Challenges with Algorithmic Program Verification

Ahmed Rezine

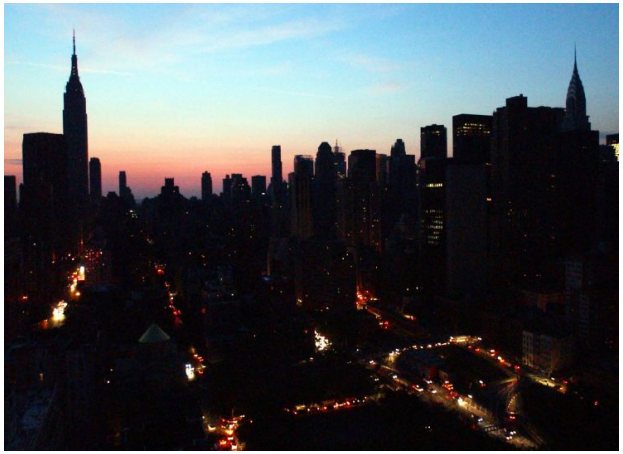
Universitetslektor

October 4, 2019

Outline

Outline

Correctness is important



03'Northeastern blackout
(race condition)

96' Ariane explosion
(uncaught exception)

Therac 25, 1985-87
(improper testing)

Shutdown Sept 2004
(automatic shutdown)

Software Verification

Bugs (cont)

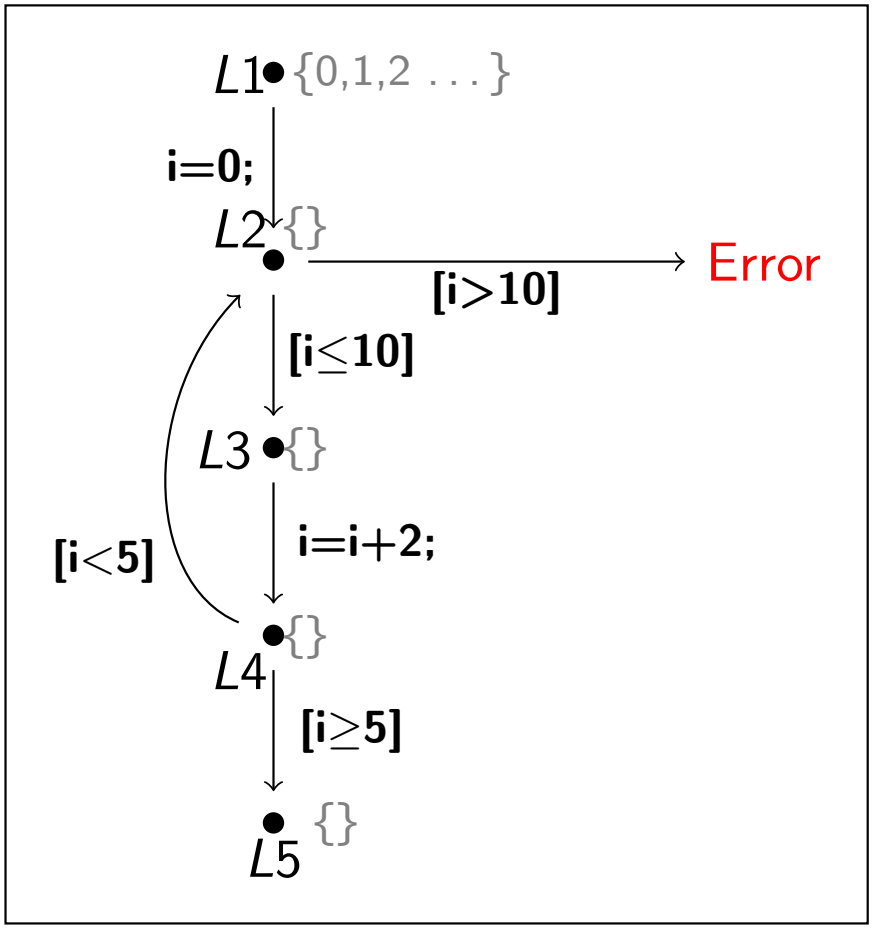
- Programmers spend far longer time fixing bugs in existing code than they do writing new one: more than half of the software-development costs of a typical project are spent on identifying and fixing defects.
- Bill Gates, April 2002: Things like even software verification, this has been the Holy Grail of computer science for many decades but now in some very key areas, for example, driver verification we're building tools that can do actual proof about the software and how it works in order to guarantee the reliability.

Outline

Static Analysis

Concrete Interpretation:

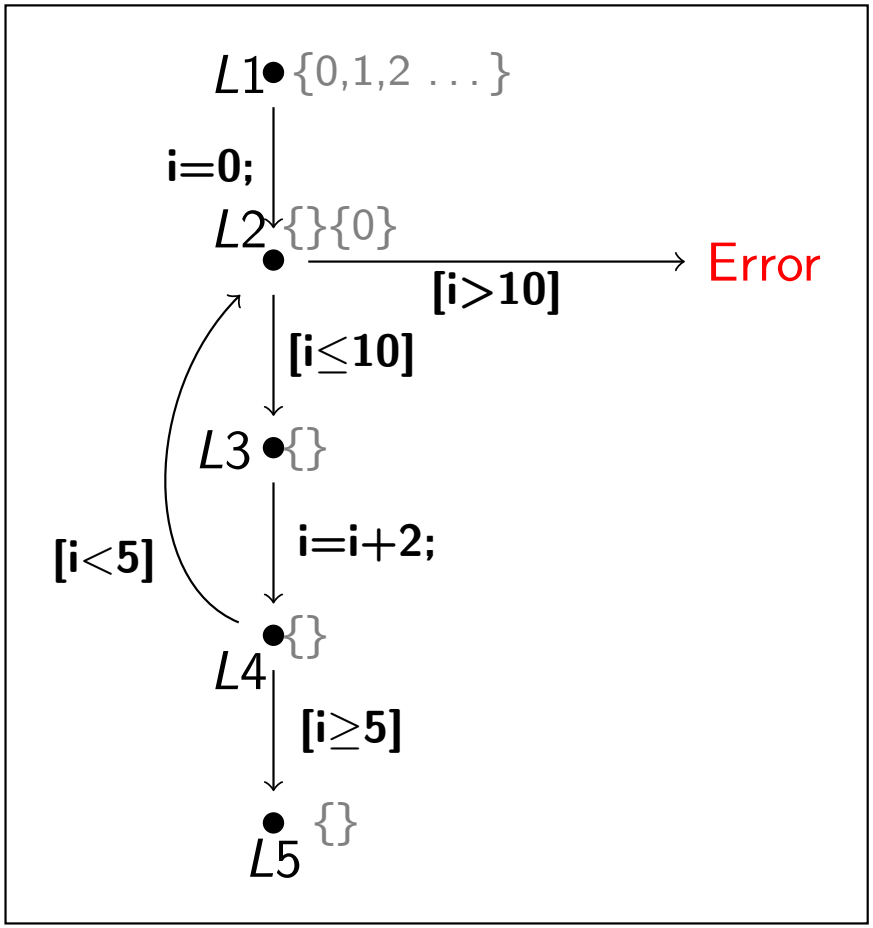
```
int i = 0;
do{
    assert(i <= 10);
    i = i+2;
  }while(i < 5)
```



Static Analysis

Concrete Interpretation:

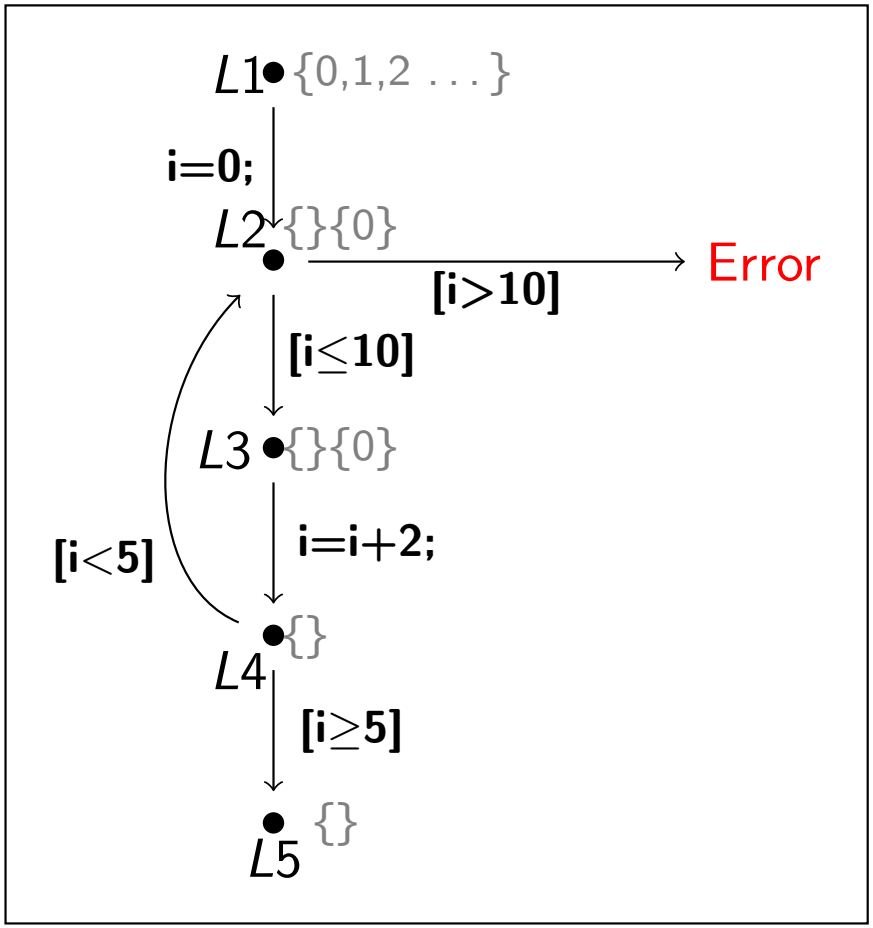
```
int i = 0;
do{
    assert(i <= 10);
    i = i+2;
  }while(i < 5)
```



Static Analysis

Concrete Interpretation:

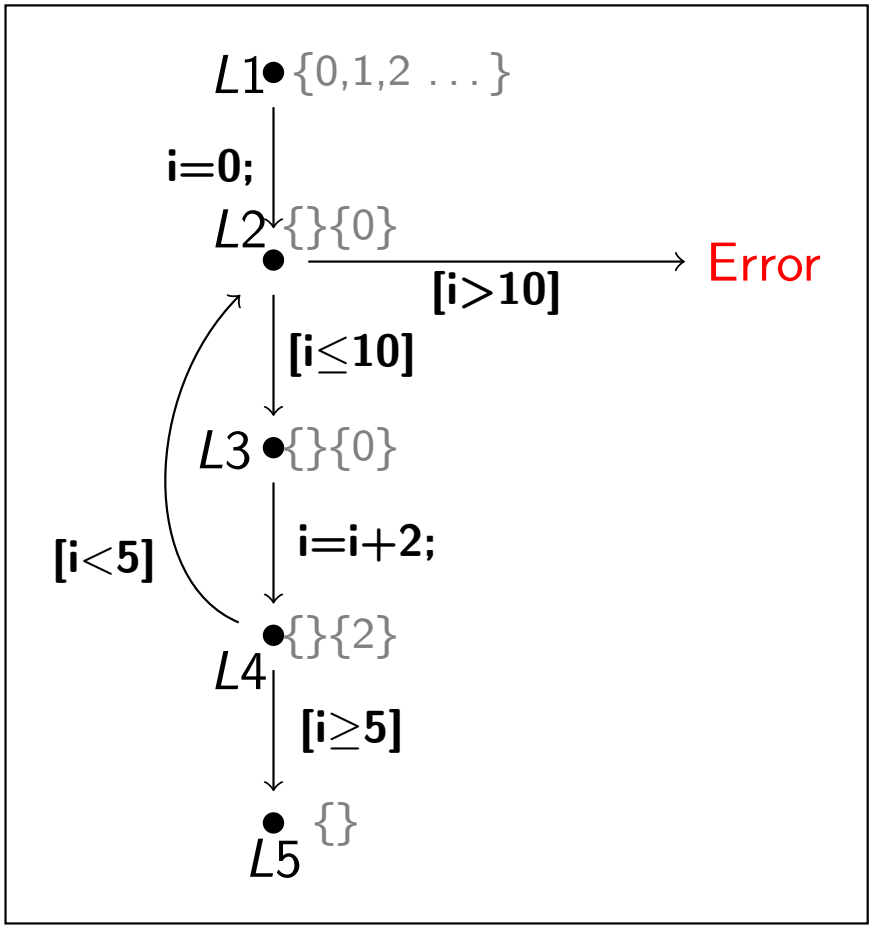
```
int i = 0;
do{
    assert(i <= 10);
    i = i+2;
  }while(i < 5)
```



