SMT and Its Application in **Software Verification (Part II)**

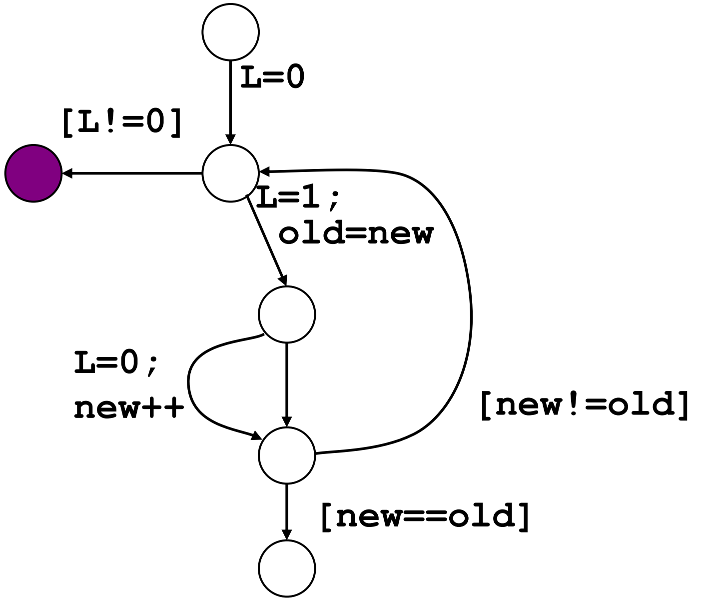
Yu-Fang Chen
IIS, Academia Sinica

Based on the slides of Barrett, Sanjit, Kroening , Rummer, Sinha, Jhala, and Majumdar, McMillan

Lazy abstraction -- an example

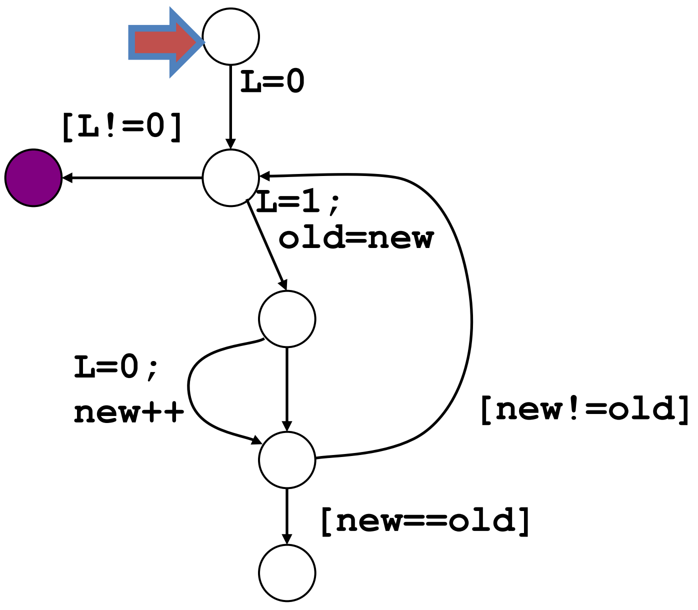
```
do{
  lock();
  old = new;
  if(*){
    unlock();
    new++;
  }
} while (new != old);
```

program fragment

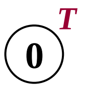


control-flow graph

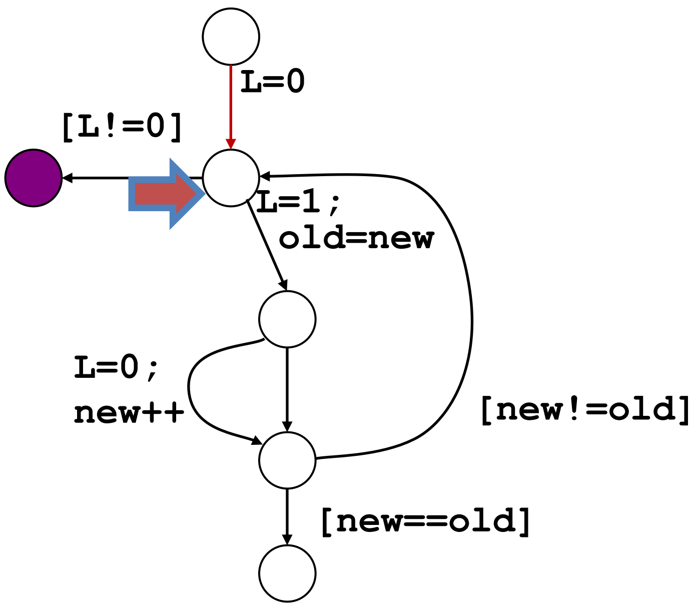
Unwinding the CFG



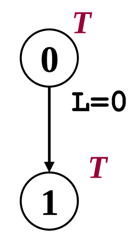
control-flow graph



Unwinding the CFG



control-flow graph



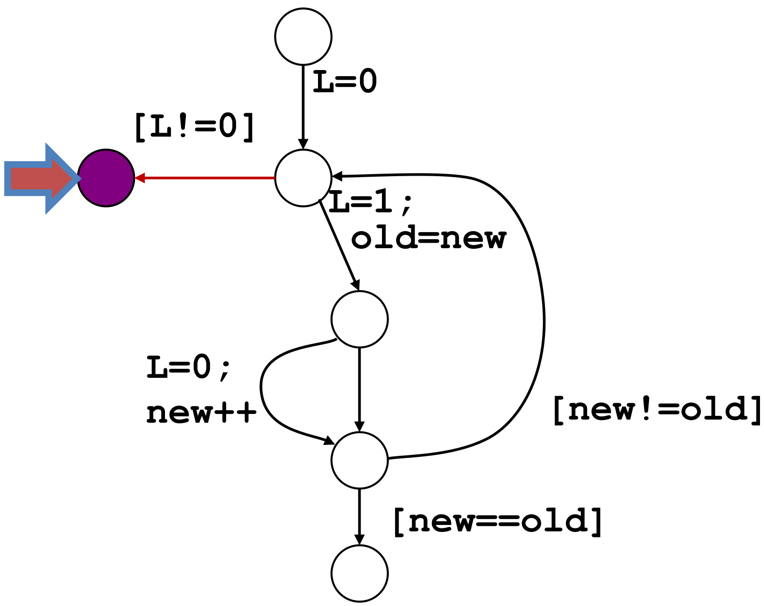
Replace all free occurrences of L in the formula with L’

Compute Post (***T***, L=0)= ***T***[L/L’] ∧L=0[L/L’]

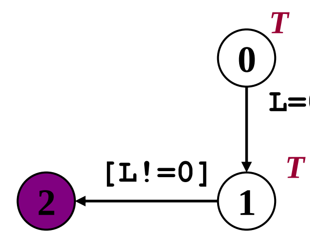
= (L=0)

Make Abstraction (L=0) → ***T*** **Pass**

Unwinding the CFG



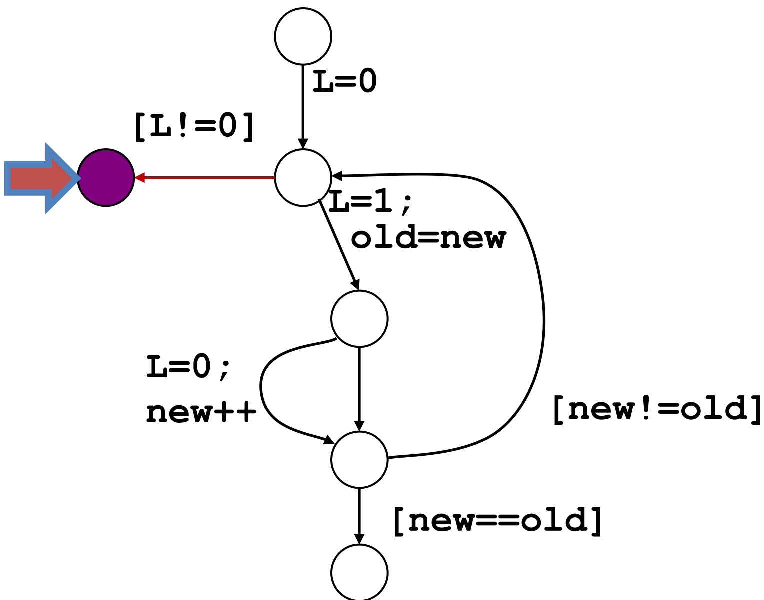
control-flow graph



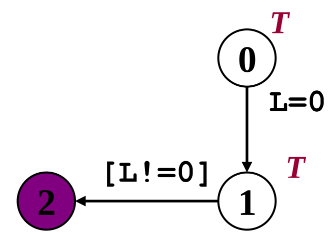
Compute Post (***T***, [L!=0])= ***T*** $\wedge$(L!=0)
= (L!=0)

ERROR state reached!

Unwinding the CFG

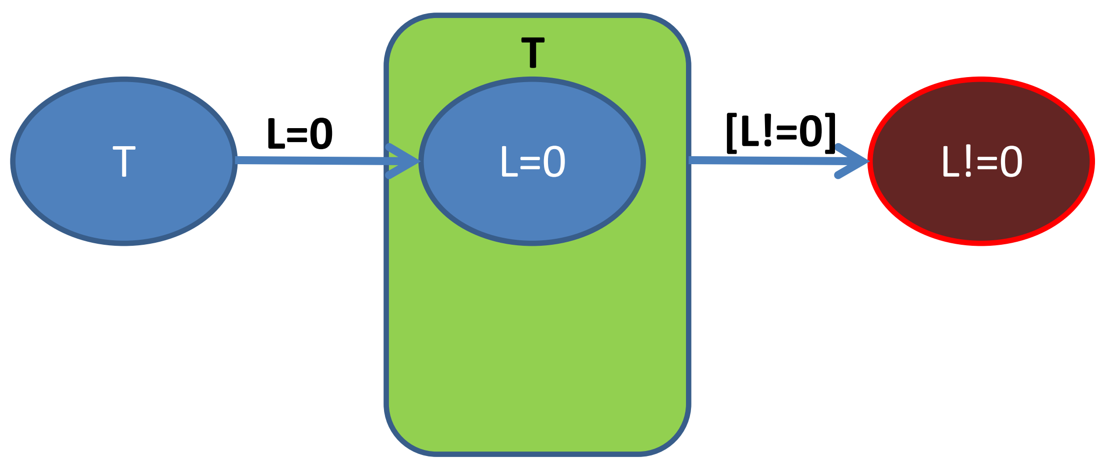


control-flow graph

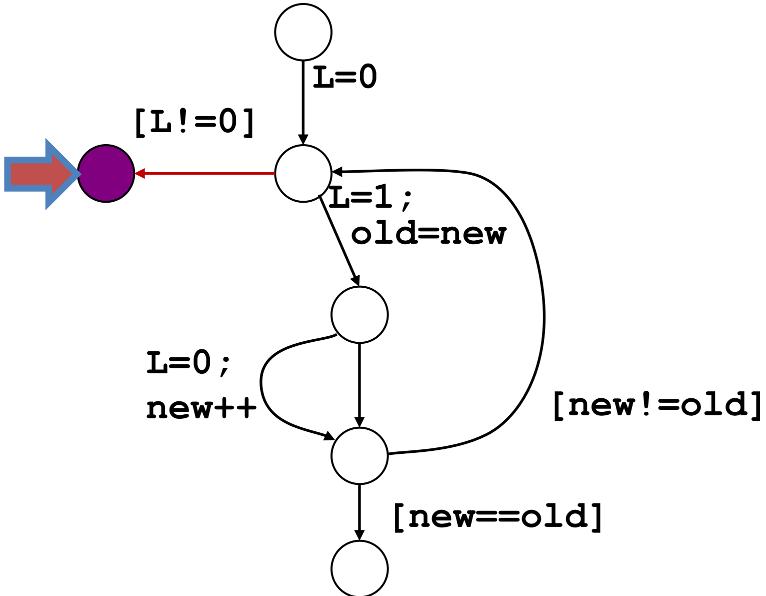


Compute Post (***T***, [L!=0])= ***T*** ∧(L!=0)
= (L!=0)

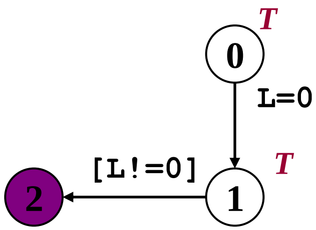
ERROR state reached!



Unwinding the CFG

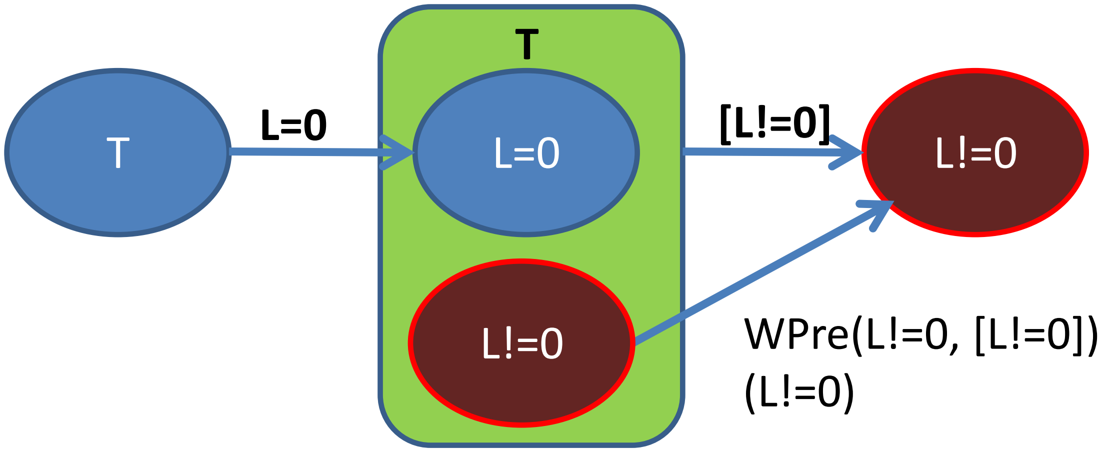


control-flow graph



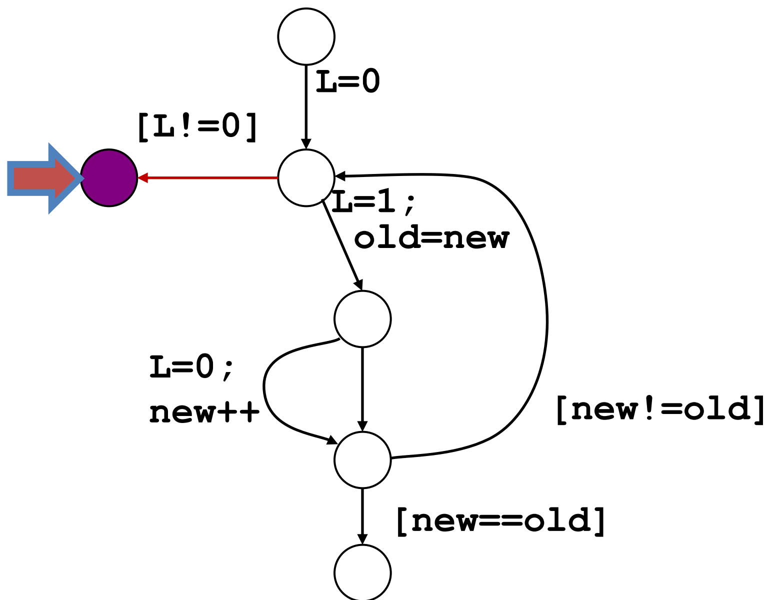
Compute Post (***T***, [L!=0])= ***T*** ∧(L!=0)
= (L!=0)

ERROR state reached!

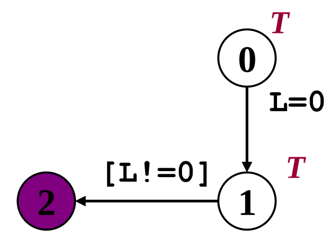


WPre(L!=0, [L!=0]) = (L!=0)

Unwinding the CFG

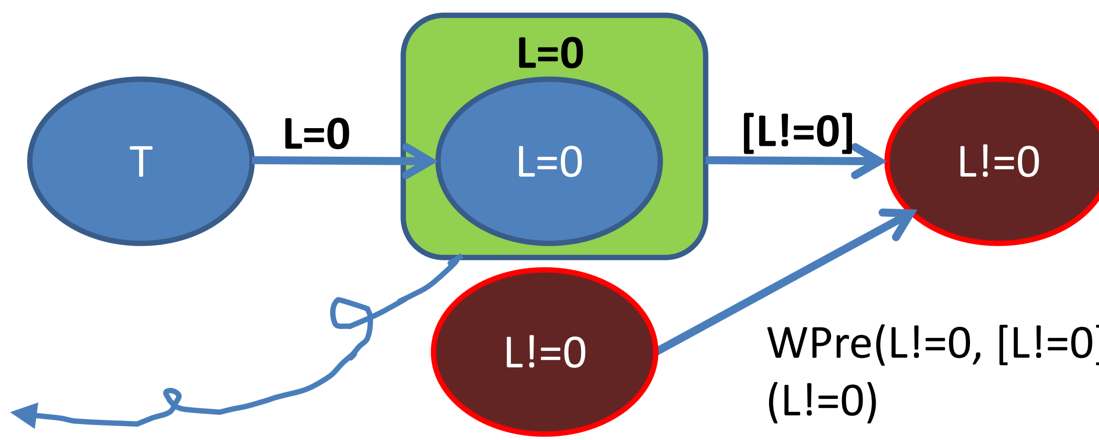


control-flow graph



Compute Post (***T***, [L!=0])= ***T*** ∧(L!=0)
= (L!=0)

ERROR state reached!

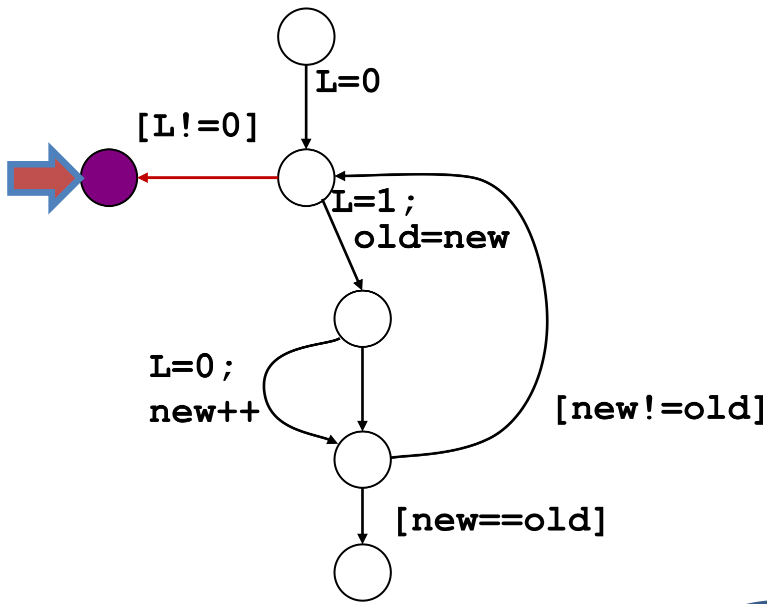


WPre(L!=0, [L!=0]) = (L!=0)

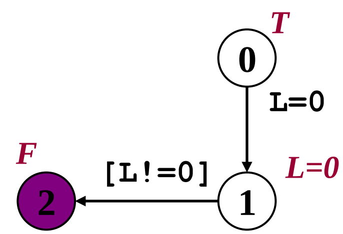
Compute Craig Interpolant: (L=0)

1. (L=0) → (L=0)
2. (L=0) ∧ (L=0) is UNSAT
3. Use only share var. of (L=0) and (L!=0)

Unwinding the CFG



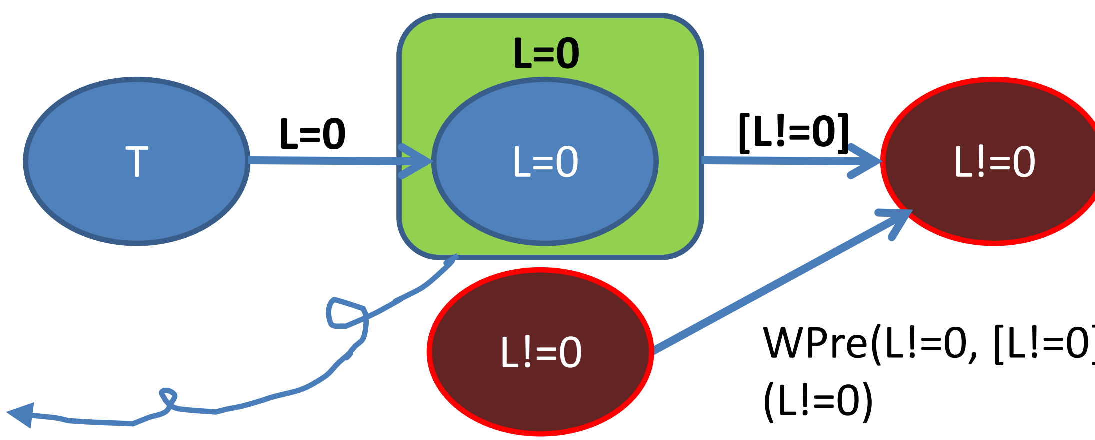
control-flow graph



Compute Post (***T***, [L!=0])= ***T*** ∧(L!=0)

= (L!=0)

ERROR state reached!

