SMT and Its Application in **Software Verification**

Yu-Fang Chen
IIS, Academia Sinica

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

Assertion in C

```
int main(){
    int x;
    scanf("%d", &x);
    assert(x > 10);  }
```

- Useful tool for debugging.
  - Can be used to describe pre- and post-conditions of a function.
  - A program terminates immediately, if it reaches a **violated** assertion.

```
[yfc@FM3 ~]$ gcc test.c
[yfc@FM3 ~]$ ./a.out
10
a.out: test.c:9: main: Assertion `x > 10' failed.
Aborted
```

Assertion in C

```
int main(){
    int x;
    scanf("%d", &x);
    while(x<10){
        x++;
    }
    assert(x > 0);
}
```

- Will this assertion be violated?

Assertion in C

```
int main(){
    int x;
    scanf("%d", &x);
    while(x<10){
        x--;
    }
    assert(x > 0);
}
```

- Will this assertion be violated?

Assertion in C

```
int main(){
    int x;
    scanf("%d", &x);
    while(x<4324358){
            x--;
    }
    assert(x > 4324358);  }
```

- Will this assertion be violated?

Assertion in C

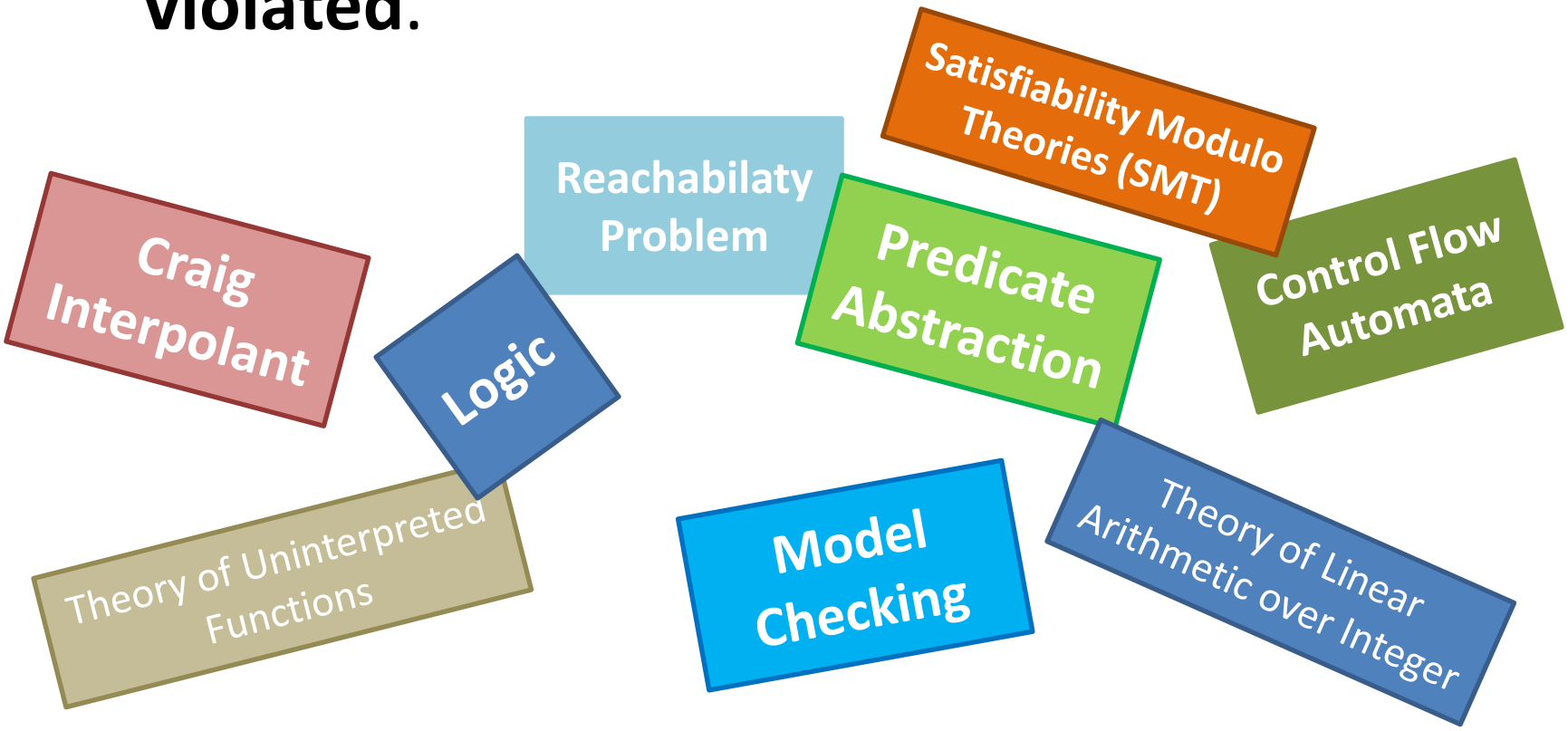
- One more example:

```
void A(bool h, bool g){
  h := !g;
  g=B(g,h);
  g=B(g,h);
  assert(g);
}
```

```
void B(bool a1,bool a2){
  if (a1)
    return B(a2,a1);
  else
    return a2;
}
```

The Problem We are Going to Solve

- Given a program with assertion, we want to **automatically detect if the assertion may be violated**.



Part I: Logic and Program Verification

First-Order Logic: A Quick Review

- Logical Symbols
  - Propositional connectives: $\vee$, $\wedge$, $\neg$, $\rightarrow$, $\leftrightarrow$
  - Variables: v1, v2, . . .
  - Quantifiers: $\forall$, $\exists$
- Non-logical Symbols
  - Functions: +, -, *, suc, …
  - Predicates: $\leq$, = , …
  - Constant symbols: 0, 1, null, …
- Example
  - 3*v2 + 5* v1 $\leq$ 54

Why This is Relevant to Software Verification?

For example:

- Given an integer program without loop and function call.
- The **assertion checking** problem can be reduced to **satisfibility** problem of a trace formula*

```
int  main(){
    int x; scanf("%d", &x);
    if(x<10)  x= x -1;
    assert(x != 9);
}
```

$$(x_0 < 10 \wedge x_1 = x_0 - 1 \wedge x_1 = 9) \wedge (x_0 \geq 10 \wedge x_0 = 9)$$

The assertion may be violated **iff** The first order formula is satisfible

Note *: a FOL formula under theory of linear integer arithmetic.

