Lazy Abstraction with Interpolants

Ken McMillan (CAV'06)

Based on presentation by Yakir Vizel

Automatic verification, June 26, 2017

Lecture 12

- Previous work develops lazy abstraction for hardware
  - Models describe hardware
  - Visible-variables abstraction
- This work presents lazy abstraction for software
  - Models describe software
  - Kind-of predicate abstraction

Modeling Programs

- Programs are represented as a Control Flow Graph (CFG)

- Sets of states are represented by formulas of the First Order Logic (FOL) over program variables, denoted L(S)

- A **transition formula** is a formula in L(S ∪ S')
  - Example: $x' = x + 1$

- $\varphi^{(n)}$ is φ at time n
  - variables in φ are primed n times

Modeling Programs

- A **program** is a tuple $(\Lambda, \Delta, l_i, l_f)$ where
  - Λ is a finite set of **program locations**
  - Δ is a set of **actions**
  - l_i is the initial location
  - l_f is the error location (l_i, l_f are both in Λ)

- An **action** is a triple (l, T, m) where l, m are respectively the entry and exit locations of the action and T is a **transition formula**

Modeling Programs

- A **path** π of a program is a sequence of transitions of the form
  $(l_0,T_0,l_1), (l_1,T_1,l_2), \ldots,(l_{n-1},T_{n-1},l_n)$
- π is an **error path** if $l_0=l_i$ and $l_n=l_f$

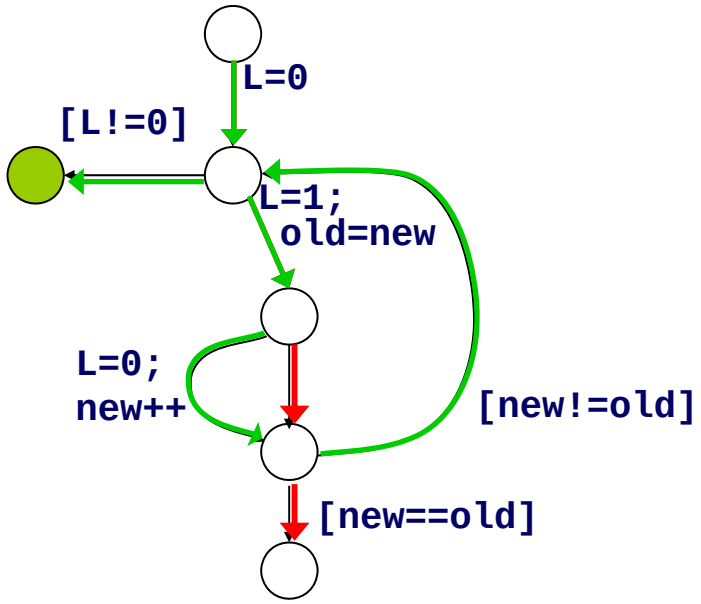
- The **unfolding** $U(\pi)$ of path π is the sequence of formulas: $T_0^{(0)}, T_1^{(1)}, \ldots, T_{n-1}^{(n-1)}$
  $T_i^{(i)}$ is shifted i time units into the future

- A path π is **feasible** when $\wedge U(\pi)$ is consistent

Example

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

program fragment



control-flow graph

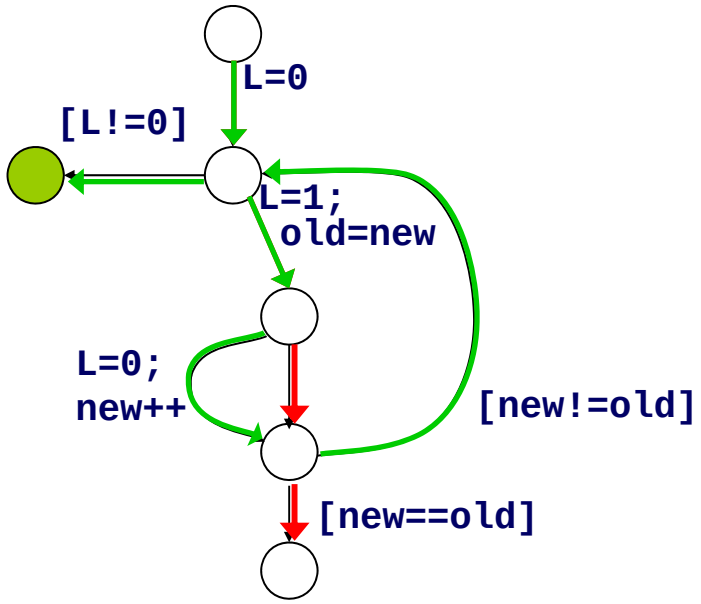
lock: set L; unlock: reset L; Initially L=0

Specification: L is always 0 on entry to lock

Example

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

program fragment



control-flow graph

Λ U(π) = (L=0) Λ (L'=1 Λ old'=new) Λ True Λ (new = old')

Λ U(π) = (L=0) Λ(L'=1 Λ old'=new) Λ (L''=0 Λ new' = new + 1) Λ (new' != old')Λ (L'' != 0)