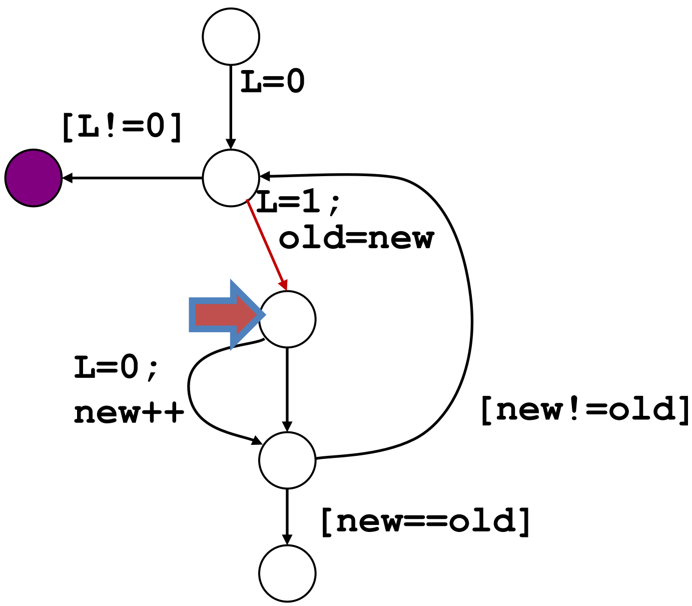


WPre(L!=0, [L!=0]) = (L!=0)

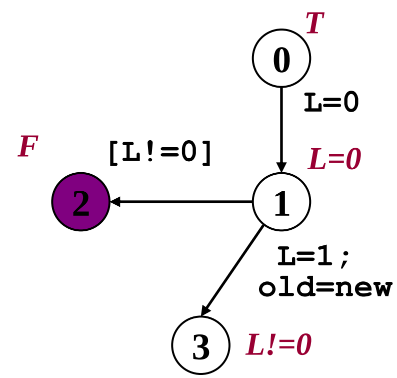
Compute Craig Interpolant: (L=0)

1. (L=0) → (L=0)
2. (L=0) ∧ (L=0) is UNSAT
3. Use only share var. of (L=0) and (L!=0)

Unwinding the CFG



control-flow graph



Compute Post (***L=0***, L=1)

= (L=0)[L/L’] ∧L=1[L/L’]

= (L’=0 ∧L=1)

Compute Post (**L’=0 ∧L=1**, old=new)

= (L’=0 ∧L=1)[old/old’] ∧ old=new[old/old’]

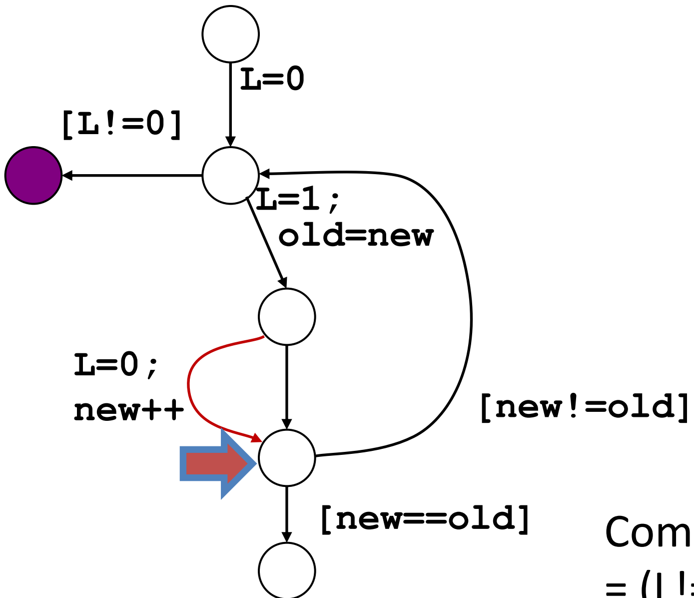
= L’=0 ∧L=1∧old=new

Make Abstraction

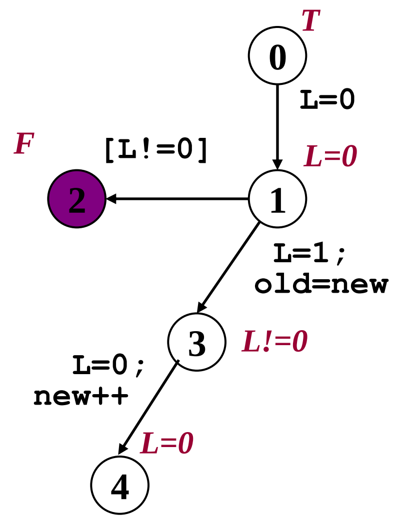
(L’=0 ∧L=1∧old=new) → **(L!=0)** Pass

(L’=0 ∧L=1∧old=new) → **(L=0)** Not Passed

Unwinding the CFG

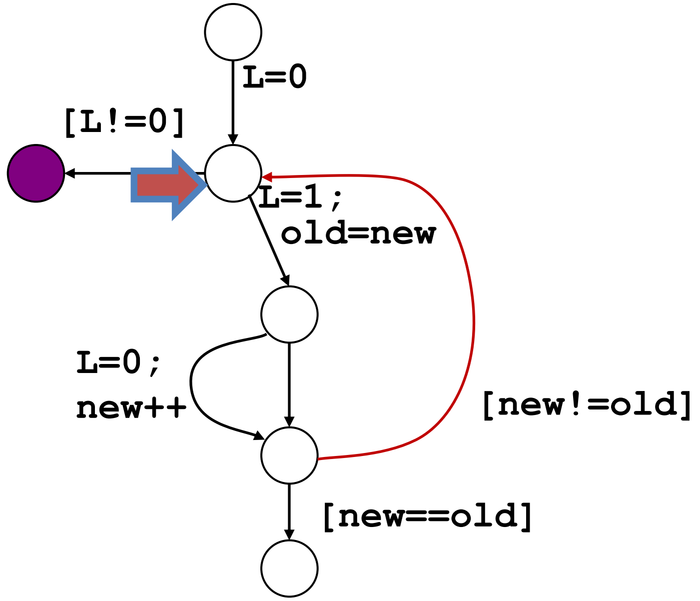


control-flow graph

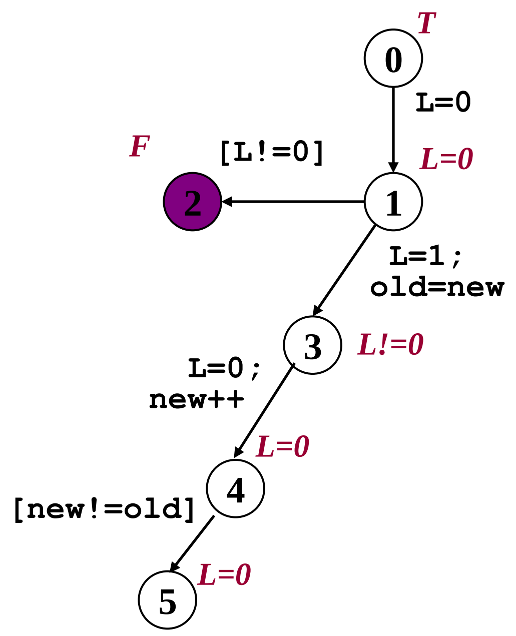


Compute Post (***L!=0***, L=0)
= (L!=0)[L/L’] ∧L=0[L/L’]
= (L’!=0 ∧L=0)
Compute Post (**L’!=0 ∧(L=0)**, new=new+1)
= (L’!=0 ∧L=0)[new/new’] ∧ new=(new+1)[new/new’]
= (L’!=0 ∧L=0 ∧ new=new’+1)
Make Abstraction
(L’!=0 ∧L=0 ∧ new=new’+1) → **(L!=0)** **Not Passed**
(L’!=0 ∧L=0 ∧ new=new’+1) → **(L=0)** **Pass**

Unwinding the CFG



control-flow graph



Compute Post (***L=0***, [new!=old])

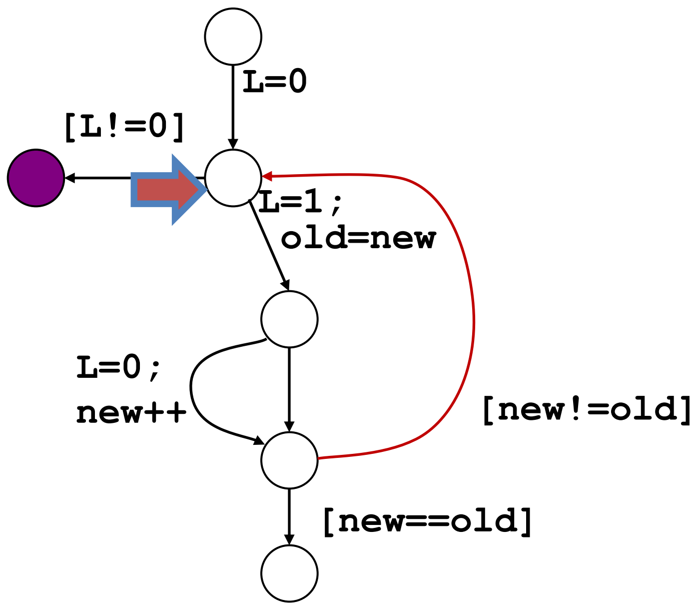
= (L=0 ∧ new!=old)

Make Abstraction

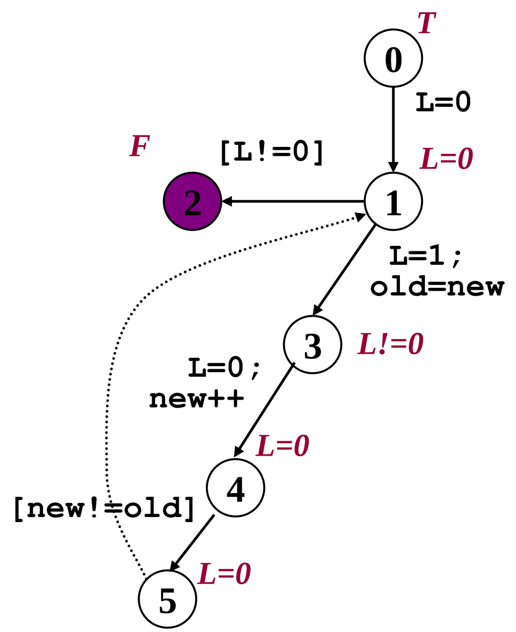
(L=0 ∧ new!=old) → **(L!=0)** **Not Passed**

(L=0 ∧ new!=old) → **(L=0)** **Pass**

Unwinding the CFG



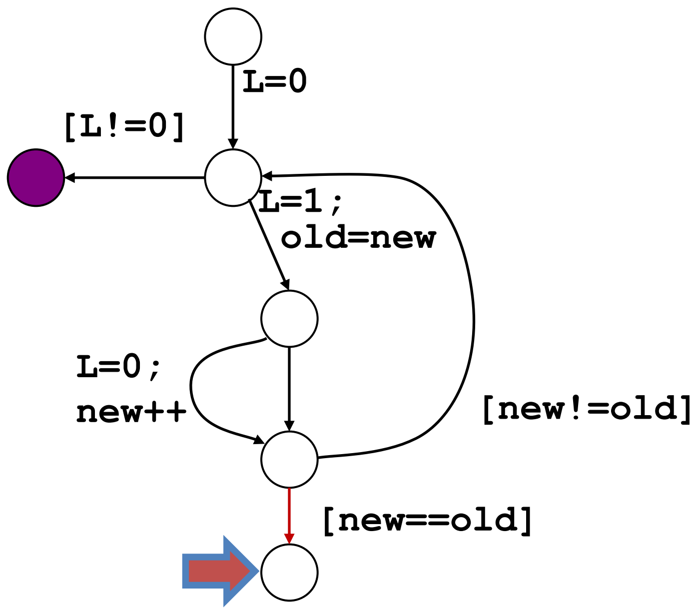
control-flow graph



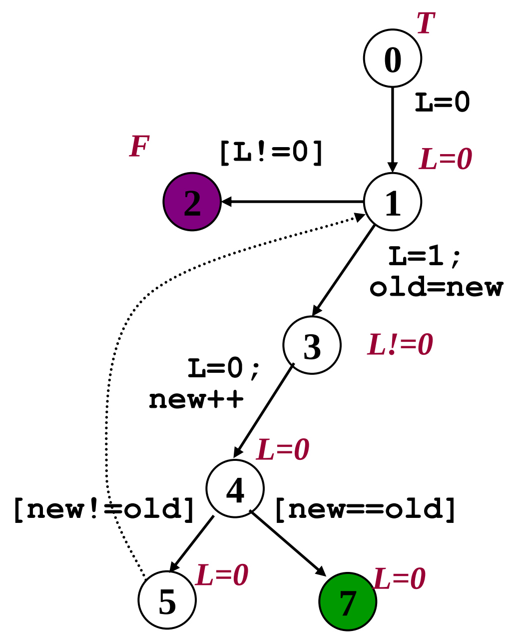
Covering: state 5 is subsumed by state 1.

L=1 → L=1 **Pass**

Unwinding the CFG



control-flow graph



Compute Post (***L=0***, [new==old])

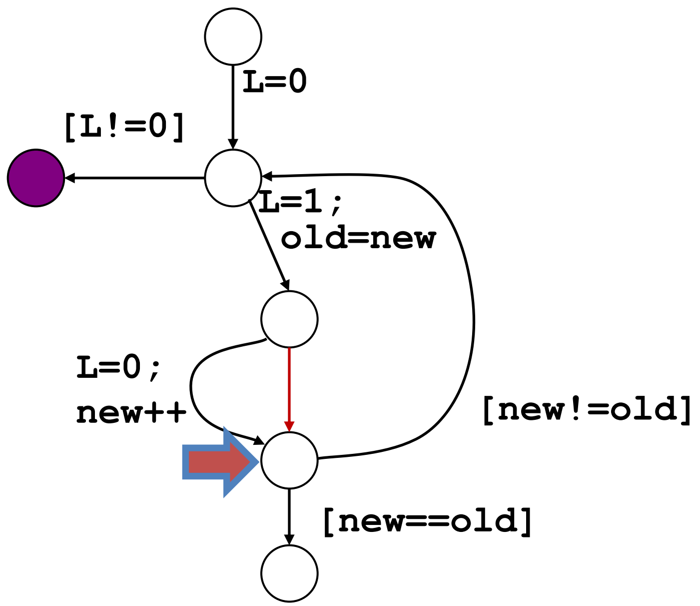
= (L=0 $\wedge$ new==old)

Make Abstraction

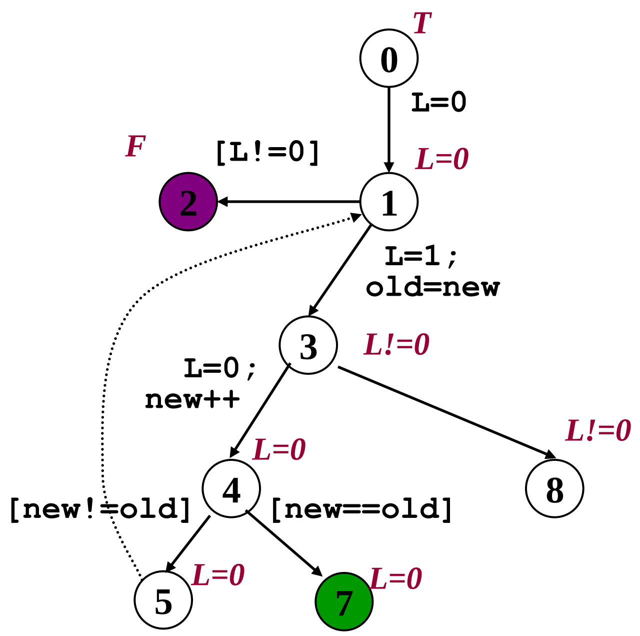
(L=0 $\wedge$ new!=old) → **(L!=0)** **Not Passed**

(L=0 $\wedge$ new!=old) → **(L=0)** **Pass**

Unwinding the CFG

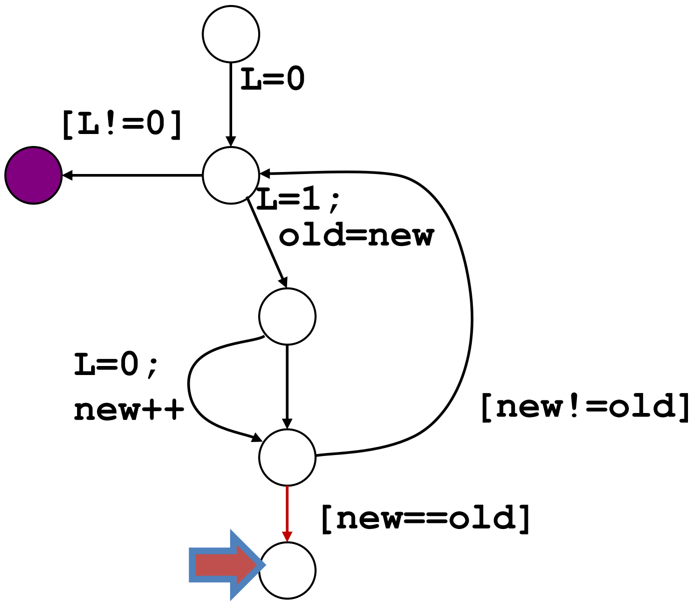


control-flow graph

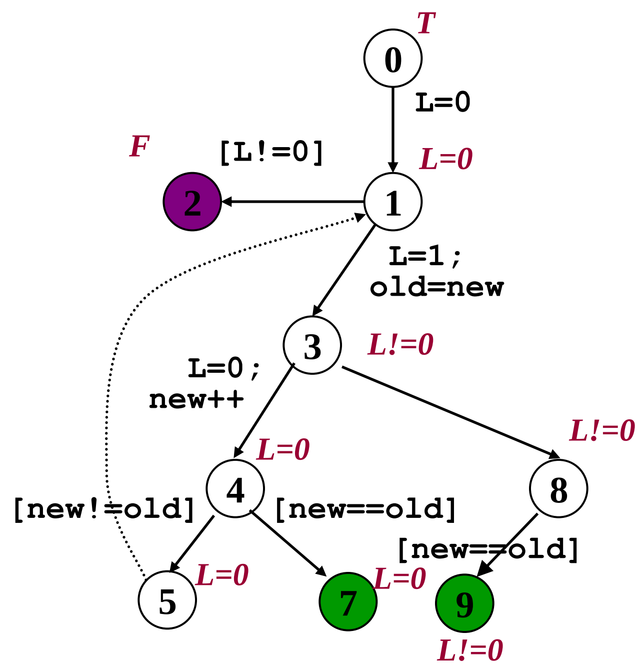


No Actions

Unwinding the CFG



control-flow graph



Compute Post (***L!=0***, [new==old])

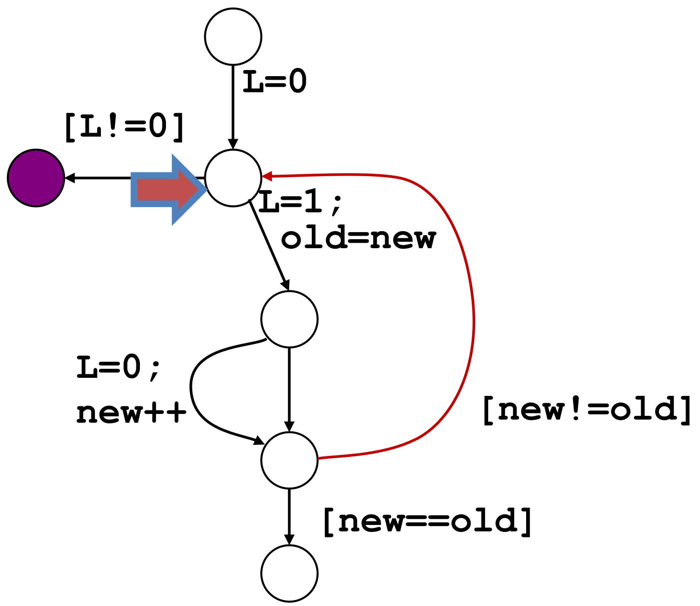
= (L!=0 ∧ new==old)

Make Abstraction

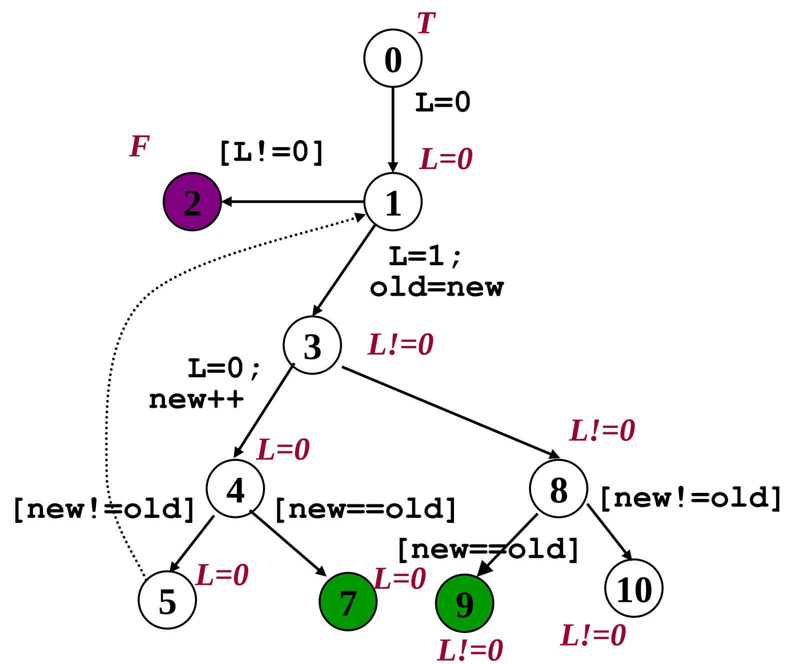
(L!=0 ∧ new==old) → **(L!=0)** Pass

(L!=0 ∧ new==old) → **(L=0)** **Not Passed**

Unwinding the CFG



control-flow graph



Compute Post (***L!=0***, [new!=old])

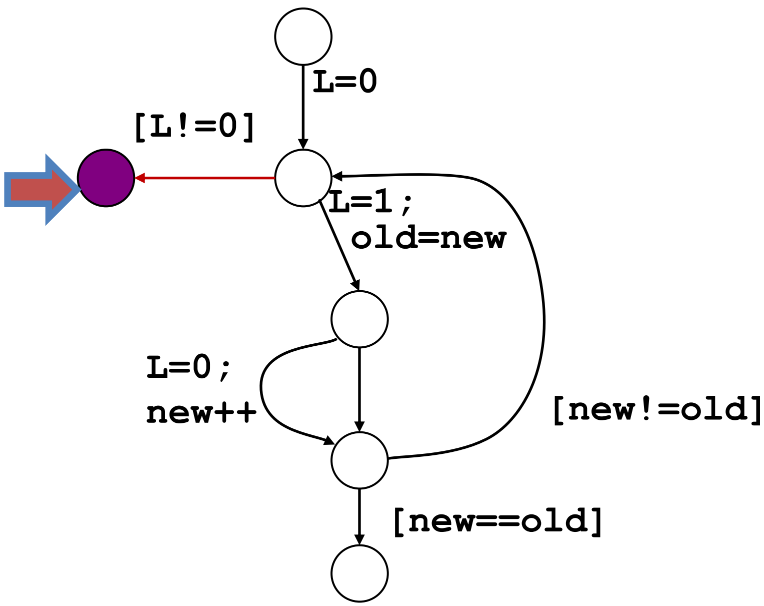
= (L!=0 ∧ new!=old)

