Program Verification II.

Critical Architectures Laboratory

Tamás Tóth

totht@mit.bme.hu

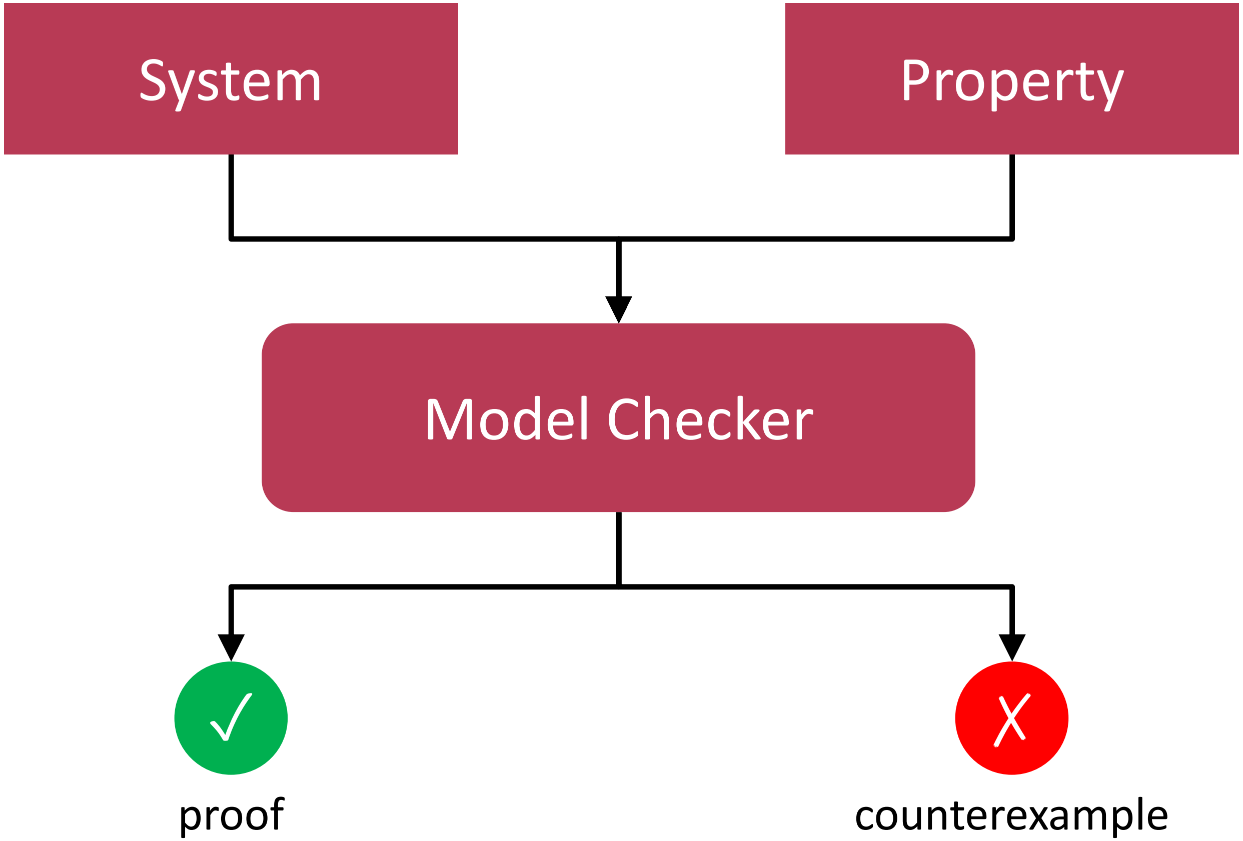
Budapest University of Technology and Economics
Fault Tolerant Systems Research Group

