. (0 \leq i \leq n - 2) \rightarrow a[i] = a_0[i + 1]$$

SIL formulae – boolean combinations of:

- Presburger constraints on bound variables and
- array properties comparing array elements within a constant sized window.

Array property is a formula of the form:

$$\forall i . \gamma(i) \rightarrow \upsilon(i), \text{ where:}$$

SIL – Single Index Logic

$$n > 0 \quad \wedge \quad \forall i. (\mathbf{0} \leq \mathbf{i} \leq \mathbf{n} - \mathbf{2}) \rightarrow a[i] = a_0[i + 1]$$

SIL formulae – boolean combinations of:

- Presburger constraints on bound variables and
- array properties comparing array elements within a constant sized window.

Array property is a formula of the form:

$$\forall i . \gamma(i) \rightarrow \upsilon(i), \text{ where:}$$

- γ is a **guard expression**
 - constraints on **scalar** variables

SIL – Single Index Logic

$$n > 0 \quad \wedge \quad \forall i. (0 \leq i \leq n - 2) \rightarrow \mathbf{a[i] = a_0[i + 1]}$$

SIL formulae – boolean combinations of:

- Presburger constraints on bound variables and
- array properties comparing array elements within a constant sized window.

Array property is a formula of the form:

$$\forall i . \gamma(i) \rightarrow \upsilon(i), \text{ where:}$$

- γ is a **guard expression**
 - constraints on **scalar** variables
- υ is a **value expression**
 - constraints on **array terms** $a[i + k]$ ($k \in \mathbb{Z}$) and the **index variable** i

SIL – Single Index Logic

$$n > 0 \quad \wedge \quad \forall i. (0 \leq \mathbf{i} \leq n - 2) \rightarrow a[\mathbf{i}] = a_0[\mathbf{i} + 1]$$

SIL formulae – boolean combinations of:

- Presburger constraints on bound variables and
- array properties comparing array elements within a constant sized window.

Array property is a formula of the form:

$$\forall \mathbf{i}. \gamma(\mathbf{i}) \rightarrow \upsilon(\mathbf{i}), \text{ where:}$$

- γ is a **guard expression**
 - constraints on **scalar** variables
- υ is a **value expression**
 - constraints on **array terms** $a[i+k]$ ($k \in \mathbb{Z}$) and the **index variable** i
 - **singly indexed**: \mathbf{i} is the only index variable

SIL – Single Index Logic

$$n > 0 \quad \wedge \quad \forall i. (0 \leq i \leq n - 2) \rightarrow a[i] = a_0[i + 1]$$

SIL formulae – boolean combinations of:

- Presburger constraints on bound variables and
- array properties comparing array elements within a constant sized window.

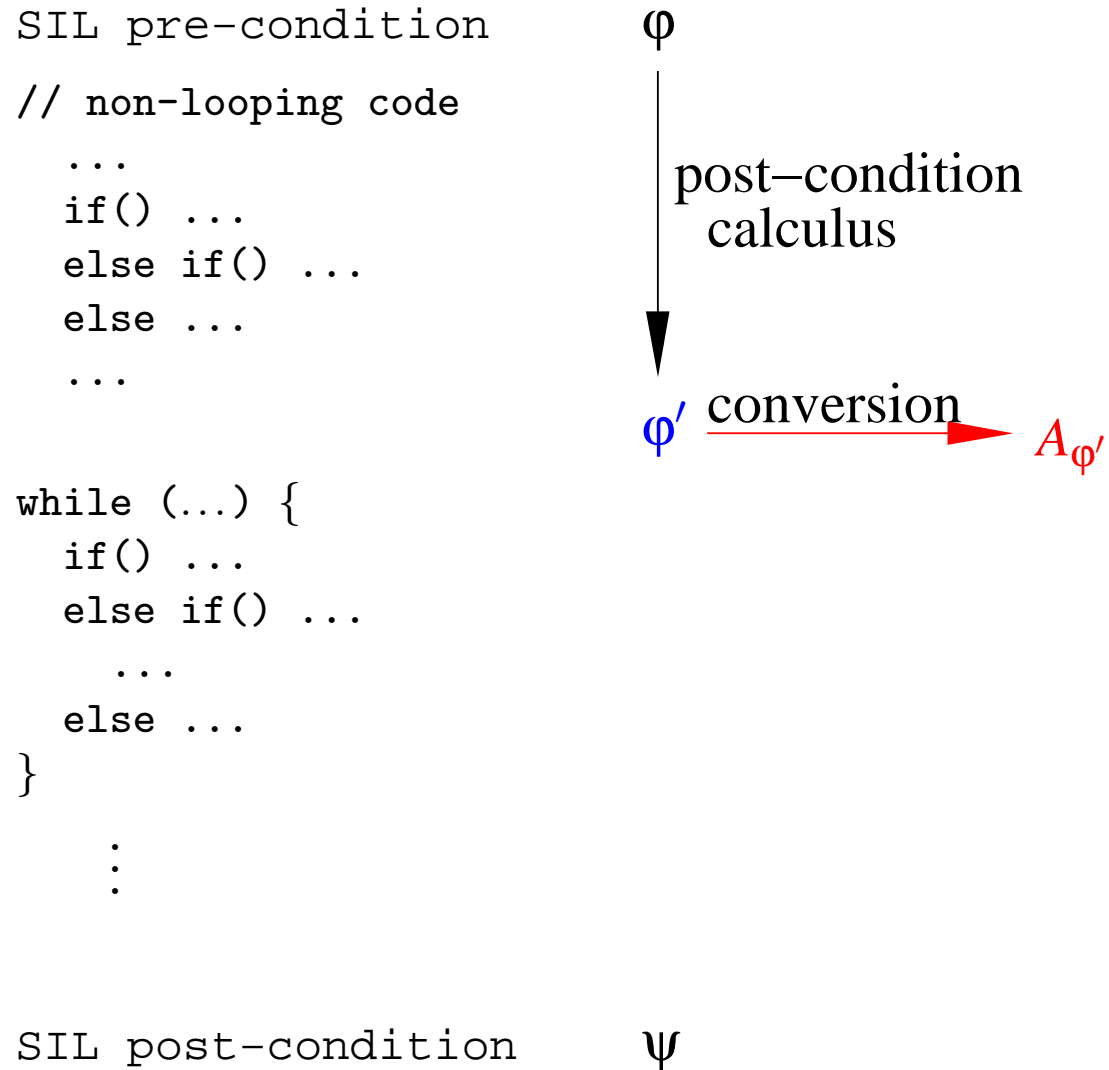
Array property is a formula of the form:

$$\forall i . \gamma(i) \rightarrow \upsilon(i), \text{ where:}$$

- γ is a **guard expression**
 - constraints on **scalar** variables
- υ is a **value expression**
 - constraints on **array terms** $a[i + k]$ ($k \in \mathbb{Z}$) and the **index variable** i
 - **singly indexed**: i is the only index variable

Difference bound constraints: $x - y \leq \alpha$, $x \leq \alpha$, $x \geq \alpha$, $\alpha \in \mathbb{Z}$

SIL to Counter Automata



SIL to Counter Automata

Encoding of arrays

- accepting runs of a CA correspond to models of a SIL formula

SIL to Counter Automata

Encoding of arrays

- accepting runs of a CA correspond to models of a SIL formula

Example: $a[0] < 5 \wedge \forall i.(0 \leq i \leq 4) \rightarrow (a[i+1] = a[i] + 2)$

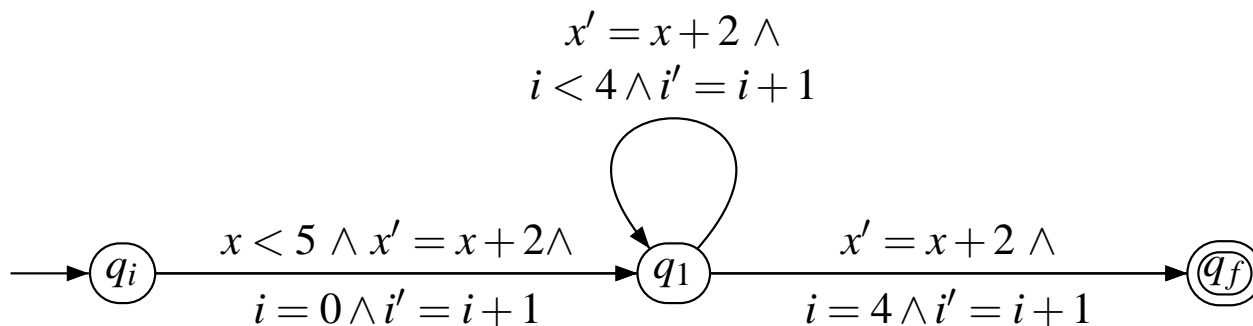
SIL to Counter Automata

Encoding of arrays

- accepting runs of a CA correspond to models of a SIL formula

Example: $a[0] < 5 \wedge \forall i.(0 \leq i \leq 4) \rightarrow (a[i+1] = a[i] + 2)$