Static Analysis

Concrete Interpretation:

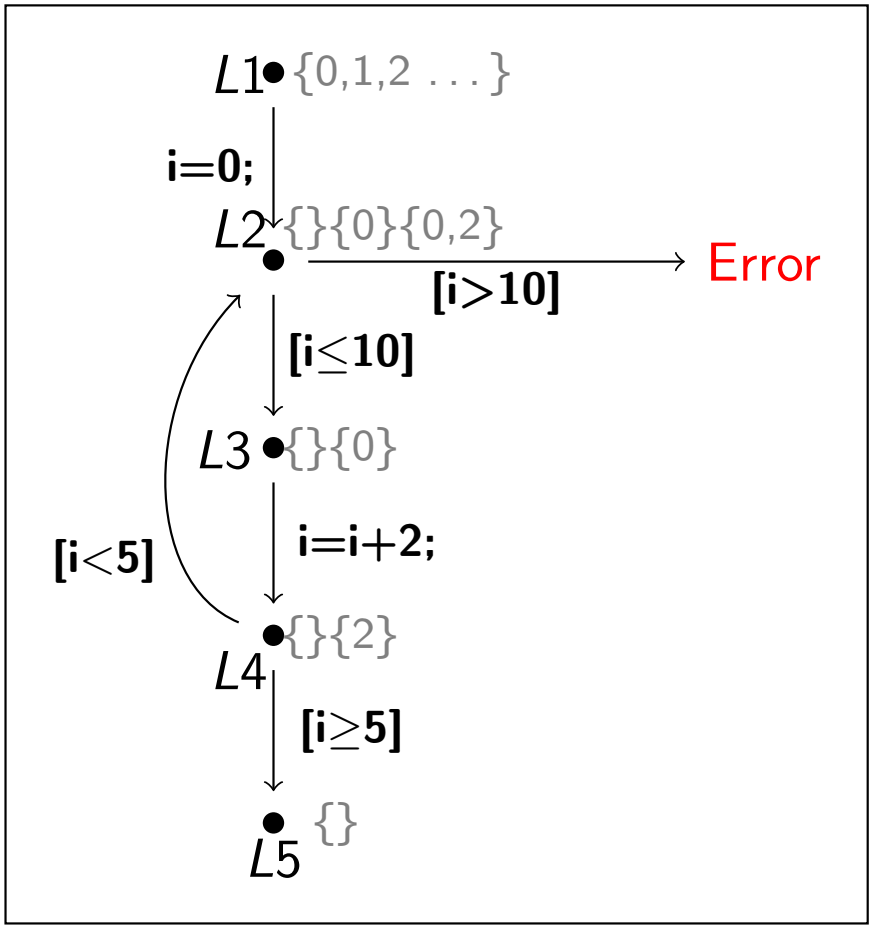
```
int i = 0;
do{
    assert(i <= 10);
    i = i+2;
  }while(i < 5)
```



Static Analysis

Concrete Interpretation:

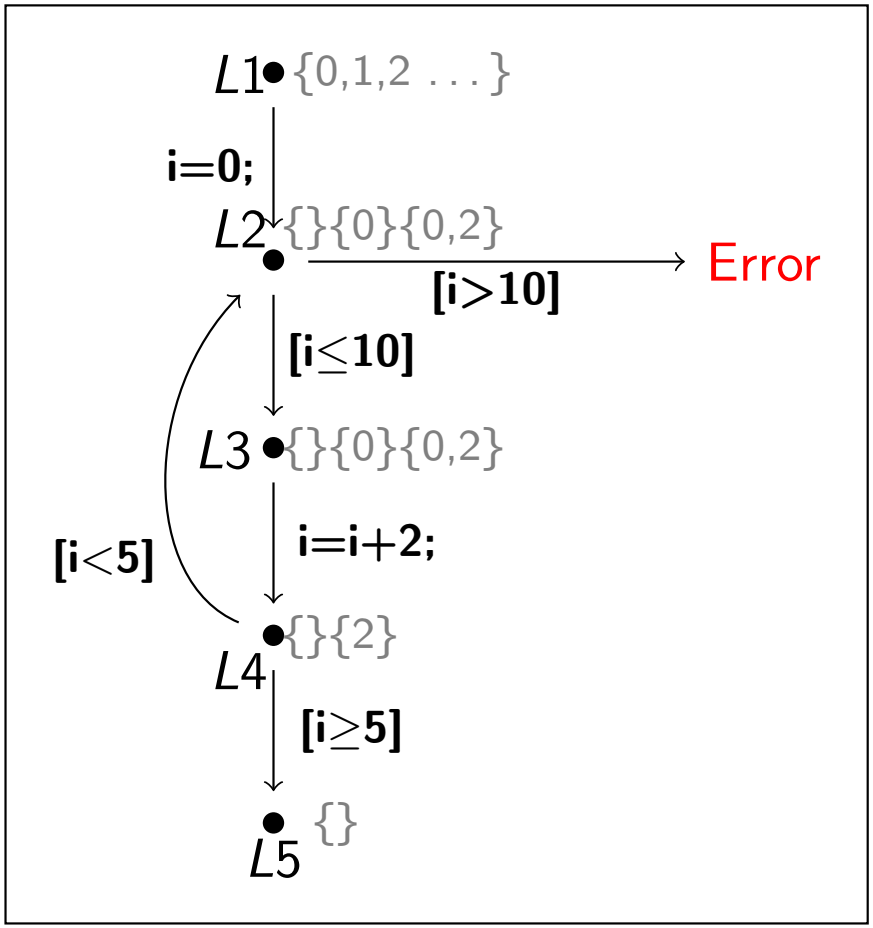
```
int i = 0;
do{
    assert(i <= 10);
    i = i+2;
  }while(i < 5)
```



Static Analysis

Concrete Interpretation:

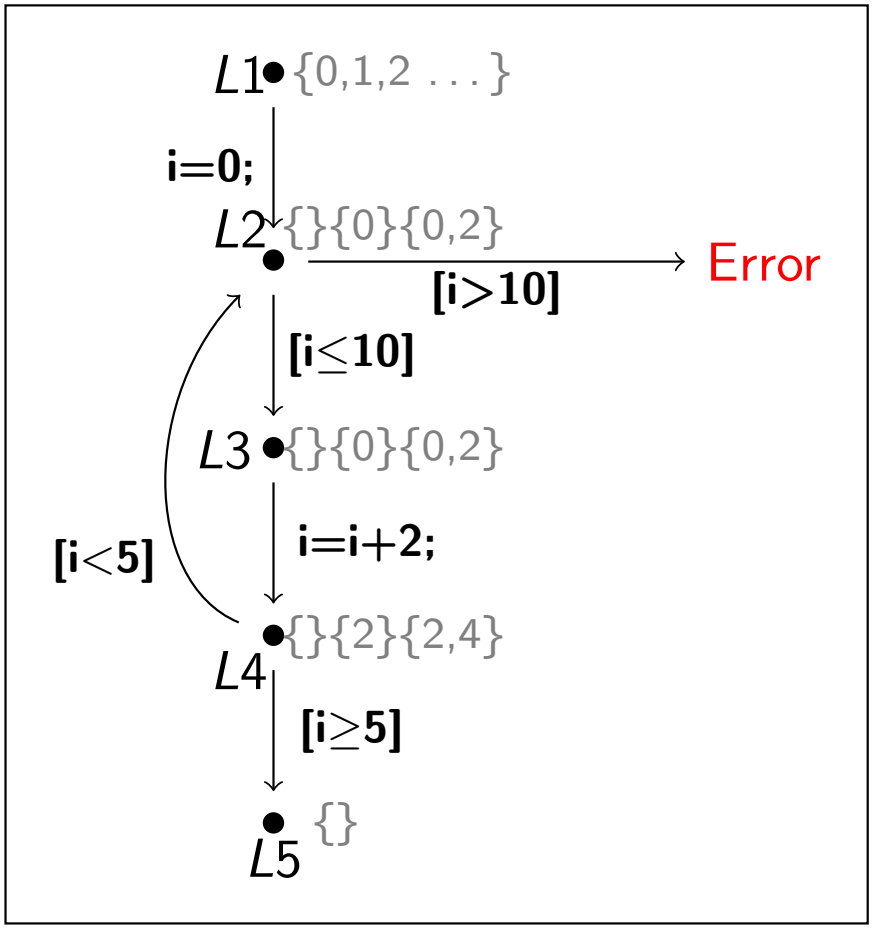
```
int i = 0;
do{
    assert(i <= 10);
    i = i+2;
  }while(i < 5)
```



Static Analysis

Concrete Interpretation:

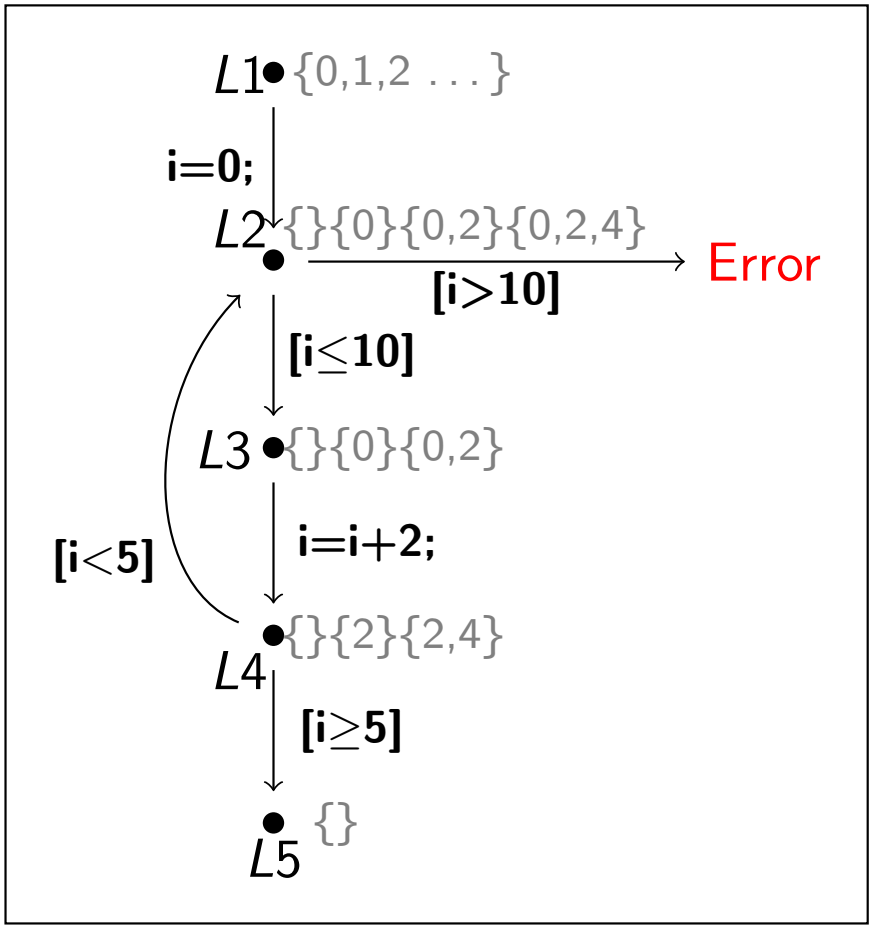
```
int i = 0;
do{
    assert(i <= 10);
    i = i+2;
  }while(i < 5)
```