Program Models

- A program is **safe** if every error path of the program is **in**feasible
- An **inductive invariant** of a program is a map **F: $\Lambda \rightarrow L(S)$** such that:
  - **$F(l_i)$ = TRUE**
  - For every action (l,T,m) in Δ, **$F(l) \wedge T$ implies $F(m)'$**
- A **safety invariant** of a program is an inductive invariant such that **$F(l_f)$ = FALSE**

- **Note:** Existence of a safety invariant for a program implies that the program is safe

Program Unwinding

- An **unwinding** of a program $A = (\Lambda, \Delta, l_i, l_f)$ is a quadruple (V,E,M_v,M_e), where
  - (V,E) is a directed tree rooted at ε,
  - $M_v: V \rightarrow \Lambda$ is the vertex map, and
  - $M_e: E \rightarrow \Delta$ is the edge map
    such that:

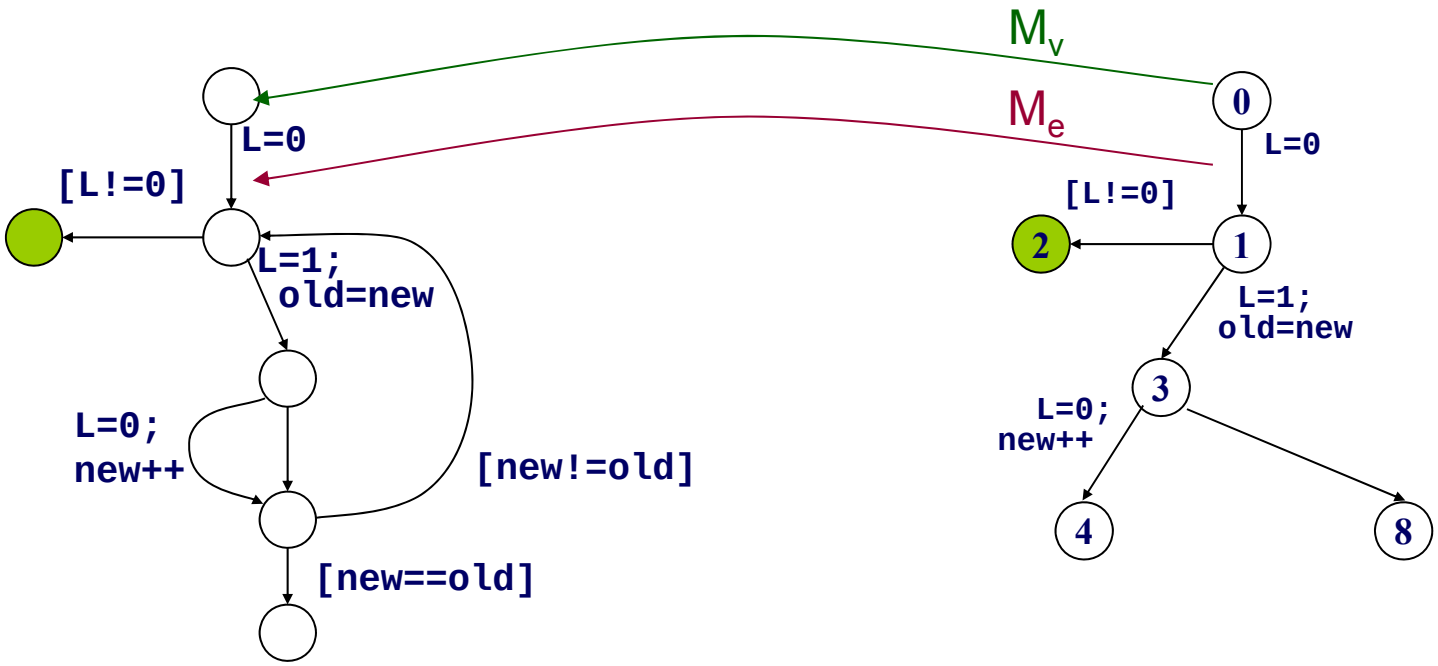
  - $M_v(\varepsilon) = l_i$
  - For every non-leaf vertex v in V, for every action $(M_v(v), T, m)$ in Δ, there exists an edge (v,w) in E such that $M_v(w) = m$ and $M_e(v,w) = T$

Program Unwinding

- For two vertices v and w of a tree, $w < v$ denotes that w is a proper ancestor of v