- array variable $a \rightarrow$ value counter x and index counter i



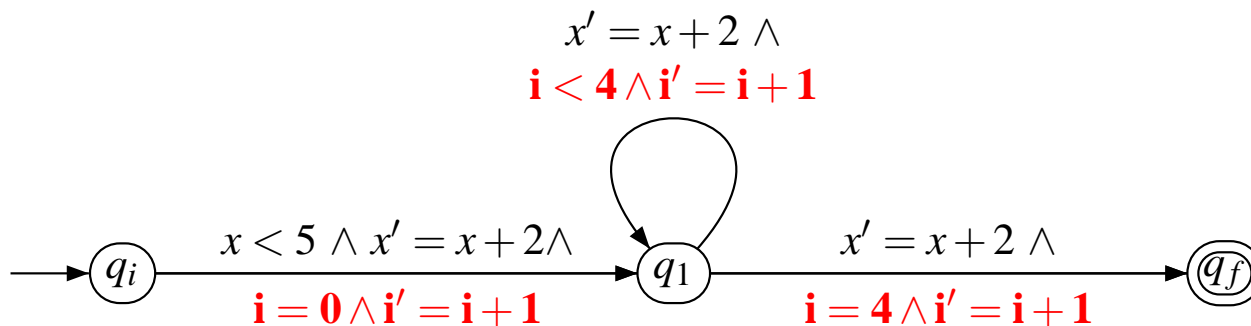
SIL to Counter Automata

Encoding of arrays

- accepting runs of a CA correspond to models of a SIL formula

Example: $a[0] < 5 \wedge \forall i.(0 \leq i \leq 4) \rightarrow (a[i+1] = a[i] + 2)$

- array variable $a \rightarrow$ value counter x and index counter i



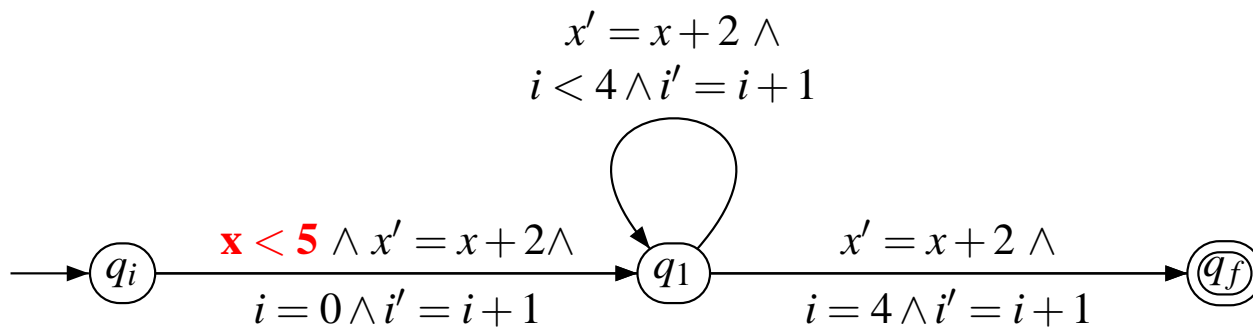
SIL to Counter Automata

Encoding of arrays

- accepting runs of a CA correspond to models of a SIL formula

Example: $a[0] < 5 \wedge \forall i.(0 \leq i \leq 4) \rightarrow (a[i+1] = a[i] + 2)$

- array variable $a \rightarrow$ value counter x and index counter i



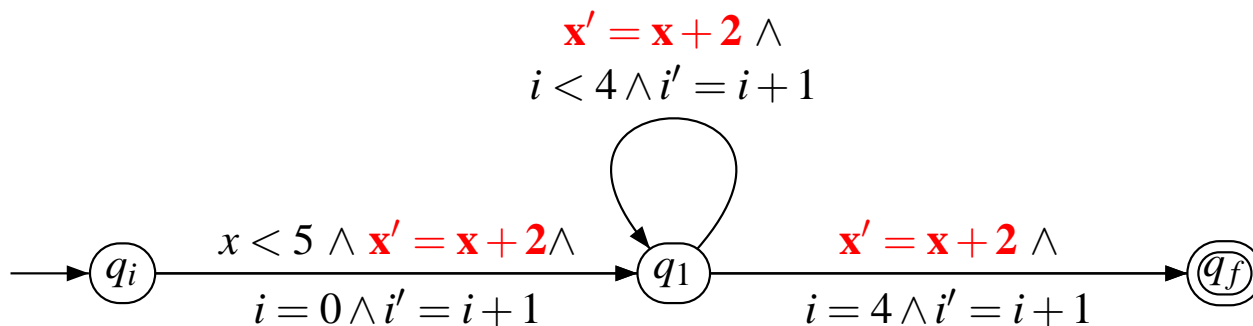
SIL to Counter Automata

Encoding of arrays

- accepting runs of a CA correspond to models of a SIL formula

Example: $a[0] < 5 \wedge \forall i.(0 \leq i \leq 4) \rightarrow (a[i+1] = a[i] + 2)$

- array variable $a \rightarrow$ value counter x and index counter i



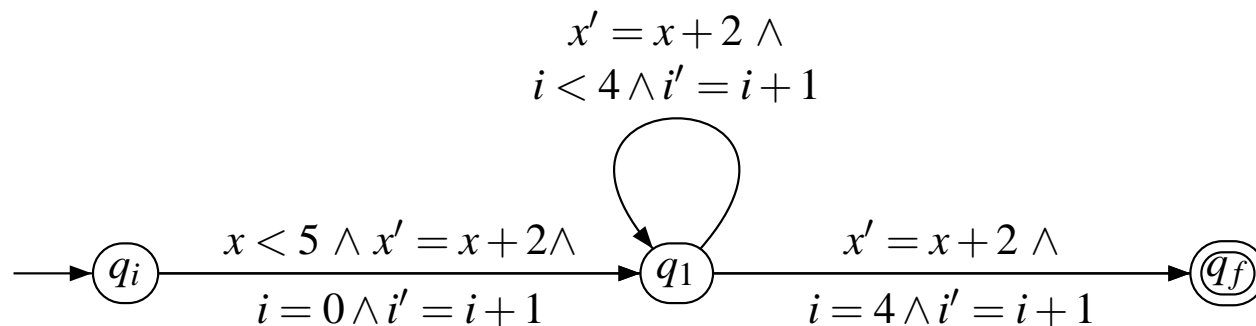
SIL to Counter Automata

Encoding of arrays

- accepting runs of a CA correspond to models of a SIL formula

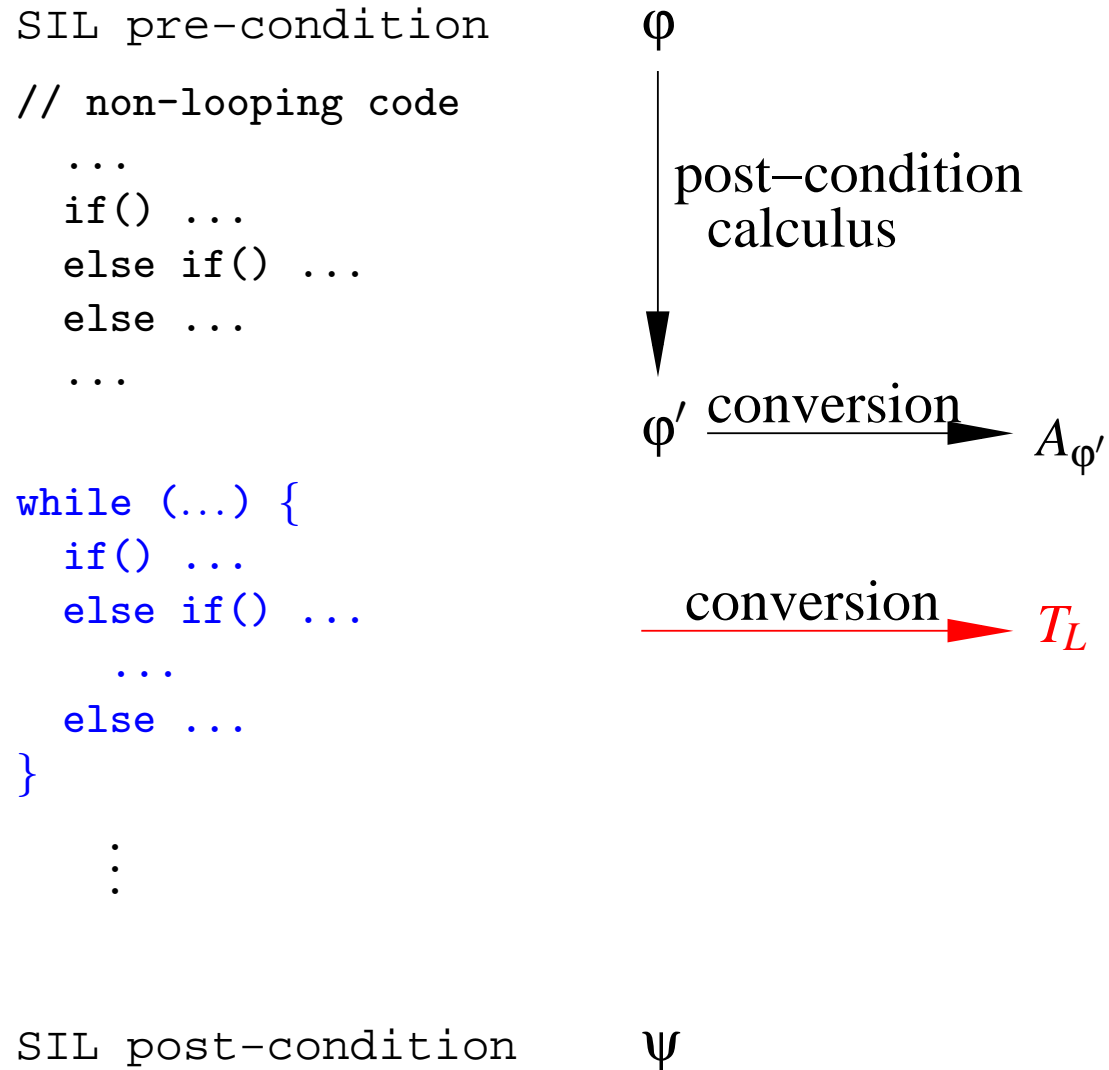
Example: $a[0] < 5 \wedge \forall i.(0 \leq i \leq 4) \rightarrow (a[i+1] = a[i] + 2)$