INTRODUCTION

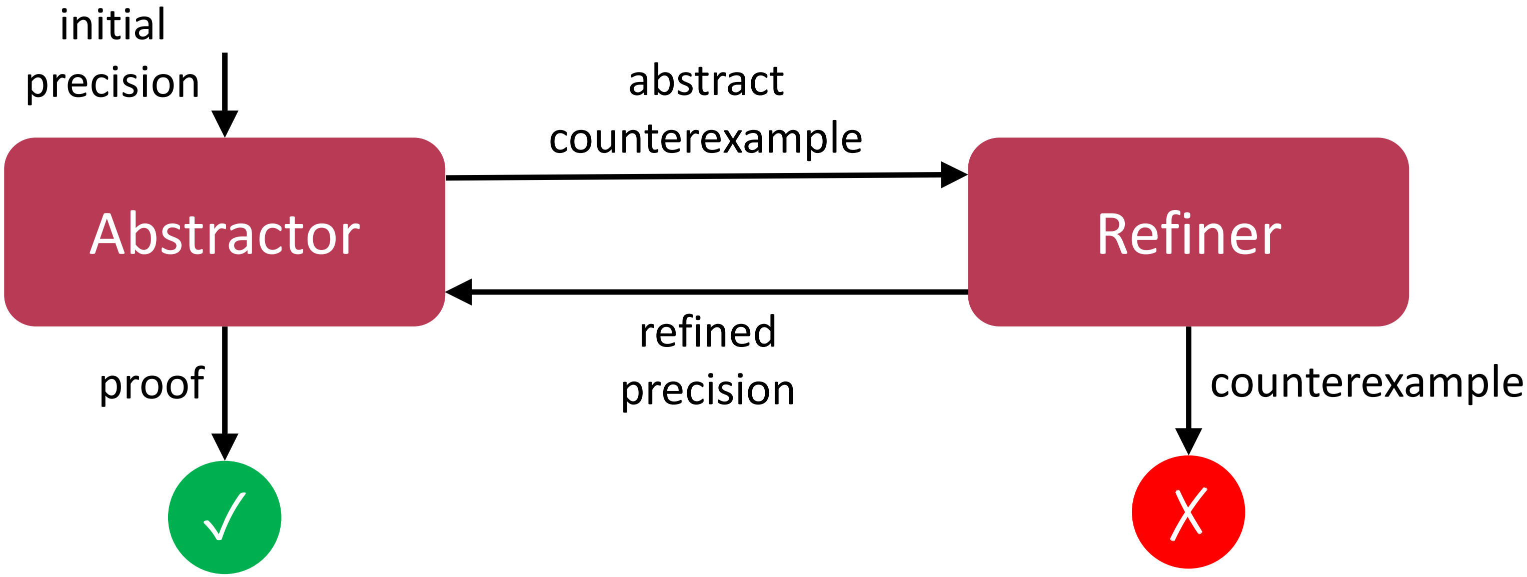
Topic of the Lab Session:

Implement a model chekcer based on Counterexample-Guided Abstraction Refinement (CEGAR)

Model Checking



CEGAR


initial precision
abstract counterexample
Abstractor
Refiner
refined precision
proof
counterexample
✓
✗

VERIFICATION WORKFLOW

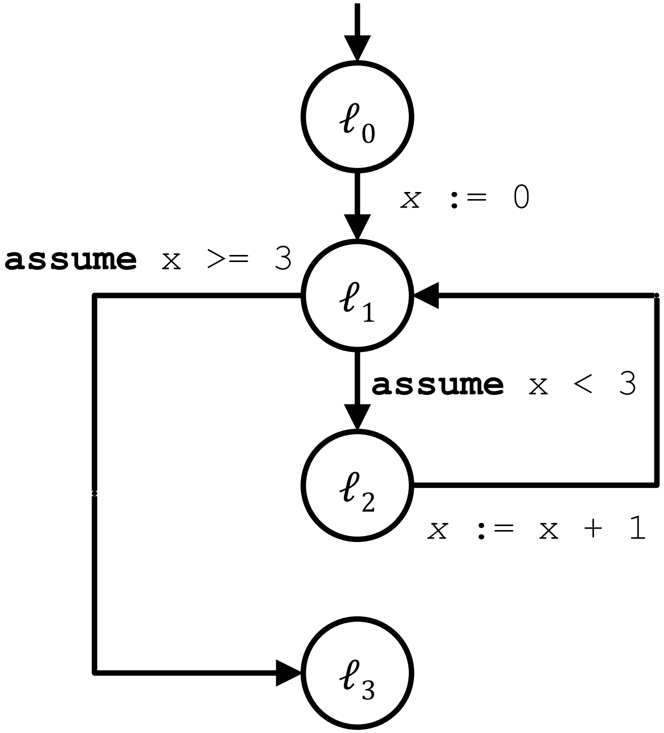
Abstraction

Given the CFA and a precision π, we build an *abstract reachability tree*

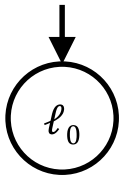
- An unwinding of the CFA to a rooted directed tree
- Each node is labeled by a set of literals over π
  - overapproximate the post-image of the parent
- Covering edges between nodes
  - the covering node is not covered
  - the nodes represent the same location
  - the label of the covering node entails the label of the covered node

Building the abstraction: step by step

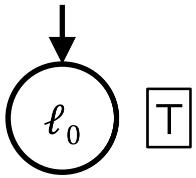
Let precision $\pi = \{x < 3\}$.