Static Analysis

Concrete Interpretation:

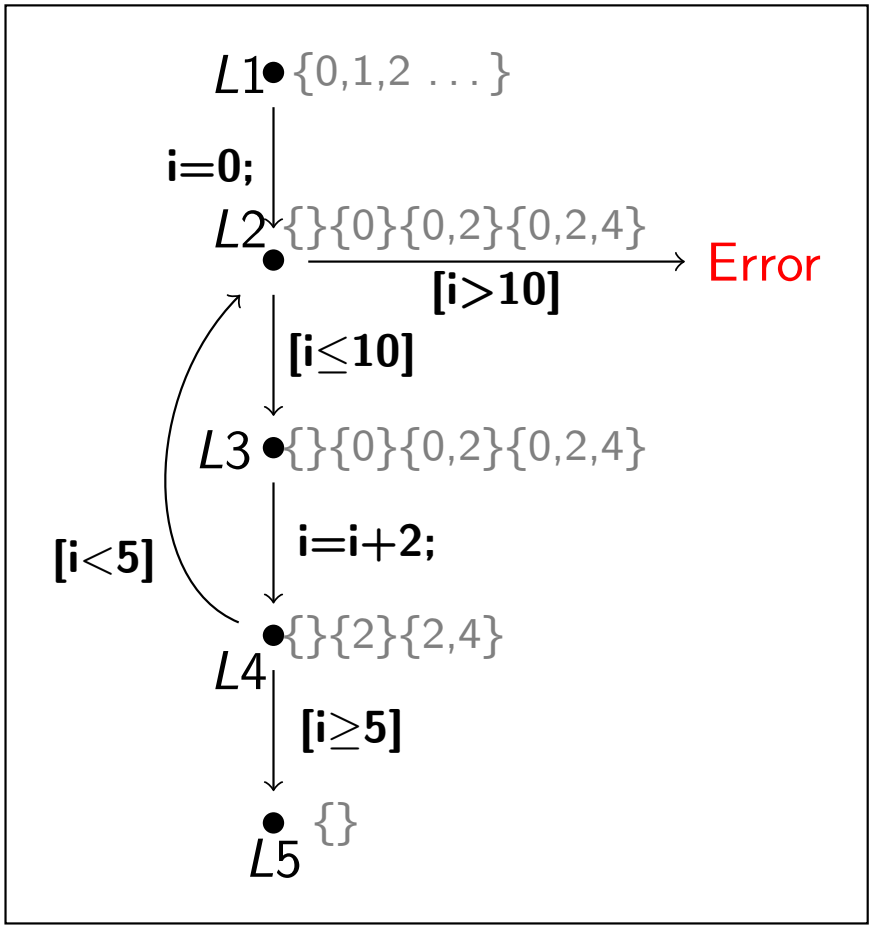
```
int i = 0;
do{
    assert(i <= 10);
    i = i+2;
  }while(i < 5)
```



Static Analysis

Concrete Interpretation:

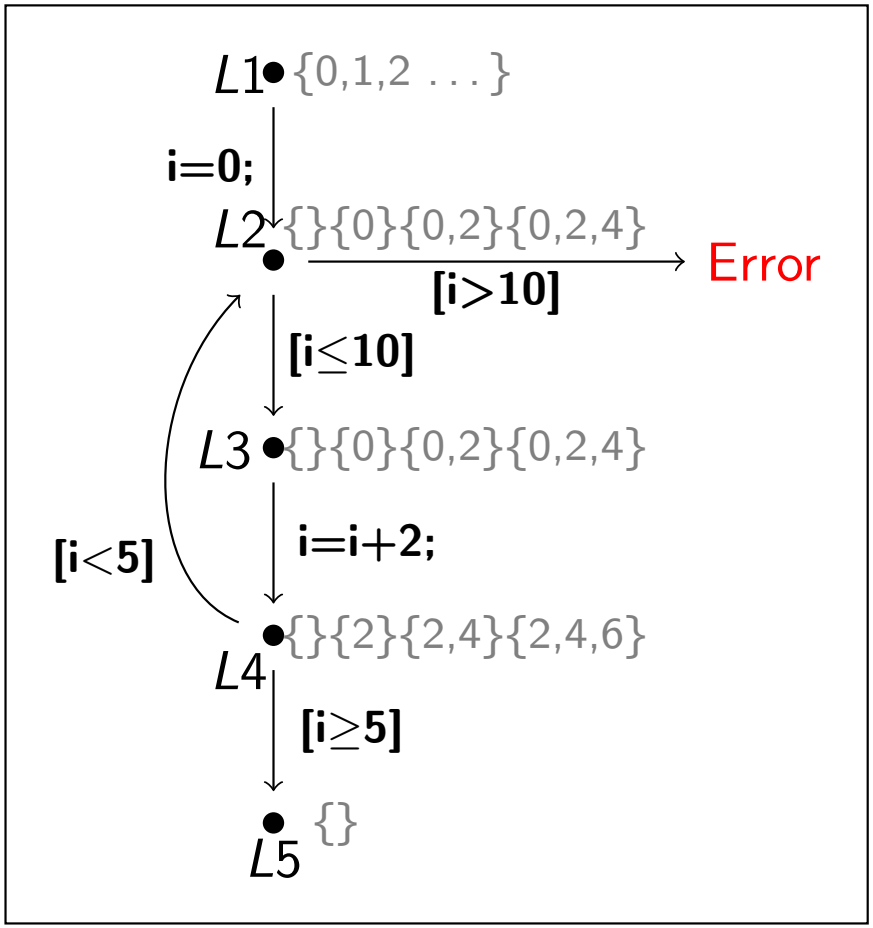
```
int i = 0;
do{
    assert(i <= 10);
    i = i+2;
  }while(i < 5)
```



Static Analysis

Concrete Interpretation:

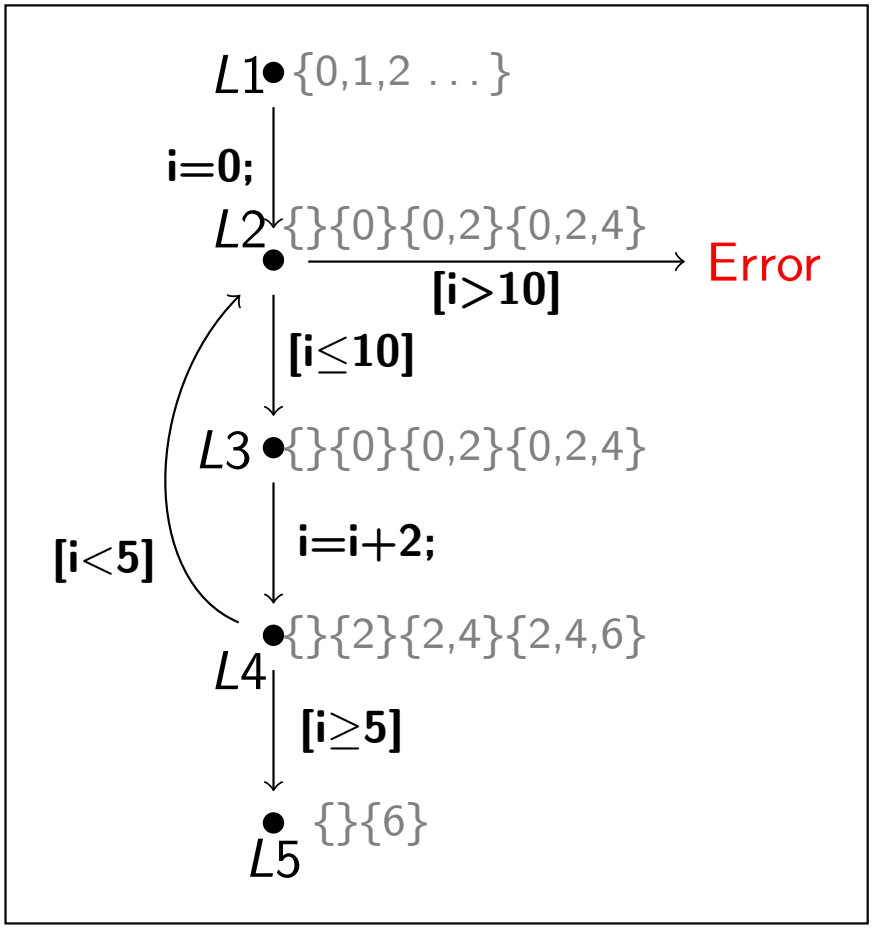
```
int i = 0;
do{
    assert(i <= 10);
    i = i+2;
  }while(i < 5)
```



Static Analysis

Concrete Interpretation:

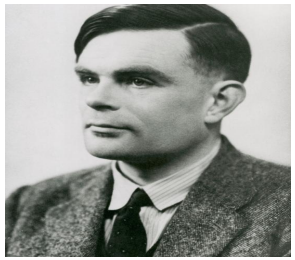
```
int i = 0;
do{
    assert(i <= 10);
    i = i+2;
  }while(i < 5)
```