Make Abstraction

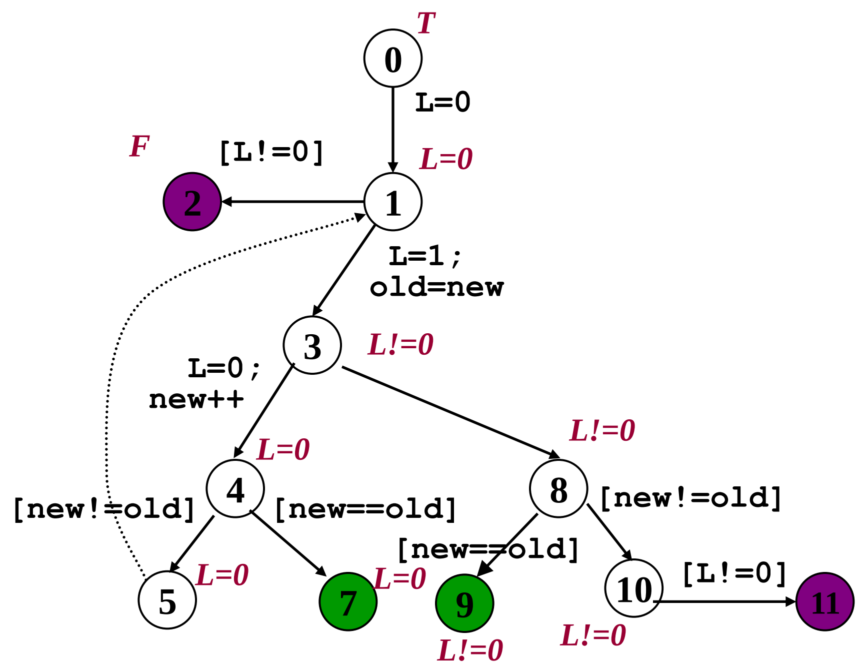
(L!=0 ∧ new!=old) → **(L!=0)** Pass

(L!=0 ∧ new!=old) → **(L=0)** **Not Passed**

Unwinding the CFG



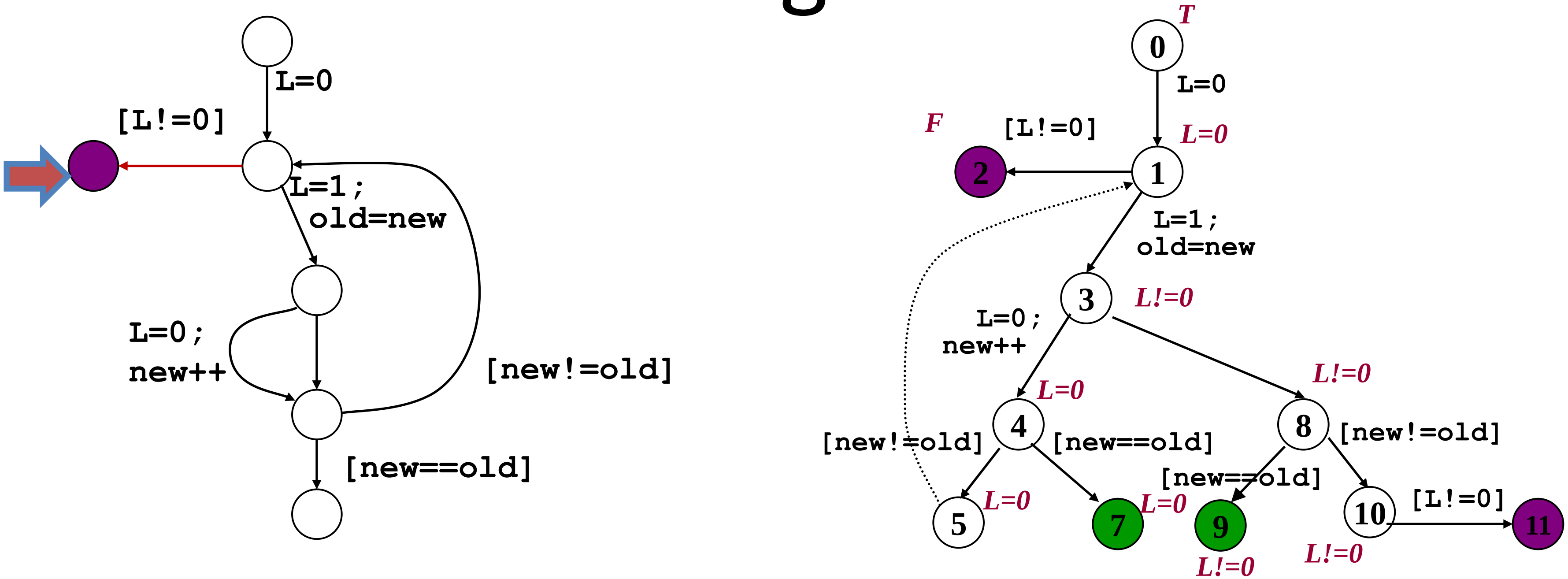
control-flow graph



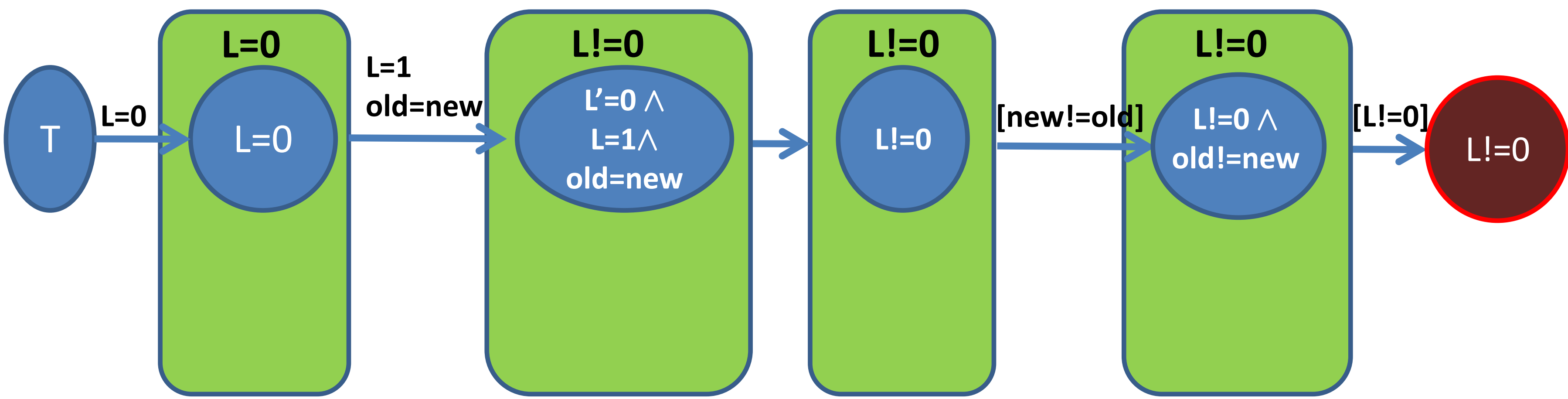
Compute Post (***L!=0***, [L!=0])
= (L!=0 $\wedge$ L!0)

ERROR state reached!