Building the abstraction: step by step

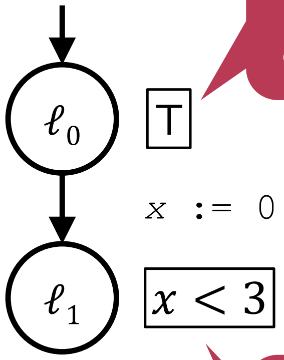


Building the abstraction: step by step



In the initial state all variables have an arbitrary value

Building the abstraction: step by step

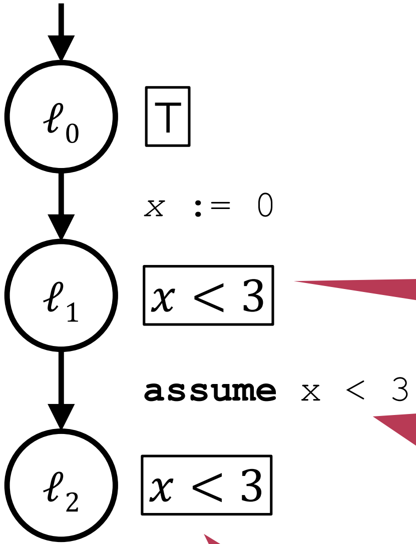


If from an arbitrary state …

… we execute the assignment …

… we only obtain states where $x < 3$ holds

Building the abstraction: step by step

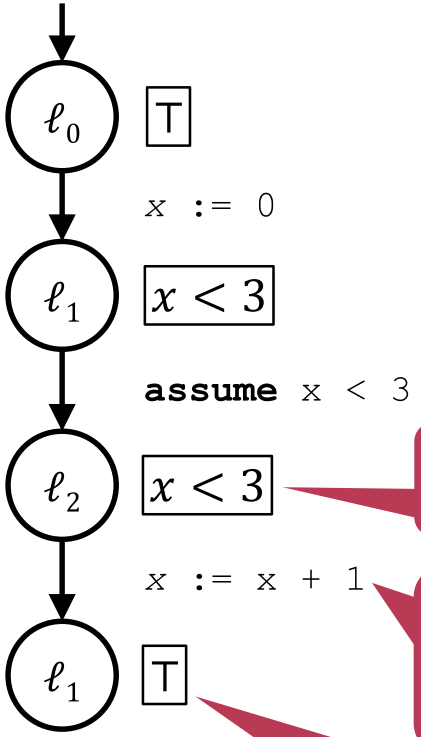


if $x < 3$ …

… then we can pass the assumption …