First order logic Theories

- A first order theory consists of
  - Variables
  - Logical symbols: $\wedge$ $\vee$ $\neg$ $\forall$ $\exists$ `(' `)'
  - **Non-logical Symbols (signature) $\sum$: Constants, predicate and function symbols**
  - **The meanings of the non-logical symbols.**

Examples

- $\sum$ = {0,1, '+', '='}
  - '0','1' are constant symbols
  - '+' is a binary function symbol
  - '=' is a binary predicate symbol

- An example of a $\sum$-formula:

$$\exists x.\ x + 0 = 1$$

Is ϕ valid?

Structures

- The most common way of specifying the meaning of symbols is to specify a **structure**

- Recall that $\phi = \exists x.\ x + 0 = 1$
- Consider the structure S:
  - Domain: $\mathcal{N}_0$
  - Interpretation of the non-logical symbols :
    - '0' and '1' are mapped to 0 and 1 in $\mathcal{N}_0$
    - '=' $\mapsto$ = (equality)
    - '+' $\mapsto$ * (multiplication)
- Now, is ϕ true in S ?

Short Summary

- A theory defines
  - the signature $\sum$ (the set of non-logical symbols) and
  - the interpretations that we can give them.

Theories through axioms

- The number of sentences that are necessary for defining a theory may be large or infinite.
- Instead, it is common to define a theory through a set of axioms.
- The theory is defined by these axioms and everything that can be inferred from them by a sound inference system.

Example 1

- Let $\sum$ = {'='}
  - An example $\sum$-formula is $((x = y) \wedge \neg (y = z)) \rightarrow \neg(x = z)$
- We would now like to define a $\sum$-theory T that will limit the interpretation of '=' to equality.
- We will do so with the equality axioms:
  - $\forall x.\ x = x$ (reflexivity)
  - $\forall x,y.\ x = y \rightarrow y = x$ (symmetry)
  - $\forall x,y,z.\ x = y \wedge y = z \rightarrow x = z$ (transitivity)
- Every assignment that satisfies these axioms also satisfies ϕ above.
- Hence ϕ is T-valid.

Example 2

- Let $\sum$ = {'<'}
- Consider the $\sum$-formula ϕ: $\forall x\ \exists y.\ y < x$
- Consider the theory T:
  - $\forall x,y,z.\ x < y \wedge y < z \rightarrow x < z$ (transitivity)
  - $\forall x,y.\ x < y \rightarrow \neg(y < x)$ (anti-symmetry)

Quantifier-free Subset

- In **Software Verification**, we will largely restrict ourselves to formulas without quantifiers (∀, ∃)—mainly for efficiency reason.
- This is called the quantifier-free fragment of first-order logic.

Some Useful Theories in Software Verification

- **Equality (with uninterpreted functions)**
- **Linear arithmetic (over $\mathbb{Q}$ or $\mathbb{Z}$)**
  - Peano Arithmetic, Presburgh Arithmetic
- Difference logic (over $\mathbb{Q}$ or $\mathbb{Z}$)
- Finite-precision bit-vectors
  - integer or floating-point
- **Arrays**
- Misc.: strings, lists, sets, …

Theory of Equality and Uninterpreted Functions (EUF)

- Signature:
  - Constants and Function symbols: f, g, etc. In principle, all possible symbols but "=" and those used for variables.
  - Predicates symbol: "="
- equality axioms:
  - $\forall x.\ x = x$ (reflexivity)
  - $\forall x,y.\ x = y \rightarrow y = x$ (symmetry)
  - $\forall x,y,z.\ x = y \wedge y = z \rightarrow x = z$ (transitivity)
- In addition, we need *congruence*: the function symbols map identical arguments to identical values, i.e., $x = y \Rightarrow f(x) = f(y)$

Example EUF Formula

$$(x = y) \wedge (y = z) \wedge (f(x) \neq f(z))$$

Transitivity:

$$(x = y) \wedge (y = z) \Rightarrow (x = z)$$

Congruence:

$$(x = z) \Rightarrow (f(x) = f(z))$$

Equivalence Checking of Program Fragments

```
int fun1(int y) {
    int x, z;
    z = y;
    y = x;
    x = z;

   return sq(x);
}

int fun2(int y) {
     return sq(y);
}
```

formula ϕ is satisfiable iff
the programs are non-equivalent

$z0 = y0 \wedge y1 = x0 \wedge x1 = z0 \wedge ret1 = sq(x1)$
$\wedge$
$ret2 = sq(y0)$
$\wedge$
$ret1 \neq ret2$

A Small Practice:

```
int f(int y) {
    int x, z;
    x = myFunc(y);
    x = myFunc(x);
    z = myFunc(myFunc(y));

   assert (x==z);
}
```

Write a formula ϕ such that

formula ϕ is satisfible $\leftrightarrow$
the assertion can be violated

Solution:

First Order Peano Arithmetic

constant function predicate

- $\sum = \{0, 1, \text{‘+’}, \text{‘*’}, \text{‘=’}\}$
- Domain: Natural numbers

Validity is

Undecidable!

- Axioms (“semantics”):
  1. $\forall x : (0 \neq x + 1)$
  2. $\forall x : \forall y : (x \neq y) \rightarrow (x + 1 \neq y + 1)$

  \+
  3. Induction
  4. $\forall x : x + 0 = x$

  *
  5. $\forall x : \forall y : (x + y) + 1 = x + (y + 1)$
  6. $\forall x : x * 0 = 0$
  7. $\forall x \forall y : x * (y + 1) = x * y + x$

These axioms define the semantics of ‘+’

First Order Presburger Arithmetic

constant function predicate

- $\sum = \{0, 1, \text{`+'}, \text{`*'}, \text{`='}\}$
- Domain: Natural numbers

Validity is

decidable!