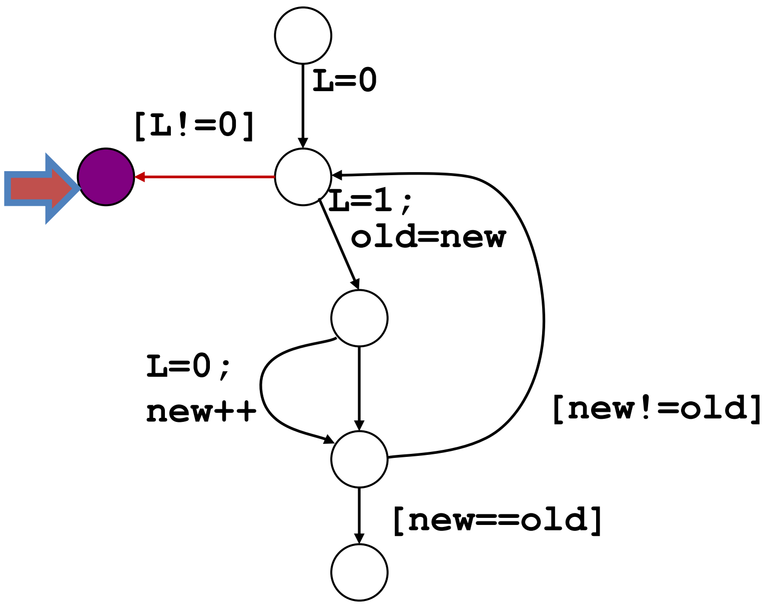
Unwinding the CFG

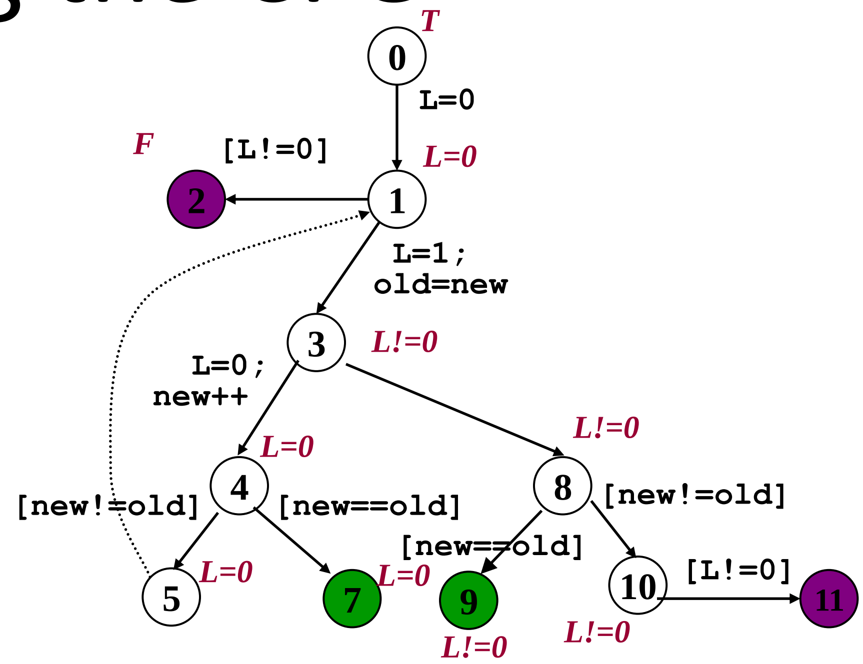


control-flow graph

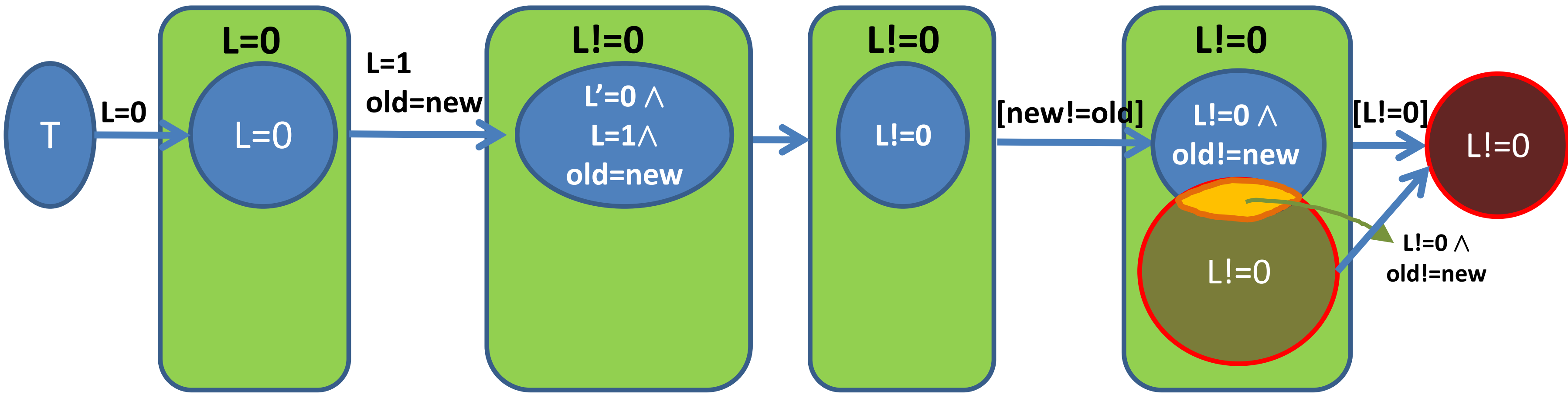


Unwinding the CFG

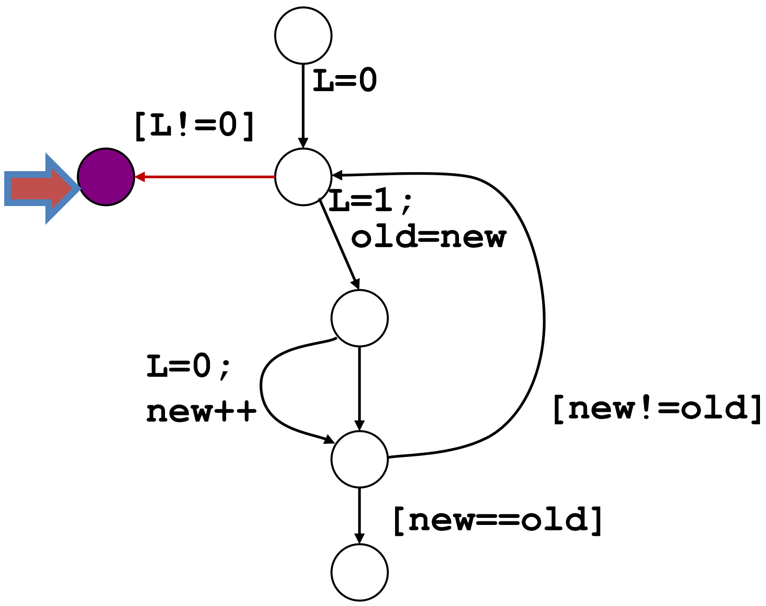




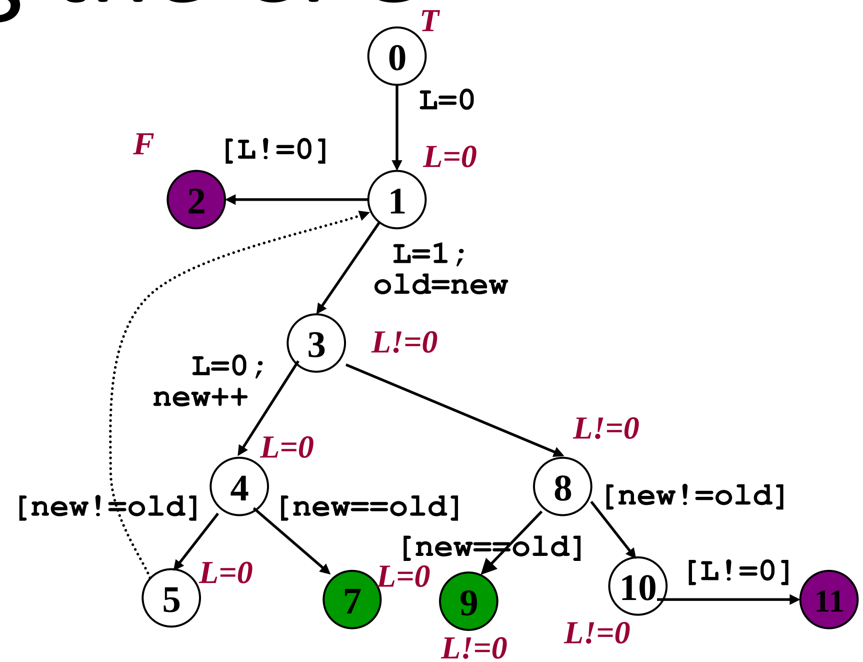
control-flow graph

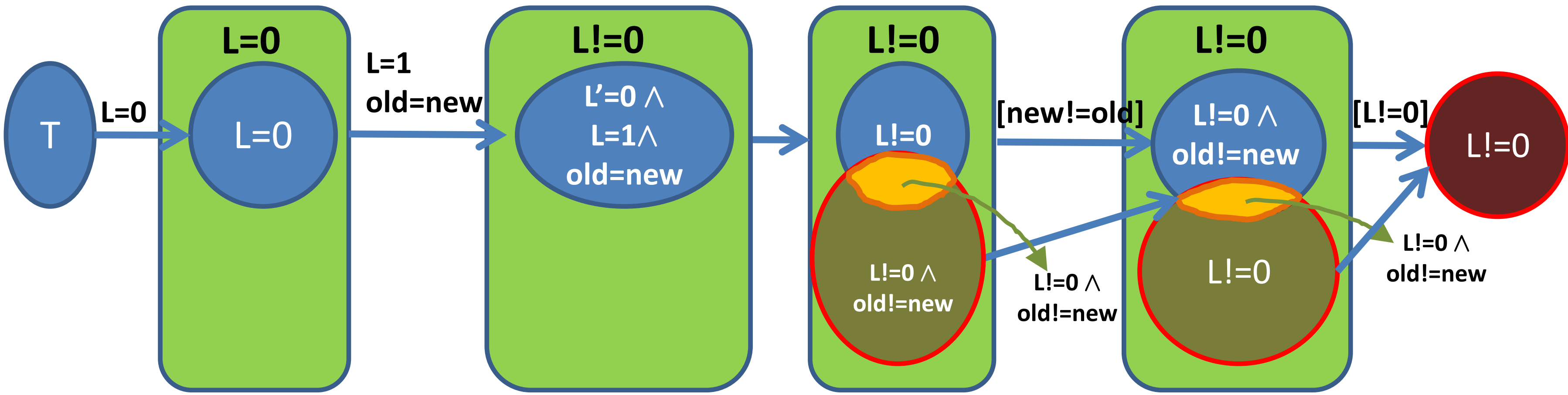


Unwinding the CFG

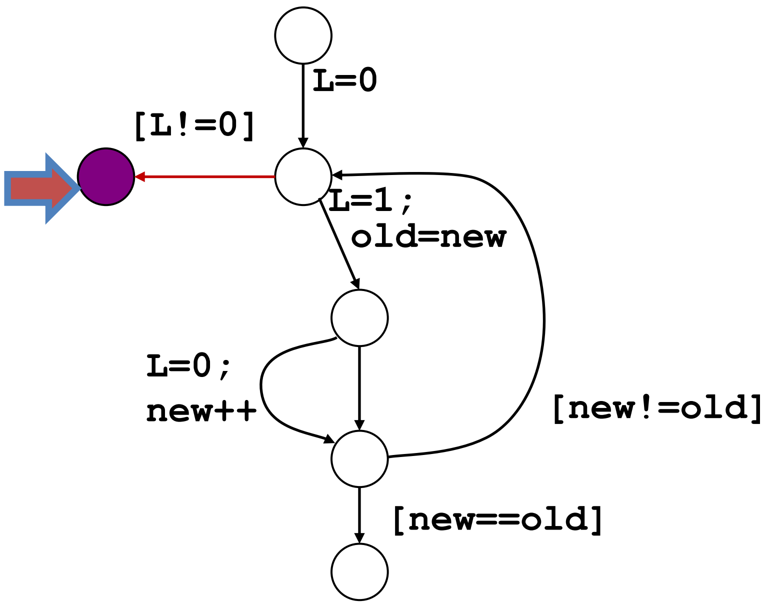


control-flow graph

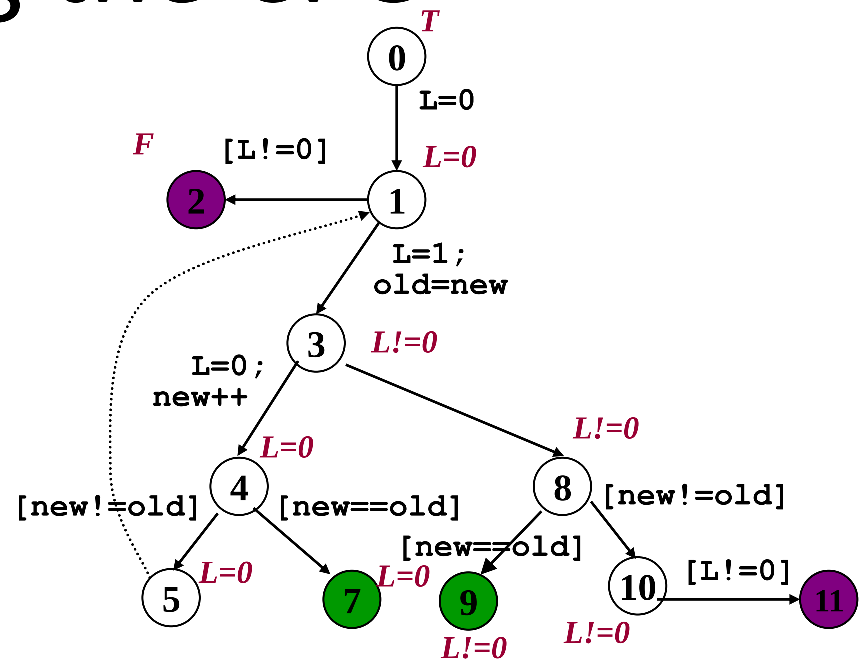


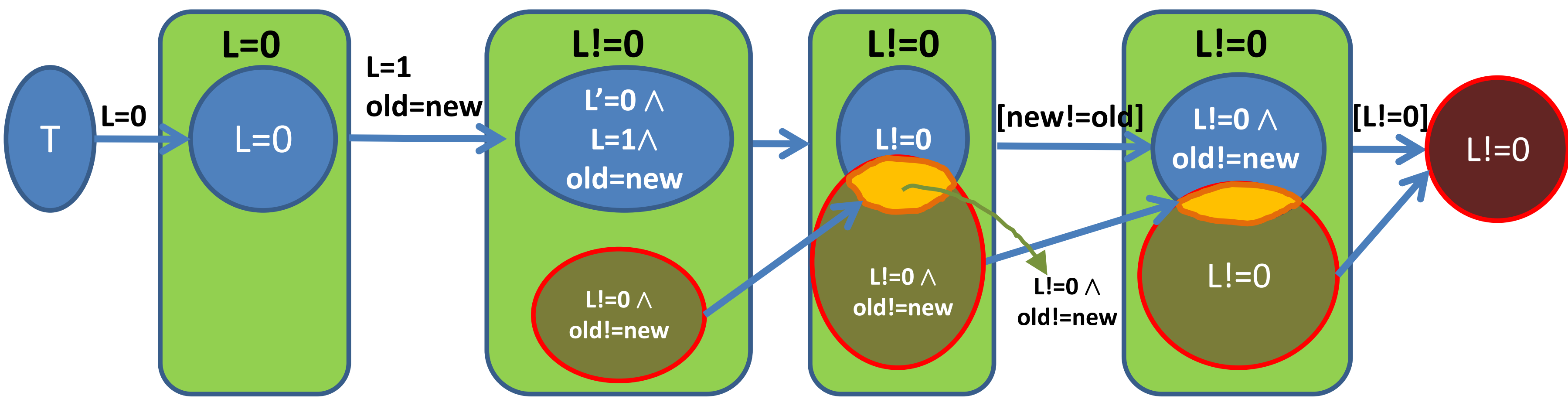


Unwinding the CFG

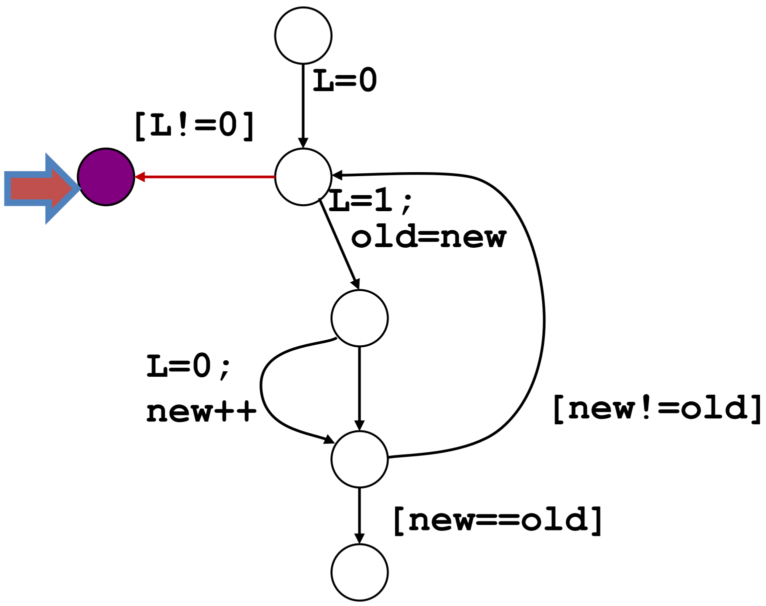


control-flow graph

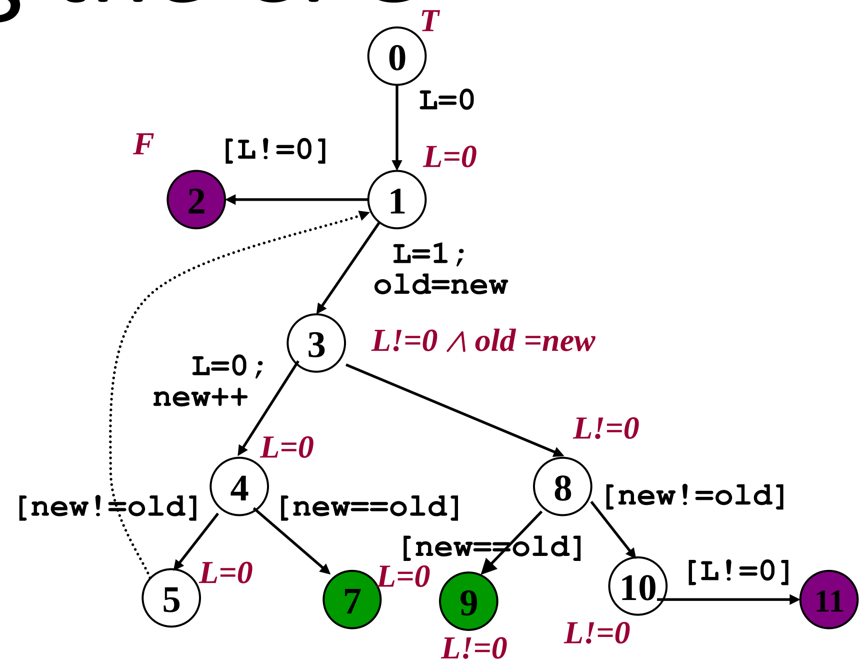


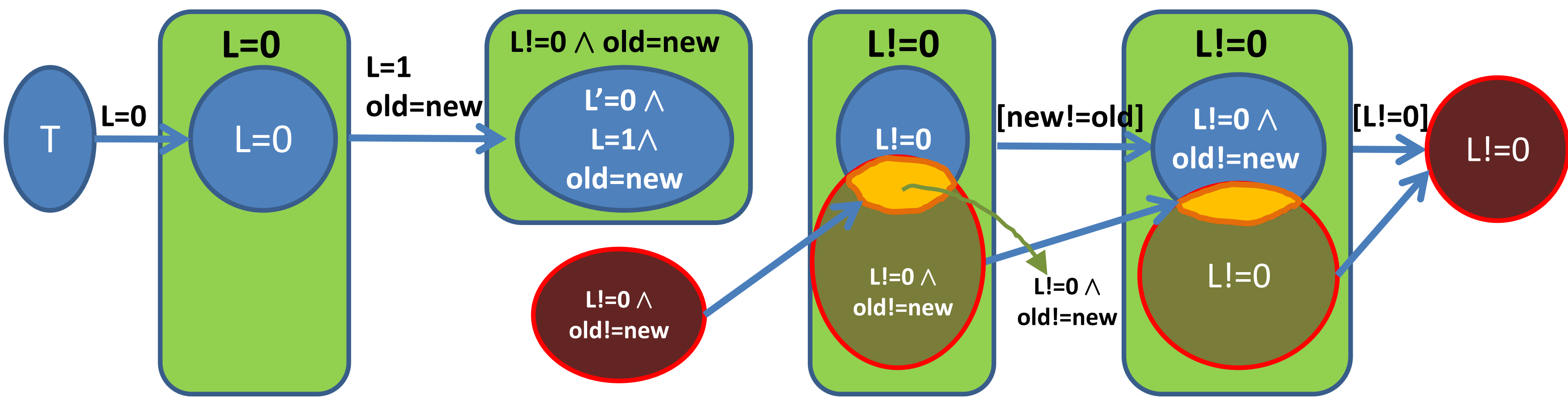


Unwinding the CFG

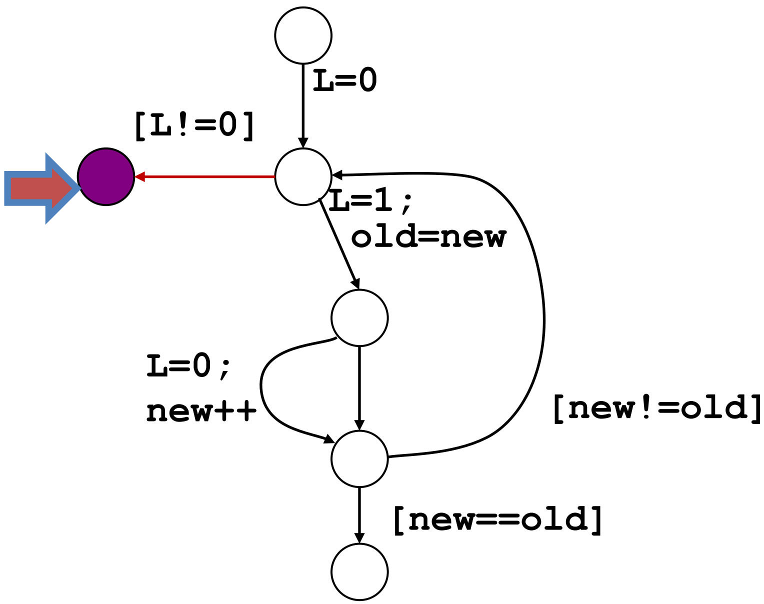


control-flow graph

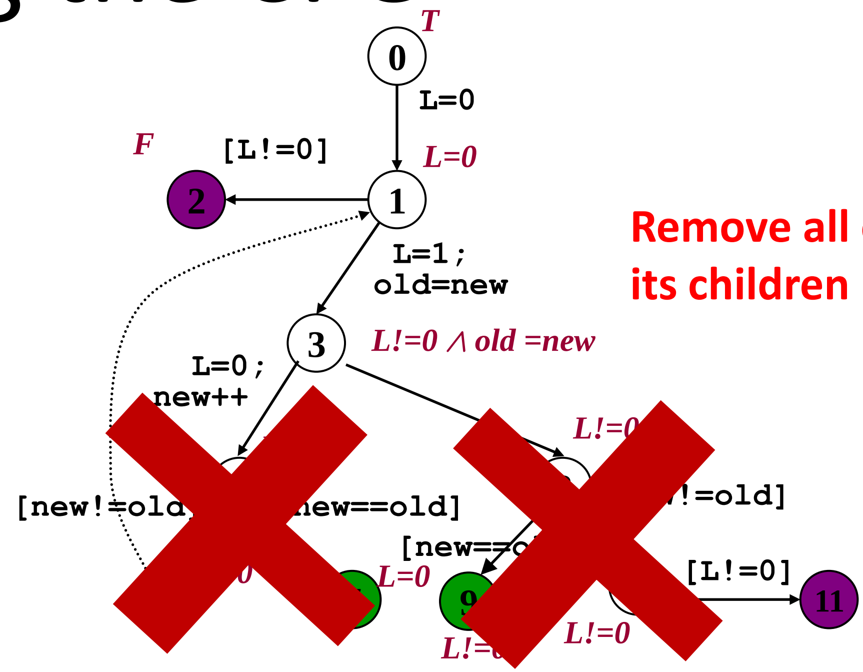




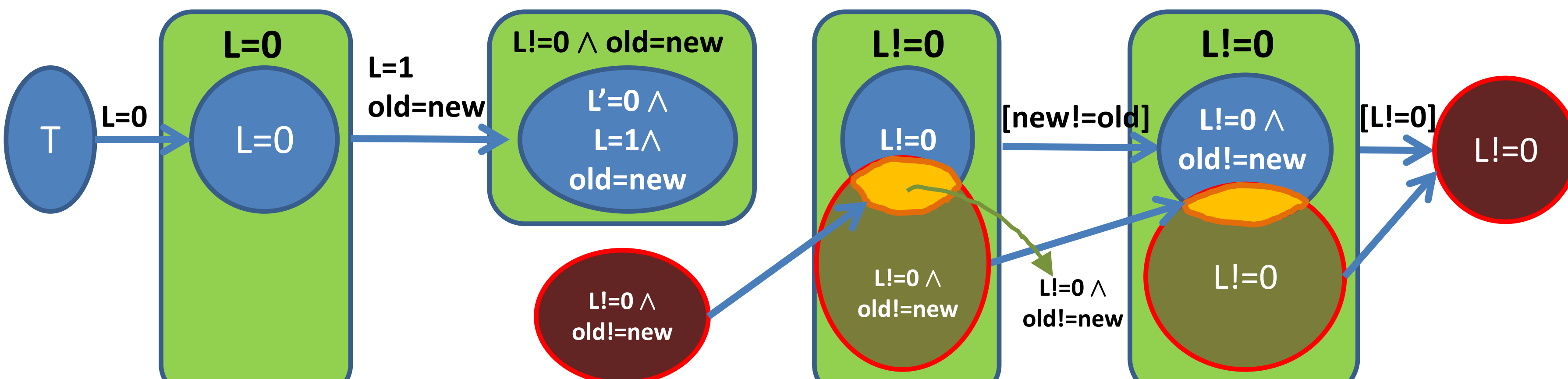
Unwinding the CFG



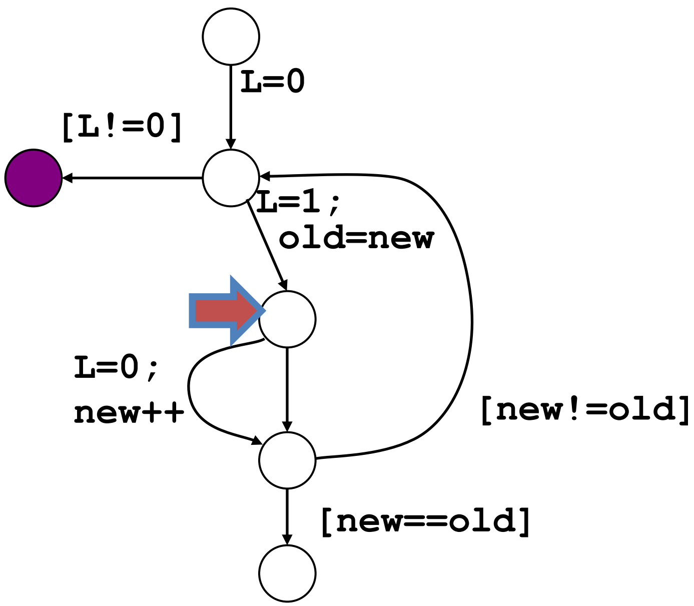
control-flow graph



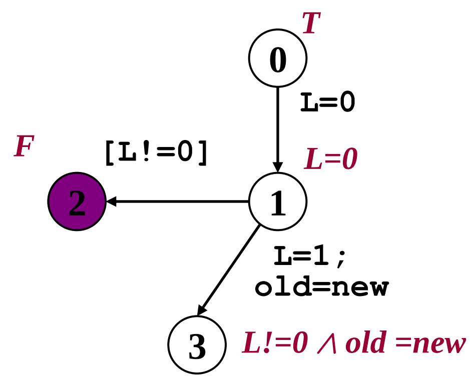
Remove all of its children



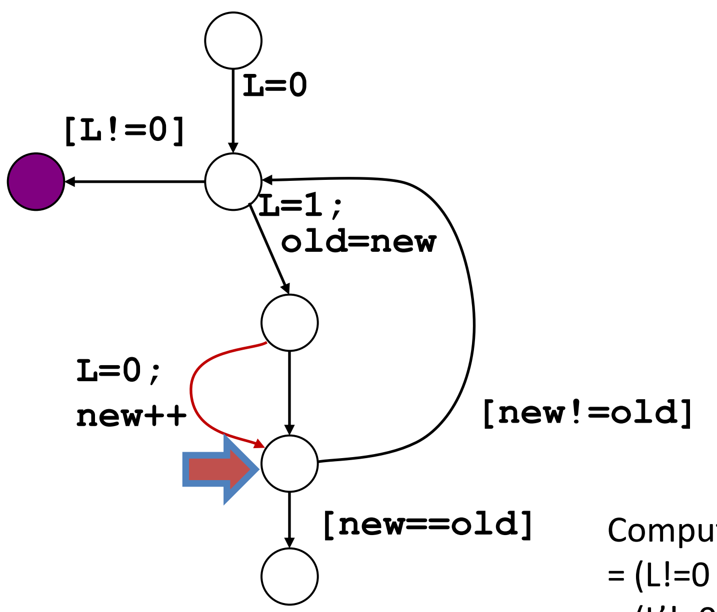
Unwinding the CFG



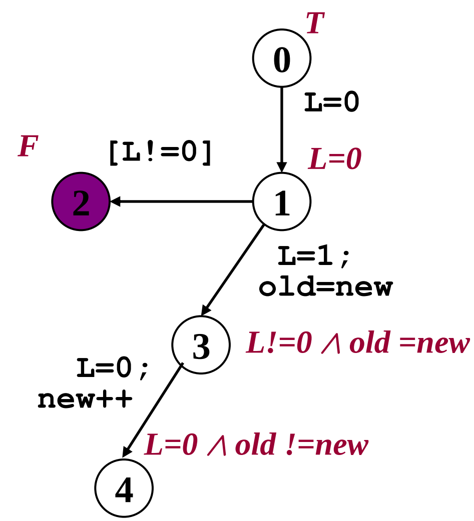
control-flow graph



Unwinding the CFG

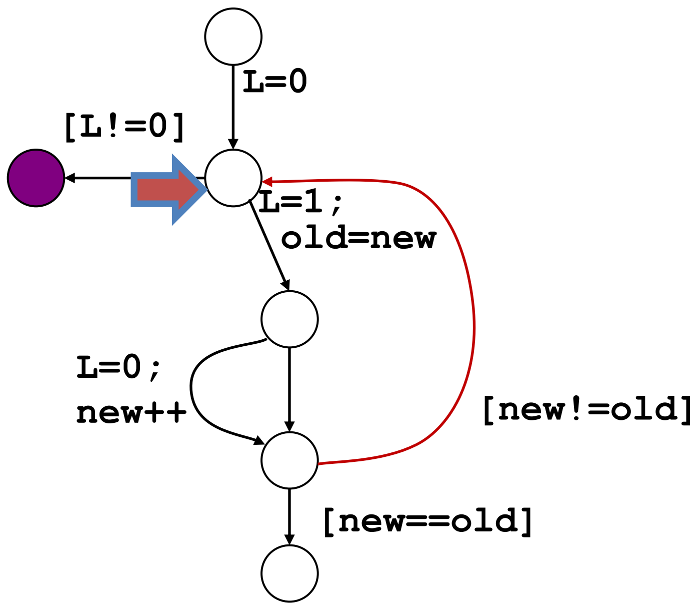


control-flow graph

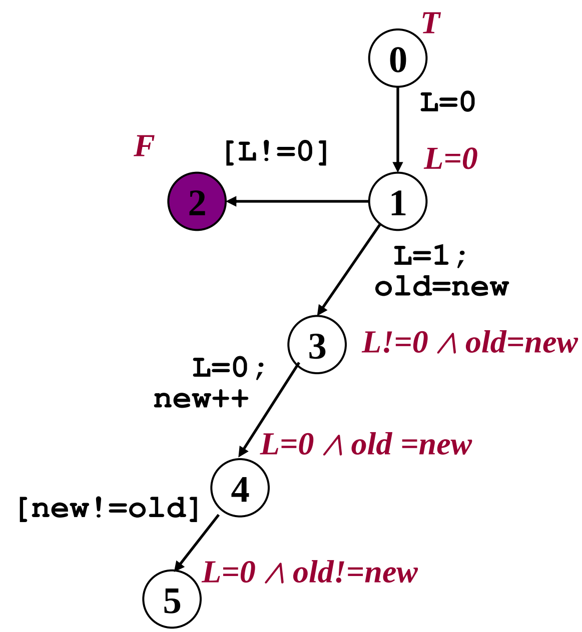


Compute Post (***L!=0*** ***∧ old =new***, L=0)
= (L!=0 ∧ old=new)[L/L'] ∧L=0[L/L']
= (L'!=0 ∧ old=new ∧L=0)
Compute Post (**L'!=0 ∧ old=new ∧(L=0)**, new=new+1)
= (L'!=0 ∧ old=new ∧L=0)[new/new'] ∧ new=(new+1)[new/new']
= (L'!=0 ∧ old=new' ∧L=0 ∧ new=new'+1)
Make Abstraction
(L'!=0 ∧ old=new' ∧L=0 ∧ new=new'+1) → **(L!=0)** **Not Passed**
(L'!=0 ∧ old=new' ∧L=0 ∧ new=new'+1) → **(L=0)** **Pass**
(L'!=0 ∧ old=new' ∧L=0 ∧ new=new'+1) → **(old=new)** **Not Passed**
(L'!=0 ∧ old=new' ∧L=0 ∧ new=new'+1) → **(old!=new)** **Pass**

Unwinding the CFG



control-flow graph



Compute Post (***L=0 ∧ old!=new***, [new!=old])

= (L=0 ∧ new!=old)

Make Abstraction

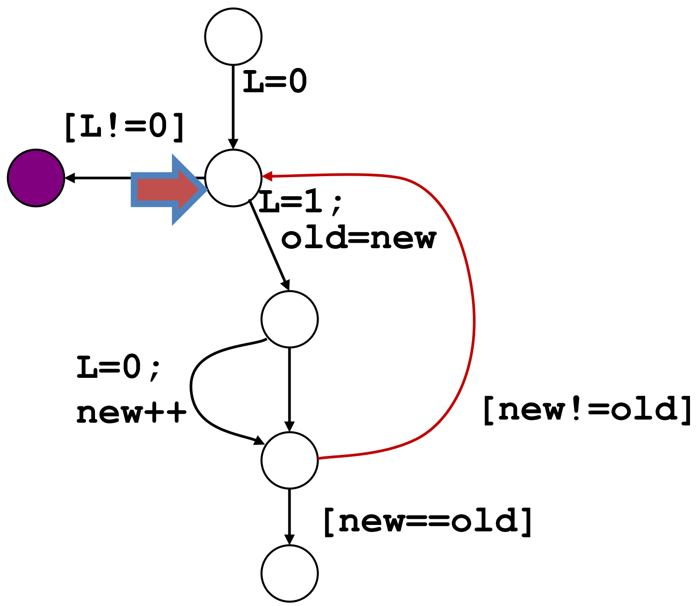
(L=0 ∧ new!=old) → **(L!=0)** Not Passed

(L=0 ∧ new!=old) → **(L=0)** Pass

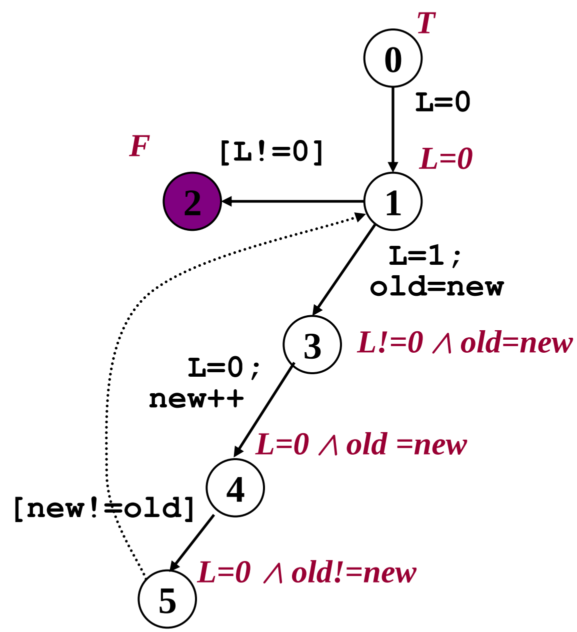
(L=0 ∧ new!=old) → **(old=new)** Not Passed

(L=0 ∧ new!=old) → **(old!=new)** Pass

Unwinding the CFG



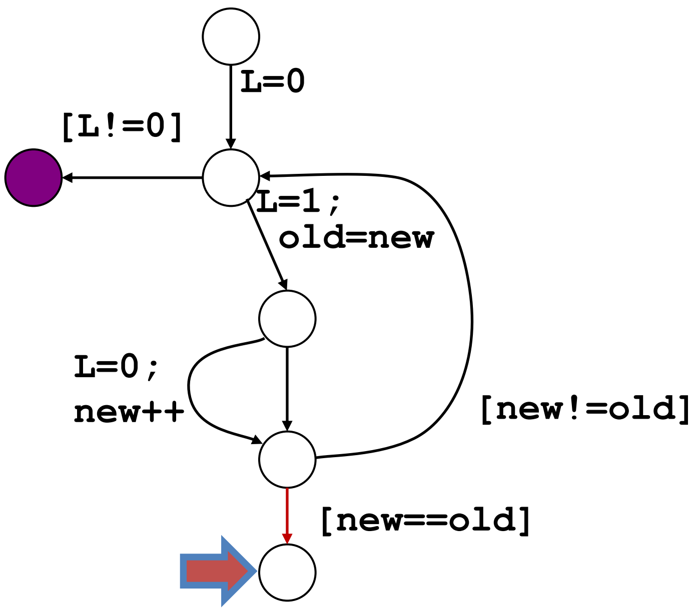
control-flow graph



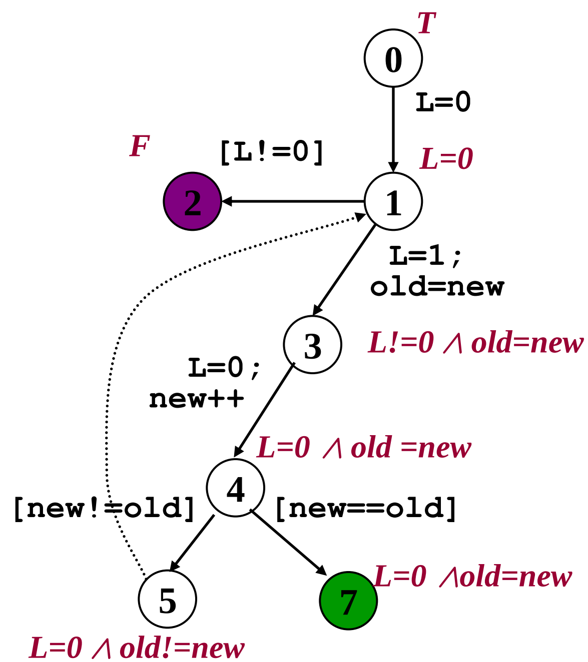
Covering: state 5 is subsumed by state 1.