… and still have $x < 3$

Building the abstraction: step by step

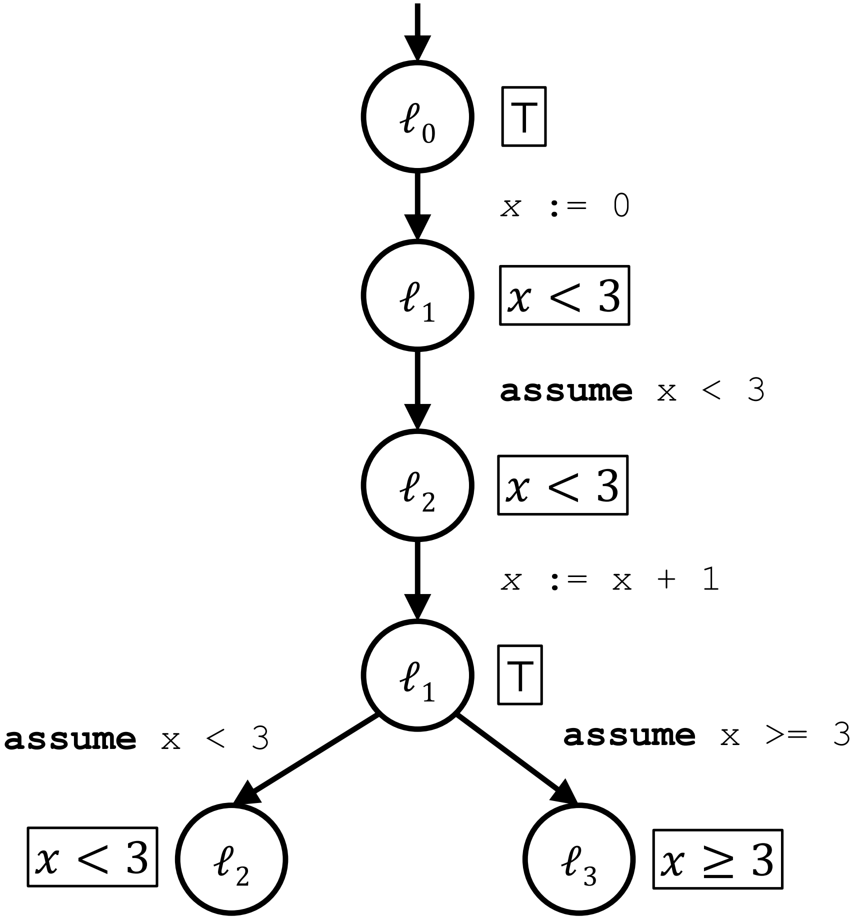


if $x < 3$ …

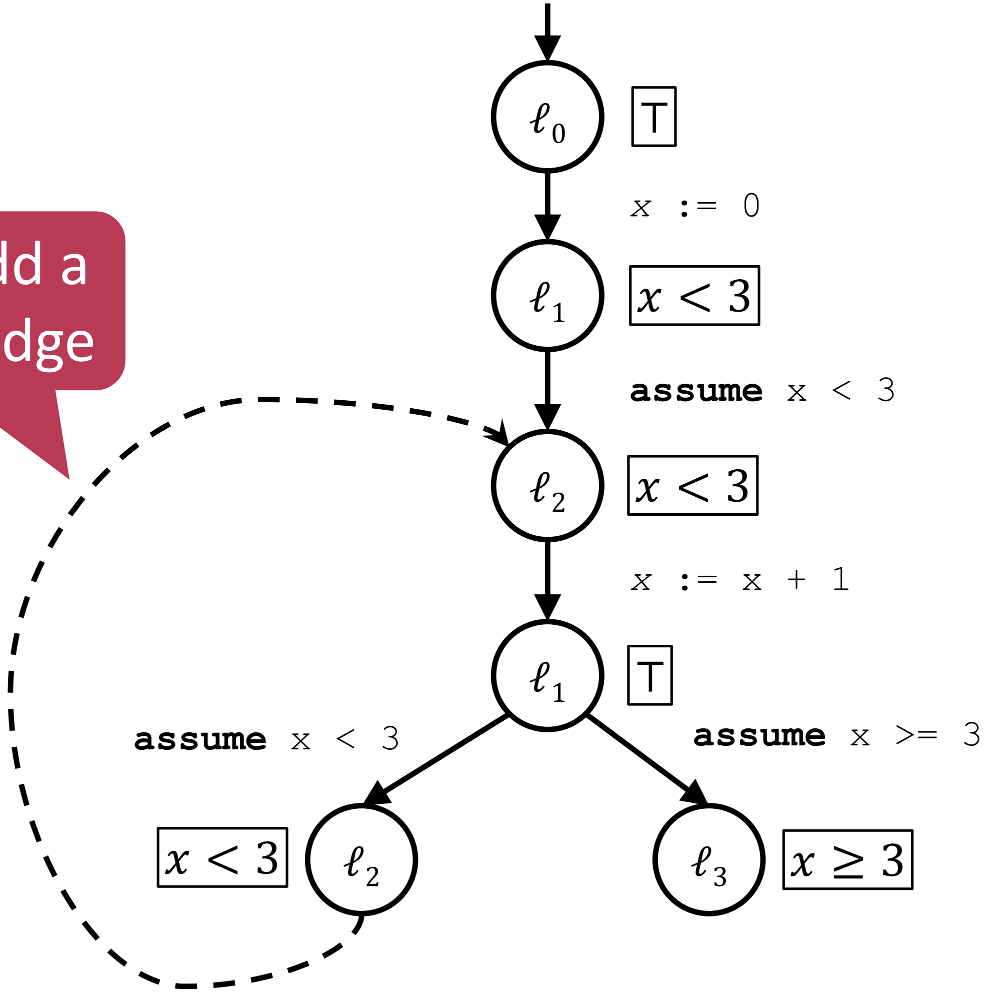
… and we increment it …

… $x < 3$ either holds or not, thus we have an arbitrary state

Building the abstraction: step by step

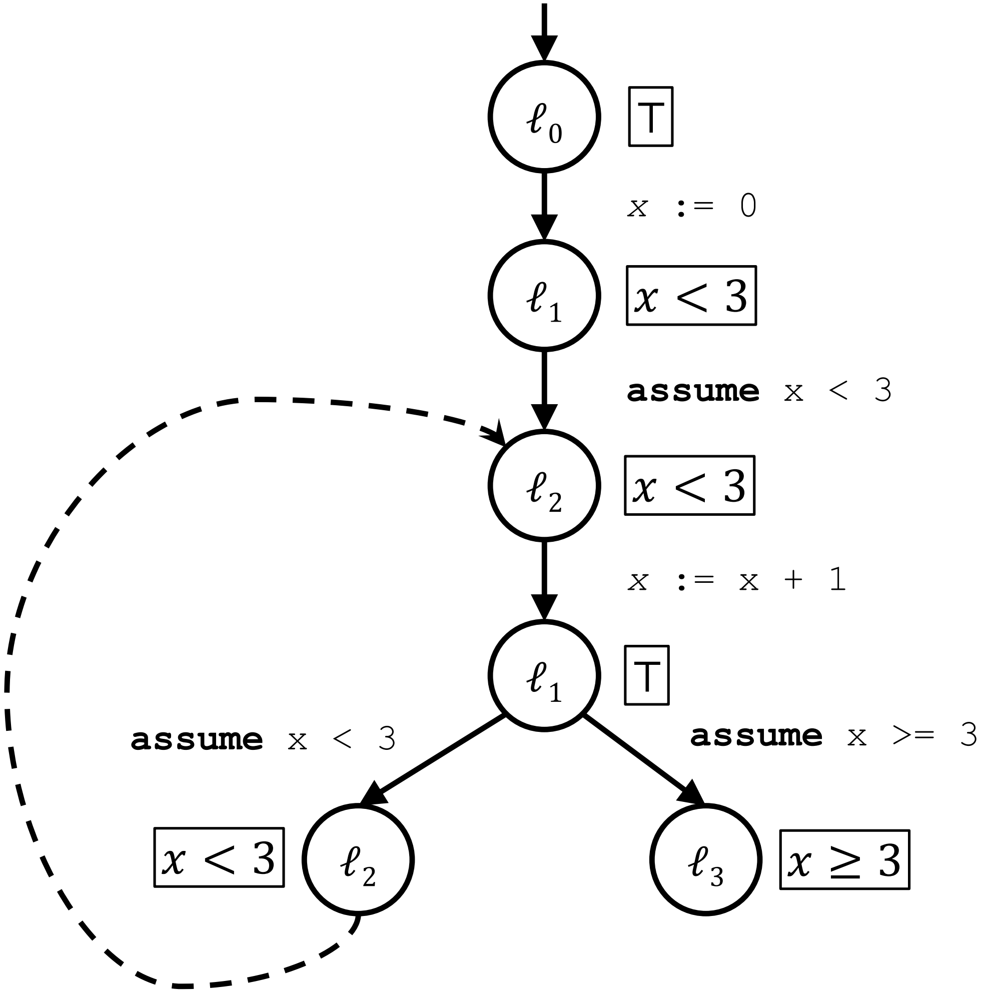


Building the abstraction: step by step



We can add a covering edge

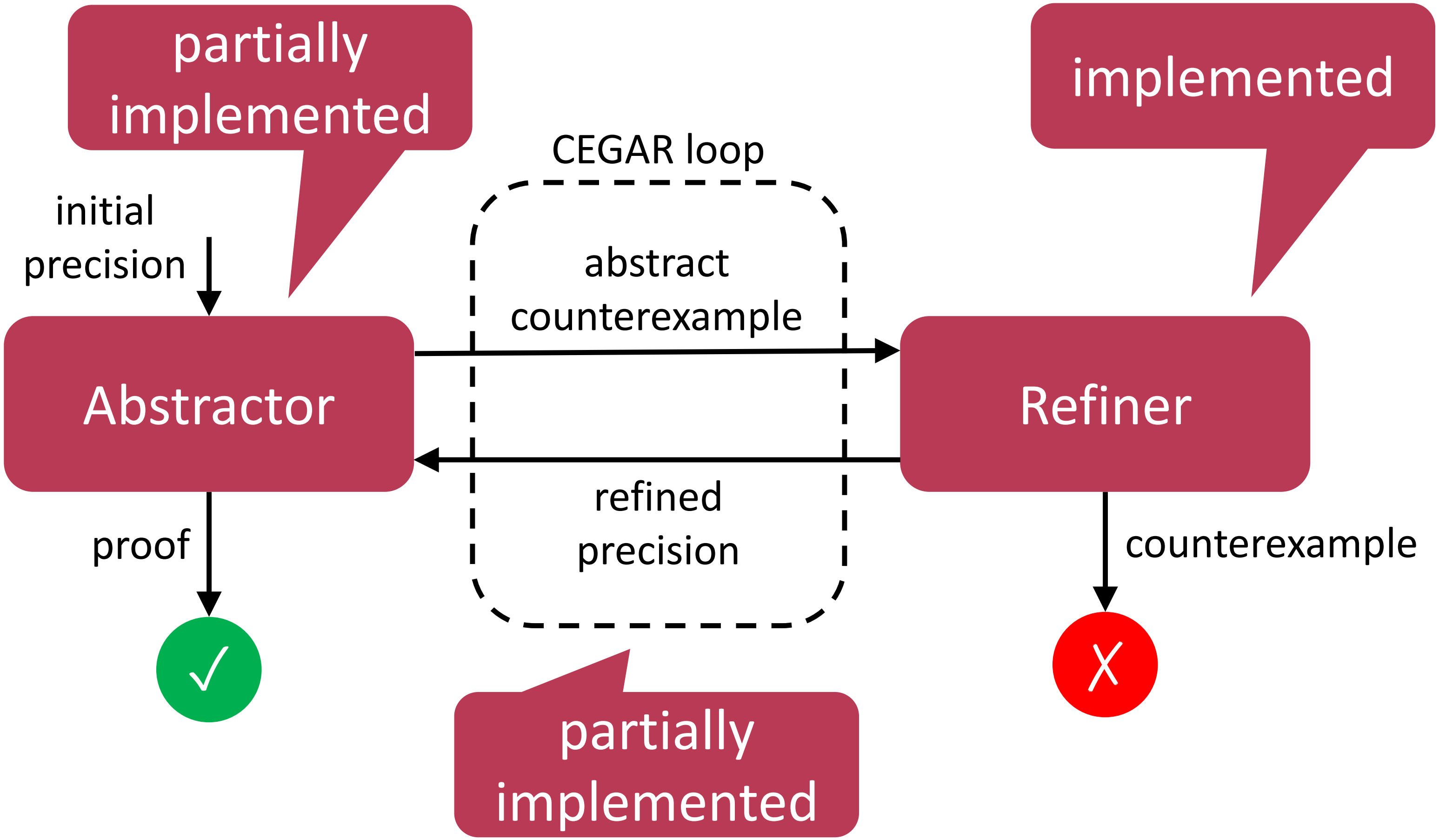
Building the abstraction: result



Refinement

- The abstract reachability tree represents an *overapproximation* of all possible behaviors
- It may contain *spurious counterexamples*:
  a path to an error location that is not feasible
- Refinement: add new predicates to the precision
- Rebuild the tree based on the new precision

CEGAR: Tasks


partially implemented
implemented
CEGAR loop
initial precision
abstract counterexample
Abstractor
Refiner
refined precision
proof
counterexample
✓
X
partially implemented