- Axioms ("semantics"):
  1. $\forall x : (0 \neq x + 1)$
  2. $\forall x : \forall y : (x \neq y) \rightarrow (x + 1 \neq y + 1)$

+
  3. Induction
  4. $\forall x : x + 0 = x$
  5. $\forall x : \forall y : (x + y) + 1 = x + (y + 1)$

*
  6. $\forall x : x * 0 = 0$
  7. $\forall x \forall y : x * (y + 1) = x * y + x$

These axioms define the semantics of '+'

Note that $3*v2 + 5* v1 \leq 54$ is a Presburger Formula

Examples in Software Verification

- Array Bound Checking:

```
void f() {
   int x[10];
   x[0]=1;
   for(int i=1; i<11; i++){
      assert(0<i<10);
      x[i]=x[i-1]+5;
      ……
   }
}
```

$i0=1 \wedge i1=2 \wedge i2=3 \wedge i3=4 \wedge \ldots \wedge i8=9 \wedge$
$0<i0<10 \wedge 0<i1<10 \wedge \ldots \wedge 0<i8<10$
is satisfible iff
the assertion may be violated

Theory of Arrays

- Two interpreted functions: select and store
  - select(A,i) Read from array A at index i
  - store(A,i,d) Write d to array A at index i
- Two main axioms:
  - select(store(A,i,d), i) = d
  - select(store(A,i,d), j) = select(A,j) for $i \neq j$
- Extentionality axiom:
  - $\forall$i. select(A,i) = select(B,i) ➔ (A = B)

Combining Theories

- **Satisfiability Modulo Theories** (**SMT**) **problem** is a decision problem for logical formulae with respect to combinations of different first order theories.
- For example: Uninterpreted Function + Linear Integer Arithmetic

$$1 \leq x \wedge x \leq 2 \ \wedge \ f(x) \neq f(1) \wedge f(x) \neq f(2)$$

Linear Integer Arithmetic (LIA)	Uninterpreted Functions(UF)

- How to Combine Theory Solvers?

  A Classical Algorithm: The Nelson-Oppen Method . Google "Nelson-Oppen" and you can get tons of slides explaining it.

Combining Theories

- **Satisfiability Modulo Theories** (**SMT**) **problem** is a decision problem for logical formulae with respect to combinations of different first order theories.
- For example: Uninterpreted Function + Linear Integer Arithmetic

$$1 \leq x \wedge x \leq 2 \ \wedge \ f(x) \neq f(1) \wedge f(x) \neq f(2)$$

Linear Integer Arithmetic (LIA)	Uninterpreted Functions(UF)

- How to Combine Theory Solvers?

In this lecture, we will not talk about the algorithm for solving SMT problem. Instead ,we will show you tools (SMT solvers) that does the job.

gle
ng it.

What you have learned so far?

- Assertions in C
- First Order Theories related to Verification
  - Equality and Uninterpreted Functions

    $f(a,b)=a \Rightarrow f(f(a, b), b)=a$
  - Linear Integer Arithmetic

    $x+5>y-7 \wedge 3y < x \wedge y> 0$
  - Arrays

    $i>j \rightarrow select(A, i) > select(A, j)$ [array A is sorted]
- The SMT Problem

Tools: Theorem Provers of FOL

- An example: Princess (by Philipp Rummer):
  - http://www.philipp.ruemmer.org/princess.shtml (JRE is required)

- Automatically check validity of a FOL formula modulo linear integer arithmetic.

$(x_0 < 10 \wedge x_1 = x_0 - 1 \wedge x_1 = 9)$
$\vee$
$(x_0 \geq 10 \wedge x_0 = 9)$

Note: f is unsatisfible iff !f is valid

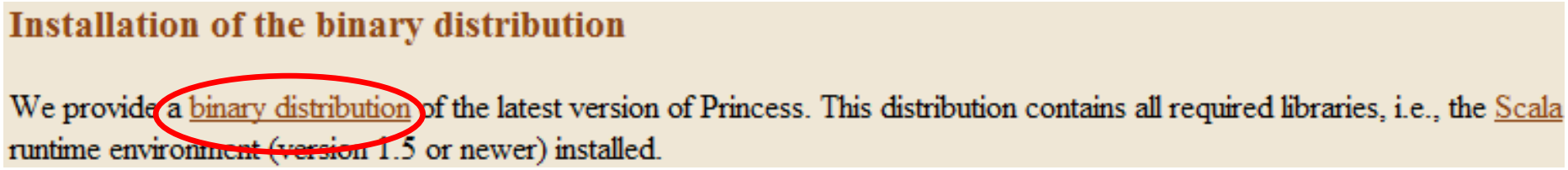
Princess Input:

```
\functions {
 int x0, x1;
}
\problem {
  \forall int x0, x1;
   !( (x0 < 10 & x1 = x0-1 & x1 =9) |  (x0 >= 10 & x0 =9) )
}
```

Tools: Theorem Provers of FOL

- An example: Princess (by Philipp Rummer):
  - http://www.philipp.ruemmer.org/princess.shtml (JRE is required)

Download the binary tarball from the web-site



Installation of the binary distribution

We provide a binary distribution of the latest version of Princess. This distribution contains all required libraries, i.e., the Scala runtime environment (version 1.5 or newer) installed.

Install JRE 1.5 or above

Run Princess:

```
set BASE=.
set CLASSPATH=%CLASSPATH%;%BASE%\dist\princess-all.jar
java ap.DialogMain
```

About Princess

- Theorem prover (SMT solver) for **first-order logic** + **linear integer arithmetic**
- Supported features:
  - Uninterpreted **predicates, functions, quantifiers**
  - Theory of **arrays** (using functions + axioms)
  - **Craig interpolation**

Declaration of functions/constants

```
\functions {
  /* Declare constants and functions occurring in the problem
   * The keyword "\partial" can be used to define functions without totality axiom,
   * "\relational" can be used to define "functions" w/o functionality axiom. */

  int x, y, z, ar;

  \partial int select(int, int);

  \partial int store(int, int, int);

  int f(int);
}

[...]
```

Constant Variables

Partial functions

Total function

Declaration of predicates

```
[...]

\predicates {
  /* Declare predicates occurring in the problem */

  p; q;
  r(int, int);
}

[...]
```

Nullary predicates (propositional variables)

Predicates/relations

Formulae/problems

```
[...]

\problem {
  /* Problem to be proven. */

  \forall int ar, ind, val;
    select(store(ar, ind, val), ind) = val
->
  \forall int ar, ind1, ind2, val;
    (ind1 != ind2 ->
     select(store(ar, ind1, val), ind2) = select(ar, ind2))

->
  x + 1 = 5
->
  select(store(ar, x, y), 4) = y

}
```