- array variable $a \rightarrow$ value counter x and index counter i



- a model $a = [3, 5, 7, 9, 11, 13]$, a run
- | | | | | | | |
|-----|---|---|---|---|----|----|
| i | 0 | 1 | 2 | 3 | 4 | 5 |
| x | 3 | 5 | 7 | 9 | 11 | 13 |

Loops to Counter Transducers



Loops to Counter Transducers

Array access

- via terms $a[i], \dots, a[i + n]$

Loops to Counter Transducers

Array access

- via terms $a[i], \dots, a[i+n]$

Encoding of an array

- 1 index counter to code i
- $2(n+1)$ array counters
 - code input/output values of $a[i], \dots, a[i+n]$ ($n \in \mathbb{Z}$)
 - the contents shifts when $i++$ is performed

Loops to Counter Transducers

Array access

- via terms $a[i], \dots, a[i+n]$

Encoding of an array

- 1 index counter to code i
- $2(n+1)$ array counters
 - code input/output values of $a[i], \dots, a[i+n]$ ($n \in \mathbb{Z}$)
 - the contents shifts when $i++$ is performed

Example

- loop: `for (i=0; i<=5; i++) a[i]=a[i]+3;`
- input state: $a = [3, 5, 7, 9, 11, 13]$, output state: $a' = [6, 8, 10, 12, 14, 16]$

- run:

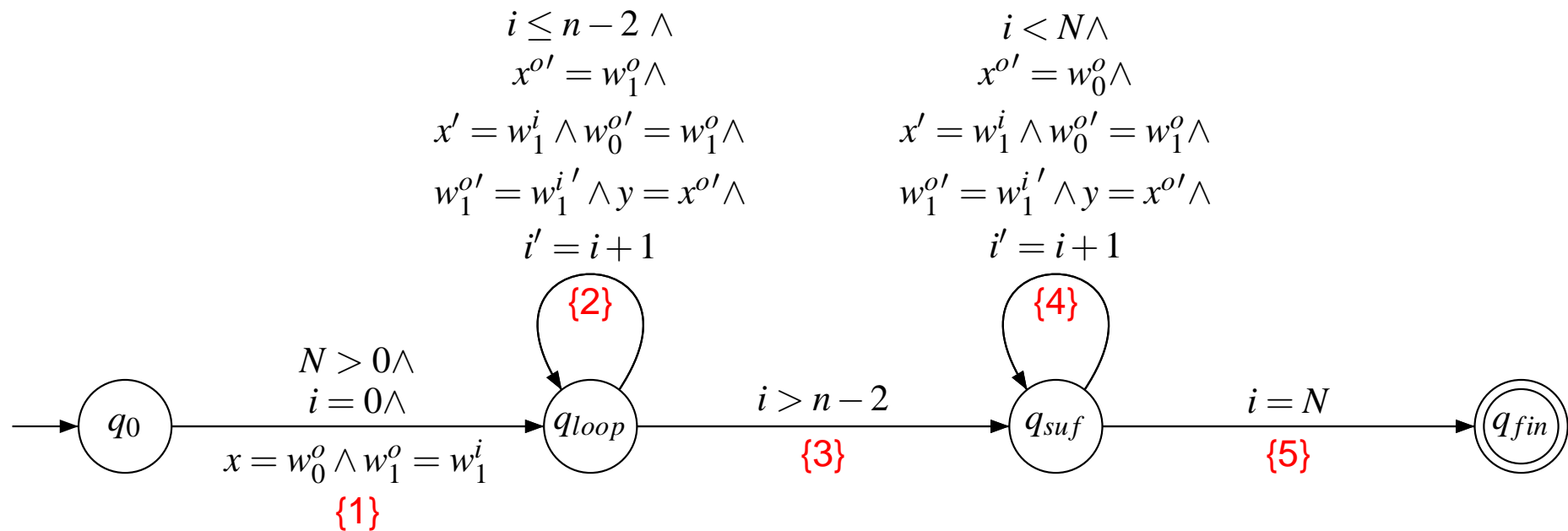
i (index)	0	1	2	3	4	5
x (input-value)	3	5	7	9	11	13
y (output-value)	6	8	10	12	14	16

Rotation Example – Loop Transducer

```

i = 0;
while (i ≤ n - 2) do
  a[i] := a[i + 1];
  i++;

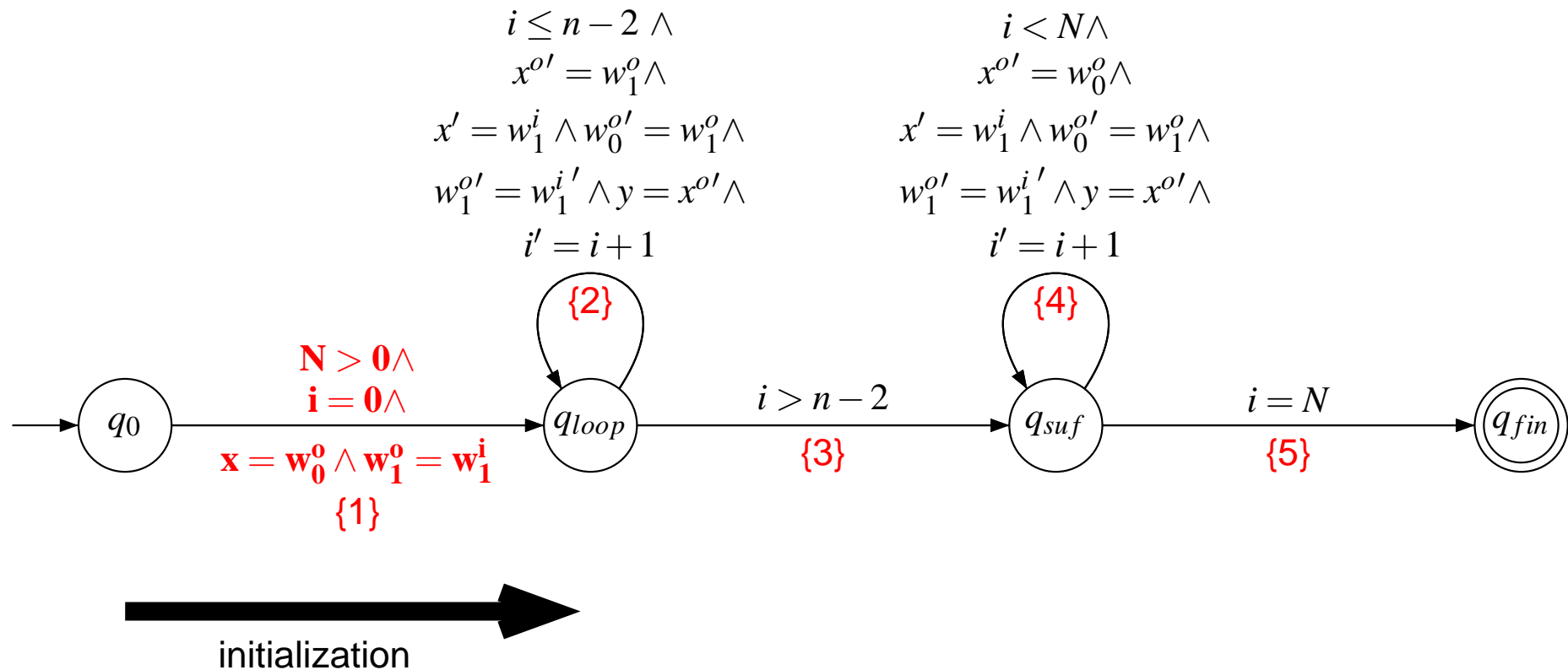
```



Rotation Example – Loop Transducer

```

i = 0;
while ( $i \leq n - 2$ ) do
  |  $a[i] := a[i + 1];$ 
  |  $i++;$ 
  