Pseudocode for the Abstractor

```
$waitlist$ := { $root$ }
while there exists an element $n$ in $waitlist$ do
        remove $n$ from $waitlist$
        if $n$ is an error node then
                return counterexample path to $n$
        else if there exists $n'$ that may cover $n$ then
                add covering edge from $n$ to $n'$
        else
                expand $n$ w. r. t. $\pi$
                add all successors of $n$ to $waitlist$
return the program is correct
```

LIST OF QUESTIONS

List of questions

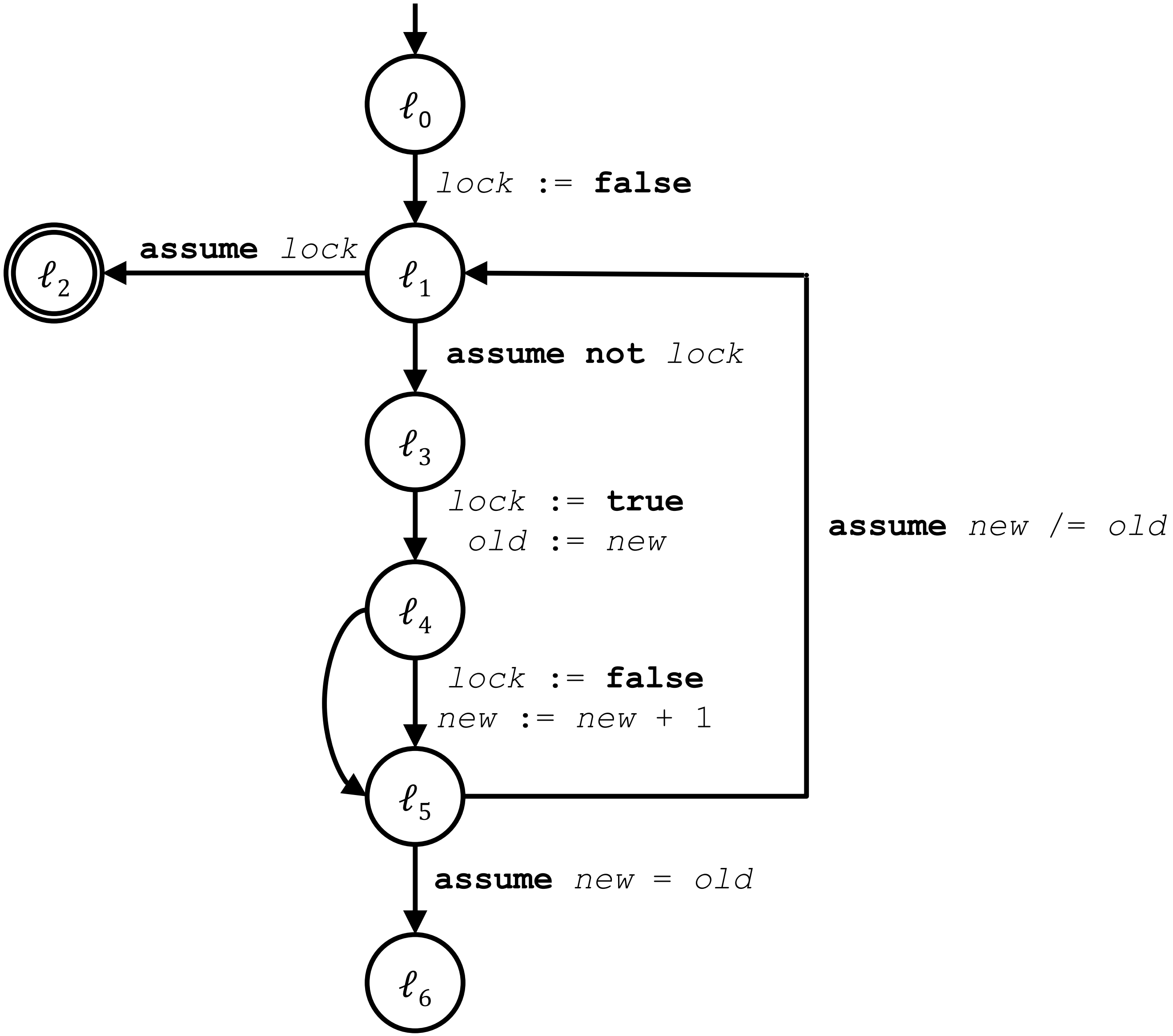
Consider the program given on the next slide.

1. Build the abstraction for $\pi = \emptyset$.
   Is the abstraction safe?
   (Does it prove the correctness of the program?)
2. Build the abstraction for $\pi = \{lock\}$.
   Is the abstraction safe?
3. Build the abstraction for $\pi = \{lock, old = new\}$.
   Is the abstraction safe?