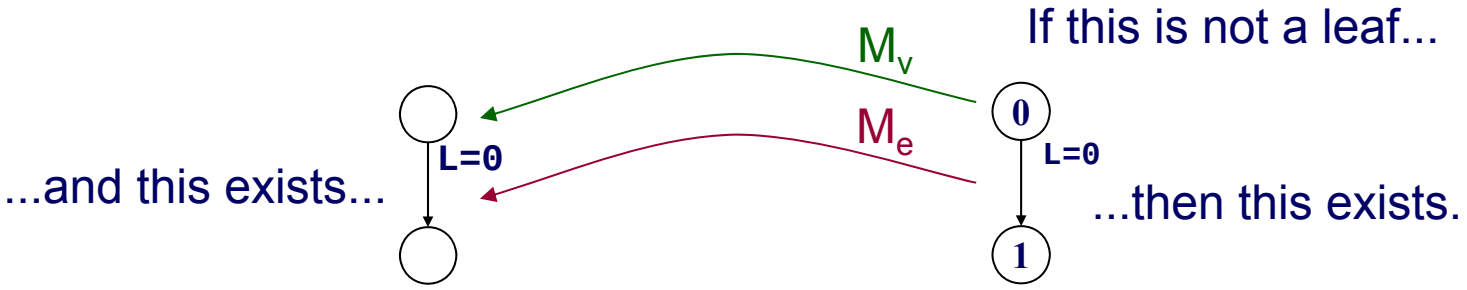
Unwinding the CFG

- An unwinding is a tree with an embedding in the CFG



Expansion

- Every non-leaf vertex of the unwinding must be fully expanded...



...but we allow unexpanded leaves (i.e., we are building a finite prefix of the infinite unwinding)

Program Unwinding

- A **labeled unwinding** of a program $A=(\Lambda,\Delta,l_i,l_f)$ is **(U,ψ,C)** where
  - $U = (V,E,M_v,M_e)$ is an unwinding of A
  - $\Psi:V\rightarrow L(S)$ is called the **vertex labeling**, and
  - C is a relation in V x V, called the **covering relation**

- A vertex v is **covered** iff there exists (w,x) in C such that $w \leq v$.

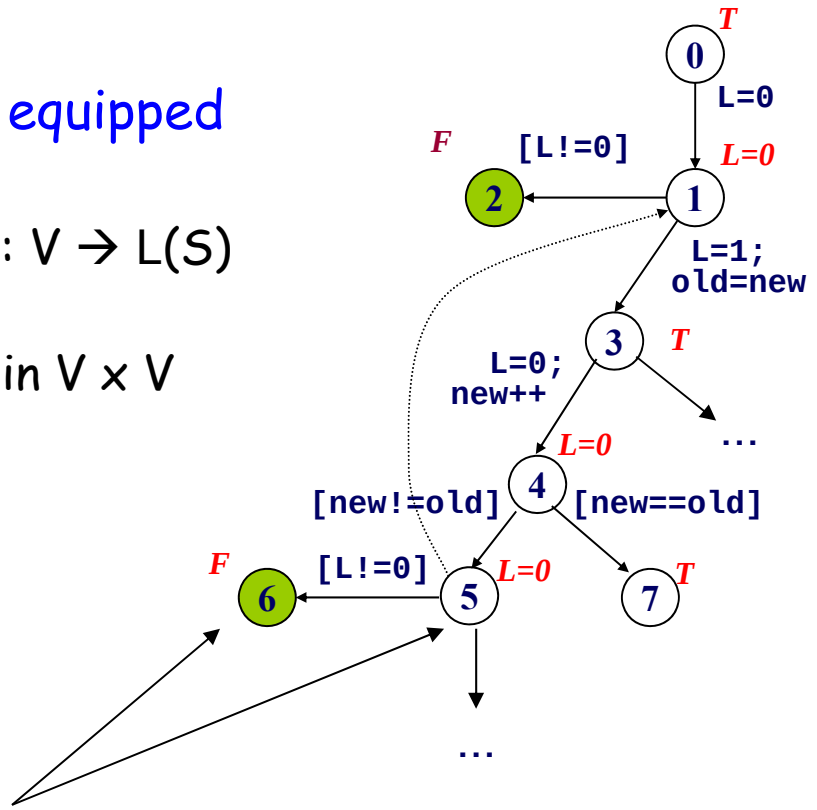
Program Unwinding

- Unwinding is **safe** iff for all vertices v in V, $M_v(v)=l_f$ implies $\Psi(v) \equiv$ **FALSE**

- Unwinding is **complete** iff every leaf v in V is covered

Labeled unwinding

- A labeled unwinding is equipped with:
  - a labeling function $\psi : V \rightarrow L(S)$ (T here is True)
  - a covering relation C in V x V



These two nodes are **covered**.

(have a ancestor at the tail of a covering arc)

Labeled Program Unwinding

- *A* labeled unwinding (U,ψ,C) of a program $A = (\Lambda, \Delta, l_i, l_f)$ where $U = (V,E,M_v,M_e)$, is **well-labeled** iff:
  - $\Psi(\varepsilon) \equiv$ TRUE, and
  - For every edge (v,w) in E, $\Psi(v) \wedge M_e(v,w)$ implies $\Psi(w)'$, and
  - For all (v,w) in *C*, $\Psi(v) \Rightarrow \Psi(w)$, and **w is not covered**