```

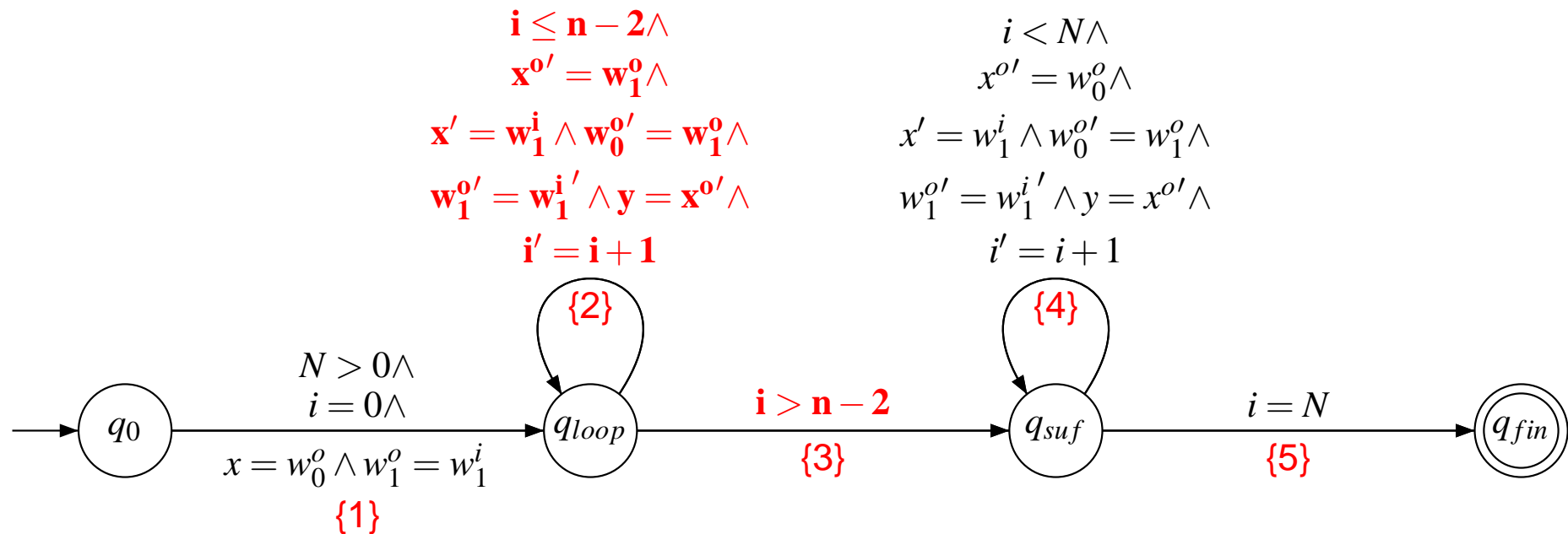


Rotation Example – Loop Transducer

```

i = 0;
while (i ≤ n - 2) do
  a[i] := a[i + 1];
  i++;

```

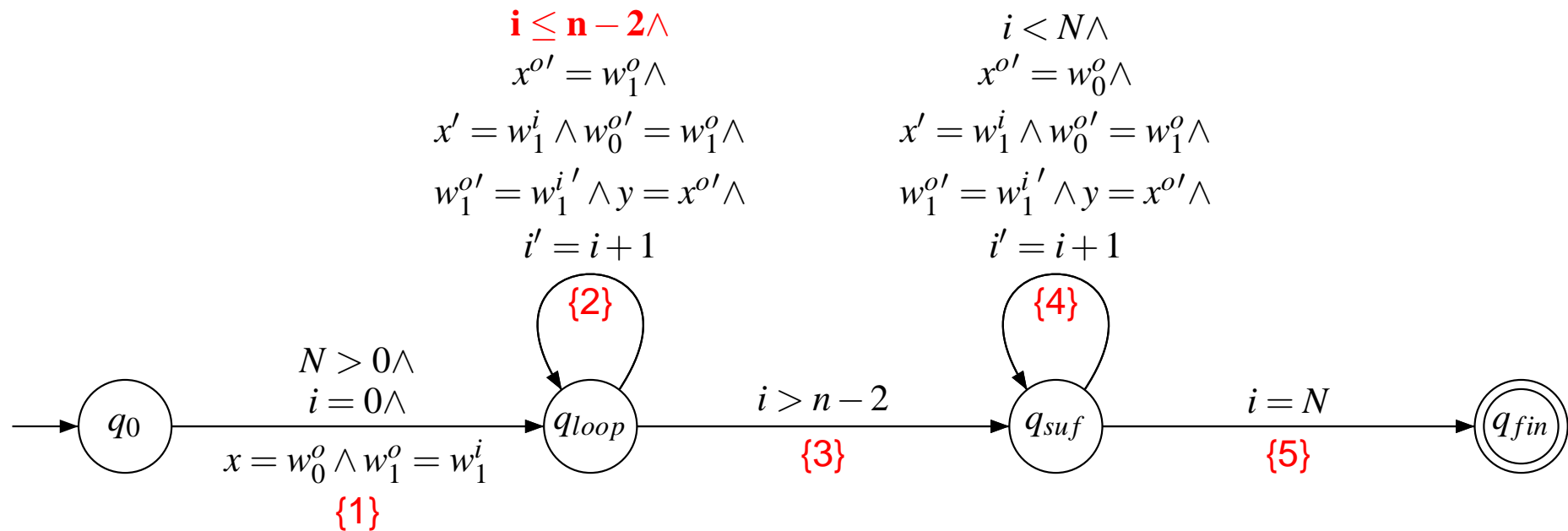


Rotation Example – Loop Transducer

```

i = 0;
while (i ≤ n - 2) do
  a[i] := a[i + 1];
  i++;

```

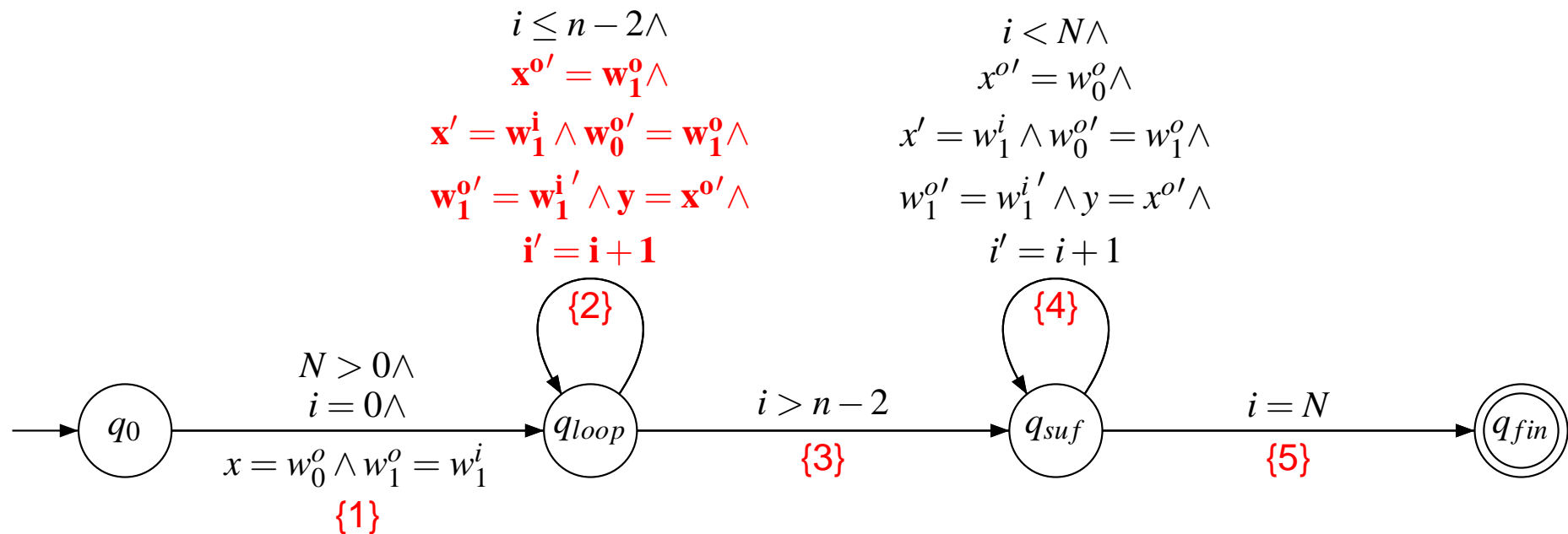


Rotation Example – Loop Transducer

```

i = 0;
while (i ≤ n - 2) do
  a[i] := a[i + 1];
  i++;