L=1 ∧ old!=new→ L=1 **Pass**

Unwinding the CFG



control-flow graph



Compute Post (***L=0***, [new==old])

= (L=0 ∧ new=old)

Make Abstraction

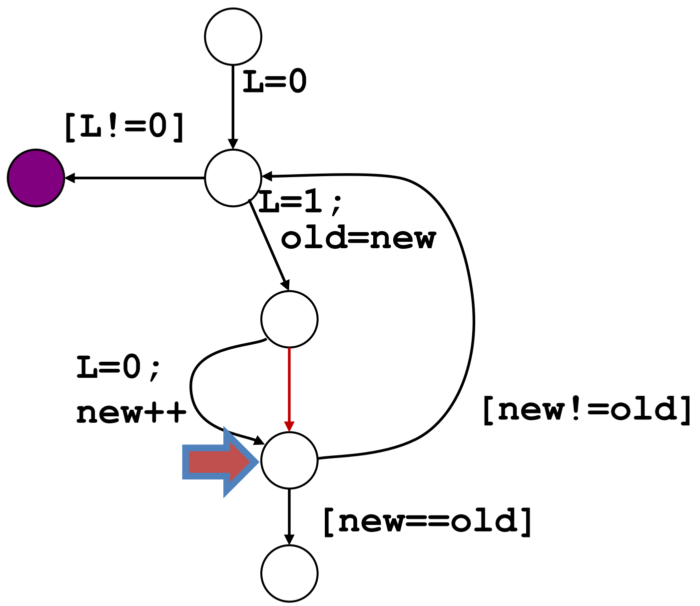
(L=0 ∧ new=old) → **(L!=0)** **Not Passed**

(L=0 ∧ new=old) → **(L=0)** **Pass**

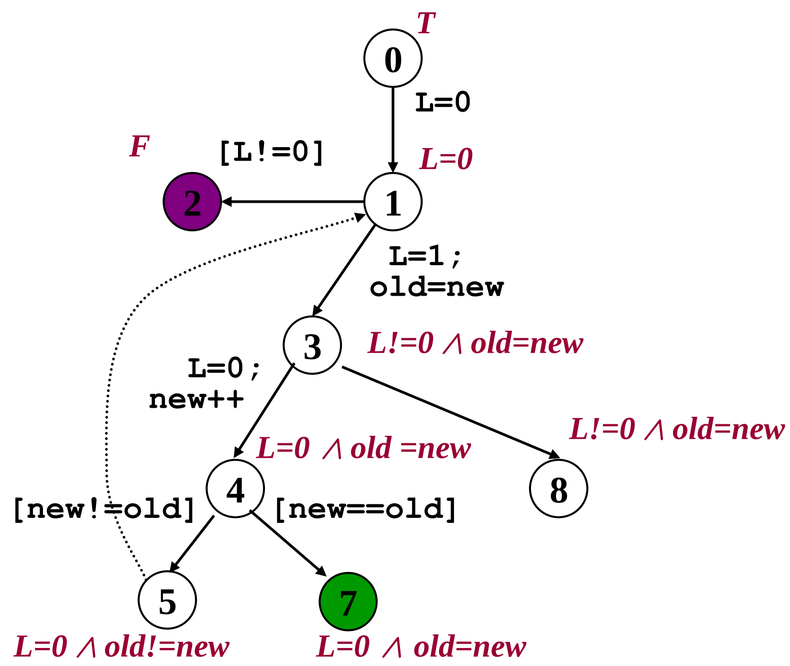
(L=0 ∧ new=old) → **(new!=old)** **Not Passed**

(L=0 ∧ new=old) → **(new=old)** **Pass**

Unwinding the CFG

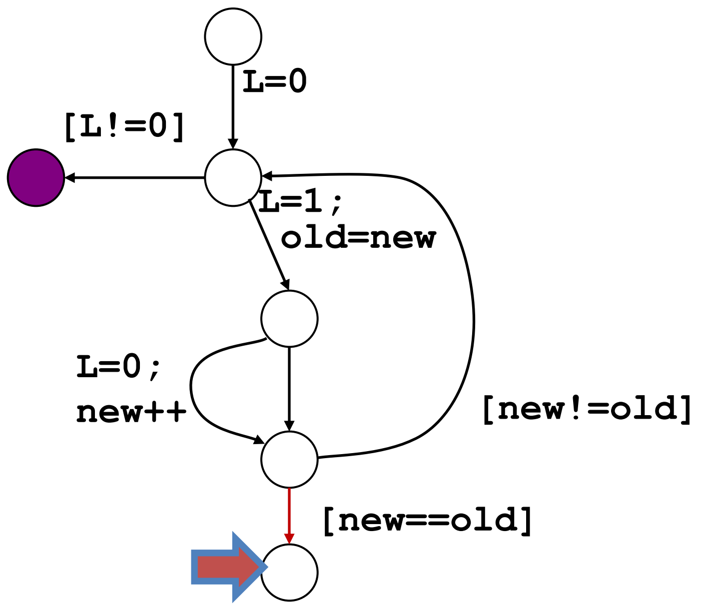


control-flow graph

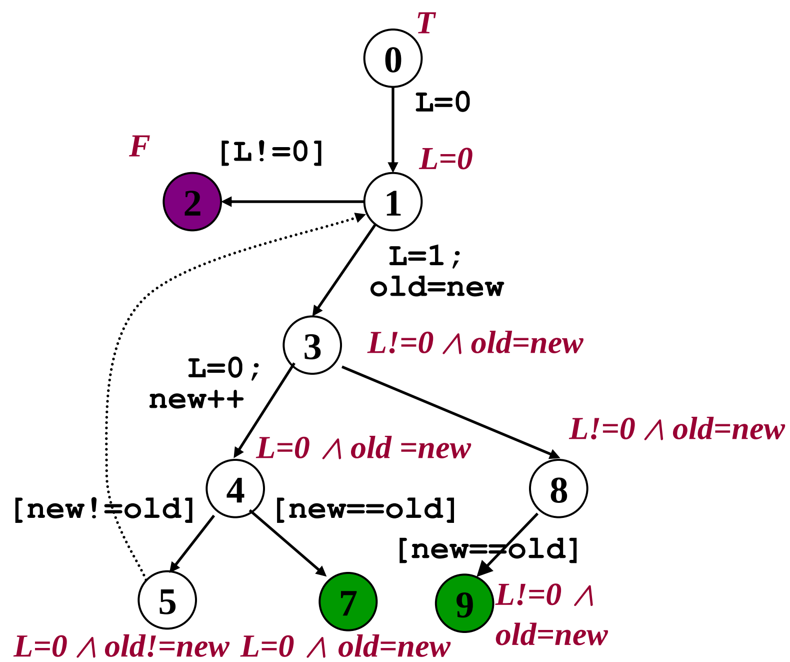


No Actions

Unwinding the CFG

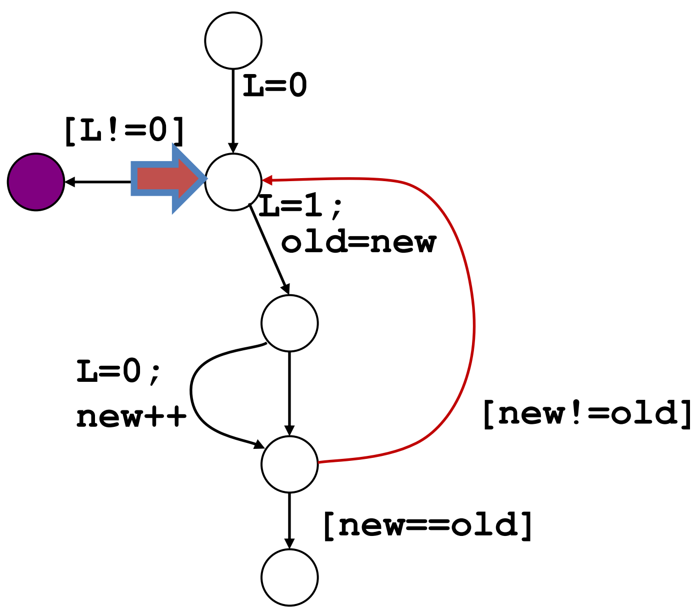


control-flow graph

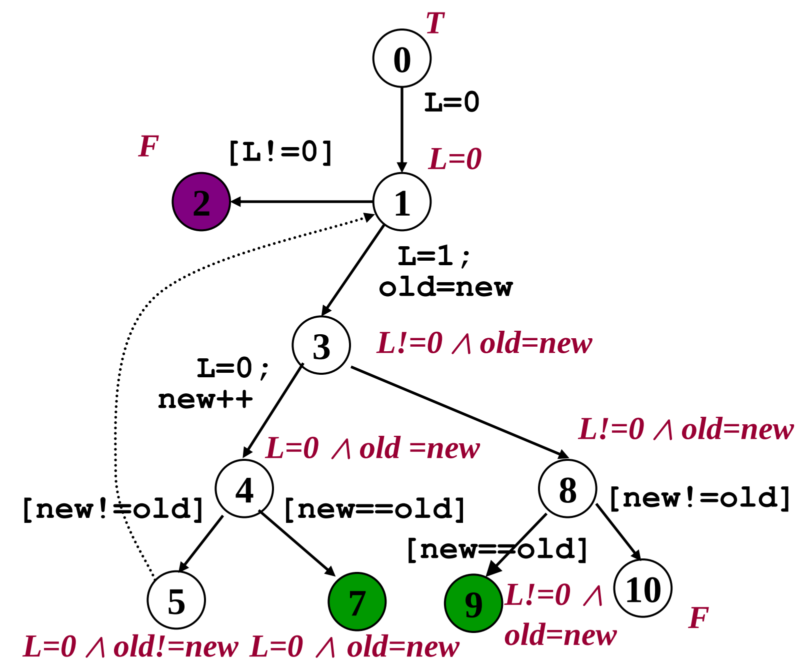


Compute Post (***L!=0***∧**new=old**, [new==old])
= (L!=0 ∧ new=old)
Make Abstraction
(L!=0 ∧ new=old) → **(L!=0)** Pass
(L!=0 ∧ new=old) → **(L=0)** Not Passed
(L!=0 ∧ new=old) → **(old=new)** Pass
(L!=0 ∧ new=old) → **(old!=new)** Not Passed

Unwinding the CFG



control-flow graph



Compute Post (***L!=0*∧new=old**, [new!=old])
= (L!=0 ∧ new=old ∧ new!=old)
= false

Another Approach: The IMPACT method

Kenneth L. McMillan: Lazy Abstraction with Interpolants. CAV 2006: 123-136

Interpolation Lemma

(Craig,57)

Interpolation Lemma

(Craig,57)

- Notation: $\mathcal{L}(\phi)$ is the set of FO formulas over the symbols of ϕ

Interpolation Lemma

(Craig,57)

- Notation: $\mathcal{L}(\phi)$ is the set of FO formulas over the symbols of ϕ
- If $A \wedge B$ = false, there exists an *interpolant* A' for (A,B) such that:

$$A \Rightarrow A'$$

$$A' \wedge B = \text{false}$$

$$A' \in \mathcal{L}(A) \cap \mathcal{L}(B)$$

Interpolation Lemma

(Craig,57)

- Notation: $\mathcal{L}(\phi)$ is the set of FO formulas over the symbols of ϕ
- If $A \wedge B$ = false, there exists an *interpolant* A' for (A,B) such that:

$$A \Rightarrow A'$$
$$A' \wedge B = \text{false}$$
$$A' \in \mathcal{L}(A) \cap \mathcal{L}(B)$$

- Example:
  - $A = p \wedge q$, $B = \neg q \wedge r$, $A' = q$

Interpolation Lemma

(Craig,57)

- Notation: $\mathcal{L}(\phi)$ is the set of FO formulas over the symbols of ϕ
- If $A \wedge B$ = false, there exists an *interpolant* A' for (A,B) such that:

$$A \Rightarrow A'$$
$$A' \wedge B = \text{false}$$
$$A' \in \mathcal{L}(A) \cap \mathcal{L}(B)$$

- Example:
  - $A = p \wedge q$, $B = \neg q \wedge r$, $A' = q$
- Interpolants from proofs
  - in certain quantifier-free theories, we can obtain an interpolant for a pair A,B from a refutation in linear time. [McMillan 05]
  - in particular, we can have linear arithmetic,uninterpreted functions, and restricted use of arrays

Interpolants for sequences

Interpolants for sequences

- Let $A_1 \ldots A_n$ be a sequence of formulas

Interpolants for sequences

- Let $A_1 \ldots A_n$ be a sequence of formulas
- A sequence $A'_0 \ldots A'_n$ is an interpolant for $A_1 \ldots A_n$ when

Interpolants for sequences

- Let $A_1 \ldots A_n$ be a sequence of formulas
- A sequence $A'_0 \ldots A'_n$ is an interpolant for $A_1 \ldots A_n$ when
  - A'_0 = True

Interpolants for sequences

- Let $A_1 \ldots A_n$ be a sequence of formulas
- A sequence $A'_0 \ldots A'_n$ is an interpolant for $A_1 \ldots A_n$ when
  - A'_0 = True
  - $A'_{i-1} \wedge A_i \Rightarrow A'_i$, for i = 1..n

Interpolants for sequences

- Let $A_1 \ldots A_n$ be a sequence of formulas
- A sequence $A'_0 \ldots A'_n$ is an interpolant for $A_1 \ldots A_n$ when
  - A'_0 = True
  - $A'_{i-1} \wedge A_i \Rightarrow A'_i$, for i = 1..n
  - A_n = False

Interpolants for sequences

- Let $A_1 \ldots A_n$ be a sequence of formulas
- A sequence $A'_0 \ldots A'_n$ is an interpolant for $A_1 \ldots A_n$ when
  - A'_0 = True
  - $A'_{i-1} \wedge A_i \Rightarrow A'_i$, for i = 1..n
  - A_n = False
  - and finally, $A'_i \in \mathcal{L}(A_1 \ldots A_i) \cap \mathcal{L}(A_{i+1} \ldots A_n)$

Interpolants for sequences

- Let $A_1 \ldots A_n$ be a sequence of formulas
- A sequence $A'_0 \ldots A'_n$ is an interpolant for $A_1 \ldots A_n$ when
  - A'_0 = True
  - $A'_{i-1} \wedge A_i \Rightarrow A'_i$, for i = 1..n
  - A_n = False
  - and finally, $A'_i \in \mathcal{L}(A_1 \ldots A_i) \cap \mathcal{L}(A_{i+1} \ldots A_n)$

A_1 A_2 A_3 … A_k

Interpolants for sequences

- Let $A_1 \ldots A_n$ be a sequence of formulas
- A sequence $A'_0 \ldots A'_n$ is an interpolant for $A_1 \ldots A_n$ when
  - A'_0 = True
  - $A'_{i-1} \wedge A_i \Rightarrow A'_i$, for i = 1..n
  - A_n = False
  - and finally, $A'_i \in \mathcal{L}(A_1 \ldots A_i) \cap \mathcal{L}(A_{i+1} \ldots A_n)$

$$A_1 \mid A_2 \mid A_3 \mid \cdots \mid A_k$$

$$\text{True} \Rightarrow A'_1 \Rightarrow A'_2 \Rightarrow A'_3 \cdots A'_{k-1} \Rightarrow \text{False}$$

Interpolants for sequences

- Let $A_1 \ldots A_n$ be a sequence of formulas
- A sequence $A'_0 \ldots A'_n$ is an interpolant for $A_1 \ldots A_n$ when
  - A'_0 = True
  - $A'_{i-1} \wedge A_i \Rightarrow A'_i$, for i = 1..n
  - A_n = False
  - and finally, $A'_i \in \mathcal{L}(A_1 \ldots A_i) \cap \mathcal{L}(A_{i+1} \ldots A_n)$

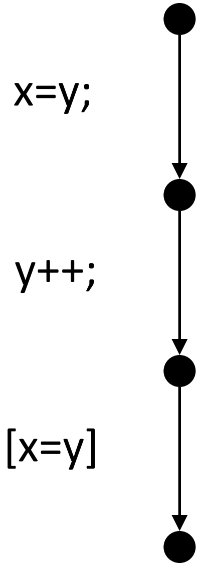
A_1 | A_2 | A_3 | $\cdots$ | A_k

True $\Rightarrow A'_1 \Rightarrow A'_2 \Rightarrow A'_3 \cdots A'_{k-1} \Rightarrow$ False

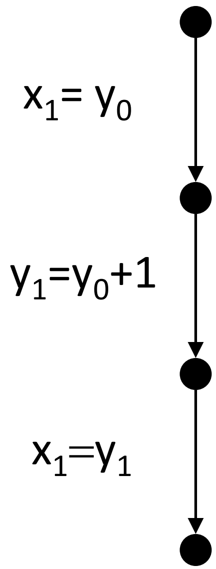
In other words, the interpolant is a structured refutation of $A_1 \ldots A_n$

Interpolants as Floyd-Hoare proofs

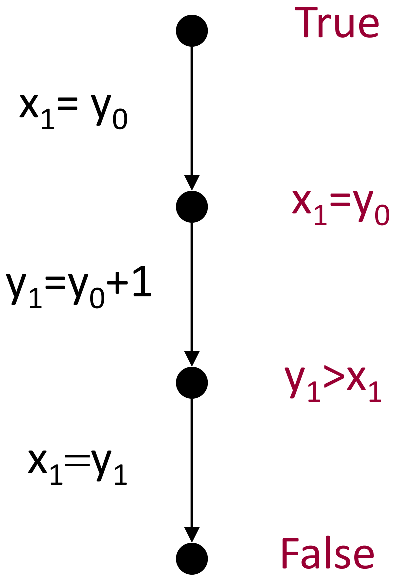
Interpolants as Floyd-Hoare proofs



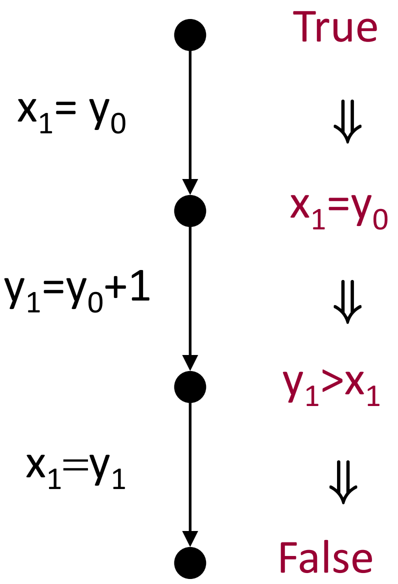
Interpolants as Floyd-Hoare proofs



Interpolants as Floyd-Hoare proofs

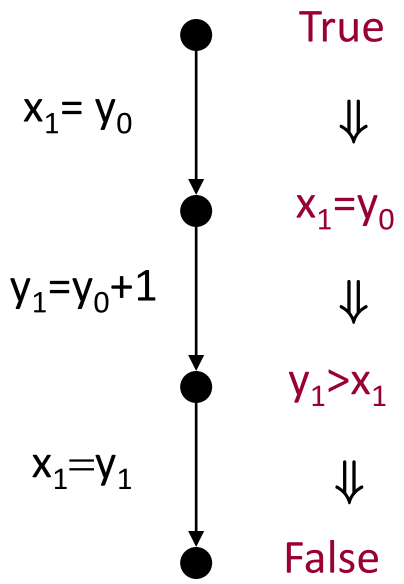


Interpolants as Floyd-Hoare proofs



1. Each formula implies the next

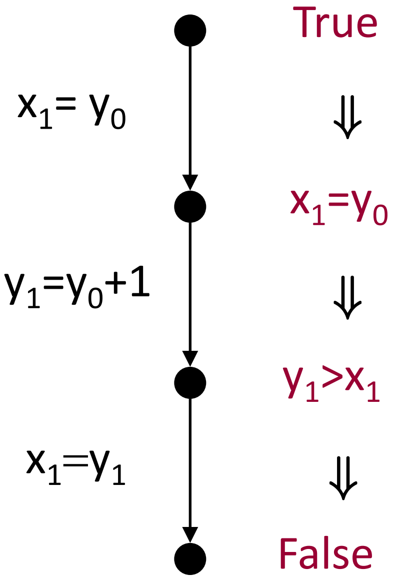
Interpolants as Floyd-Hoare proofs



1. Each formula implies the next

2. Each is over common symbols of prefix and suffix

Interpolants as Floyd-Hoare proofs



1. Each formula implies the next

2. Each is over common symbols of prefix and suffix

3. Begins with true, ends with false

Interpolants as Floyd-Hoare proofs

	True
$x_1 = y_0$	$\Downarrow$
	$x_1 = y_0$
$y_1 = y_0 + 1$	$\Downarrow$
	$y_1 > x_1$
$x_1 = y_1$	$\Downarrow$
	False

1. Each formula implies the next
2. Each is over common symbols of prefix and suffix
3. Begins with true, ends with false

Path refinement procedure

SSA sequence → Prover —proof→ Interpolation —structured proof→ Path Refinement

Interpolation

Lazy abstraction -- an example

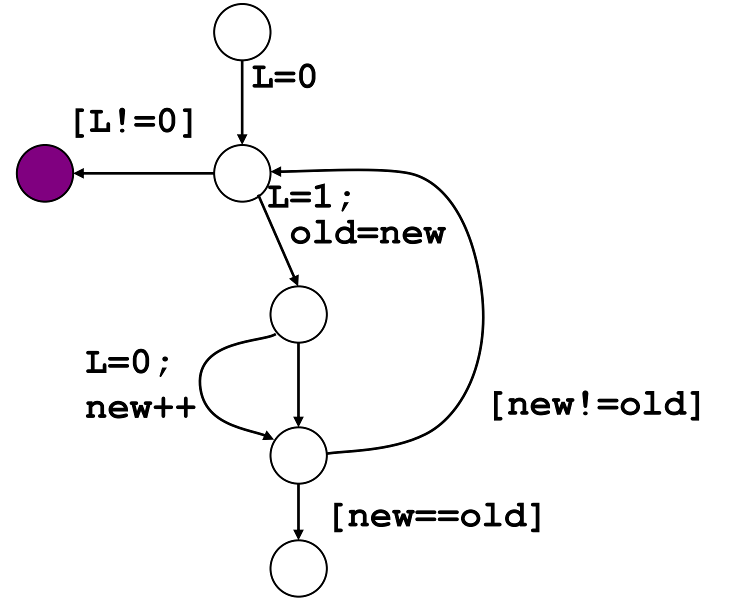
```
do{
  lock();
  old = new;
  if(*){
    unlock;
    new++;
  }
} while (new != old);
```

program fragment

Lazy abstraction -- an example

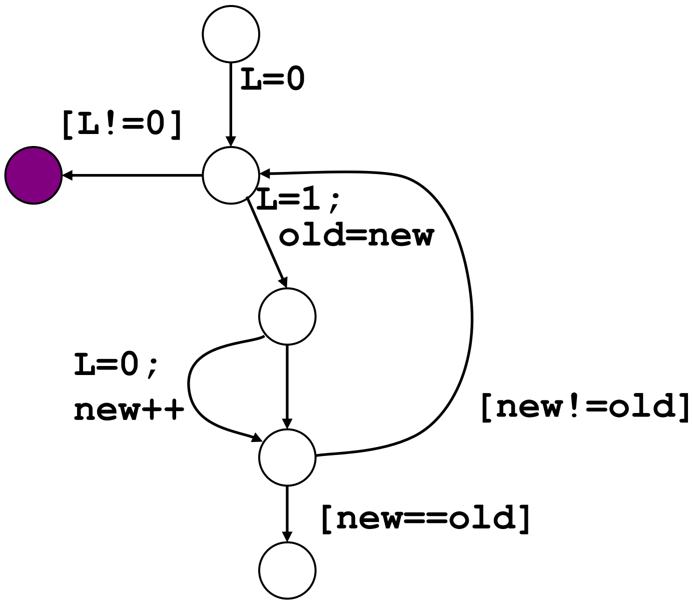
```
do{
  lock();
  old = new;
  if(*){
    unlock;
    new++;
  }
} while (new != old);
```