Array axioms

Implied formula

Syntax of formulae

Syntax	Meaning
A & B	Conjunction
A | B	Disjunction
!A	Negation
A -> B	Implication
A <-> B	Equivalence
\forall int x, y, z; A \exists int x, y, z; A	Quantifiers
\part[p0] A	Labelled formula
s = t, s != t s <= t, s < t	(Dis)equations, inequalities
p(s, t, ...)	Application of predicates
…, -2, -1, 0, 1, 2, ...	Integer literals (constants)
s + t, s * t	Sums, products (only linear expressions are allowed)
x, y, z	Variables, constants
f(s, t, ...)	Application of functions

Some Practice

```
int  main(){
    int x; scanf("%d", &x);
    if(x<10)  x= x -1;
    assert(x != 9);
}
```

$(x_0 < 10 \wedge x_1 = x_0 - 1 \wedge x_1 = 9)$
$\vee$
$(x_0 \geq 10 \wedge x_0 = 9)$

The assertion may be violated  The first order formula is satisfible

Some Practice

```
int  main(){
    int x; scanf("%d", &x);
    if(x<10)  x= x -1;
    assert(x != 9);
}
```

$(x_0 < 10 \wedge x_1 = x_0 - 1 \wedge x_1 = 9)$
$\vee$
$(x_0 \geq 10 \wedge x_0 = 9)$

The assertion may be violated **iff** The first order formula is satisfible

ex1.pri

```
\functions {
  int x0, x1;
}
\problem {
  \forall int x0, x1;
   !((x0<10 & x1=x0-1 & x1=9) & (x0>=10 & x0=9))
}
```

Some Practice

```
int fun1(int y) {
    int x, z;
    z = y;
    y = x;
    x = z;

   return sq(x);
}

int fun2(int y) {
     return sq(y);
}
```

Some Practice

```
int fun1(int y) {
    int x, z;
    z = y;
    y = x;
    x = z;

   return sq(x);
}

int fun2(int y) {
     return sq(y);
}
```

formula ϕ is satisfiable iff
the programs are non-equivalent

Some Practice

```
int fun1(int y) {
    int x, z;
    z = y;
    y = x;
    x = z;

   return sq(x);
}

int fun2(int y) {
     return sq(y);
}
```

formula ϕ is satisfiable iff
the programs are non-equivalent

$$z0 = y0 \wedge y1 = x0 \wedge x1 = z0 \wedge ret1 = sq(x1)$$
$$\wedge$$
$$ret2 = sq(y0)$$
$$\wedge$$
$$ret1 \neq ret2$$

Some Practice

```
int fun1(int y) {
    int x, z;
    z = y;
    y = x;
    x = z;

   return sq(x);
}

int fun2(int y) {
     return sq(y);
}
```

formula ϕ is satisfiable iff
the programs are non-equivalent

$$z0 = y0 \wedge y1 = x0 \wedge x1 = z0 \wedge ret1 = sq(x1)$$
$$\wedge$$
$$ret2 = sq(y0)$$
$$\wedge$$
$$ret1 \neq ret2$$

ex2.pri

```
\functions {
  int z0, y0, y1, x0, x1, ret1,ret2;
  int sq(int);
}
\problem {
  \forall int z0, y0, y1, x0, x1, ret1,ret2; !(
    z0=y0 & y1 = x0 & x1 =z0 & ret1 =sq(x1)
& ret2=sq(y0) & ret1 != ret2
)}
```

Some Practice

```
int f(int y) {
    int x, z;
    x = myFunc(y);
    x = myFunc(x);
    z = myFunc(myFunc(y));
    assert (x==z);
}
```

Some Practice

```
int f(int y) {
   int x, z;
   x = myFunc(y);
   x = myFunc(x);
   z = myFunc(myFunc(y));
   assert (x==z);
}
```

Write a formula ϕ such that

formula ϕ is satisfible $\leftrightarrow$
the assertion can be violated

Some Practice

```
int f(int y) {
   int x, z;
   x = myFunc(y);
   x = myFunc(x);
   z = myFunc(myFunc(y));
   assert (x==z);
}
```

Write a formula ϕ such that

formula ϕ is satisfible $\leftrightarrow$
the assertion can be violated

$x0=myFunc(y0) \wedge x1=myFunc(x0) \wedge$
$z0 = myFunc(myFunc(y0)) \wedge x1!=z0$

Some Practice

```
int f(int y) {
   int x, z;
   x = myFunc(y);
   x = myFunc(x);
   z = myFunc(myFunc(y));
   assert (x==z);
}
```

Write a formula ϕ such that

formula ϕ is satisfible $\leftrightarrow$
the assertion can be violated

$x0=myFunc(y0) \wedge x1=myFunc(x0) \wedge$
$z0 = myFunc(myFunc(y0)) \wedge x1!=z0$

ex3.pri

```
\functions {
  int z0, y0, x0, x1;
  int myFunc(int);
}
\problem {
  \forall int z0, y0, x0, x1; !(
   x0=myFunc(y0) & x1=myFunc(x0) & z0 =myFunc(myFunc(y0)) & x1 != z0 )}
```

Homework

1. Define a small program verification problem.
2. Reduce the problem to a satisfibility problem of FOL
3. Solve it using Princess (send me your pri file via email: yfc@iis.sinica.edu.tw)

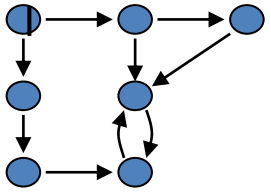
Samples can be found the previous 3 pages.