```

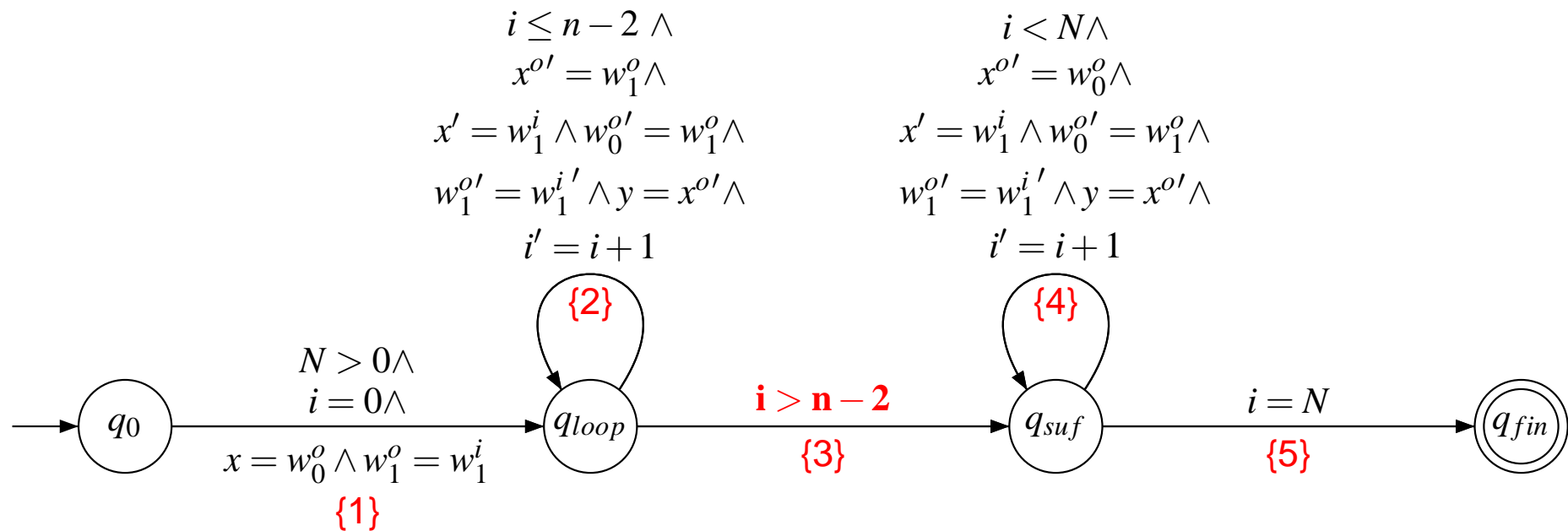


Rotation Example – Loop Transducer

```

i = 0;
while (i ≤ n - 2) do
  a[i] := a[i + 1];
  i++;

```

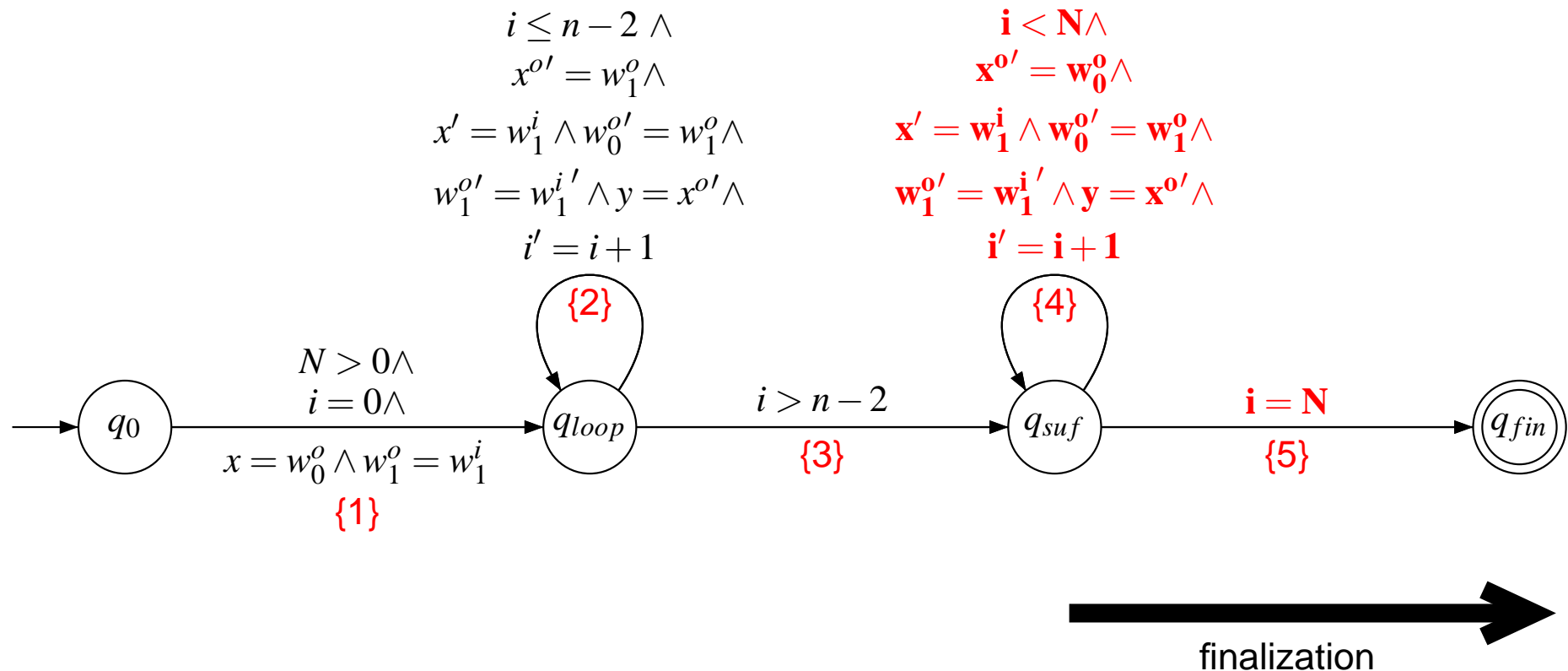


Rotation Example – Loop Transducer

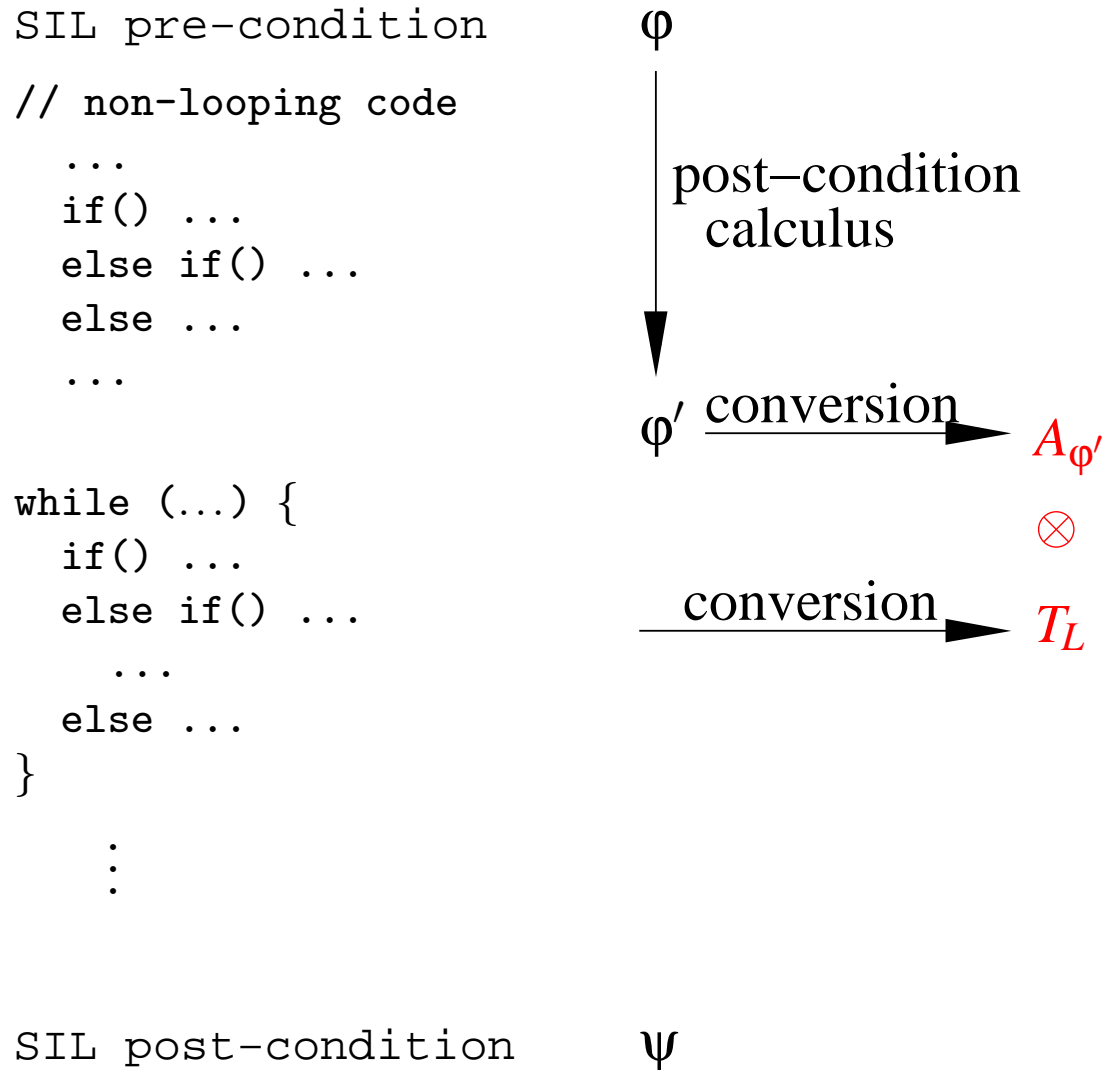
```

i = 0;
while (i ≤ n - 2) do
  a[i] := a[i + 1];
  i++;

```



Post-image of a Transducer



Post-image of a Transducer

SIL pre-condition

// non-looping code

```
...
if() ...
else if() ...
else ...
...
```

```
while (...) {
  if() ...
  else if() ...
  ...
  else ...
}
```