program fragment



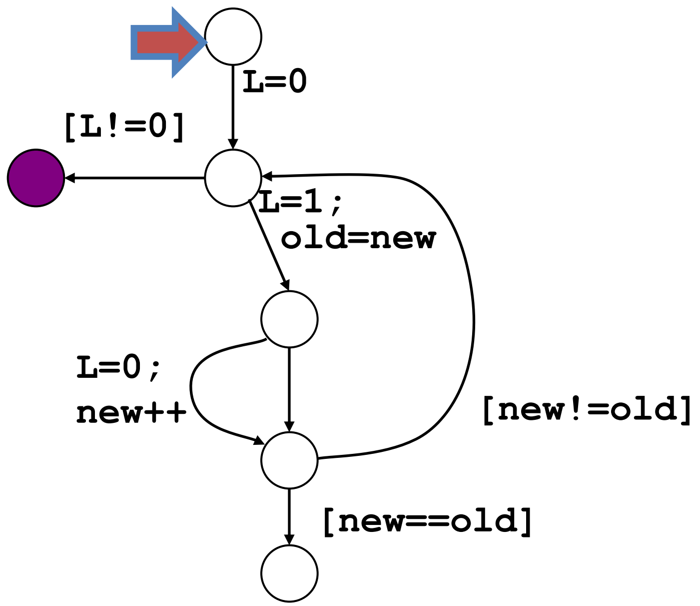
control-flow graph

Unwinding the CFG

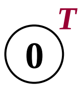


control-flow graph

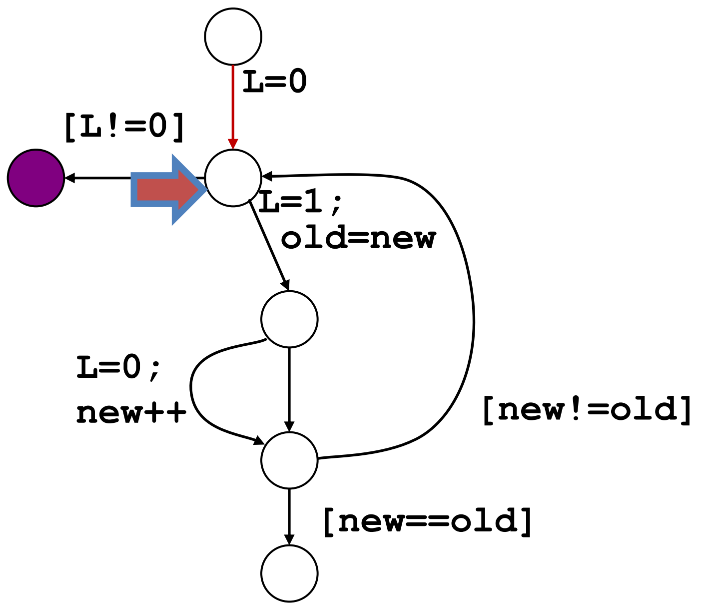
Unwinding the CFG



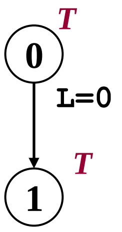
control-flow graph



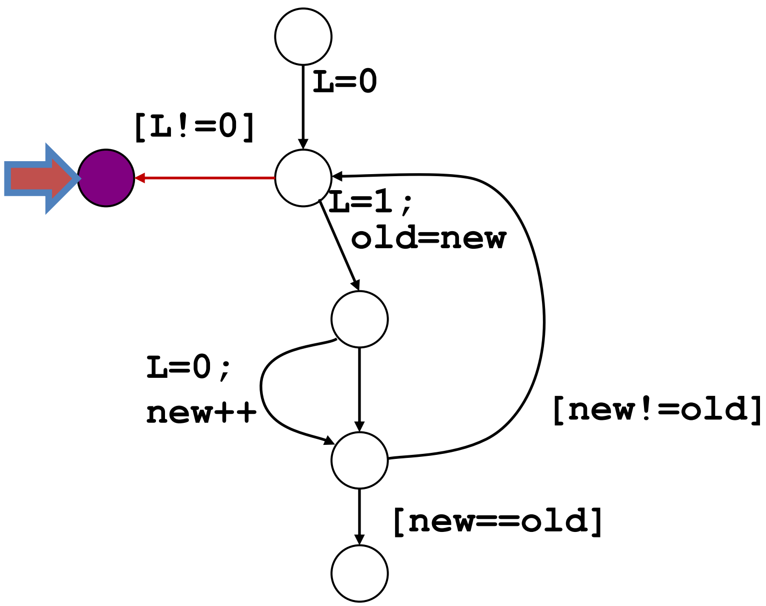
Unwinding the CFG



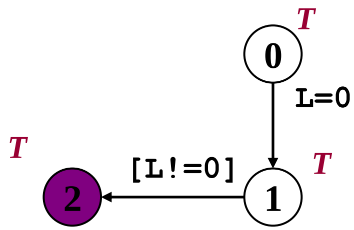
control-flow graph



Unwinding the CFG

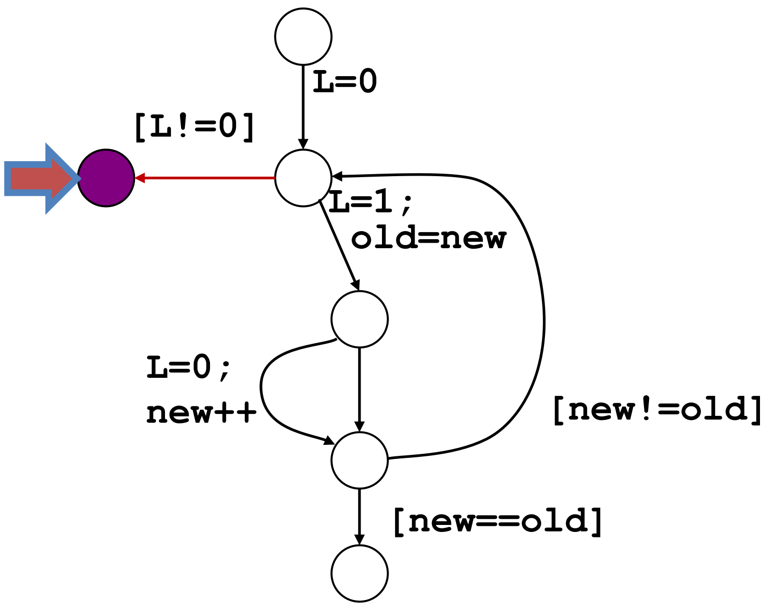


control-flow graph

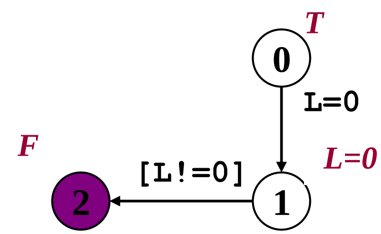


Interpolant for $(L_0 = 0) \wedge (L_0 != 0)$ =?

Unwinding the CFG



control-flow graph

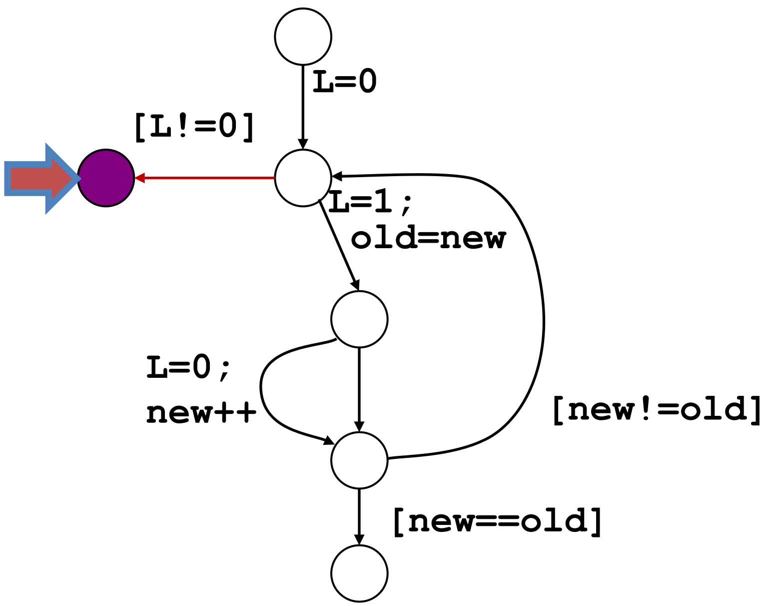


Label error state with false, by refining labels on path

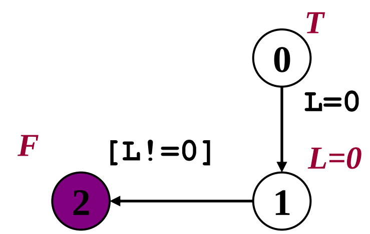
Interpolant for $(L_0 = 0) \wedge (L_0 != 0)$ =?

T L_0=0 *F*

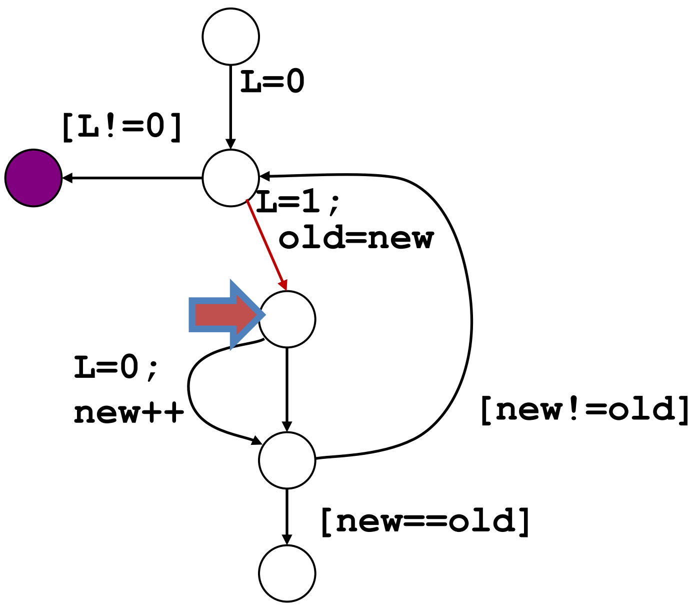
Unwinding the CFG



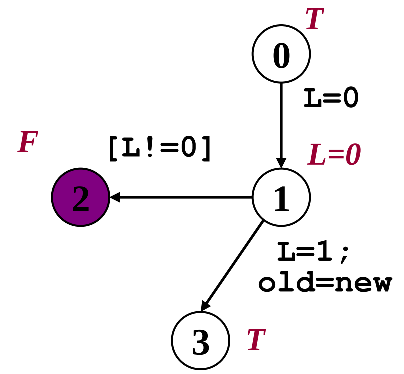
control-flow graph



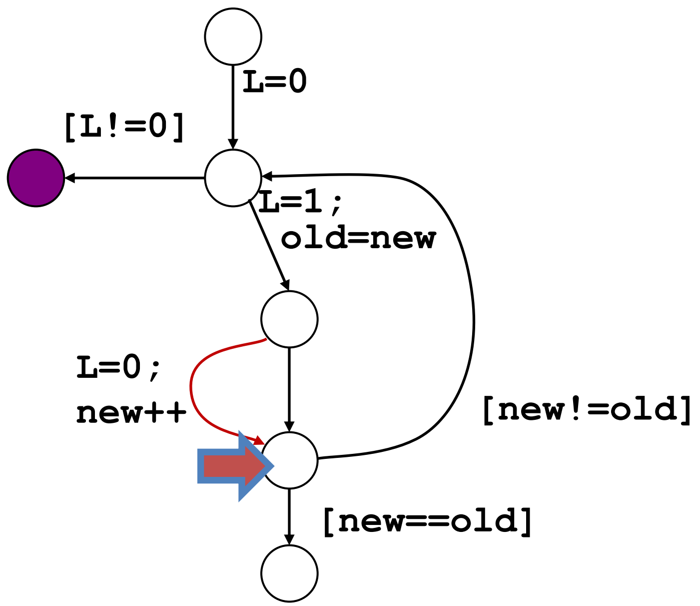
Unwinding the CFG



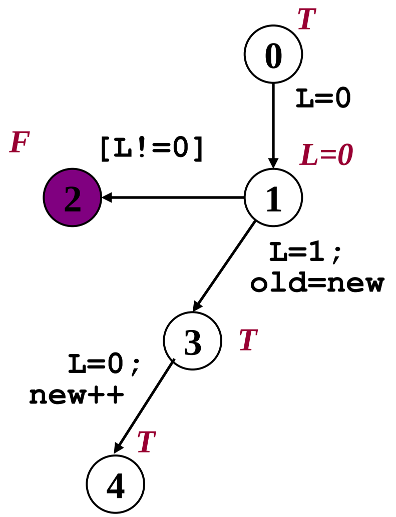
control-flow graph



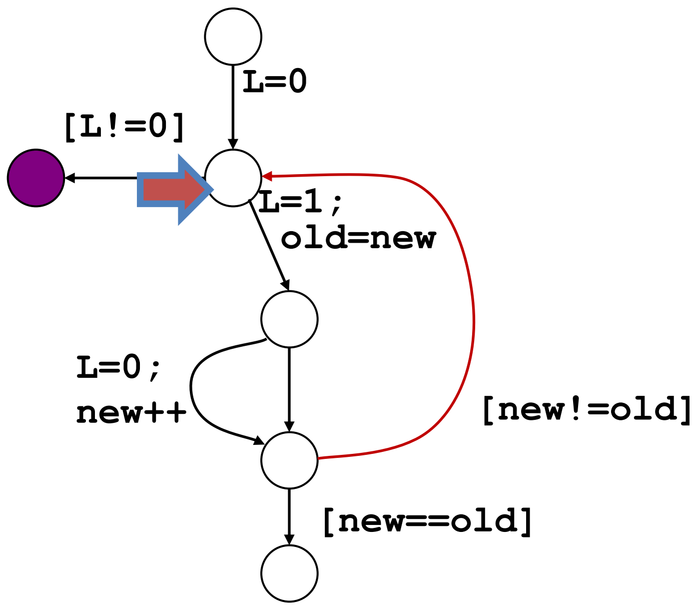
Unwinding the CFG



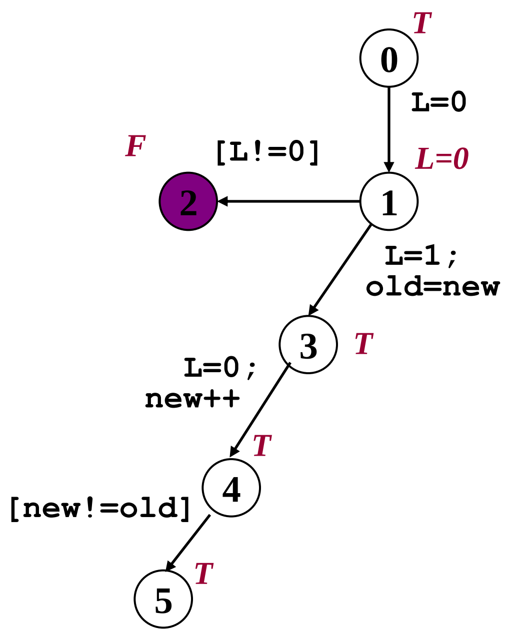
control-flow graph



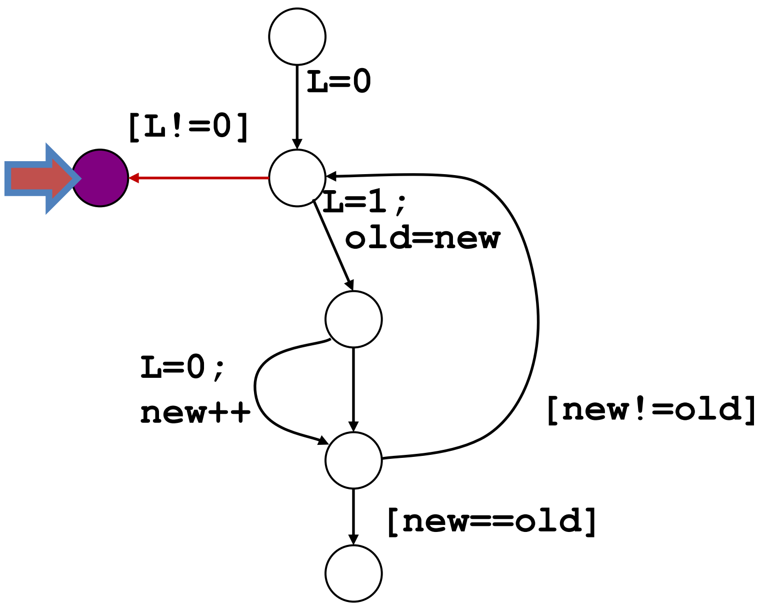
Unwinding the CFG



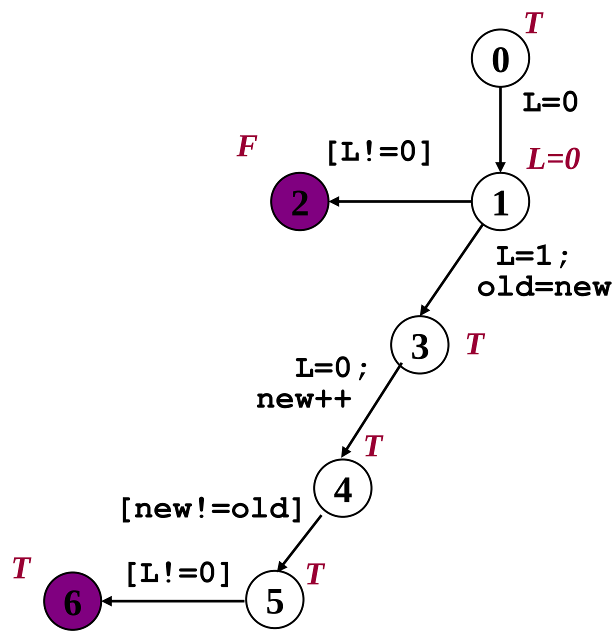
control-flow graph



Unwinding the CFG

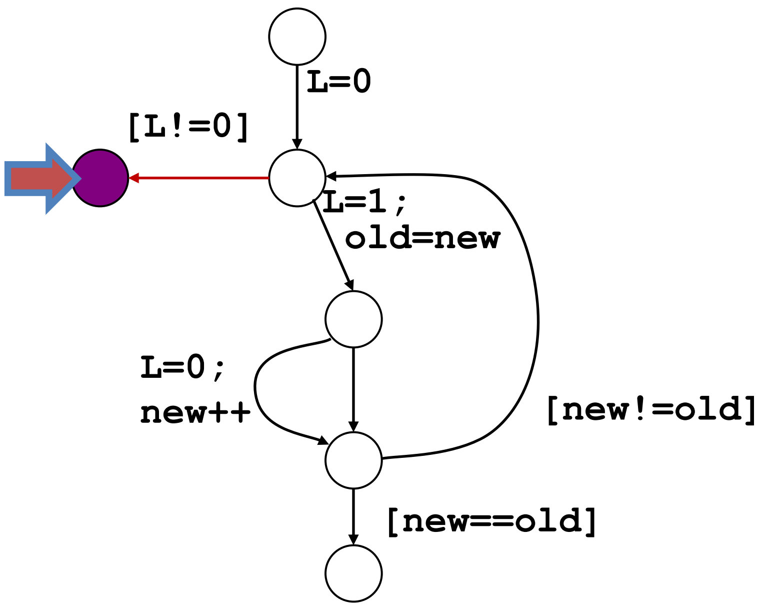


control-flow graph

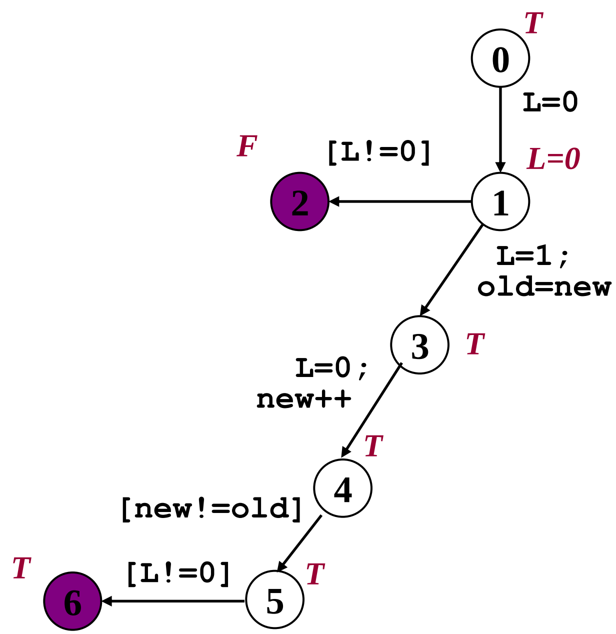


Interpolant for $(L_0=0) \wedge (L_1=1 \wedge old_0=new_0) \wedge (L_2=0 \wedge new_1=new_0+1) \wedge (new_1 != old_0) \wedge (L_2 != 0)$ =?

Unwinding the CFG



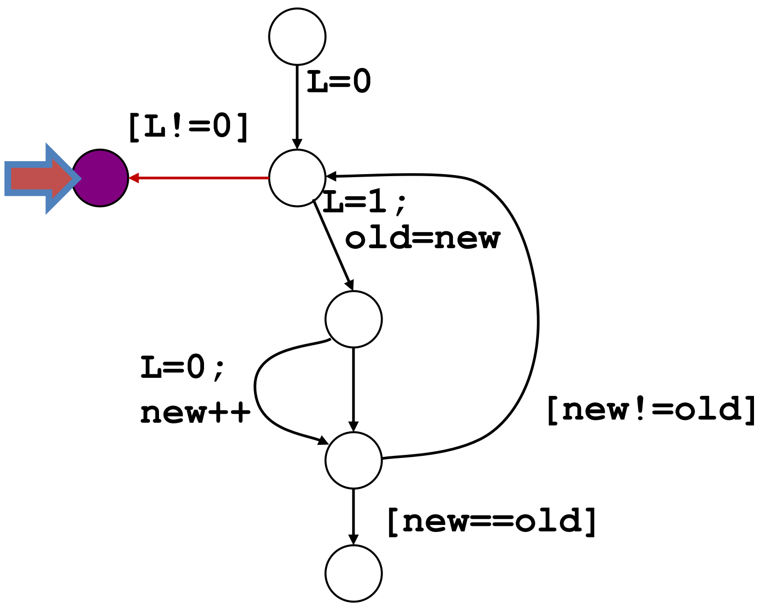
control-flow graph



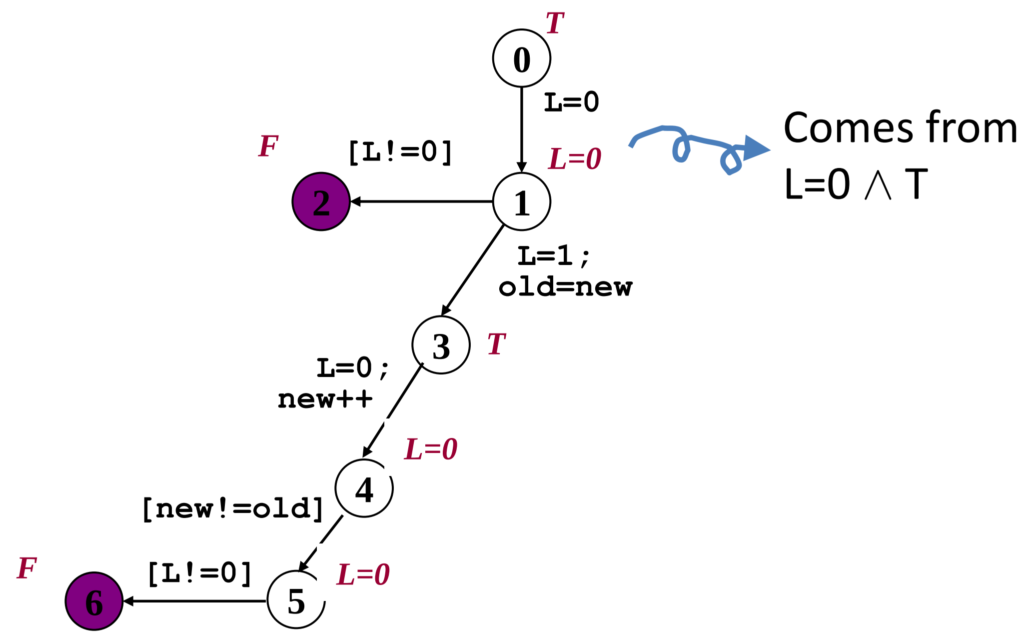
Interpolant for $(L_0 = 0) \wedge (L_1 = 1 \wedge old_0 = new_0) \wedge (L_2 = 0 \wedge new_1 = new_0 + 1) \wedge (new_1 != old_0) \wedge (L_2 != 0)$ =?