Labeled Program Unwinding

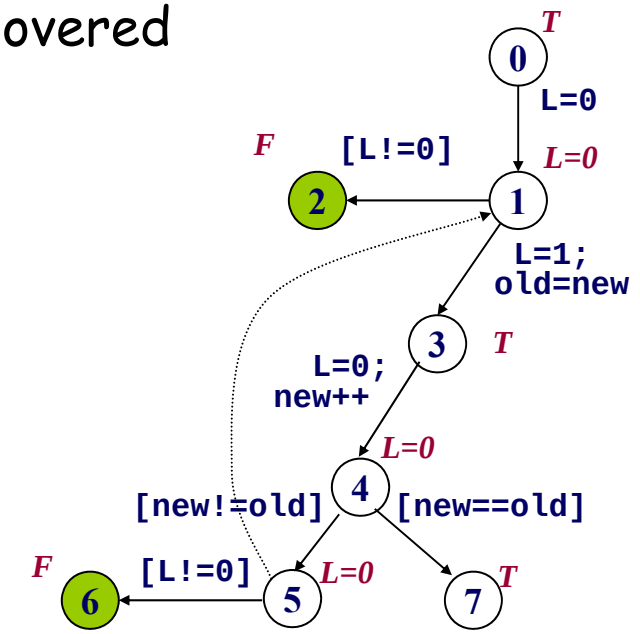
Main Theorem:

If there exists a safe, complete, well-labeled unwinding of program A, then A is safe

Recall: A program is **safe** if every error path of the program is **in**feasible

Well-Labeled Unwinding

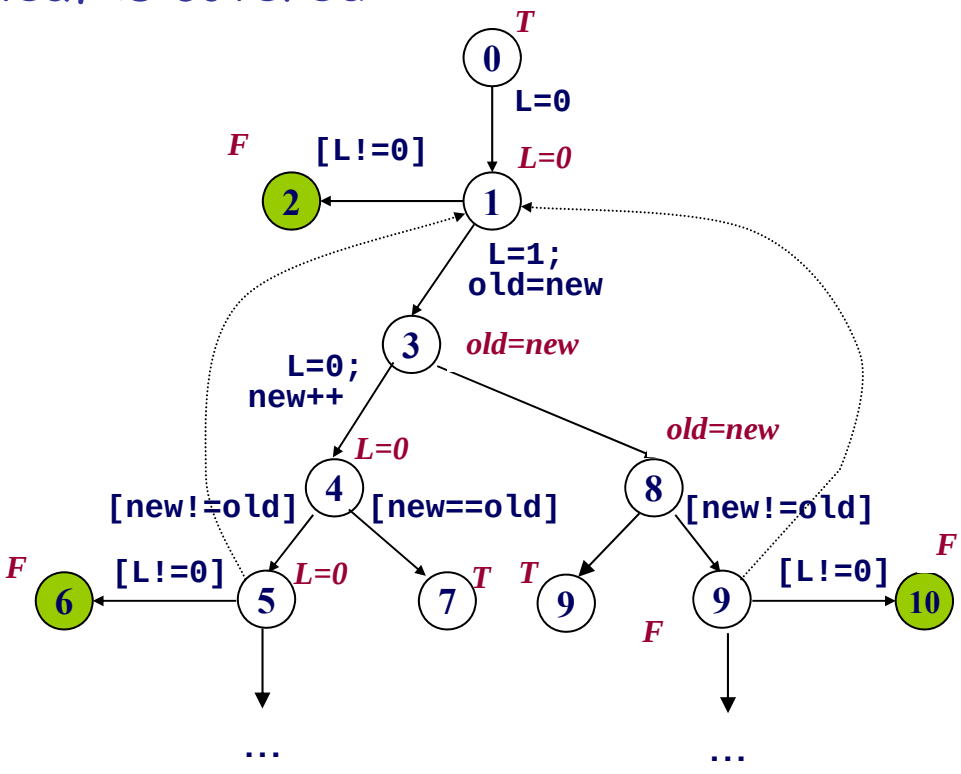
- An unwinding is well-labeled when
  - $\psi(\varepsilon)$ = True
  - every edge is a valid Hoare triple
  - if (x,y) in C then y is not covered



Safe and Complete

safe if every error vertex is labeled False

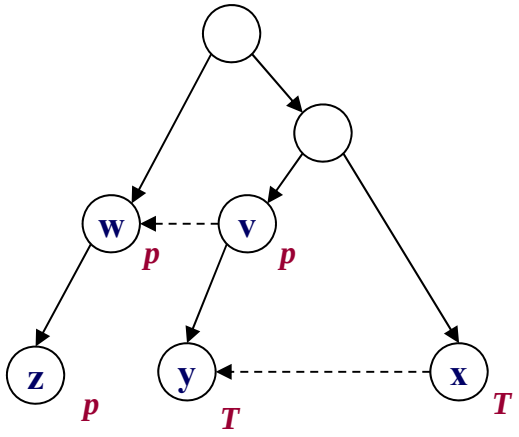
complete if every non-terminal leaf is covered



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

Why a Covered Vertex Cannot Cover?

- y covers x, w covers v $\Rightarrow$ y is covered ($v \leq y$)
  - $\Psi(x) \Rightarrow \Psi(y)$
    - Every state reachable from x is reachable from y.
  - $\Psi(v) \Rightarrow \Psi(w)$
    - Every state reachable from v is reachable from w.
- Any state reachable from y should be reachable from w through its descendent z.
- NOT every state reachable from x is also reachable from z.
- z is the only vertex that is not covered.