⋮

SIL post-condition

ϕ

post-condition
calculus

ϕ' conversion $\rightarrow A_{\phi'}$

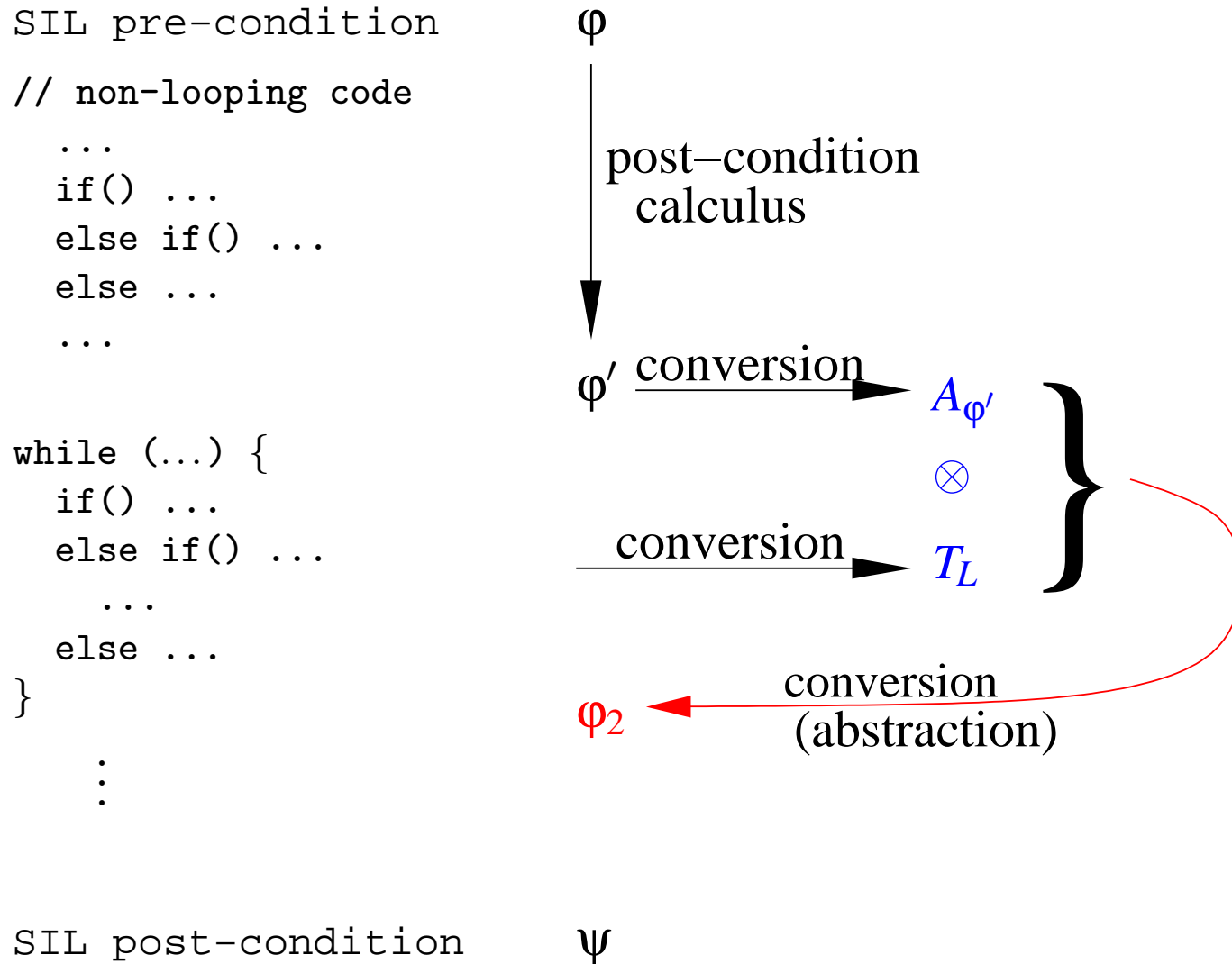
\otimes

conversion $\rightarrow T_L$

$A_{\phi'}$	input states
T_L	transition relation
$A_{\phi'} \otimes T_L$	output states

ψ

From CA to SIL – Abstraction (1)



From CA to SIL – Abstraction (1)

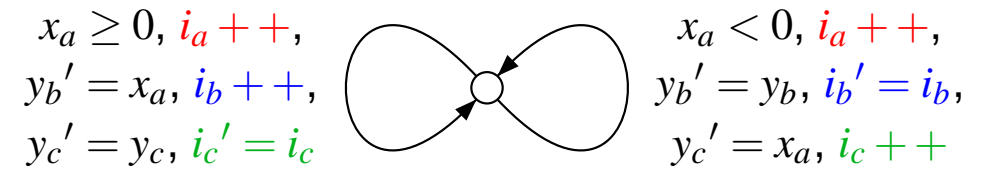
Main idea

- post-condition automata can be directly translated back to SIL when:
 - no nested loops appear in the control structure
 - the only loops are self-loops
 - all transitions on loops are DBM constraints
- each self-loop is translated to an array-property formula
- connect array property formulae by \wedge (sequences) and \vee (branches)

From CA to SIL – Abstraction (2)

Abstraction steps

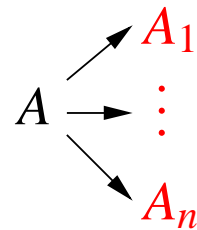
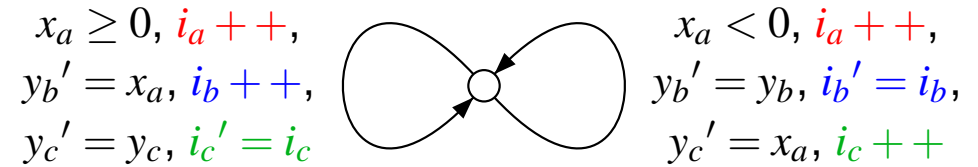
- partition arrays into classes whose index advances at the same time



From CA to SIL – Abstraction (2)

Abstraction steps

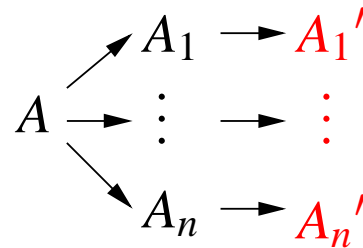
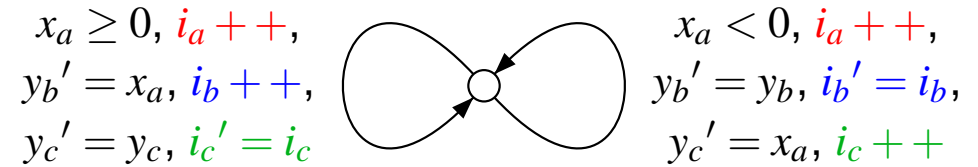
- partition arrays into classes whose index advances at the same time
- make a copy A_i for each class



From CA to SIL – Abstraction (2)

Abstraction steps

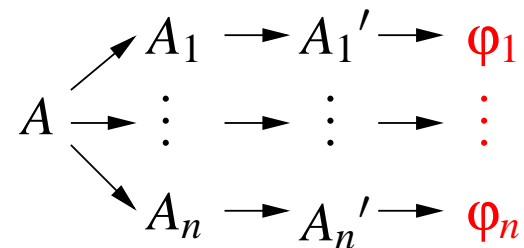
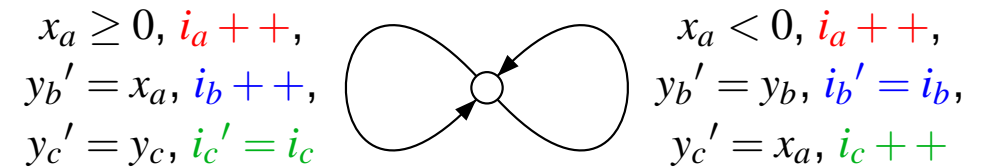
- partition arrays into classes whose index advances at the same time
- make a copy A_i for each class
- create an abstraction A_i' of each copy A_i (over-approximate)



From CA to SIL – Abstraction (2)

Abstraction steps

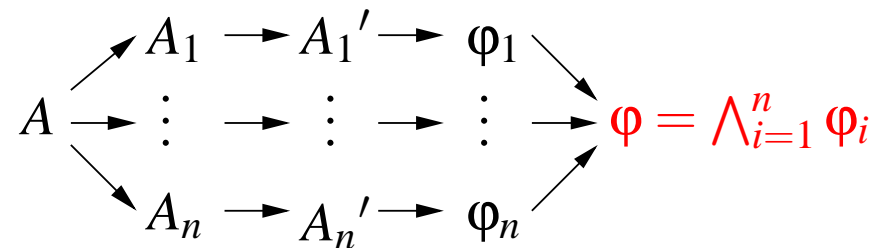
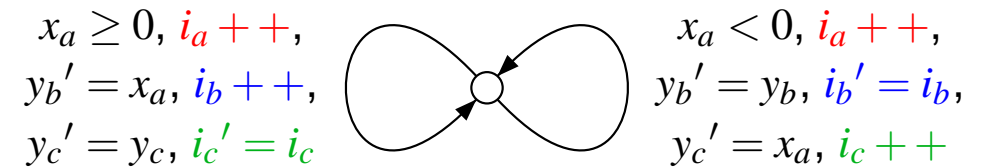
- partition arrays into classes whose index advances at the same time
- make a copy A_i for each class
- create an abstraction A_i' of each copy A_i (over-approximate)
- translate each A_i' to φ_i