Score of you homework
(60%) correctness
(40%) creativity

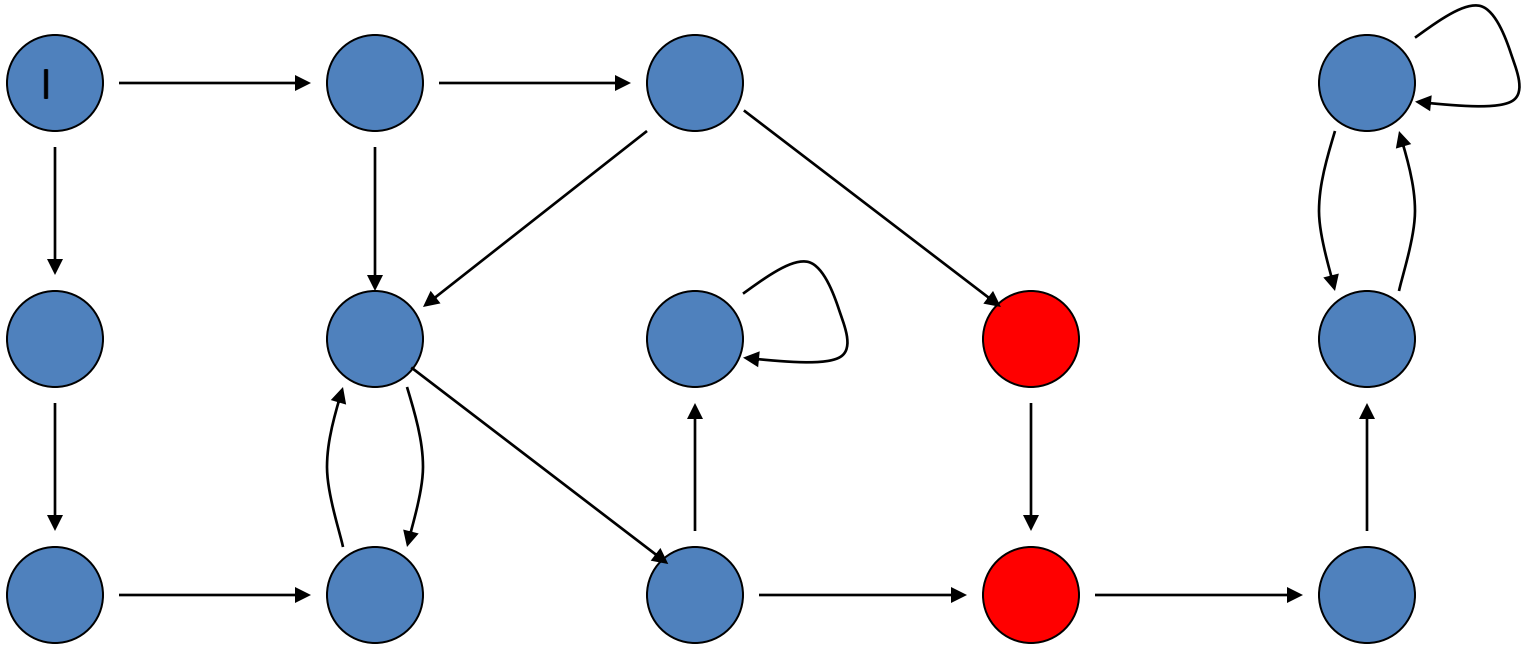
Part II: Software Model Checking

Model Checking

- Given a:
  - Finite state transition system M
  - A property p
    - Reachability (Simplest form)
    - CTL, LTL, CTL* (Prof. Farn Wang)
    - Automata (Prof. Yih-Kuen Tsay)
- The model checking problem:
  - Does M satisfy p?



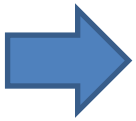
Model Checking (safety)



= bad state

Assertion Checking as Reachability

```
int main(){
    int x;
    scanf("%d", &x);
    while(x<10){
        x++;
    }
    assert(x > 0);
}
```

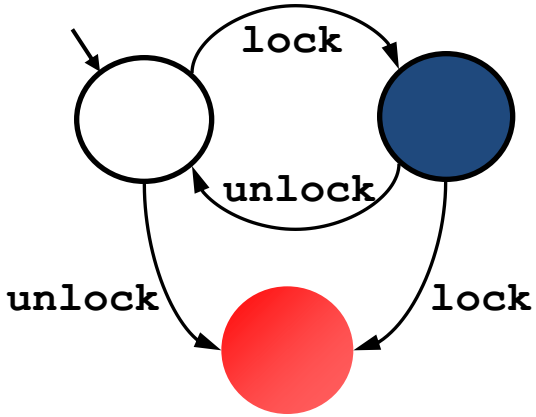


```
int main(){
    int x;
    scanf("%d", &x);
    while(x<10){
        x++;
    }
    if(! (x > 0) ){
        printf("ERROR"):
    }
}
```

We will show you how to translate a program to a state transition system later

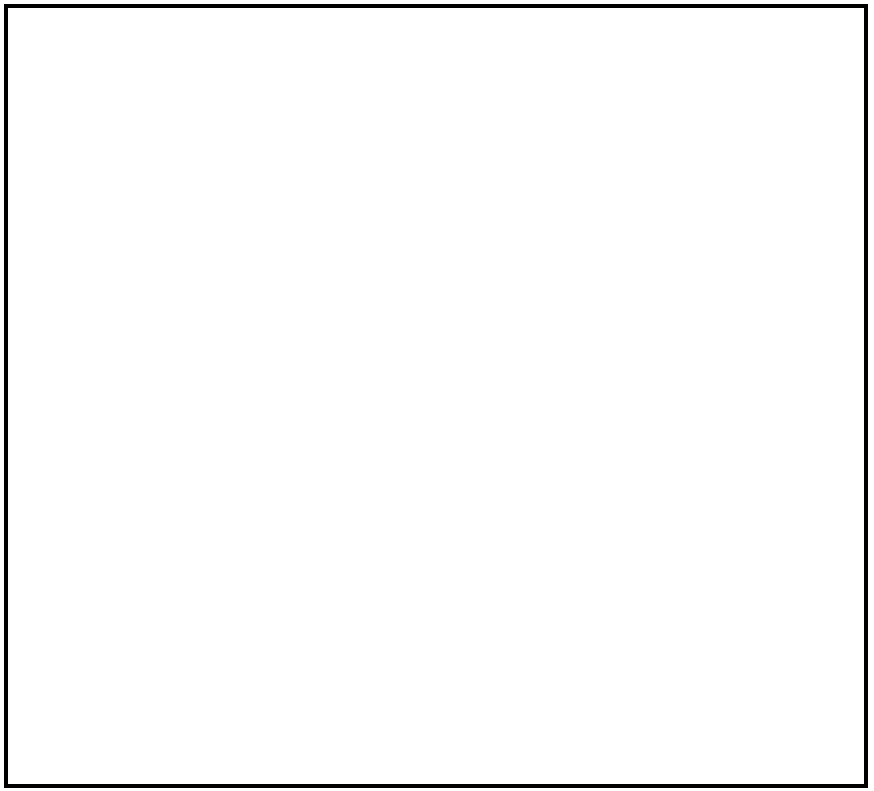
Example

```
Example ( ) {
1: do{
       assert(!lock)
       lock=true;
       old = new;
       q = q->next;
2:     if (q != NULL){
3:        q->data = new;
          assert(lock)
          lock=false;
          new ++;
       }
4: } while(new != old);
5:  assert(lock)
    lock=false;
    return;
}
```