Interpolants for Sequences

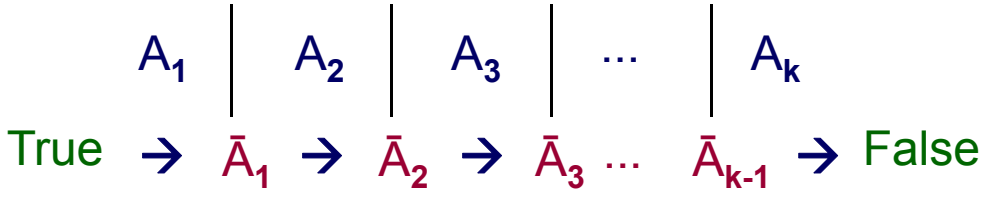
- To handle program paths, a generalization of interpolant is needed

- Given a sequence of formulas $\Gamma = A_1, A_2, \ldots, A_n$, we say that $\bar{A}_0, \bar{A}_1, \ldots, \bar{A}_n$ is an **interpolant for Γ** when:
  - $\bar{A}_0$ = TRUE and $\bar{A}_n$ = FALSE,
  - For all $1 \le i \le n$, $\bar{A}_{i-1} \wedge A_i$ implies $\bar{A}_i$, and
  - For all $1 \le i \le n$, $\bar{A}_i$ is in $L(A_1,\ldots,A_i) \cap L(A_{i+1},\ldots,A_n)$

- If Γ is quantifier-free we can derive a quantifier-free interpolant for Γ (from the refutation of Γ)

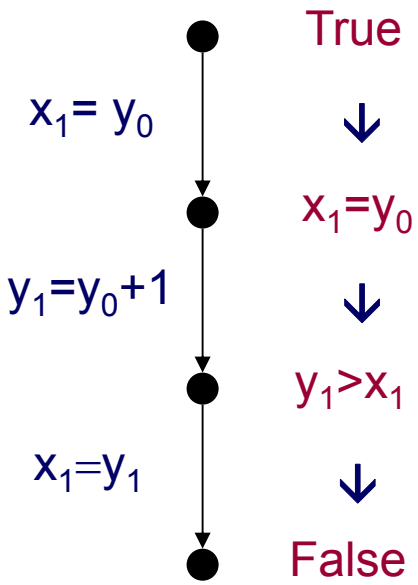
Interpolants for Sequences

- An intuition…



- So this is a structured refutation of $A_1,\ldots,A_k$

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

Path refinement procedure

Interpolation

Lazy PA with Interpolants

- Procedure Expand (v in V)
  if v is uncovered leaf then
    for all actions $(M_v(v), T, m)$ in Δ
      add a new vertex w to V and a new edge (v,w) to E;
      $M_v(w) = m$ and $\psi(w) =$ **True**;
      $M_e(v,w) = T$;

Lazy PA with Interpolants

- Procedure Refine (v in V)

  if $M_v(v) = l_f$ and $\psi(v) \mathrel{!}= FALSE$ *then*

  let $\pi = (v_0, T_0, v_1)\ldots(v_{n-1}, T_{n-1}, v_n)$ *be the unique path from* ε *to v.*

  if $U(\pi)$ has an interpolant $A'_0,\ldots,A'_n$ then

  for i=0…n:

  let $\Phi = A'^{(-i)}_i$ * remove primes

  if $\psi(v_i)$ does not imply Φ then

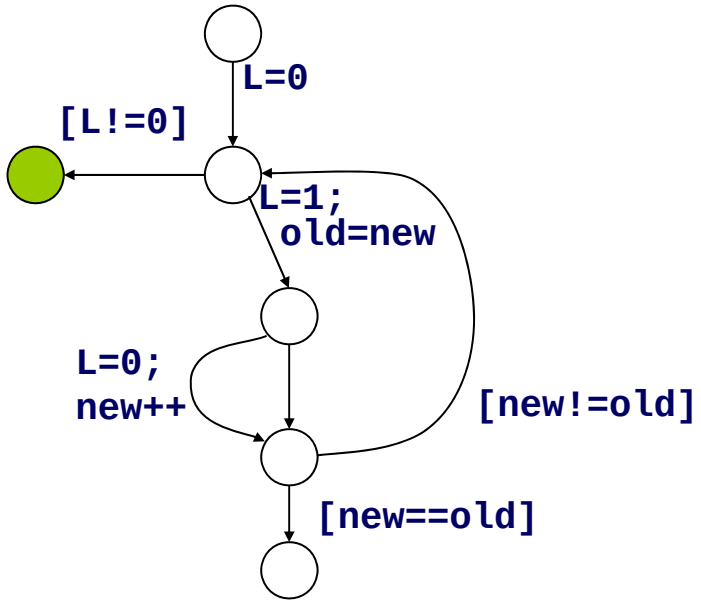
  remove all pairs $(\cdot, v_i)$ from C