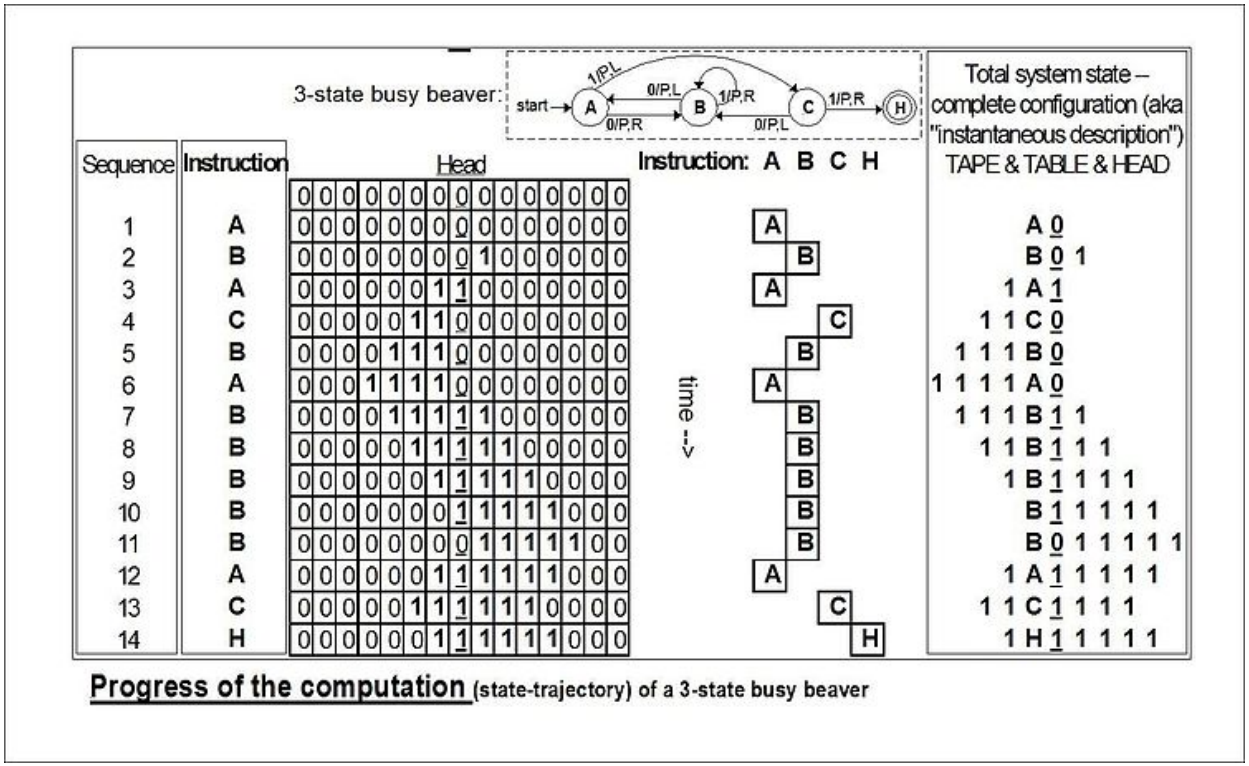


The problem is that this is not always possible :(

Program Verification is a difficult problem

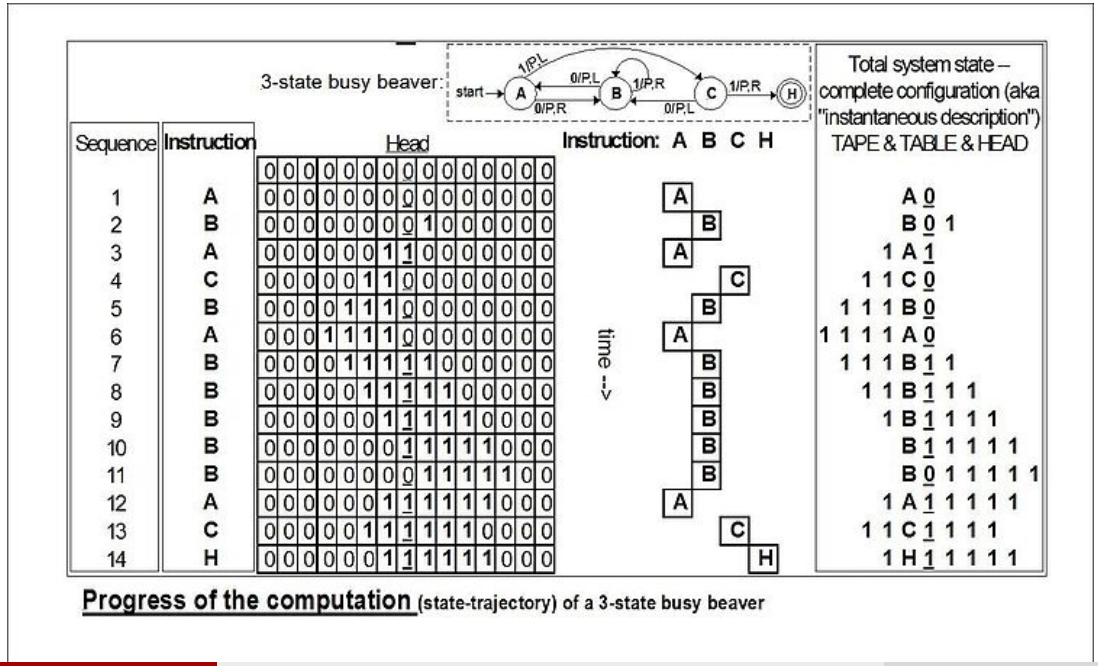
- A Turing machine is a finite state machine augmented with an arbitrary large memory.
- It captures the notion of programs and what can be executed by computers.





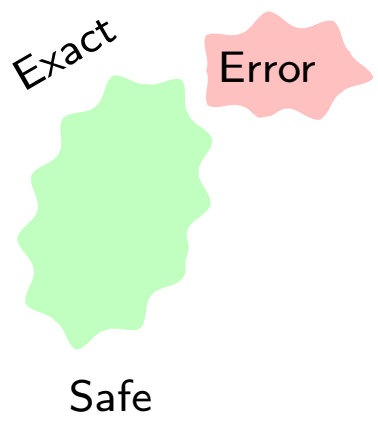
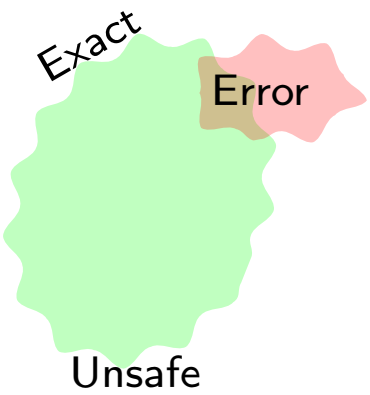
Static Program Analysis is a difficult problem

- Checking whether all possible behaviors are error-free is so hard that if we could write a program that could always do it for arbitrary computer programs then we would always be able to answer whether a Turing machine halts.
- This problem is proven to be undecidable, i.e., there is no algorithm that is guaranteed to terminate and to give an exact answer to the problem.

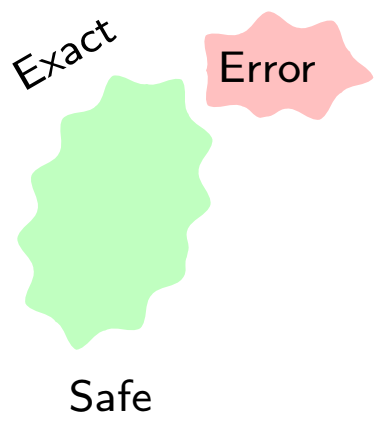
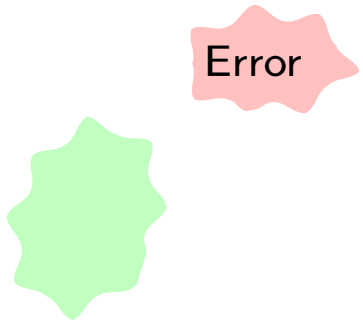
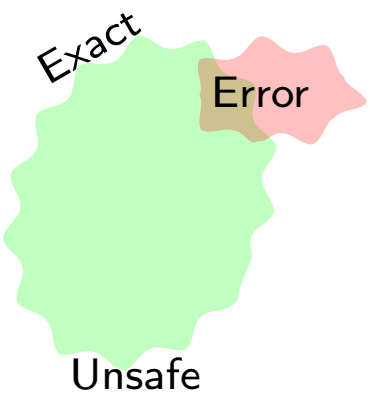


Progress of the computation (state-trajectory) of a 3-state busy beaver

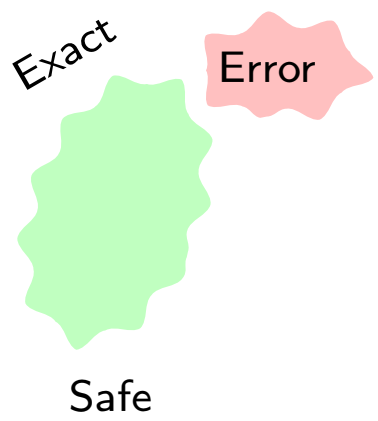
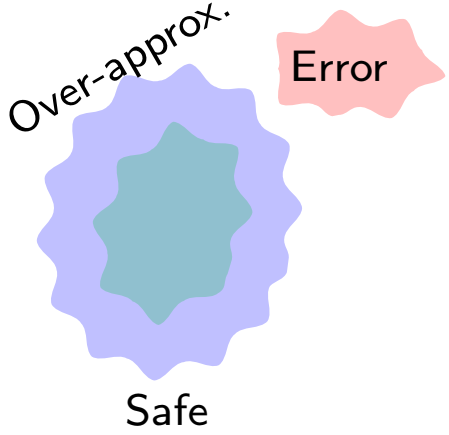
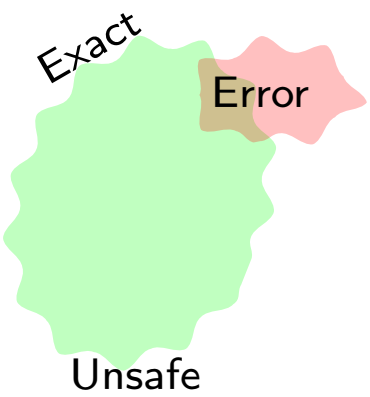
Approximations

Conclusive		Inconclusive
		
		

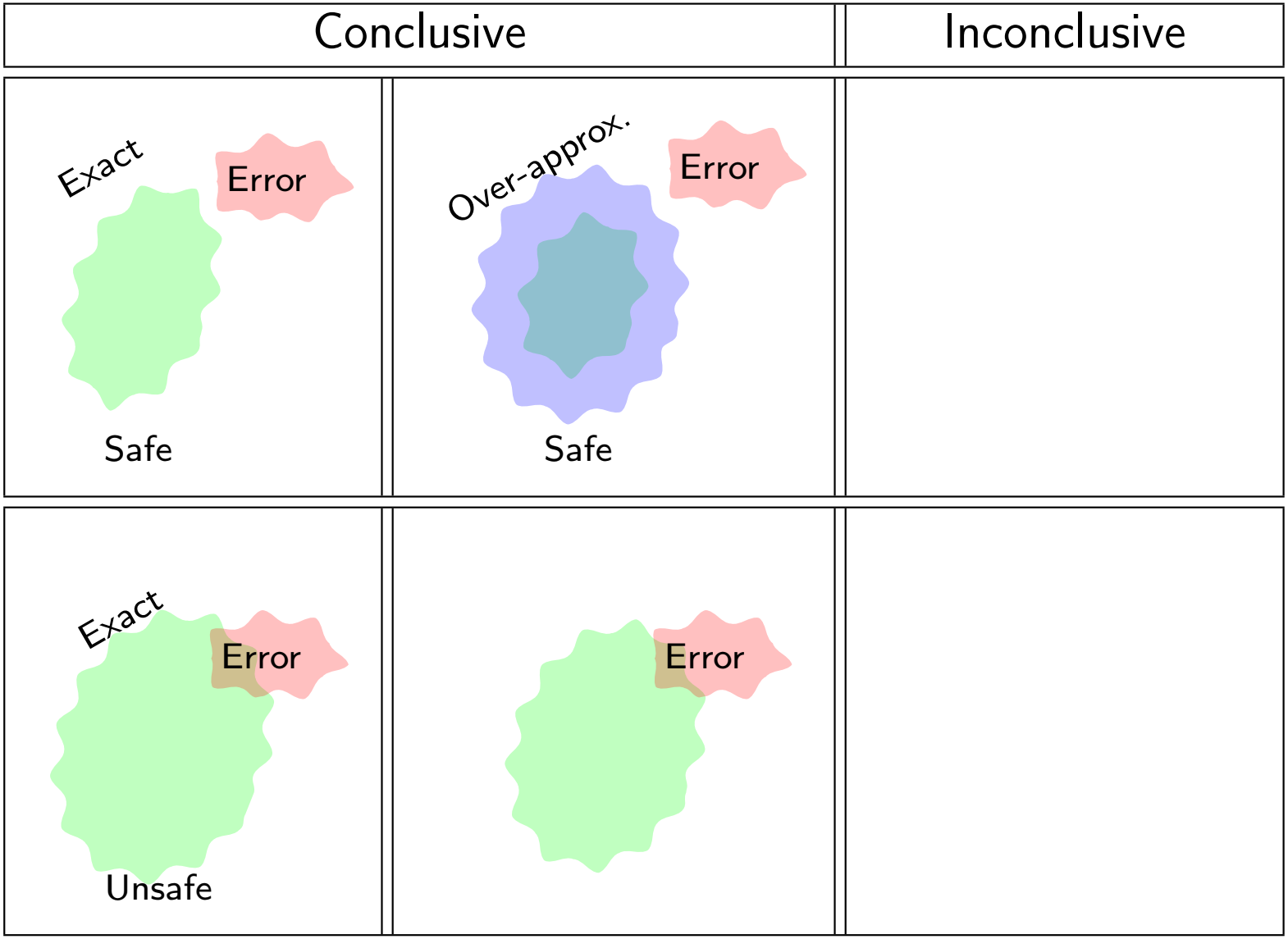
Approximations

Conclusive		Inconclusive
		
		