T   *T*   *T*   *L_2=0*   *L_2=0*   *F*

Unwinding the CFG



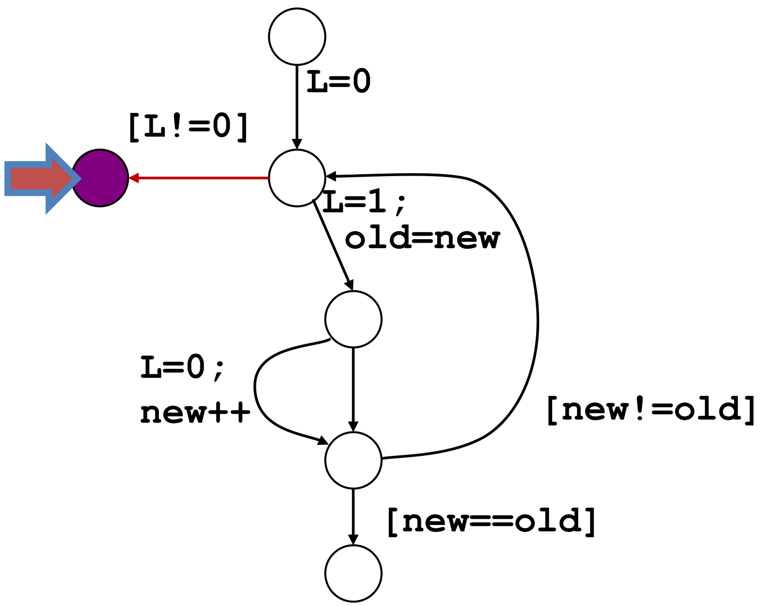
control-flow graph



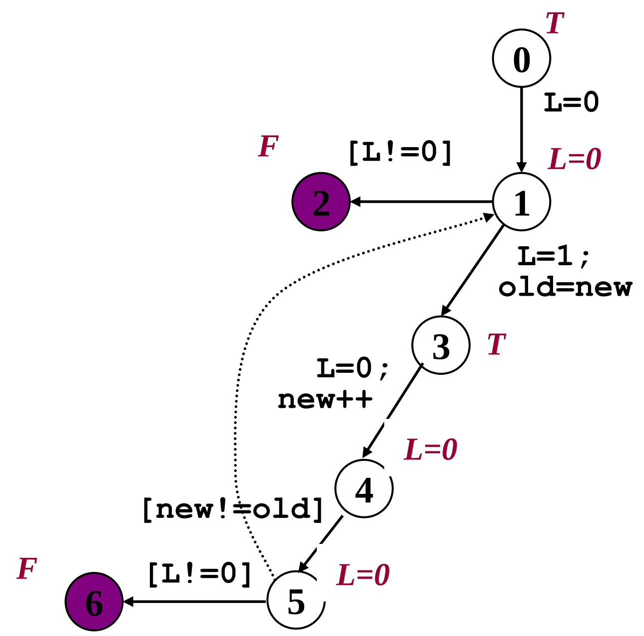
Interpolant for $(L_0 = 0) \wedge (L_1=1 \wedge old_0=new_0) \wedge (L_2=0 \wedge new_1=new_0+1) \wedge (new_1 != old_0) \wedge (L_2 != 0)$ =?

T *T* *T* $L_2 = 0$ $L_2 = 0$ *F*

Unwinding the CFG

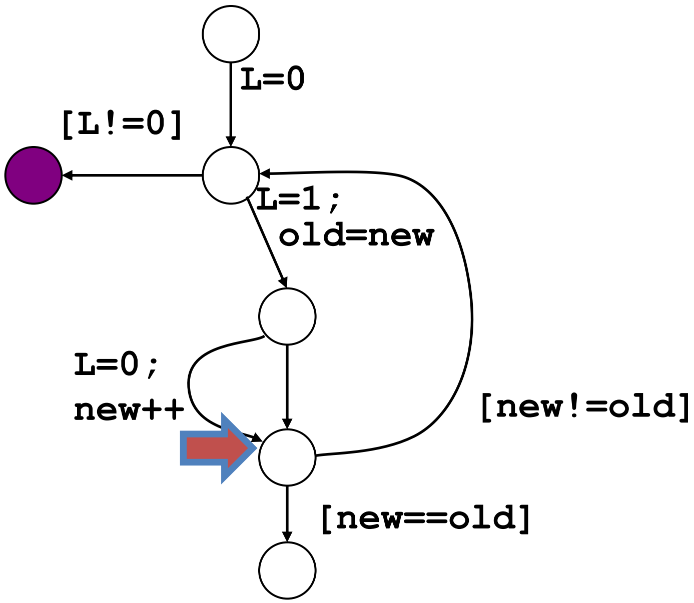


control-flow graph

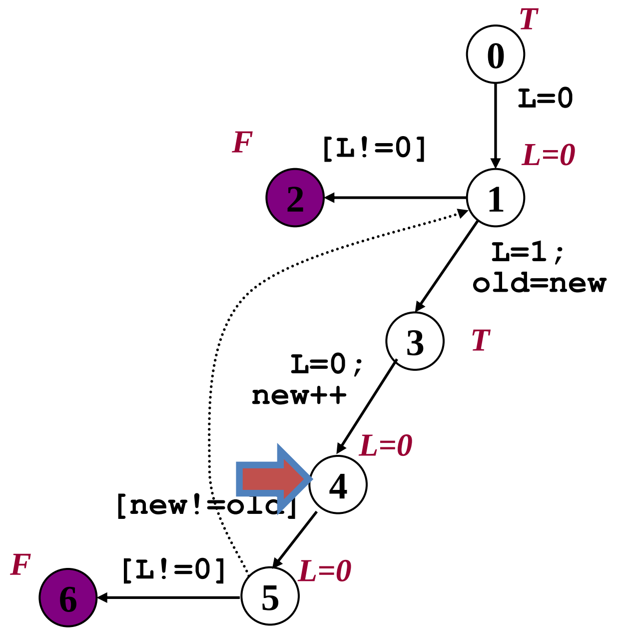


Covering: state 5 is subsumed by state 1.