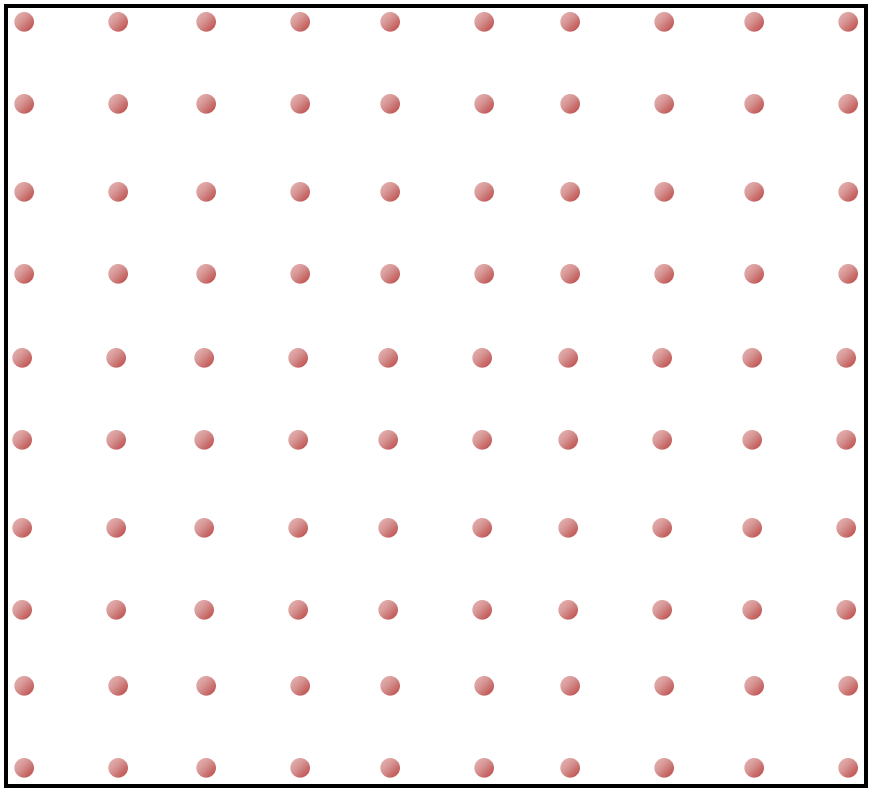
Fill up a linked list with increasing values

What a program *really* is…



```
Example ( ) {
1: do{
       assert(!lock)
       lock=true;
       old = new;
       q = q->next;
2:     if (q != NULL){
3:        q->data = new;
          assert(lock)
          lock=false;
          new ++;
       }
4: } while(new != old);
5:  assert(lock);
    lock=false;
    return;}
```

What a program *really* is…



State

pc ↦ 3

lock ↦ ●

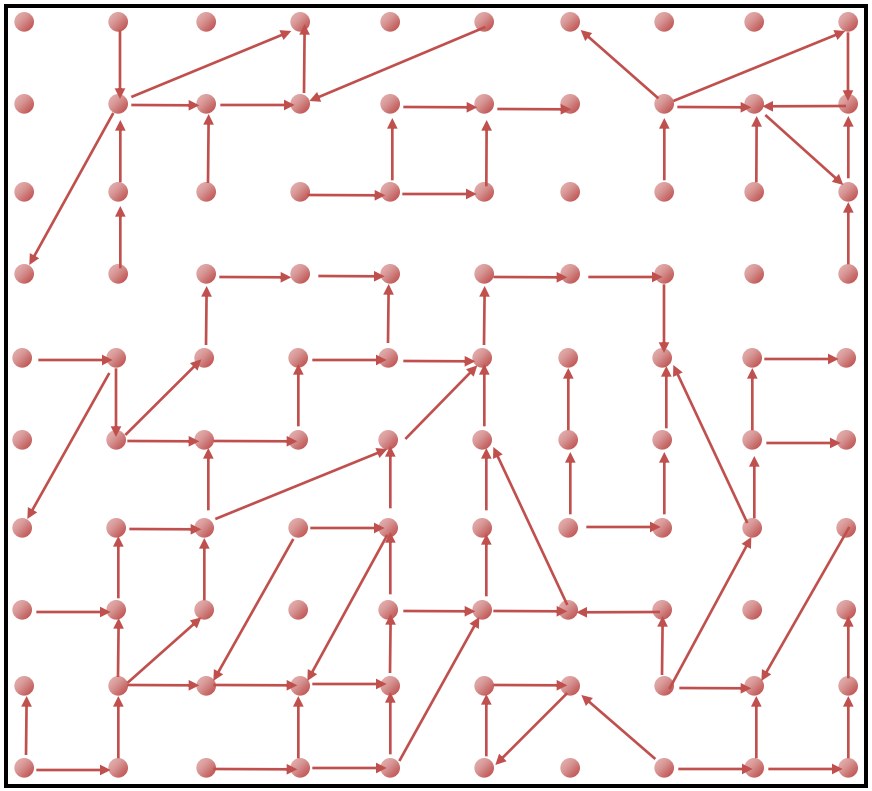
old ↦ 5

new ↦ 5

q ↦ 0x133a

```
Example ( ) {
1: do{
       assert(!lock)
       lock=true;
       old = new;
       q = q->next;
2:     if (q != NULL){
3:        q->data = new;
          assert(lock)
          lock=false;
          new ++;
       }
4: } while(new != old);
5:  assert(lock);
    lock=false;
    return;}
```