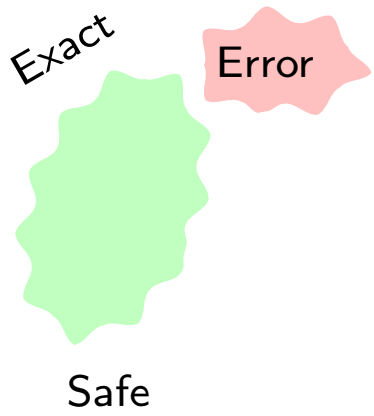
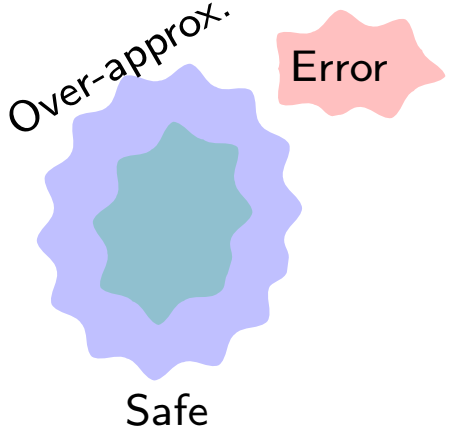
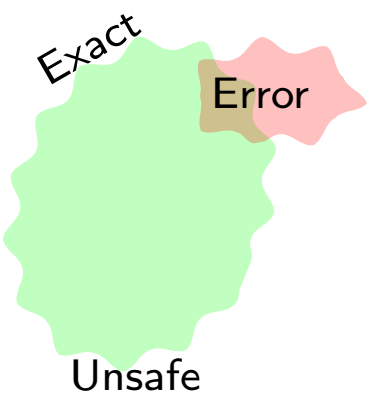
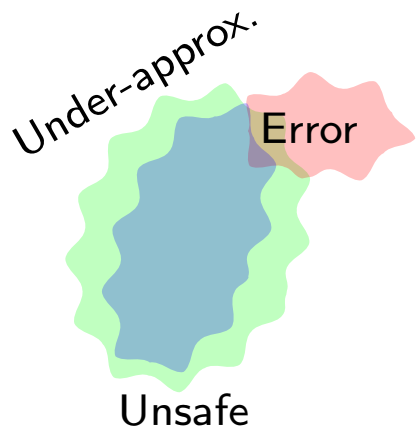
Approximations

Conclusive		Inconclusive
		
		

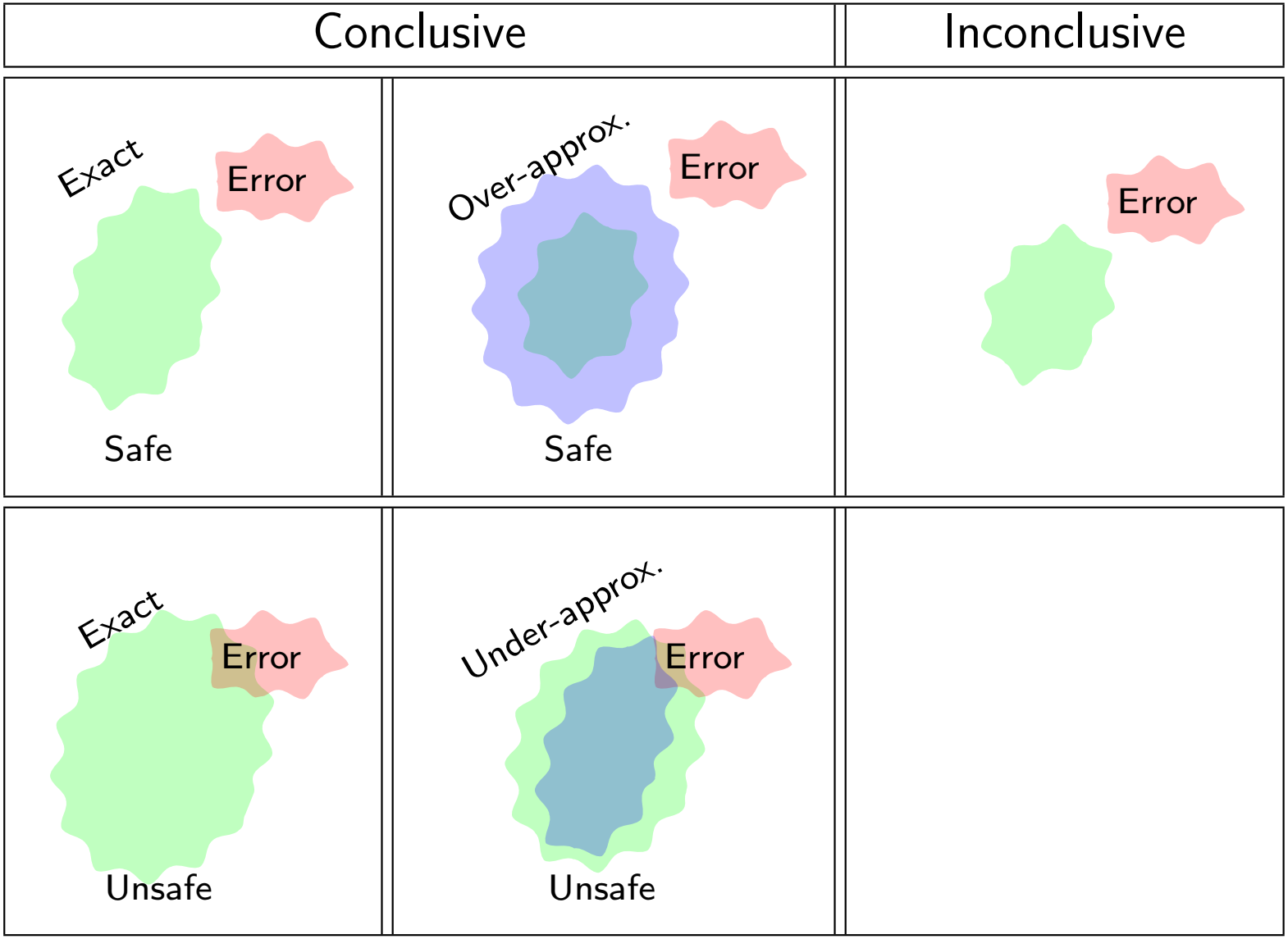
Approximations



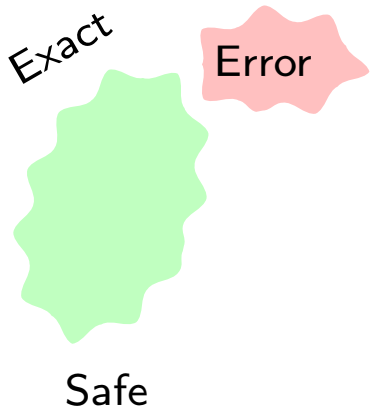
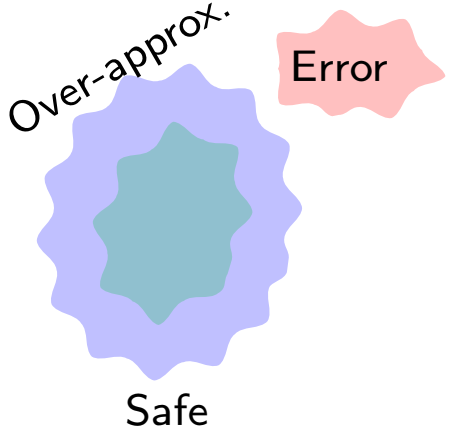
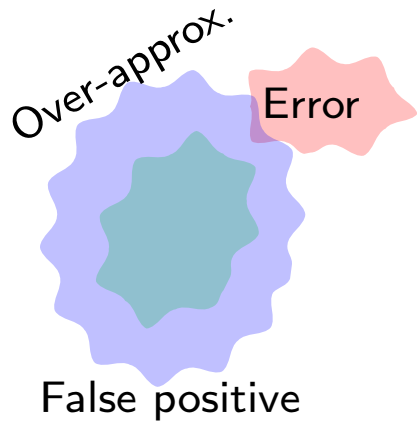
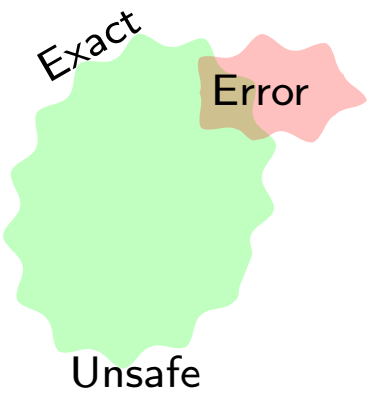
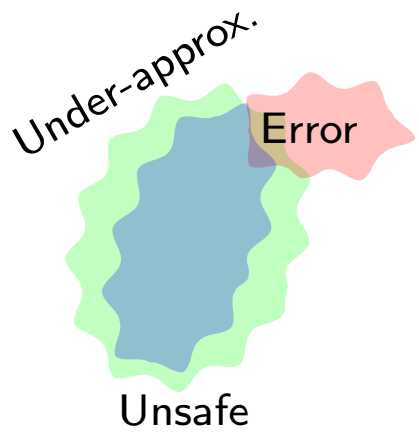
Approximations

Conclusive		Inconclusive
		
		