Example

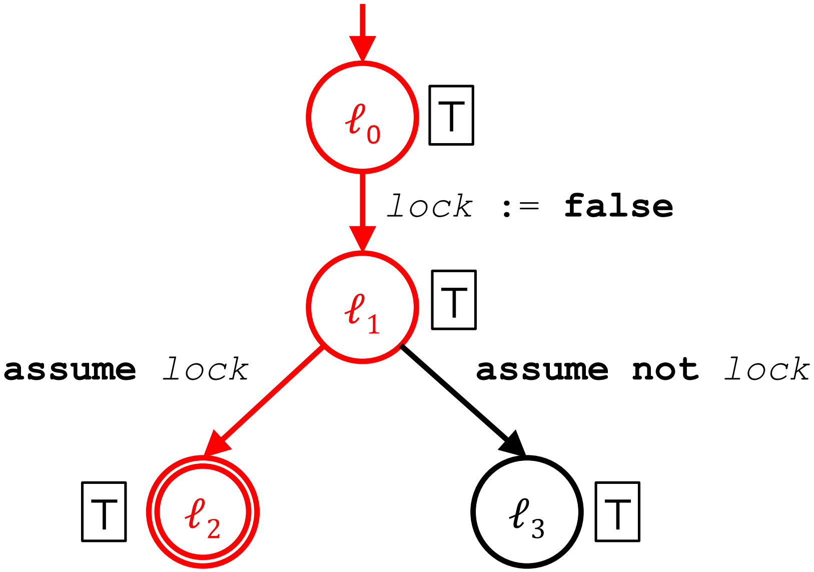
```
lock = false;
do {
  assert(!lock);
  lock = true;
  old = new;
  if (*) {
    lock = false;
    new++;
  }
} while (new != old);
```

Example


ℓ_0
lock := false
assume lock
ℓ_2
ℓ_1
assume not lock
ℓ_3
lock := true
old := new
assume new /= old
ℓ_4
lock := false
new := new + 1
ℓ_5
assume new = old
ℓ_6

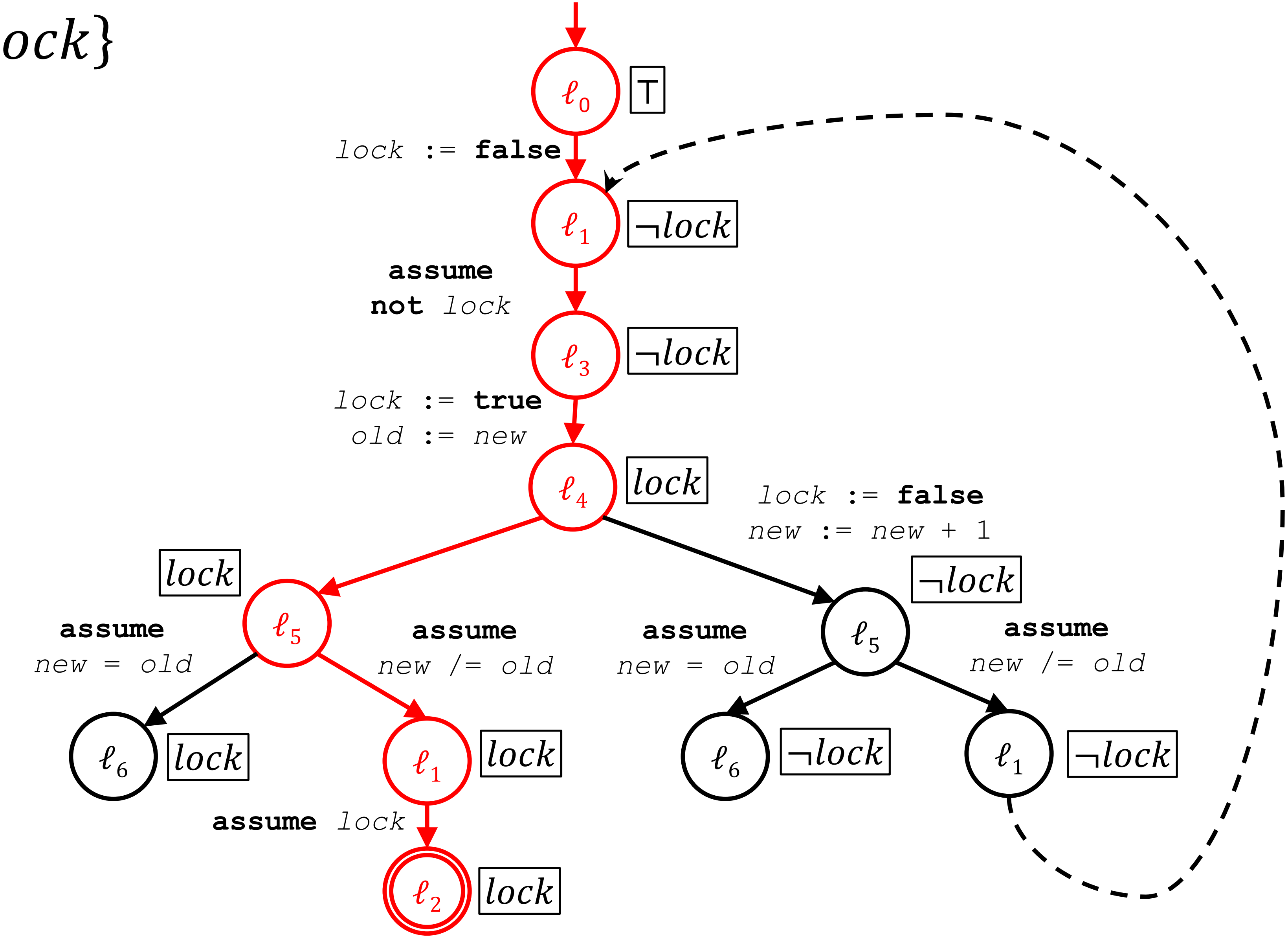
Solution (1)