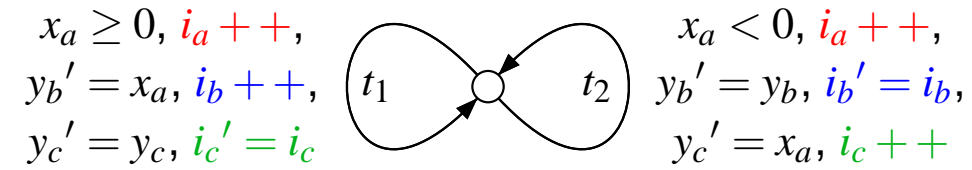
From CA to SIL – Abstraction (2)

Abstraction steps

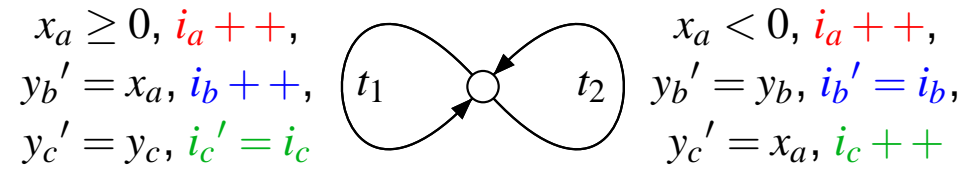
- partition arrays into classes whose index advances at the same time
- make a copy A_i for each class
- create an abstraction A_i' of each copy A_i (over-approximate)
- translate each A_i' to φ_i
- resulting formula is $\bigwedge \varphi_i$



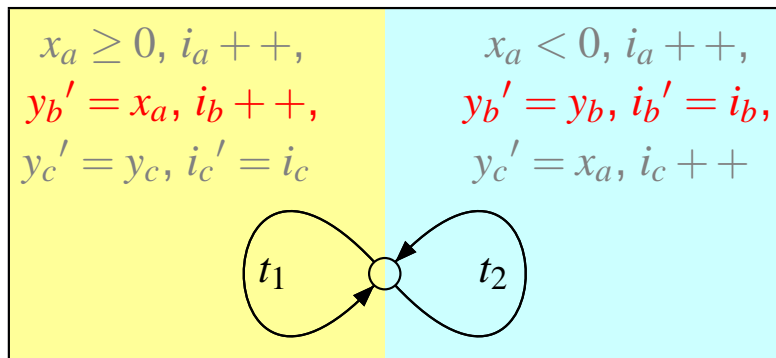
From CA to SIL – Abstraction (3)



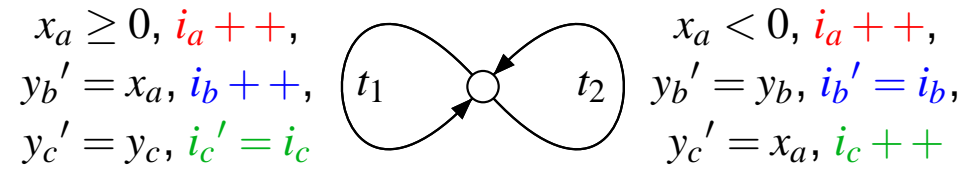
From CA to SIL – Abstraction (3)



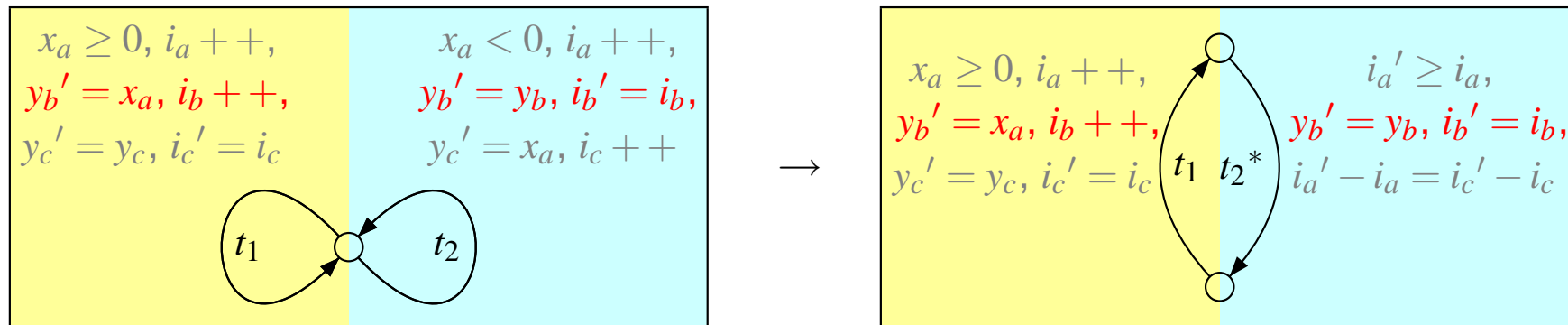
[i_b] class



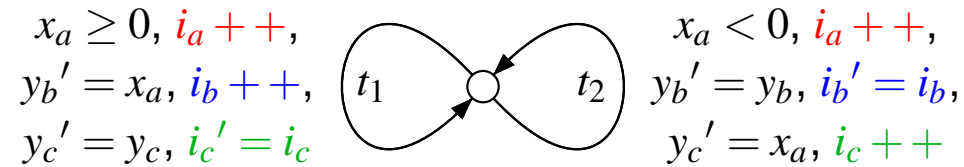
From CA to SIL – Abstraction (3)



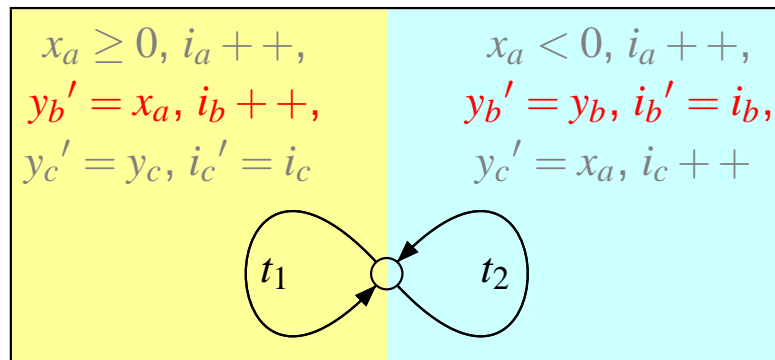
$[i_b]$ class



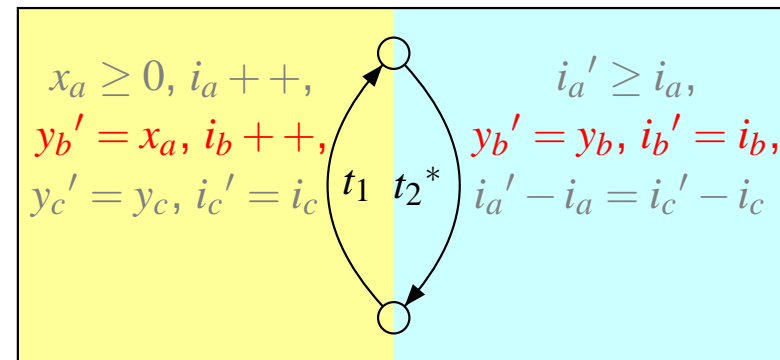
From CA to SIL – Abstraction (3)



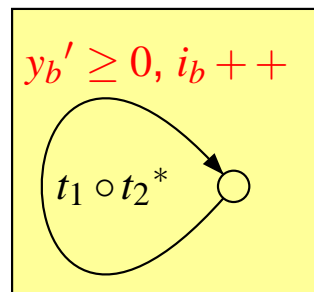
$[i_b]$ class



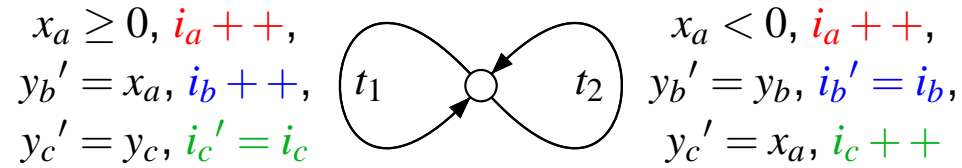
→



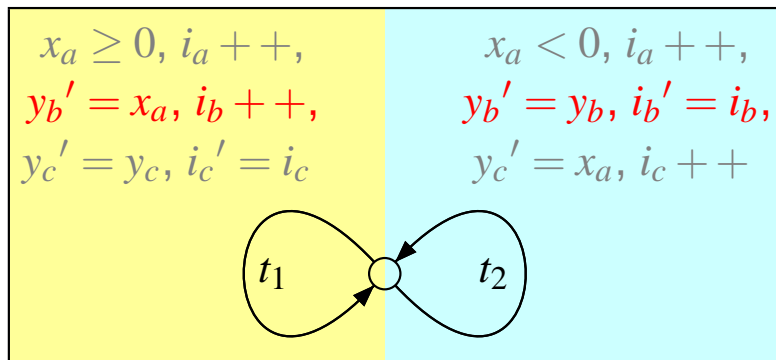
→



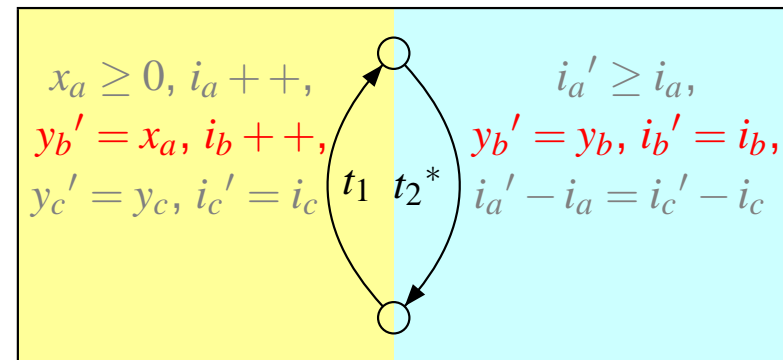
From CA to SIL – Abstraction (3)