  set $\psi(v_i) = \psi(v_i) \wedge \Phi$

The Example

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

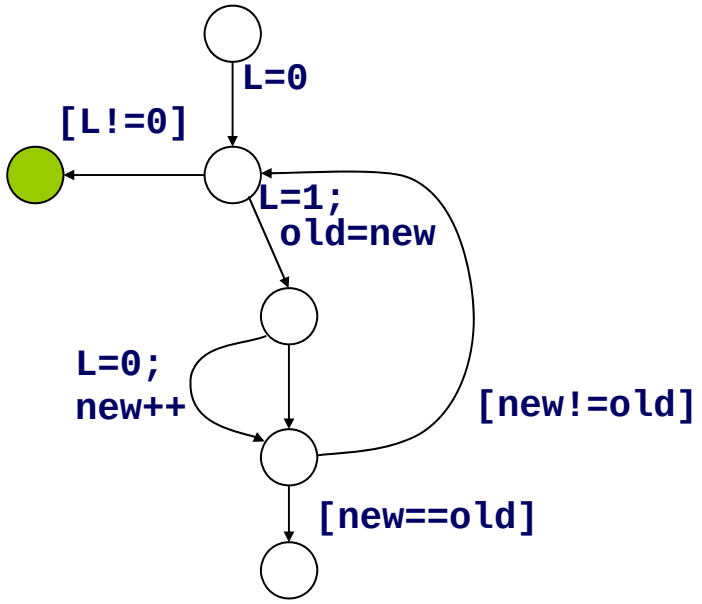
program fragment



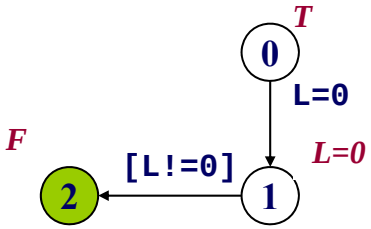
control-flow graph

- Property: lock() is not called if the lock is already being held

Unwinding the CFG

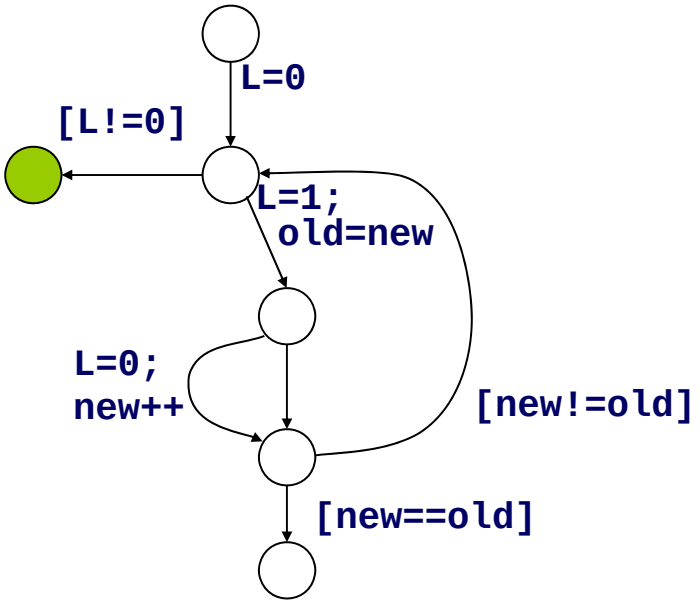


control-flow graph