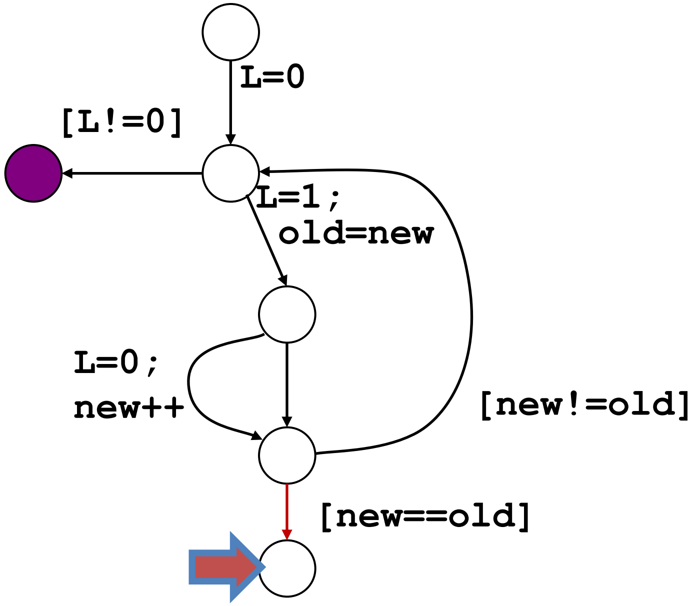
Unwinding the CFG



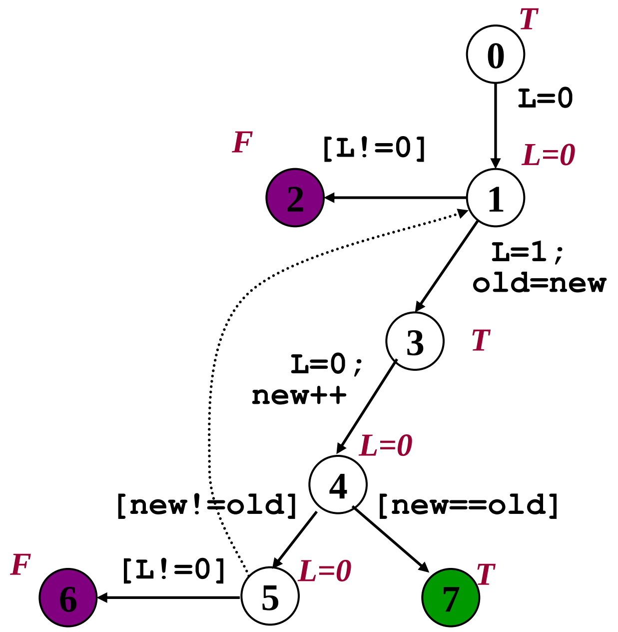
control-flow graph



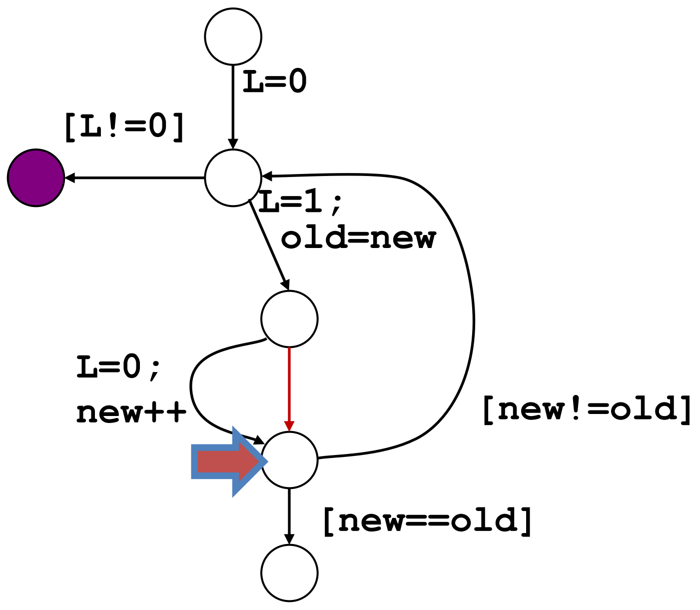
Unwinding the CFG



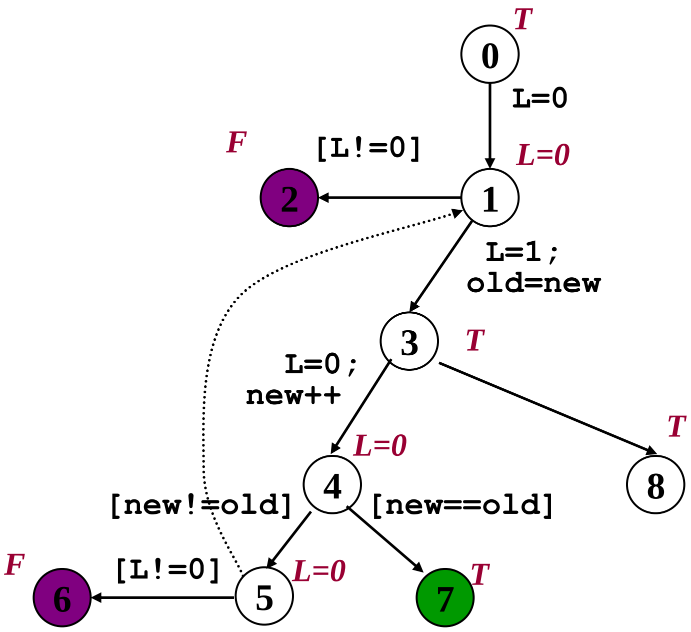
control-flow graph



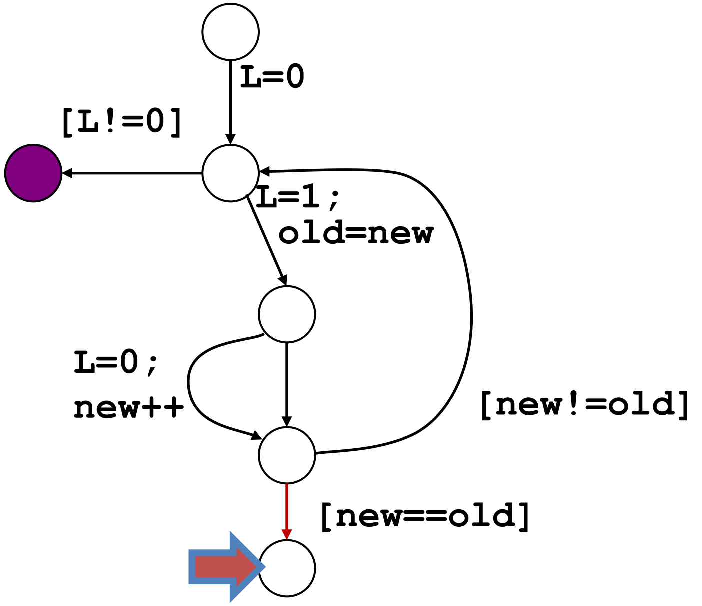
Unwinding the CFG



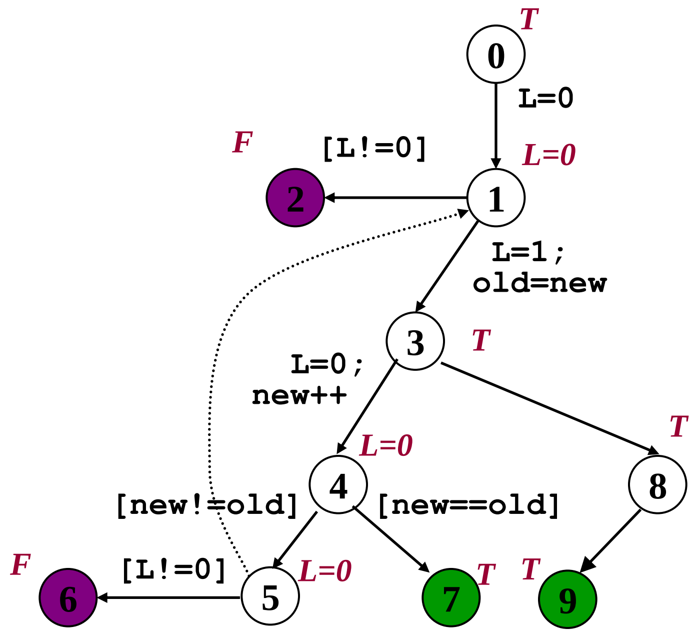
control-flow graph



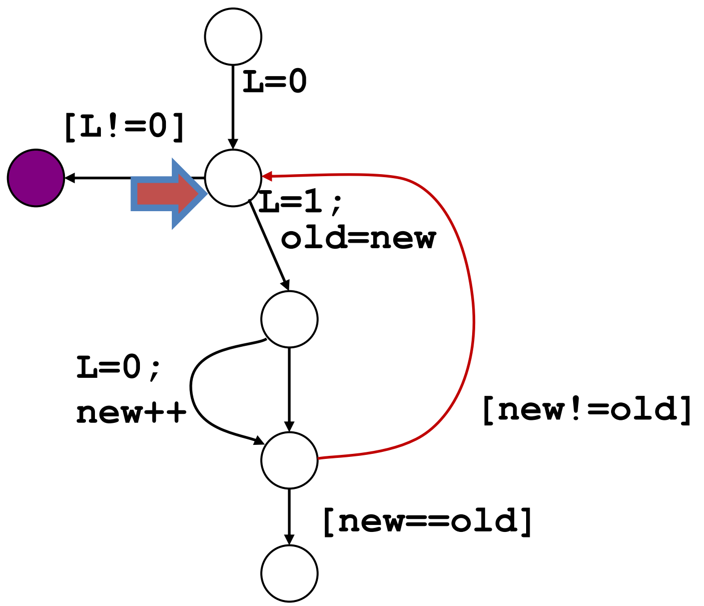
Unwinding the CFG



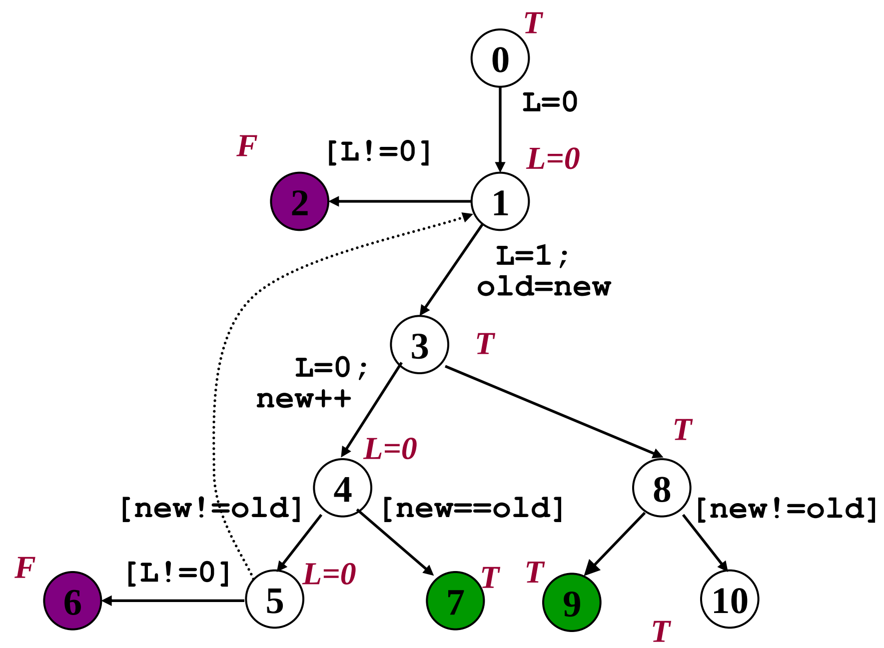
control-flow graph



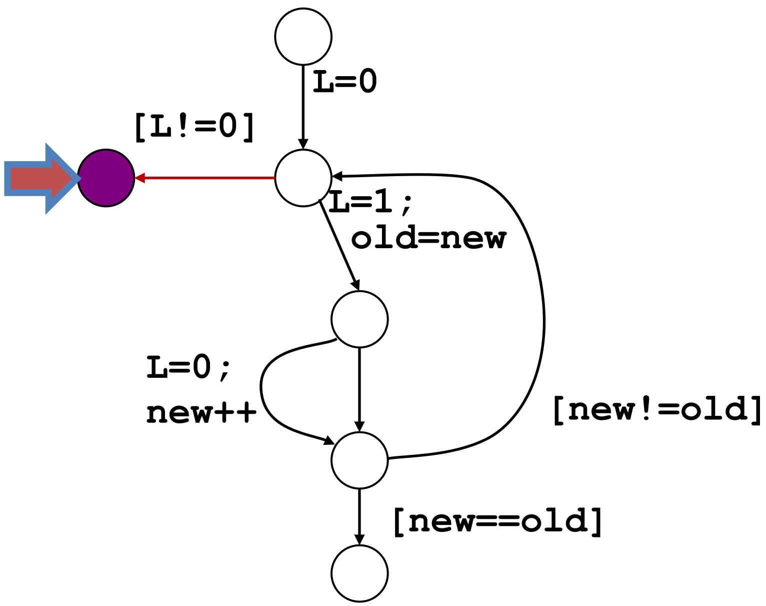
Unwinding the CFG



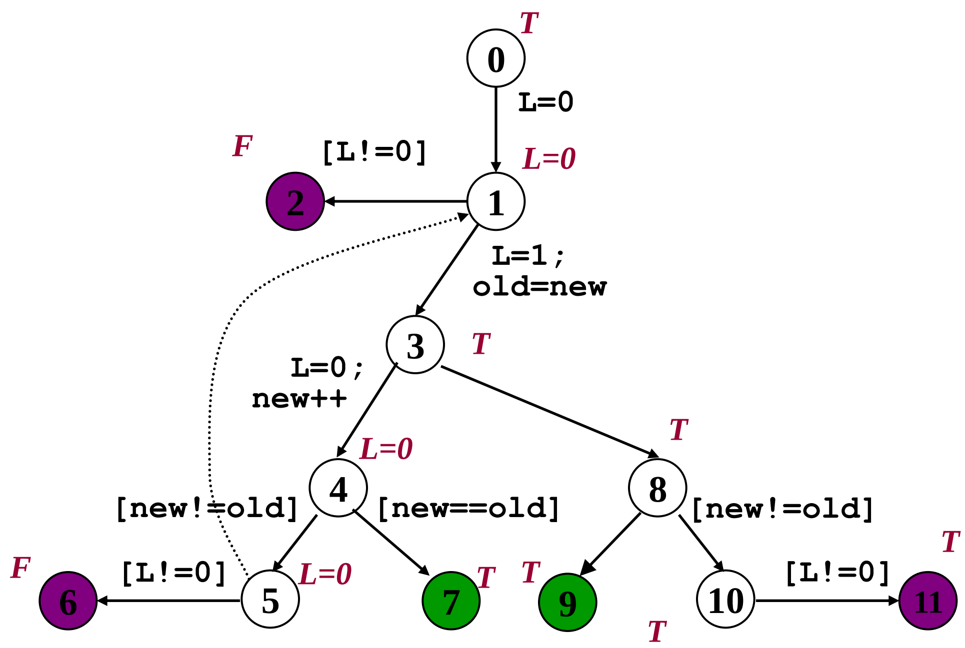
control-flow graph



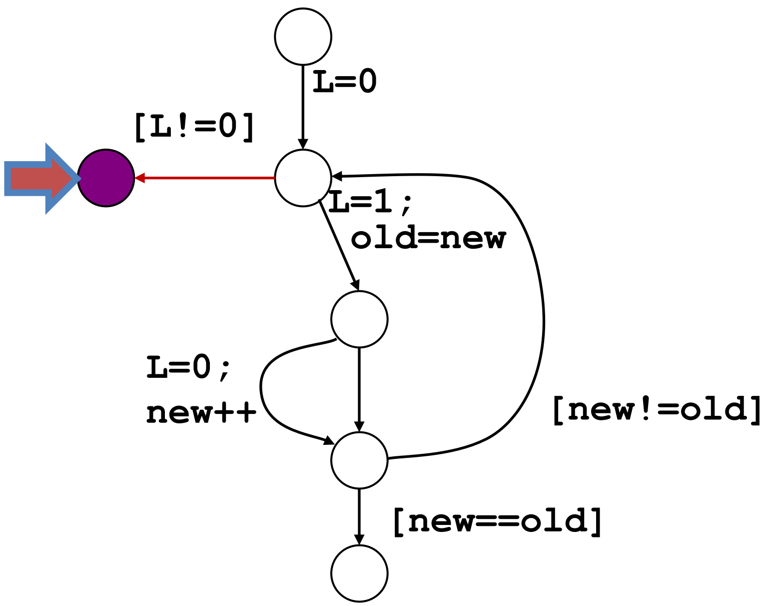
Unwinding the CFG



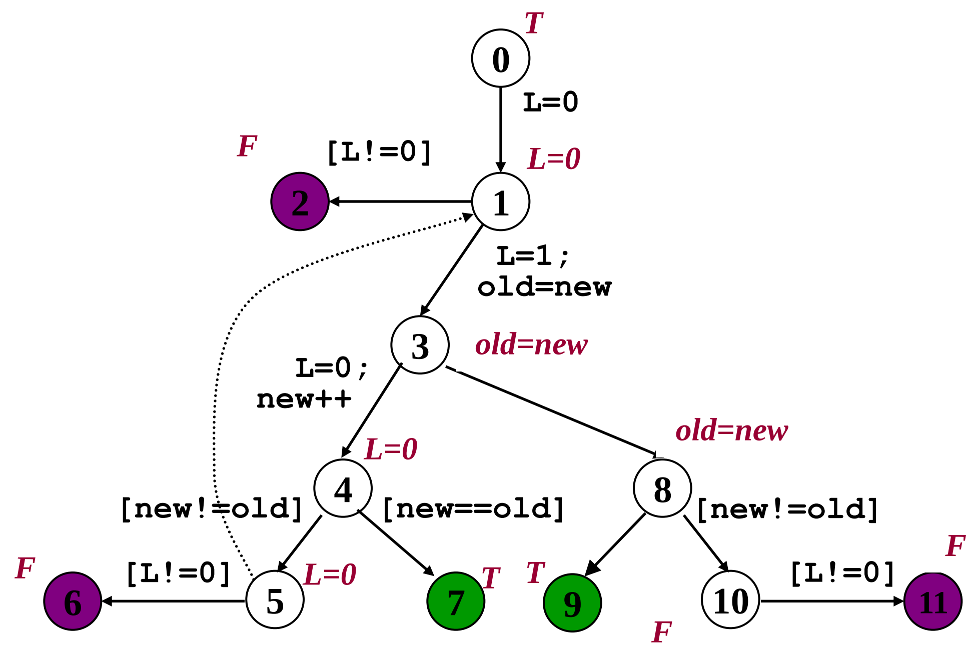
control-flow graph



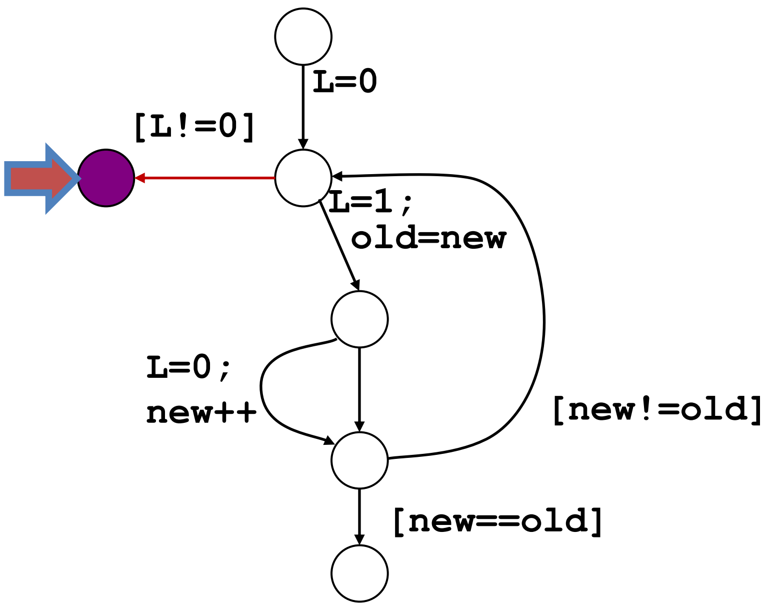
Unwinding the CFG



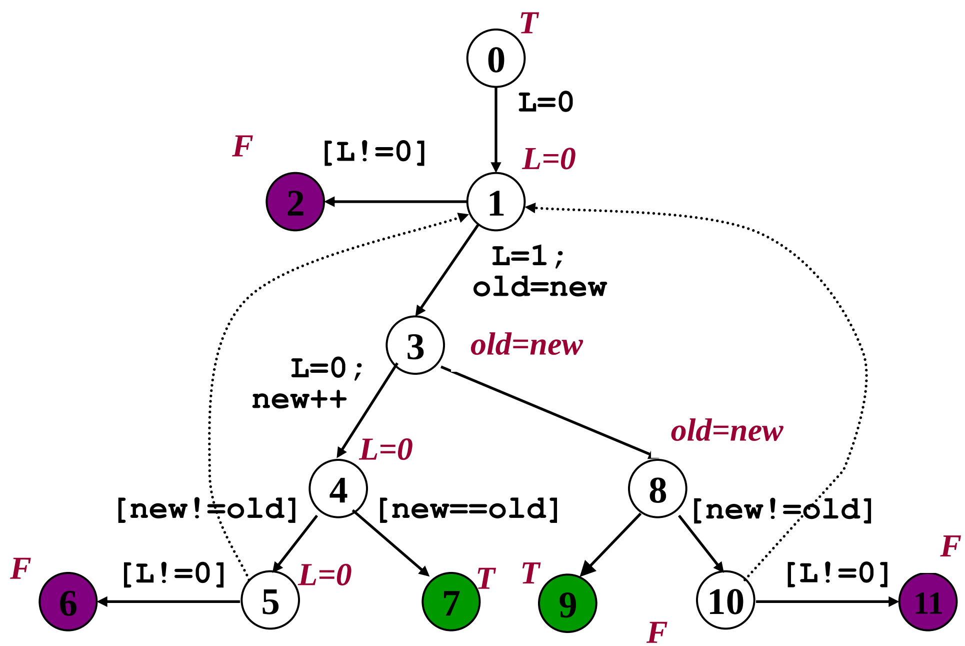
control-flow graph



Unwinding the CFG



control-flow graph



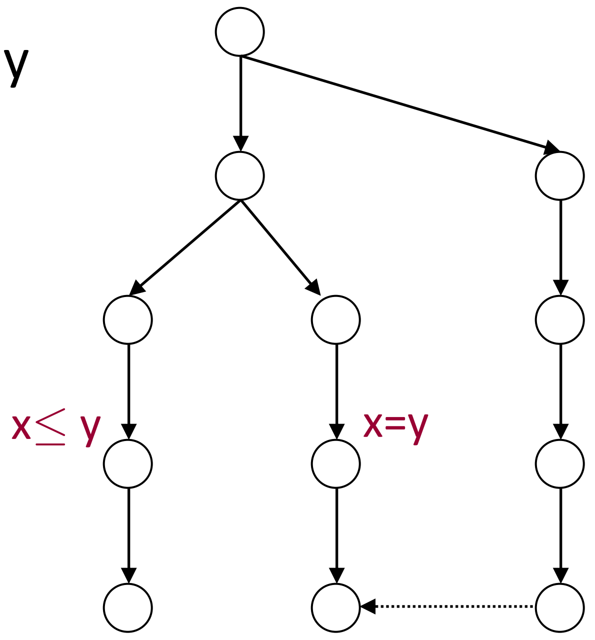
Another cover. Unwinding is now complete.

Covering step

- If $\psi(x) \Rightarrow \psi(y)$...
  - add covering arc $x \triangleright y$
  - remove all $z \triangleright w$ for w descendant of y

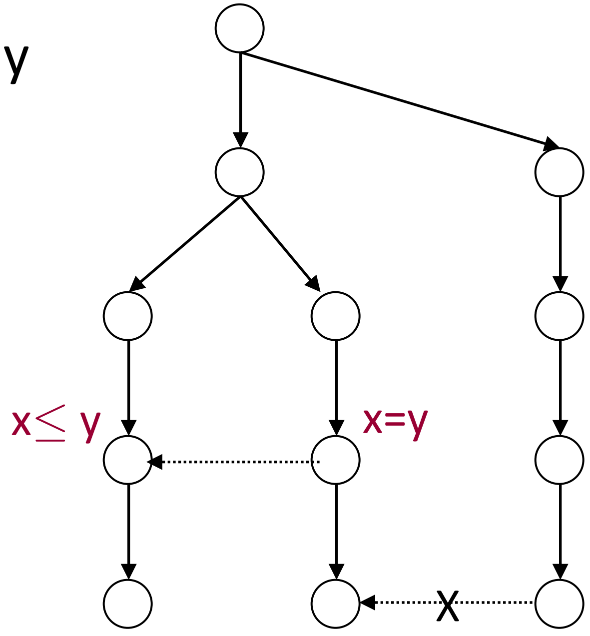
Covering step

- If $\psi(x) \Rightarrow \psi(y)$...
  - add covering arc $x \triangleright y$
  - remove all $z \triangleright w$ for w descendant of y



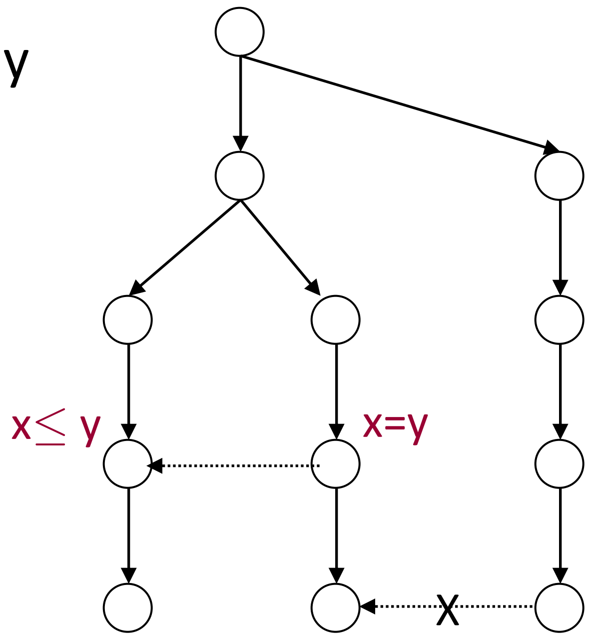
Covering step

- If $\psi(y) \Rightarrow \psi(x)$...
  - add covering arc $x \triangleright y$
  - remove all $w \triangleright z$ for w descendant of y



Covering step

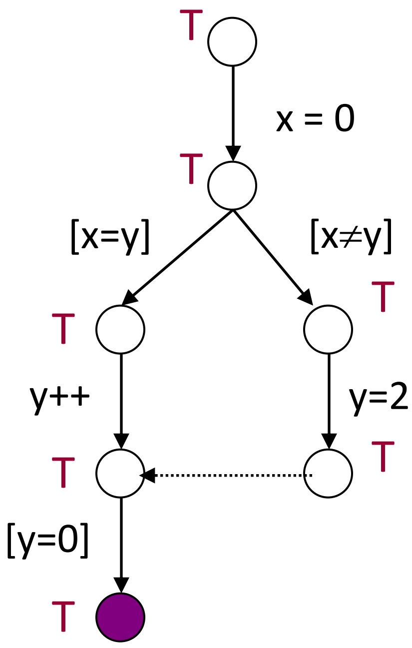
- If $\psi(y) \Rightarrow \psi(x)$...
  - add covering arc $x \triangleright y$
  - remove all $w \triangleright z$ for w descendant of y



We restrict covers to be descending in a suitable total order on vertices.
This prevents covering from diverging.

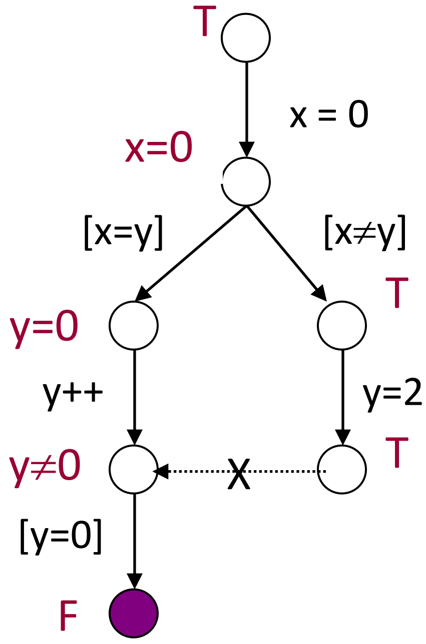
Refinement step

- Label an error vertex False by refining the path to that vertex with an interpolant for that path.
- By refining with interpolants, we avoid predicate image computation.



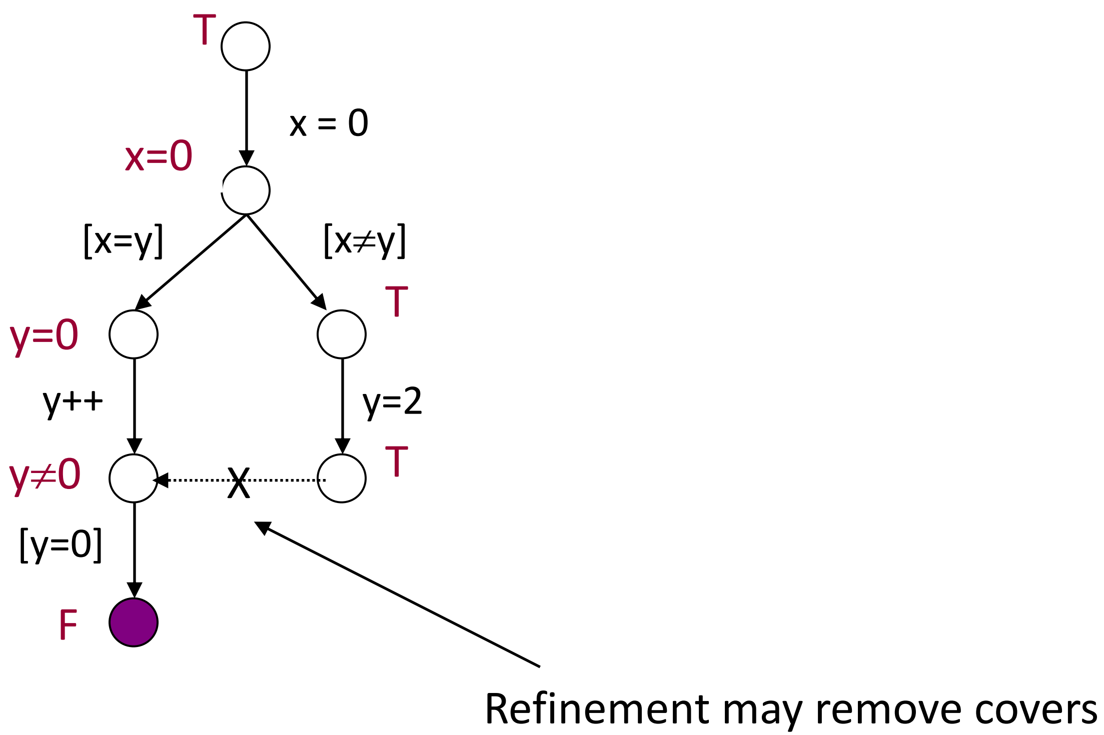
Refinement step

- Label an error vertex False by refining the path to that vertex with an interpolant for that path.
- By refining with interpolants, we avoid predicate image computation.



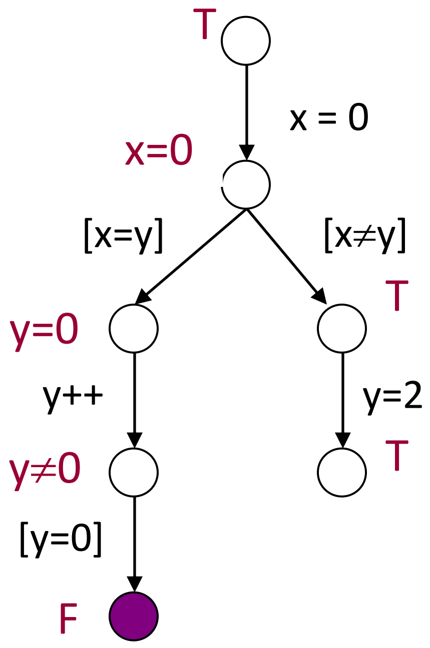
Refinement step

- Label an error vertex False by refining the path to that vertex with an interpolant for that path.
- By refining with interpolants, we avoid predicate image computation.



Forced cover

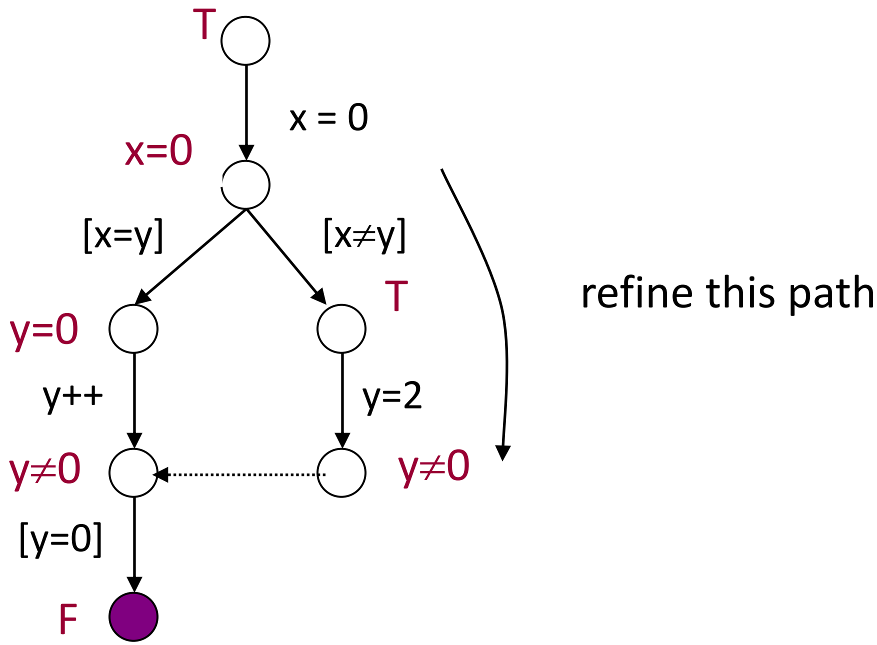
- Try to refine a sub-path to force a cover
  - show that path from nearest common ancestor of x,y proves $\psi(x)$ at y



Forced cover allow us to efficiently handle nested control structure

Forced cover

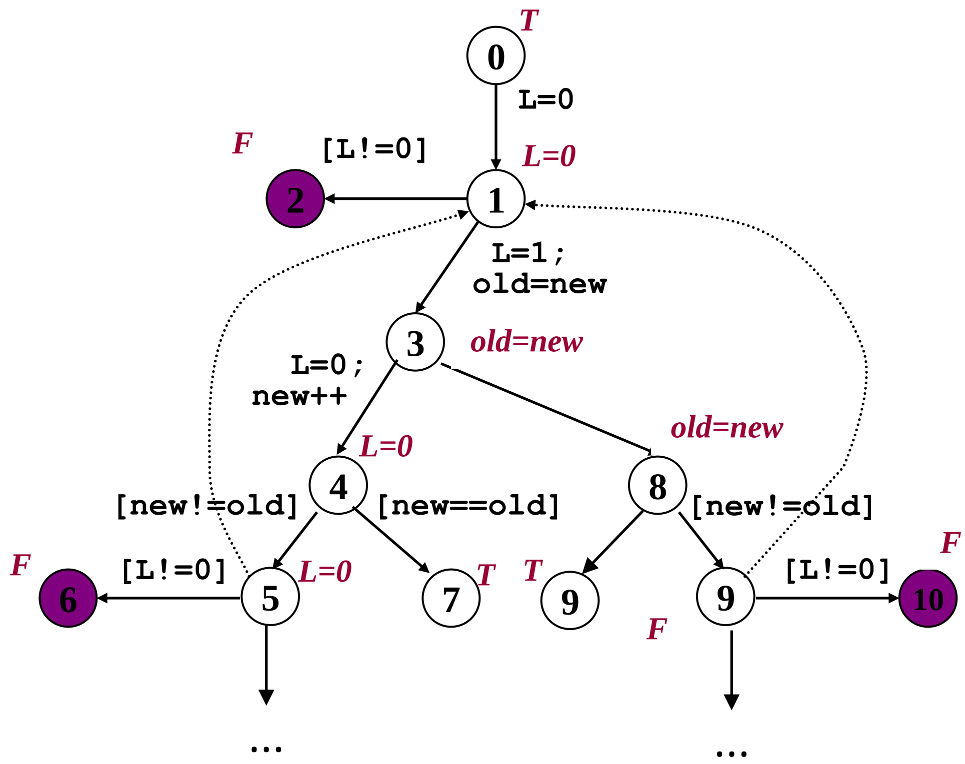
- Try to refine a sub-path to force a cover
  - show that path from nearest common ancestor of x,y proves $\psi(x)$ at y



Forced cover allow us to efficiently handle nested control structure

Safe and complete

- An unwinding is
  - *safe* if every error vertex is labeled False
  - *complete* if every nonterminal leaf is covered



Theorem: A CFG with a safe complete unwinding is safe.

Unwinding steps

- Three basic operations:
  - Expand a nonterminal leaf
  - Cover: add a covering arc
  - Refine: strengthen labels along a path so error vertex labeled False

Overall algorithm

1. Do as much covering as possible
2. If a leaf can't be covered, try forced covering
3. If the leaf still can't be covered, expand it
4. Label all error states False by refining with an interpolant
5. Continue until unwinding is safe and complete

Interpolant Seqeunce in Princess

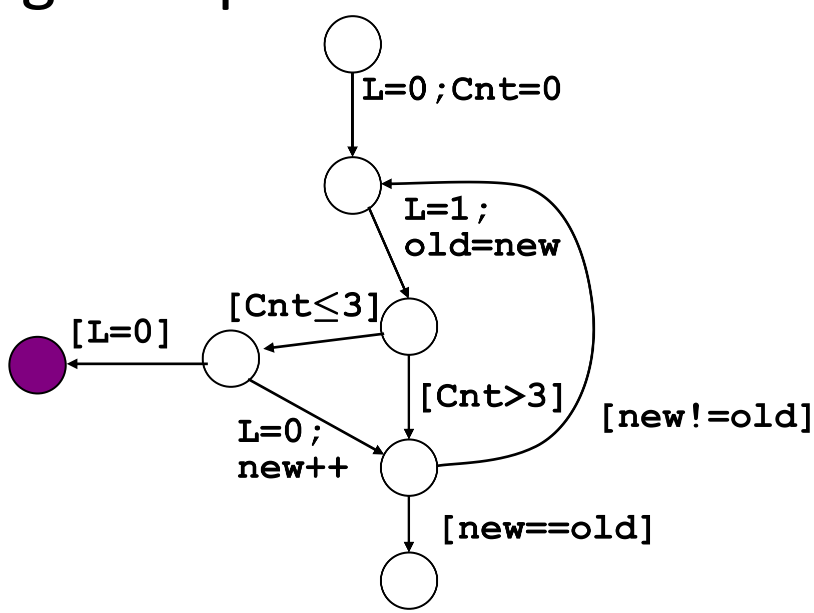
```
\functions {
  int L0, L1, old0, new0, L2, new1;
}
\problem {
 \part[p1]        (L0 =0) &
 \part[p2]       (L1 =1 & old0=new0) &
 \part[p3]       (L2=0 & new1 =new0+1) &
 \part[p4]       (new1 != old0) &
 \part[p5]       (L2!=0)
                   ->
 false
}

\interpolant {p1; p2, p3, p4, p5}
\interpolant {p1, p2; p3, p4, p5}
\interpolant {p1, p2, p3; p4, p5}
\interpolant {p1, p2, p3, p4; p5}
```

Interpolant for

$(L_0 = 0) \wedge (L_1 = 1 \wedge old_0 = new_0)$
$\wedge (L_2 = 0 \wedge new_1 = new_0 + 1) \wedge (new_1 != old_0)$
$\wedge (L_2 != 0)$

Homework

- Run the two versions of verification algorithms on the following control flow graph, using Princess for computing interpolants



control-flow graph