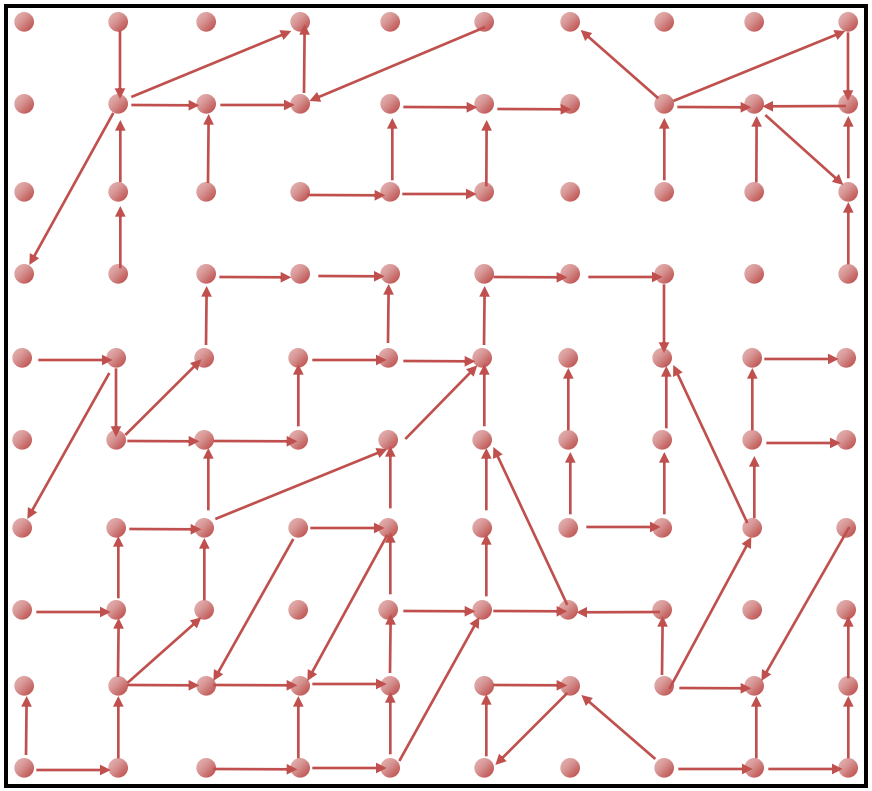
What a program *really* is…



pc ↦ 3
lock ↦ ●
old ↦ 5
new ↦ 5
q ↦ 0x133a

```
Example ( ) {
1: do{
       assert(!lock)
       lock=true;
       old = new;
       q = q->next;
2:     if (q != NULL){
3:        q->data = new;
          assert(lock)
          lock=false;
          new ++;
       }
4: } while(new != old);
5:  assert(lock);
    lock=false;
    return;}
```

What a program *really* is…



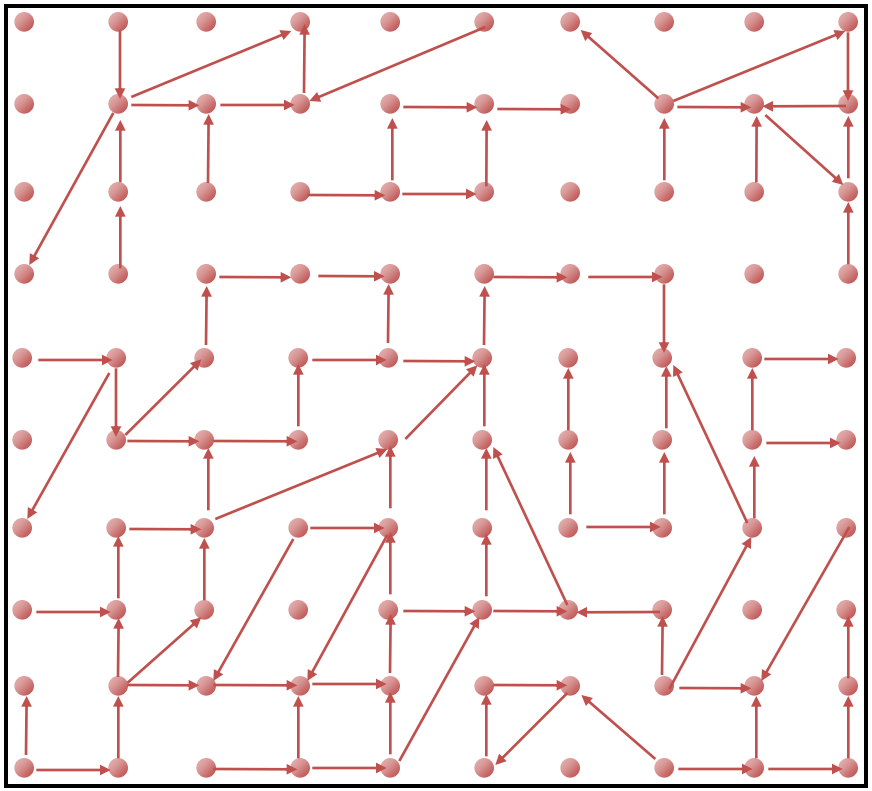
State

pc	↦	3
lock	↦	●
old	↦	5
new	↦	5
q	↦	0x133a

```
3: lock=false;
   new++;
4:} …
```

```
Example ( ) {
1: do{
      assert(!lock)
      lock=true;
      old = new;
      q = q->next;
2:    if (q != NULL){
3:       q->data = new;
         assert(lock)
         lock=false;
         new ++;
      }
4: } while(new != old);
5:  assert(lock);
    lock=false;
    return;}
```