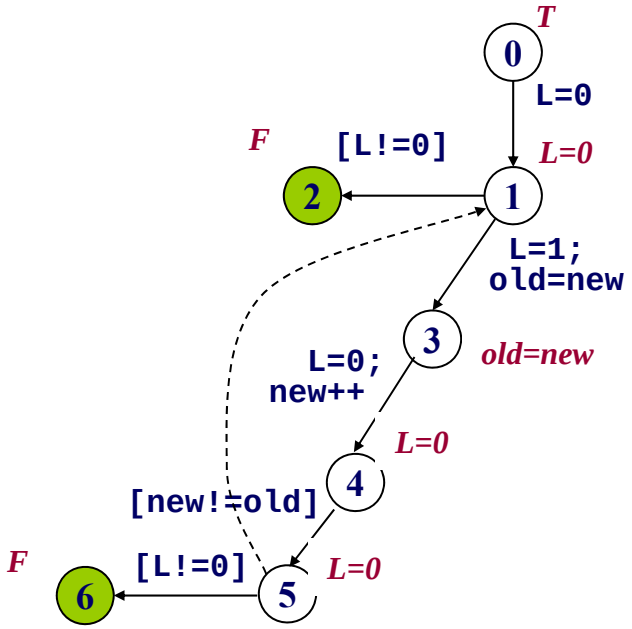


Label error state with false, by refining labels on path

Unwinding the CFG

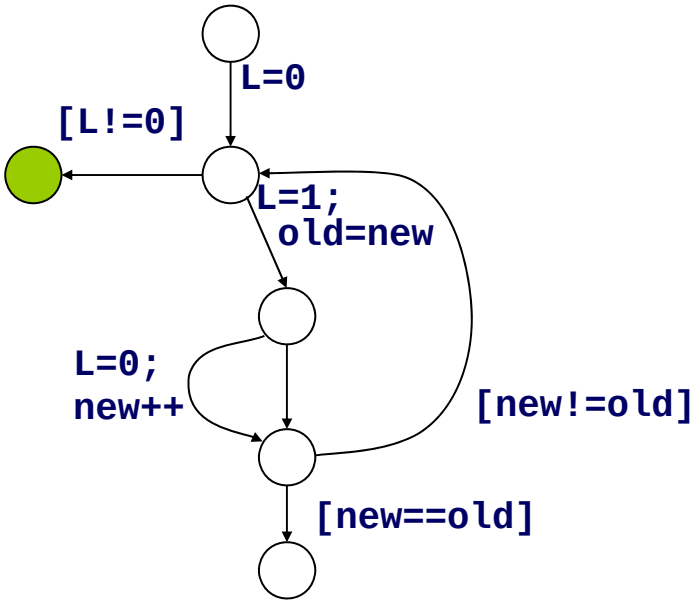


control-flow graph

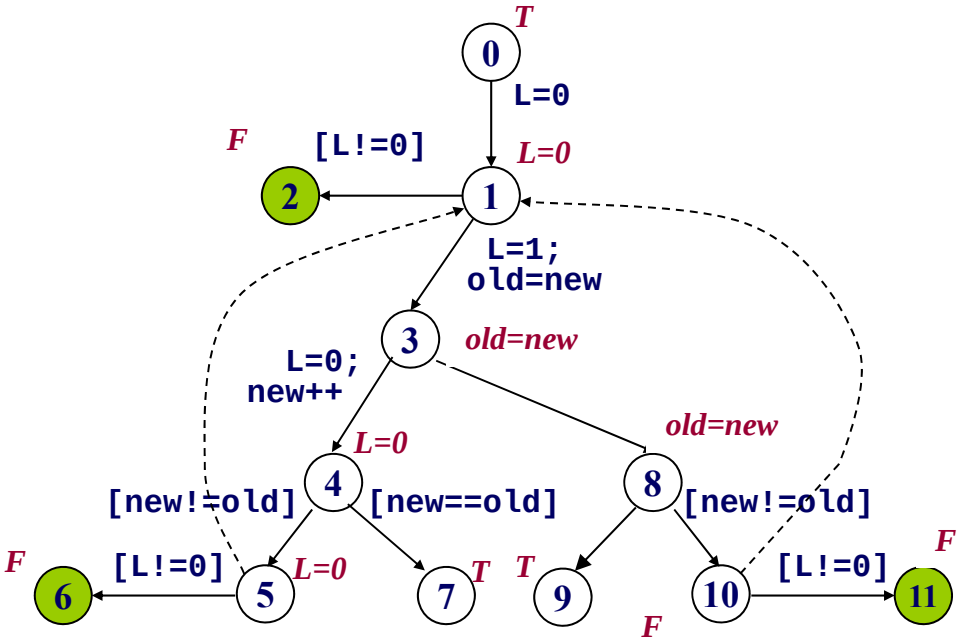


Covering: state 5 is subsumed by state 1.

Unwinding the CFG



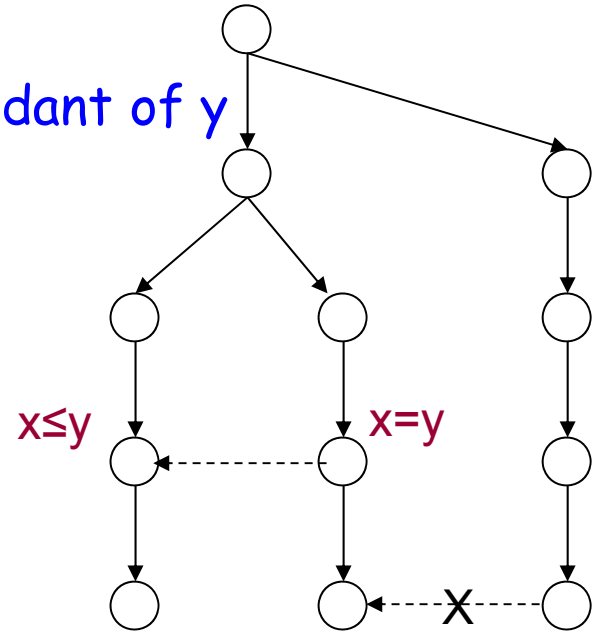
control-flow graph



Another cover. Unwinding is now complete.