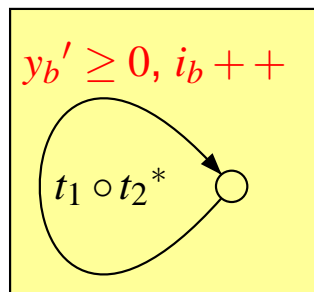
$[i_b]$ class



→



→



→

$$\forall i. (i \geq 1) \rightarrow (b[i] \geq 0)$$

SIL Entailment

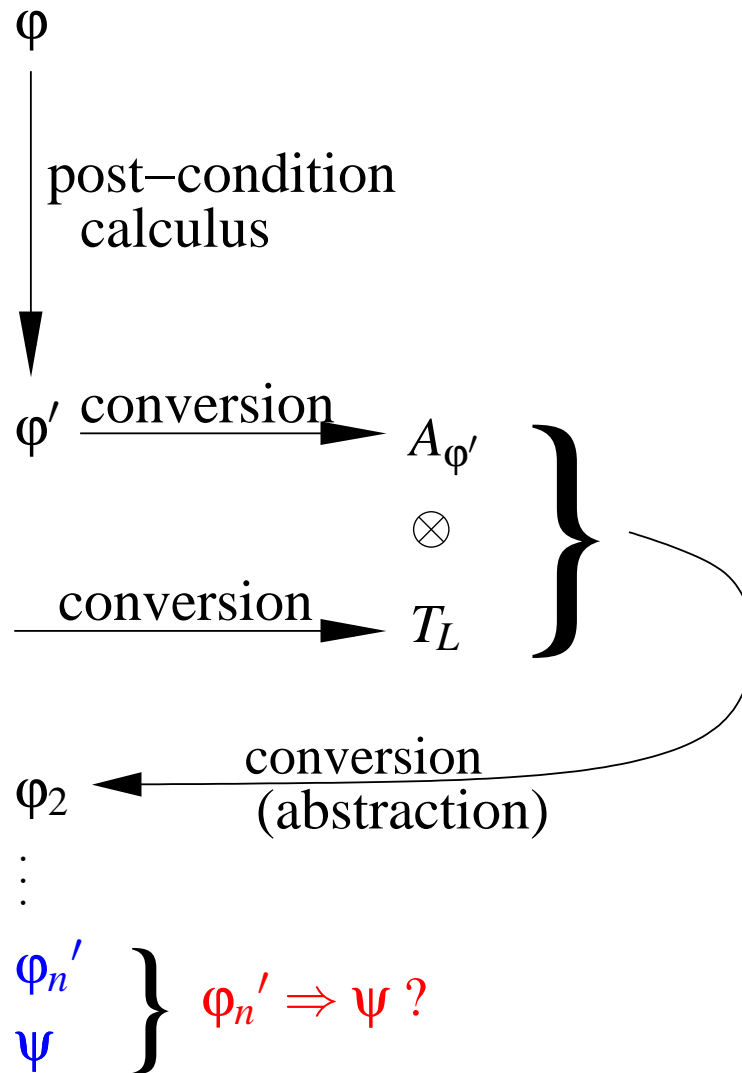
```

SIL pre-condition
// non-looping code
...
if() ...
else if() ...
else ...
...

while (...) {
  if() ...
  else if() ...
  ...
  else ...
}
...

SIL post-condition

```



SIL Entailment

Given

- ϕ_n' – inferred post-condition in SIL
- ψ – user-specified post-condition in SIL

Entailment check

- validity of $\phi_n' \rightarrow \psi$
- unsatisfiability of $\phi_n' \wedge \neg\psi$
- satisfiability of SIL is decidable

Results

Size of automata encoding the loop post-condition

program	control states	counters
init	4	8
partition	4	24
insert	7	19
rotate	4	15

Results

Size of automata encoding the loop post-condition

program	control states	counters
init	4	8
partition	4	24
insert	7	19
rotate	4	15

Abstraction technique

- sufficient to prove user post-conditions

program	post-condition
init	$\forall i.(0 \leq i \leq n - 1) \Rightarrow (a[i] = 0)$
partition	$\forall i.(0 \leq i \leq n_b - 1) \Rightarrow (b[i] \geq 0) \wedge \forall i.(0 \leq i \leq n_c - 1) \Rightarrow (c[i] < 0)$
insert	$\forall i.(0 \leq i \leq n - 2) \Rightarrow (a[i] \leq a[i + 1]) \wedge \forall i.(i = k) \Rightarrow (a[i] = e)$
rotate	$\forall i.(0 \leq i \leq n - 2) \Rightarrow (a[i] = a_0[i + 1]) \wedge \forall i.(i = n - 1) \Rightarrow (a[i] = e)$

Summary

Approach

- two-way automata-logic connection (counter automata, SIL logic)

Summary

Approach

- two-way automata-logic connection (counter automata, SIL logic)

verification problem involving arrays (uninterpreted functions)
↓
reachability problem on CA (non-deterministic integer programs)

Summary

Approach

- two-way automata-logic connection (counter automata, SIL logic)

verification problem involving arrays (uninterpreted functions)
↓
reachability problem on CA (non-deterministic integer programs)

Highlights

- translation from loops to CA without fixpoint computation

Summary

Approach

- two-way automata-logic connection (counter automata, SIL logic)

verification problem involving arrays (uninterpreted functions)
↓
reachability problem on CA (non-deterministic integer programs)

Highlights

- translation from loops to CA without fixpoint computation
- inferred post-conditions in a decidable logic (SIL)

Summary

Approach

- two-way automata-logic connection (counter automata, SIL logic)

verification problem involving arrays (uninterpreted functions)
↓
reachability problem on CA (non-deterministic integer programs)

Highlights

- translation from loops to CA without fixpoint computation
- inferred post-conditions in a decidable logic (SIL)
- leveraging on the progress on CA tools

Ongoing / Future Work

FLATA – a counter automata tool

- goals
 - checking of safety properties
 - computation of the input-output relation
- approach
 - fast algorithm for transitive closure of a DBM relation
 - techniques for elimination of control states
 - reduce number of counters

Ongoing / Future Work

FLATA – a counter automata tool

- goals
 - checking of safety properties
 - computation of the input-output relation
- approach
 - fast algorithm for transitive closure of a DBM relation
 - techniques for elimination of control states
 - reduce number of counters

More general classes of programs

- nested loops
- function calls

Termination checking

Related Work

Abstract interpretation

A Framework for Numeric Analysis of Array Operations

D. Gopan, T.W. Reps, and S. Sagiv

In *POPL'05*. ACM, 2005

Discovering Properties about Arrays in Simple Programs

N. Halbwachs and M. Péron

In *Proc. of PLDI'08*. ACM, 2008

Lifting Abstract Interpreters to Quantified Logical Domains

S. Gulwani, B. McCloskey, and A. Tiwari

In *POPL'08*. ACM, 2008

Related Work

Predicate abstraction / Interpolation

Array Abstractions from Proofs

R. Jhala and K. McMillan

In *CAV'07, LNCS 4590*. Springer, 2007

Quantified Invariant Generation Using an Interpolating Saturation Prover

K. McMillan

In *Proc. of TACAS'08, LNCS 4963*. Springer, 2008

Related Work

Logics / Theorem proving

What's Decidable About Arrays?

A.R. Bradley, Z. Manna, and H.B. Sipma

In *Proc. of VMCAI'06*, LNCS 3855. Springer, 2006

Rewriting Systems with Data: A Framework for Reasoning about Systems
with Unbounded Structures over Infinite Data Domains

A. Bouajjani, P. Habermehl, Y. Jurski, and M. Sighireanu

In *Proc. FCT'07*, LNCS 4639, 2007

Finding Loop Invariants for Programs over Arrays Using
a Theorem Prover

L. Kovács and A. Voronkov

In *Proc. of FASE'09*, LNCS 5503, Springer, 2009

A Logic-Based Framework for Reasoning about Composite Data Structures

A. Bouajjani, C. Dragoi, C. Enea, M. Sighireanu

In *Proc. of CONCUR'09*, LNCS 5710, Springer, 2009

Questions?