$\pi = \emptyset$



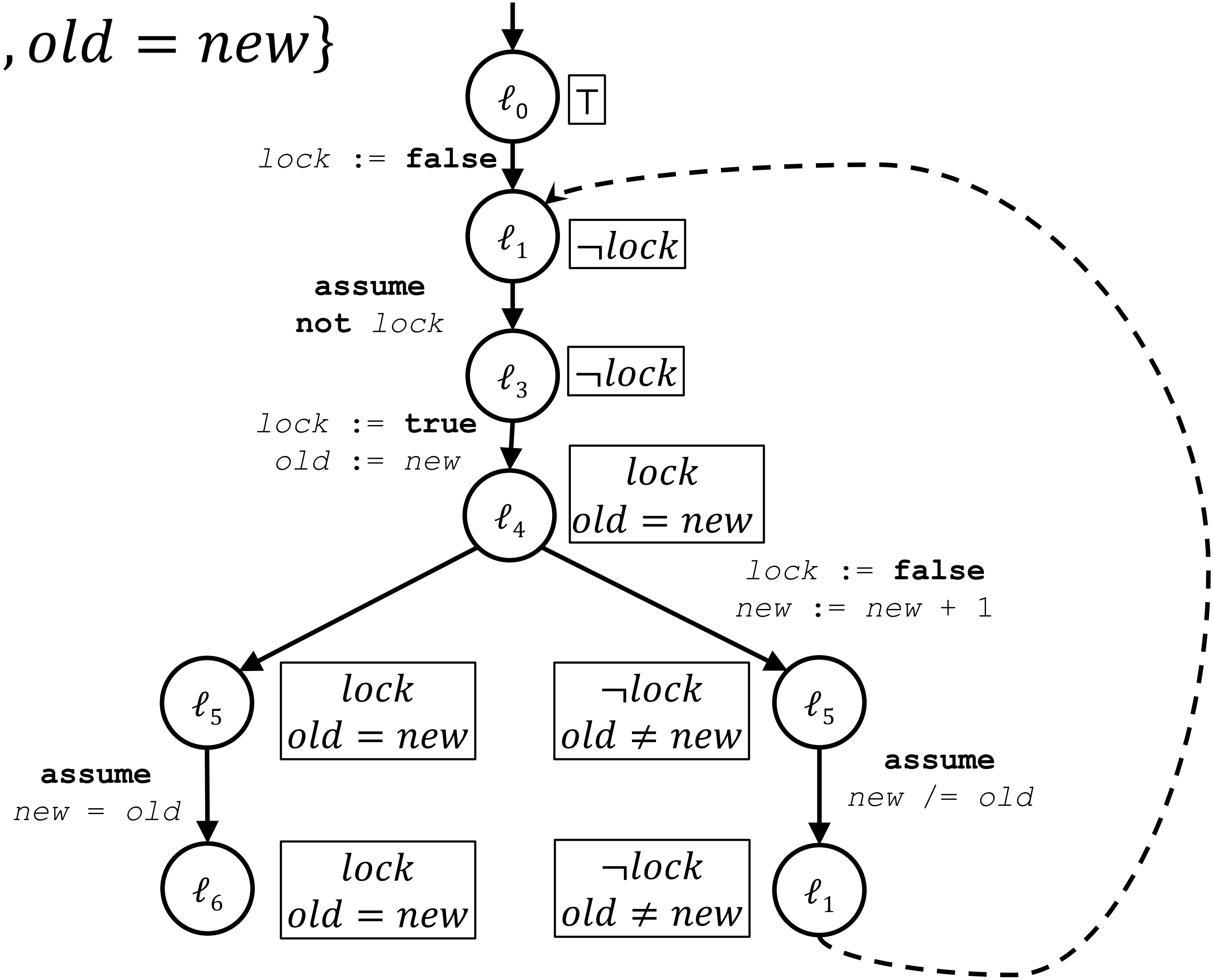
Solution (2)

$\pi = \{lock\}$



Solution (3)

$\pi = \{lock, old = new\}$



ℓ_0
$\top$
lock := false
ℓ_1
$\neg lock$
assume not lock
ℓ_3
$\neg lock$
lock := true
old := new
ℓ_4
$lock$
$old = new$
lock := false
new := new + 1
ℓ_5
$lock$
$old = new$
$\neg lock$
$old \neq new$
ℓ_5
assume new = old
assume new /= old
ℓ_6
$lock$
$old = new$
$\neg lock$
$old \neq new$
ℓ_1