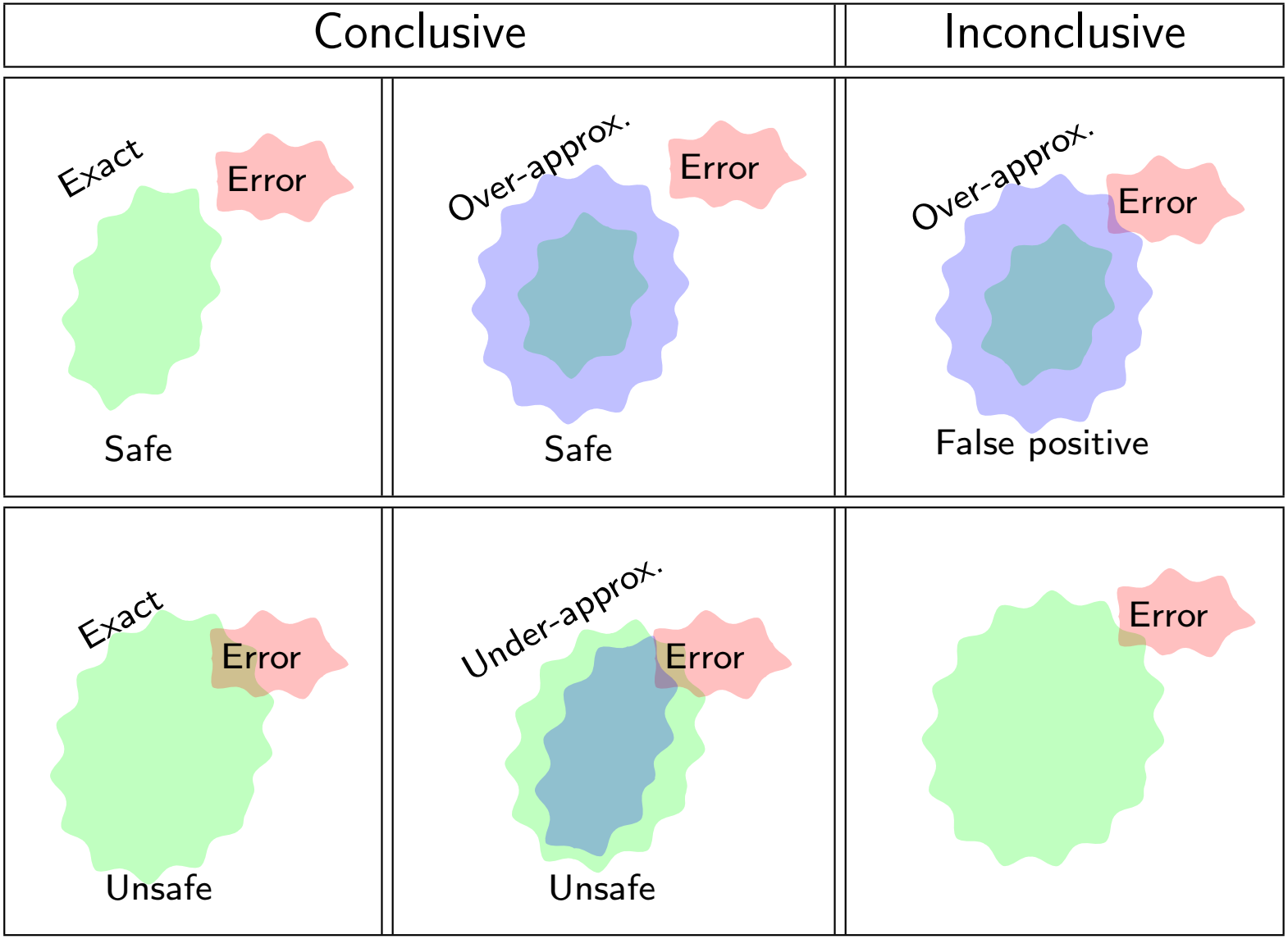
Approximations



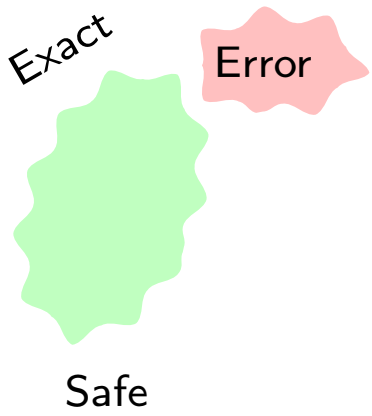
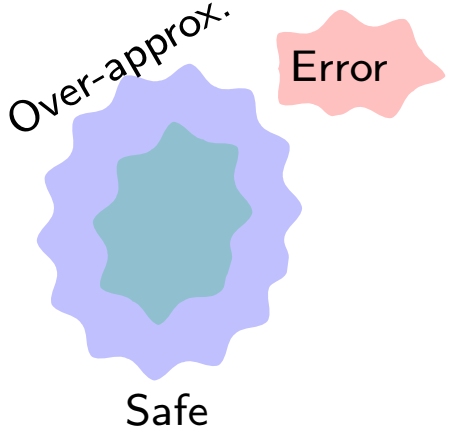
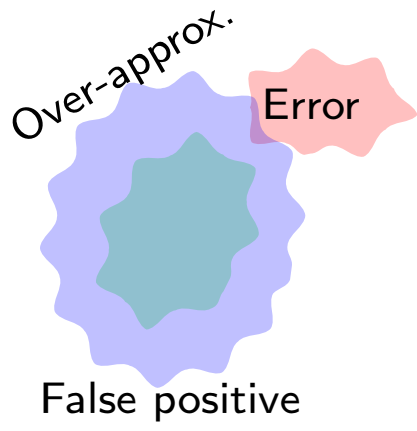
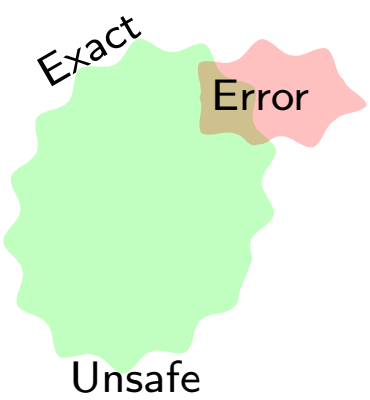
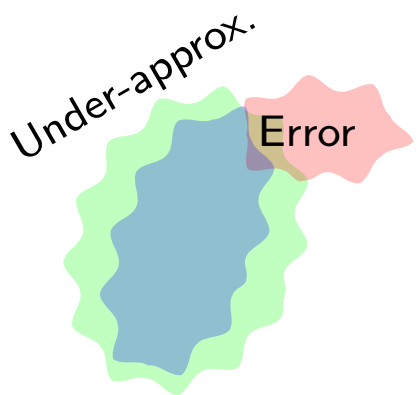
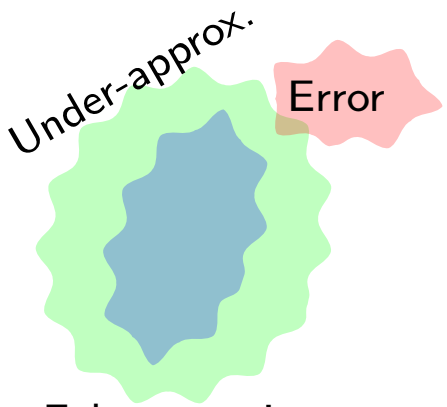
Approximations

Conclusive		Inconclusive
		
		

Approximations



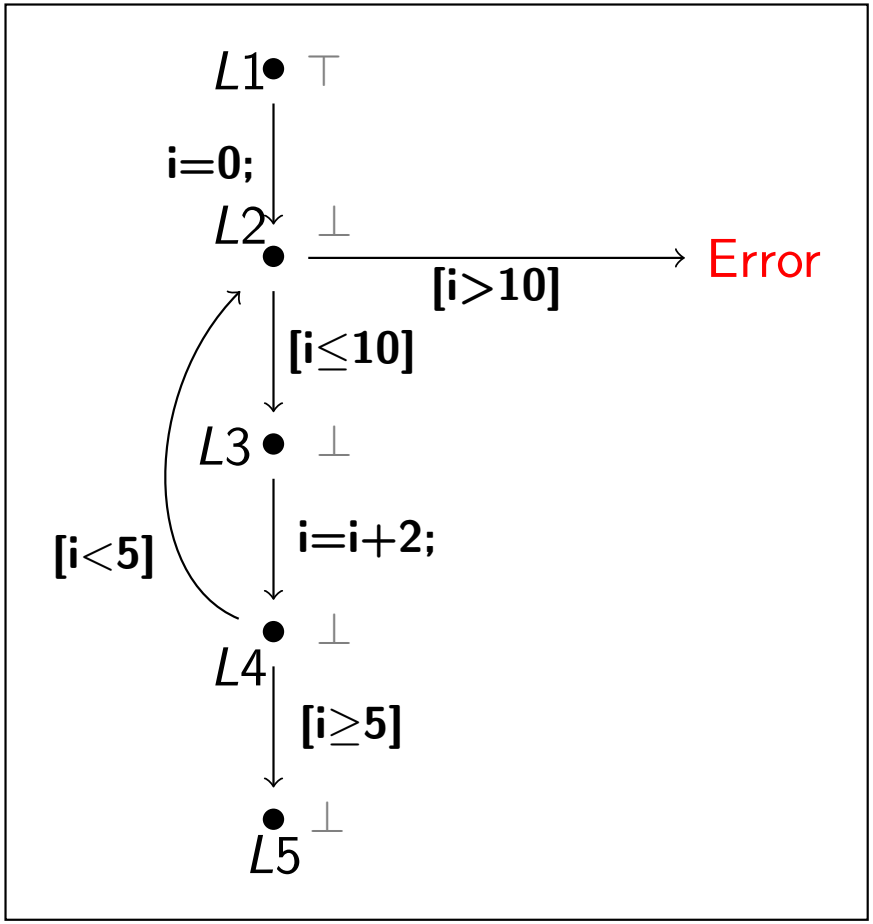
Approximations

Conclusive		Inconclusive
		
		

Software Verification

Abstract Interpretation:

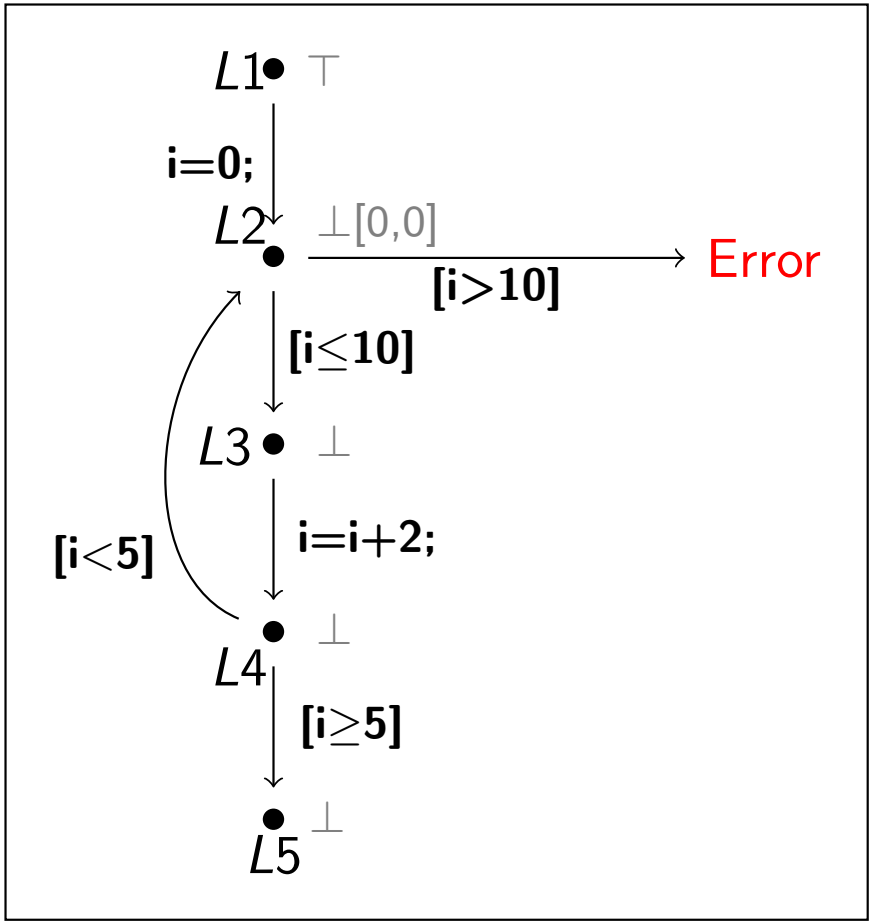
```
int i = 0;
do{
    assert(i <= 10);
    i = i+2;
  }while(i < 5)
```



Software Verification

Abstract Interpretation:

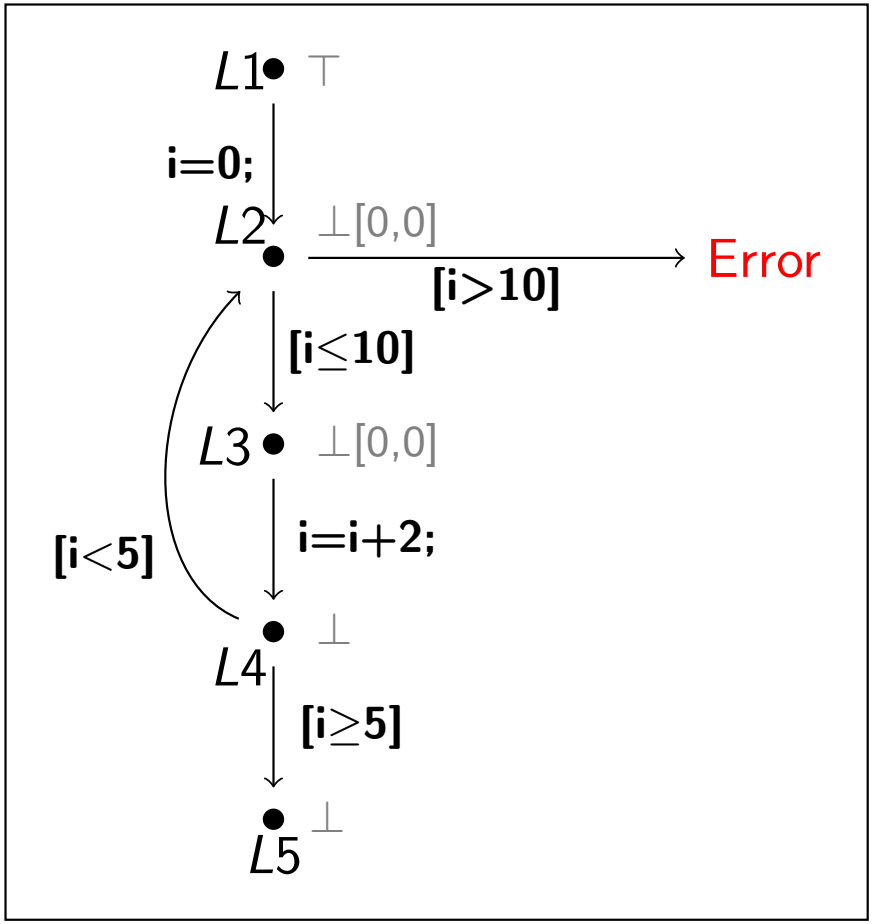
```
int i = 0;
do{
    assert(i <= 10);
    i = i+2;
  }while(i < 5)
```



Software Verification

Abstract Interpretation:

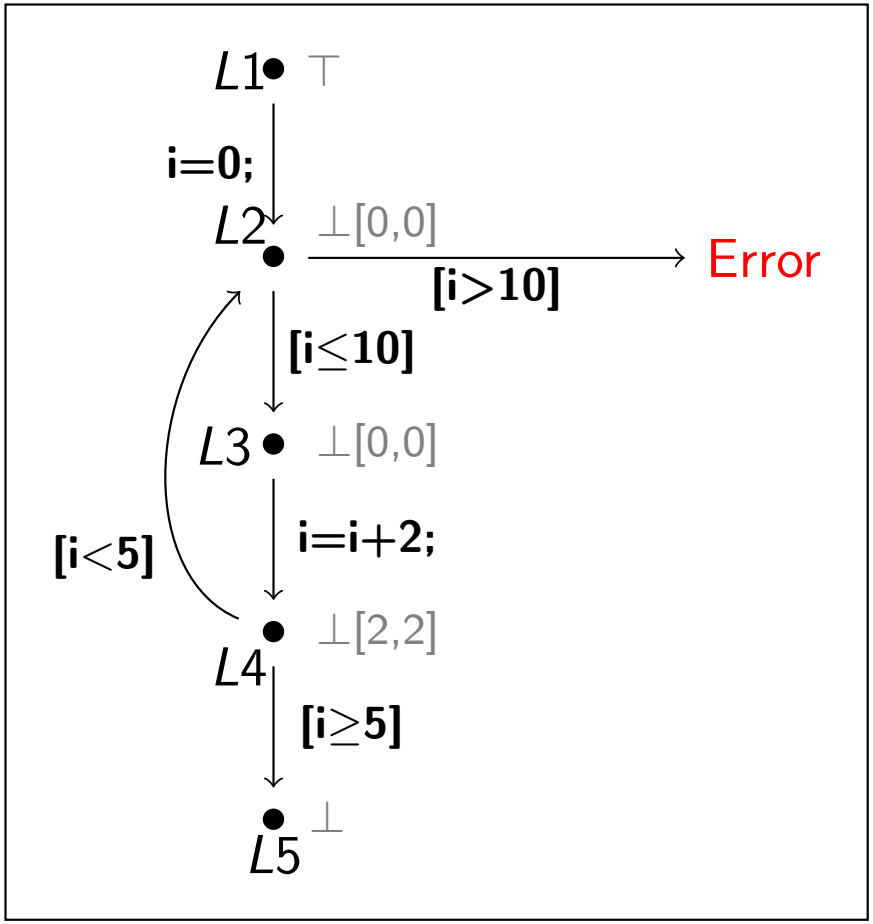
```
int i = 0;
do{
    assert(i <= 10);
    i = i+2;
  }while(i < 5)
```



Software Verification

Abstract Interpretation:

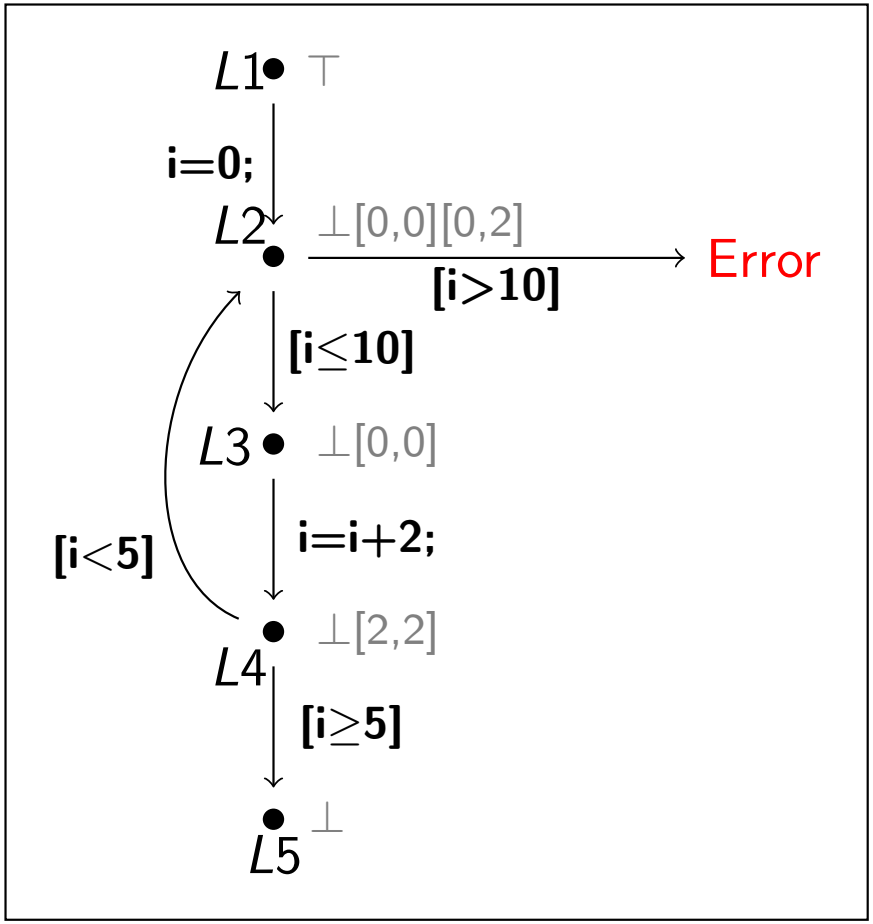
```
int i = 0;
do{
    assert(i <= 10);
    i = i+2;
  }while(i < 5)
```



Software Verification

Abstract Interpretation:

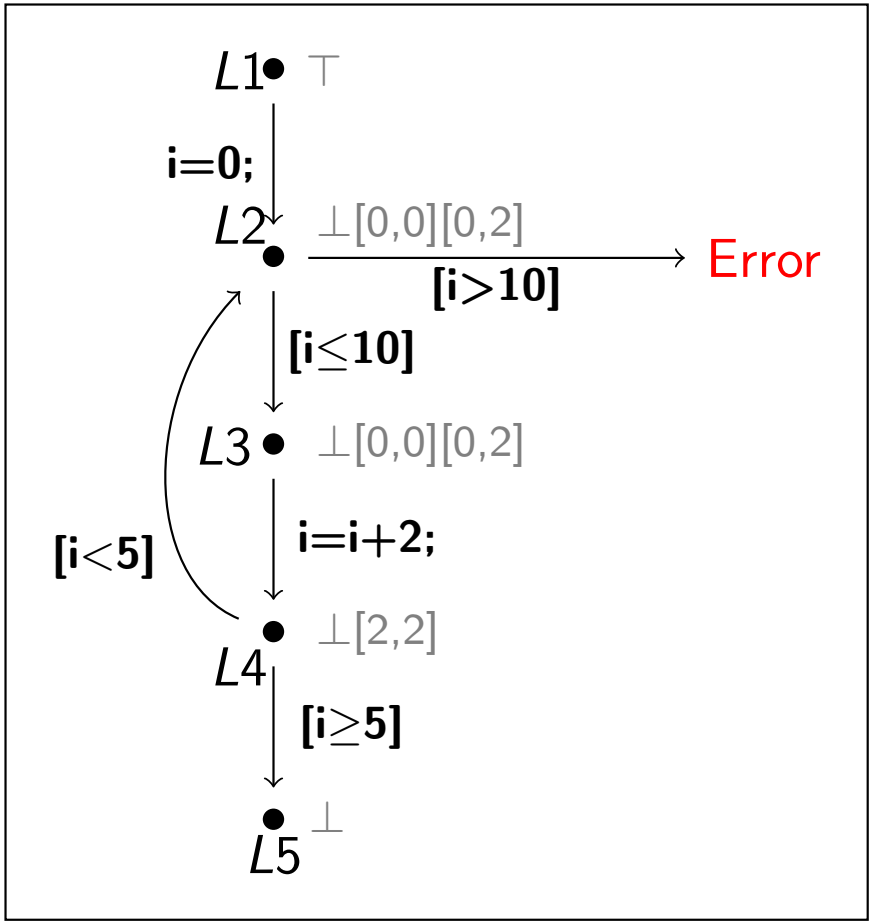
```
int i = 0;
do{
    assert(i <= 10);
    i = i+2;
  }while(i < 5)
```



Software Verification

Abstract Interpretation:

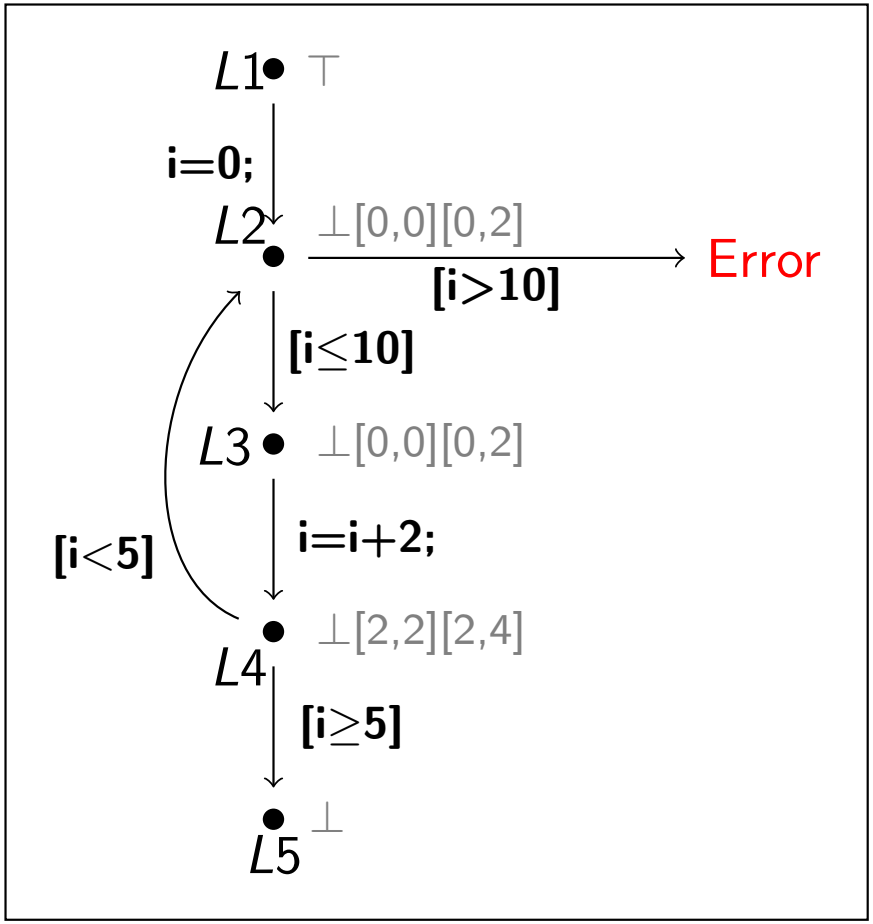
```
int i = 0;
do{
    assert(i <= 10);
    i = i+2;
  }while(i < 5)
```



Software Verification

Abstract Interpretation:

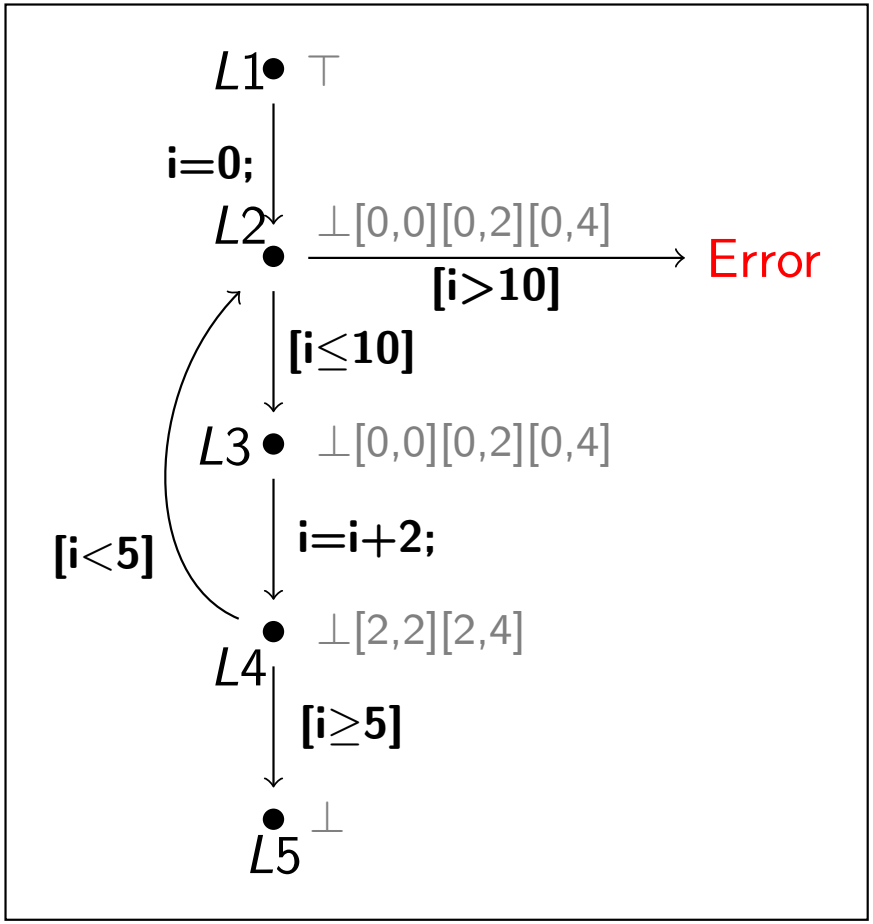
```
int i = 0;
do{
    assert(i <= 10);
    i = i+2;
  }while(i < 5)
```