Covering Step

- If $\psi(x) \rightarrow \psi(y)$...
  - add covering arc (x, y) to C
  - remove all (z, w) in C for w descendant of y



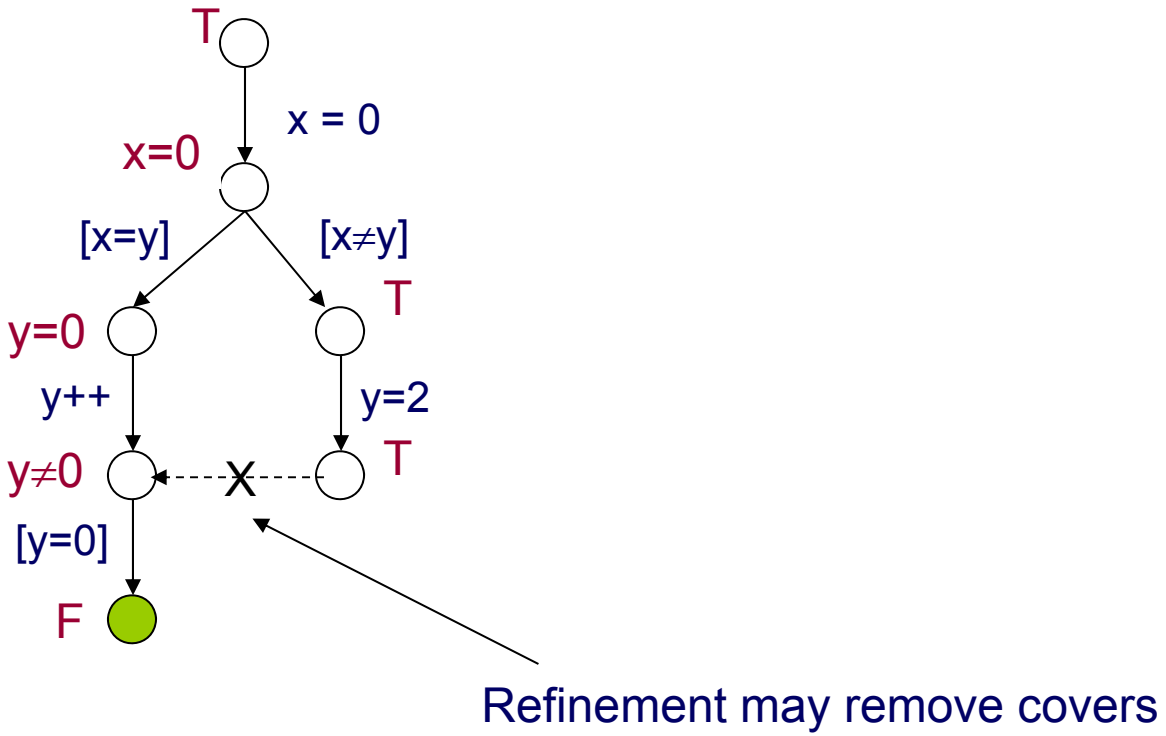
We restrict covers to be descending in a suitable total order on vertices. This prevents covering from diverging.

Covering Step

- Covering one vertex may result in uncovering others.
  - Applying covering non-deterministically may not terminate.
- A total order is defined ◄ on the vertices.
  - Respects the ancestor relation.
    - $v \leq w \rightarrow v \blacktriangleleft w$
- Can be defined by a preorder traversal of the tree.
- Cover is only applied to (v,w) if w ◄ v.
  - If by adding (v,w) we remove (x,y) where $v \leq y$. By transitivity we get v ◄ x.
  - Covering a vertex v can result in uncovering vertices greater then v.
- Therefore, we cannot apply covering infinitely.

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.



The Complete Algorithm

- *A vertex v is said to be **closed** if either it is covered or no covering arc (v,w) can be added to C (while maintaining well-labeledness).*
  - *Procedure Close(v in V)*
    For all w in V s.t. w ◄ v and $M_v(v) = M_v(w)$:
      Cover(v,w)
  - *Procedure DFS(v in V)*
    Close(v)
    if v is uncovered then
      if $M_v(v) = l_f$ then
        Refine(v);
        for all w ≤ v: Close(w)
        Expand(v);
        for all children w of v: DFS(w)

The Complete Algorithm (2)

- *Procedure Unwind()*

  $V = \{\varepsilon\}$, $E = \Phi$, $\psi(\varepsilon) = True$, $C = \Phi$

  While there exists an uncovered leaf v in V:

  for all w in V s.t. $w \leq v$: Close(w);

  DFS(v);

- *Theorem: If procedure Unwind terminates without aborting on a program A, then A is safe.*
  - *Proof: Expand, Refine and Cover alter the unwinding and all preserve well-labeledness, the resulting unwinding is well-labeled.*
  - *All vertices are refined → The unwinding is safe*
  - *Terminates when there are no more uncovered leaves → Complete.*

Something about Interpolant

- A(X,Y) ∧ B(Y,Z) ≡ FALSE
  - There exists I(Y) such that
    - A(X,Y) → I(Y)
    - I(Y) ∧ B(Y,Z) ≡ FALSE
- The "best" interpolant: $I(Y) = (\exists \overline{X})(A(\overline{X}, Y))$
  - $A(X, Y) \Rightarrow (\exists \overline{X})(A(\overline{X}, Y))$
  - $(\exists \overline{X})(A(\overline{X}, Y)) \wedge B(Y, Z) \equiv TRUE \Rightarrow A(X, Y) \wedge B(X, Y) \equiv TRUE$
- Interpolantion is an Existential Quantification