Software Verification

Abstract Interpretation:

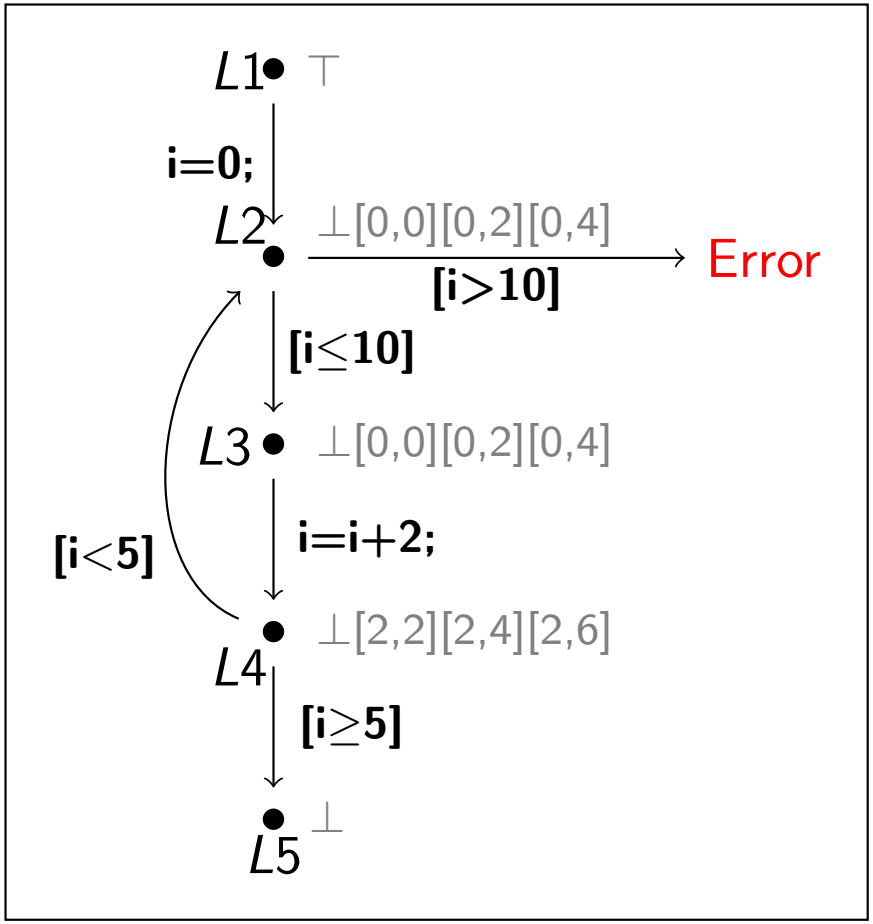
```
int i = 0;
do{
    assert(i <= 10);
    i = i+2;
  }while(i < 5)
```



Software Verification

Abstract Interpretation:

```
int i = 0;
do{
    assert(i <= 10);
    i = i+2;
  }while(i < 5)
```

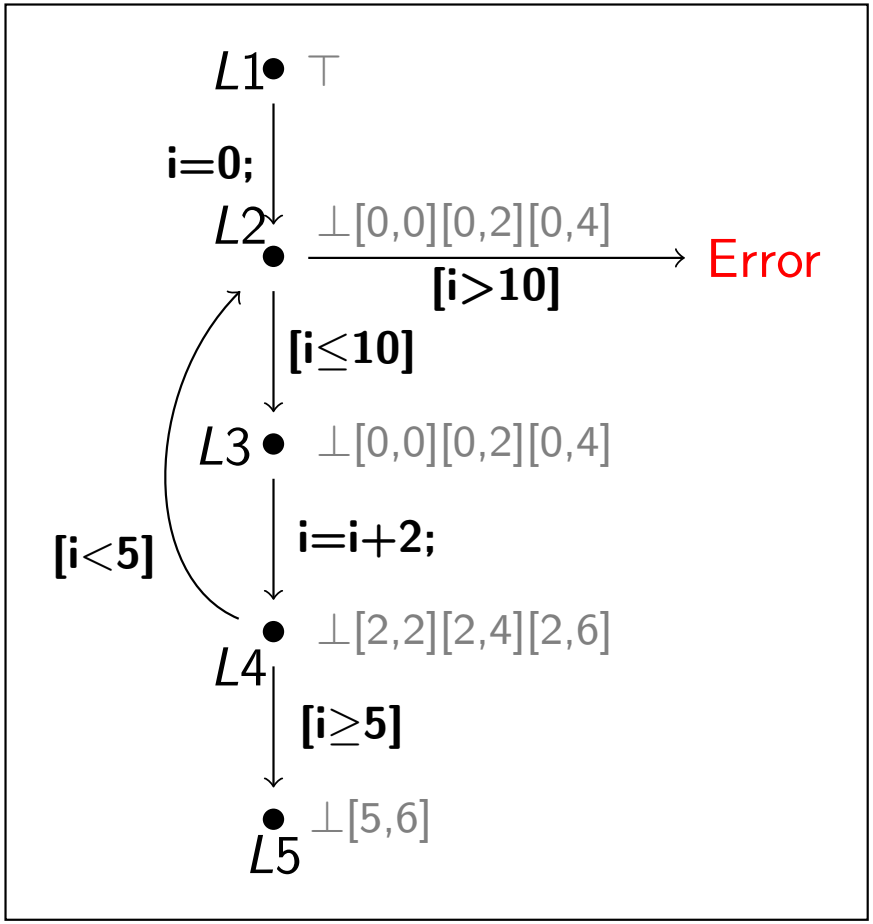


Software Verification

Abstract Interpretation:

```
int i = 0;
do{
    assert(i <= 10);
    i = i+2;
  }while(i < 5)
```

- intervals: [min,max]

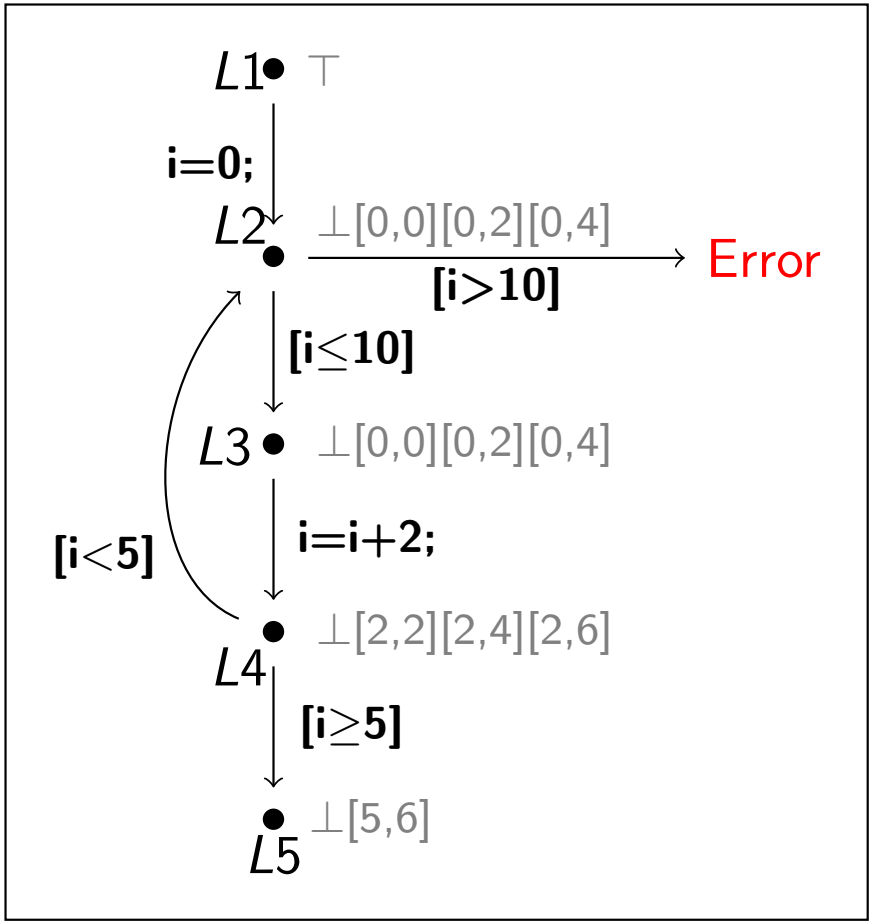


Software Verification

Abstract Interpretation:

```
int i = 0;
do{
    assert(i <= 10);
    i = i+2;
}while(i < 5)
```

- intervals: [min,max]
- DBMs: $x - y \leq c$
- octagons: $\pm x \pm y \leq c$
- polyhedra:
  $a_1 x_1 + a_2 x_2 + \ldots \leq c$
- ...



Outline

String constraints for verification

```
1  function checkMail(mail){
2    var index = mail.indexOf("@");
3    var prefix = mail.substr(0,index);
4    var suffix = email.substr(index + 1);
5
6    return suffix.equals("liu.se") && prefix.length <= 20;
7  }
```

```
1 $query = "SELECT... where mail='\$mail' and pass='$pass'";
2 $result = mysql_query($query);
3
4 # consider the malicious mail adress:  ' OR 1=1 -- @liu.se
```

String constraints for verification

```
1  // Pre = (true)
2  String s= '';
3  // P1 = (s ∈ ϵ)
4  while(*){
5      // P2 = (s = u · v ∧ u ∈ a* ∧ v ∈ b* ∧ |u| = |v|)
6      s= 'a' + s + 'b';
7  }
8  // P3 = P2
9  assert(!s.contains('ba') && (s.length() % 2) == 0);
```

vc_1 : $post(Pre, \mathtt{s} = \text{""}) \implies P_1$
$(s \in \epsilon) \implies (s \in \epsilon)$

vc_2 : $P_1 \implies P_2$
$(s \in \epsilon) \implies (s = u \cdot v \wedge u \in a^* \wedge v \in b^* \wedge |u| = |v|)$

vc_3 : $post(P_2, \mathtt{s} = \text{"a"} \cdot \mathtt{s} \cdot \text{"b"}) \implies P_2$
$(s = u \cdot v \wedge u \in a^* \wedge v \in b^* \wedge |u| = |v|) \wedge (s' = a \cdot s \cdot b) \implies (s' = u \cdot v \wedge u \in a^* \wedge v \in b^* \wedge |u| = |v|)$

vc_4 : $P_2 \implies P_3$
$(s = u \cdot v \wedge u \in a^* \wedge v \in b^* \wedge |u| = |v|) \implies (s = u \cdot v \wedge u \in a^* \wedge v \in b^* \wedge |u| = |v|)$

vc_5 : $post(P_3, \mathtt{assume(s.contains("ba")||!(s.length()\%2 == 0))}) \implies false$
$(s = u \cdot v \wedge u \in a^* \wedge v \in b^* \wedge |u| = |v|) \wedge ((s = r \cdot b \cdot a \cdot t) \vee (|s| = x + x)) \implies false$