What a program *really* is…

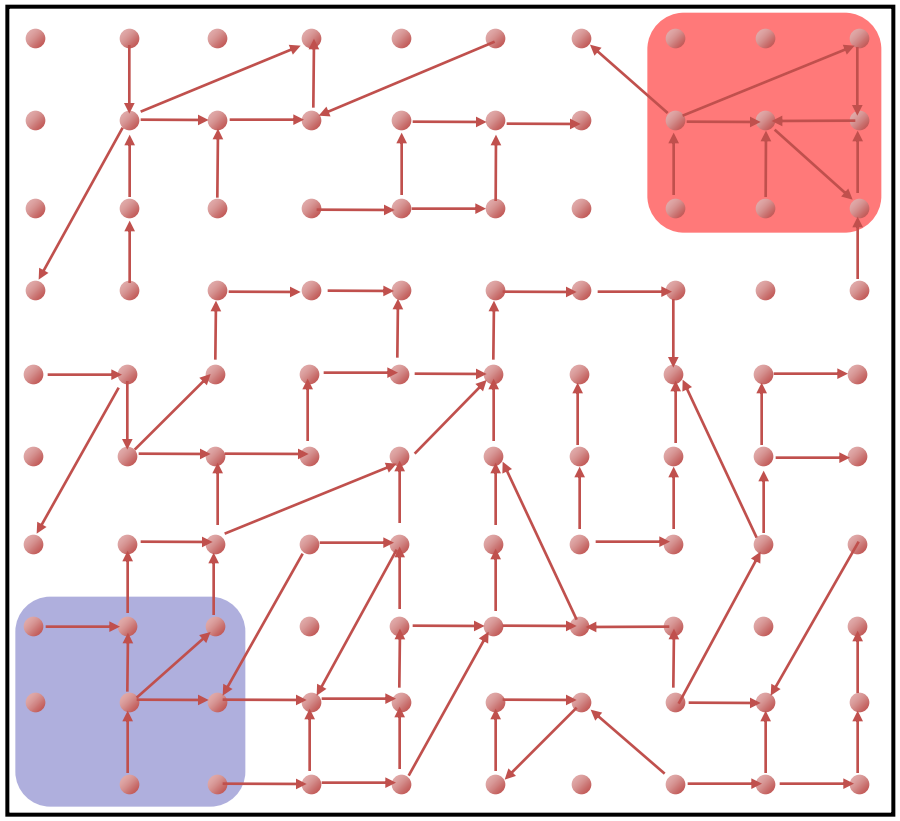


State

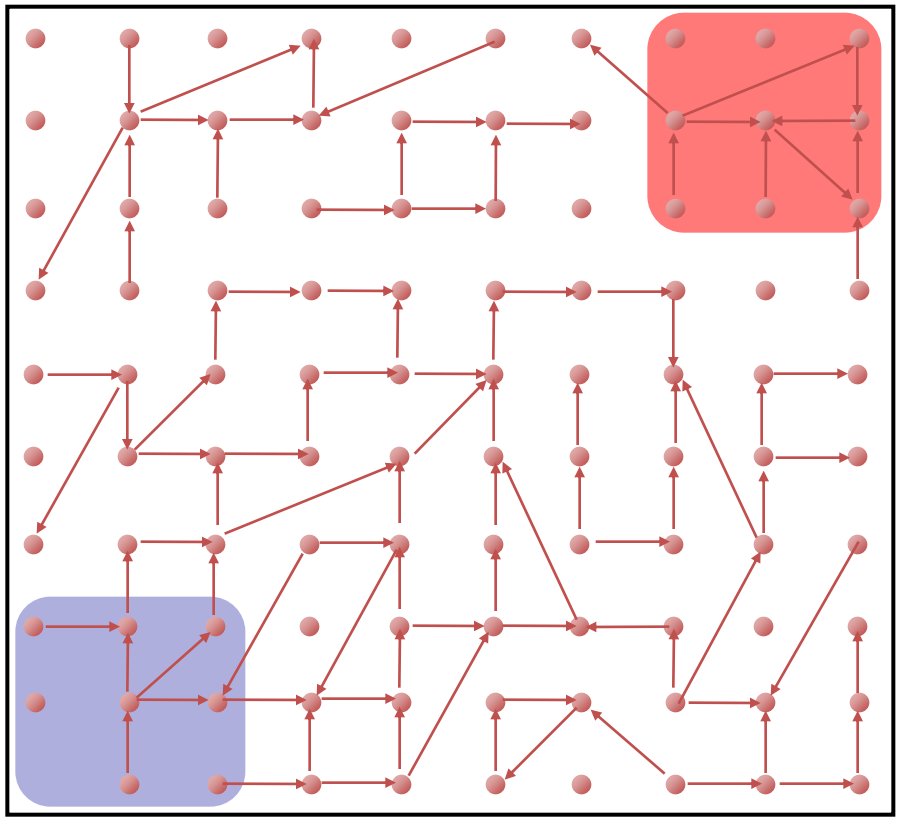
pc	↦ 3		***pc*** ↦ 4
lock	↦ ●	`3: lock=false;`	lock ↦ ○
old	↦ 5	`new++;`	old ↦ 5
new	↦ 5	`4:} …`	new ↦ 6
q	↦ 0x133a		q ↦ 0x133a

```
Example ( ) {
1: do{
       assert(!lock)
       lock=true;
       old = new;
       q = q->next;
2:     if (q != NULL){
3:        q->data = new;
          assert(lock)
          lock=false;
          new ++;
       }
4: } while(new != old);
5:  assert(lock);
    lock=false;
    return;}
```

The Safety Verification Problem

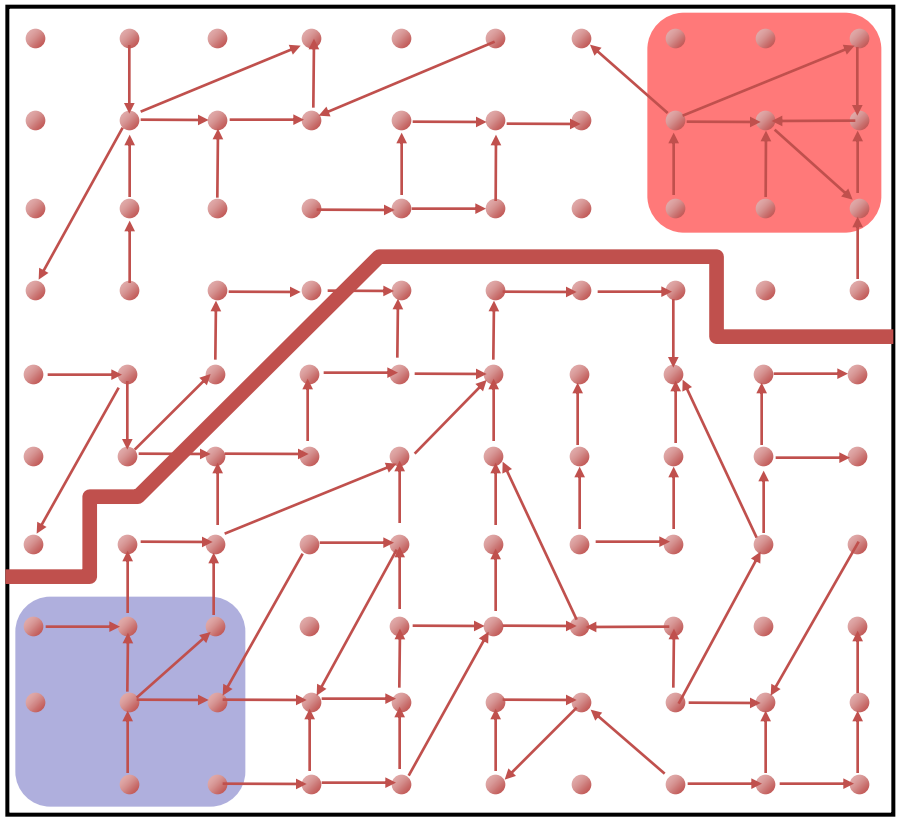


The Safety Verification Problem



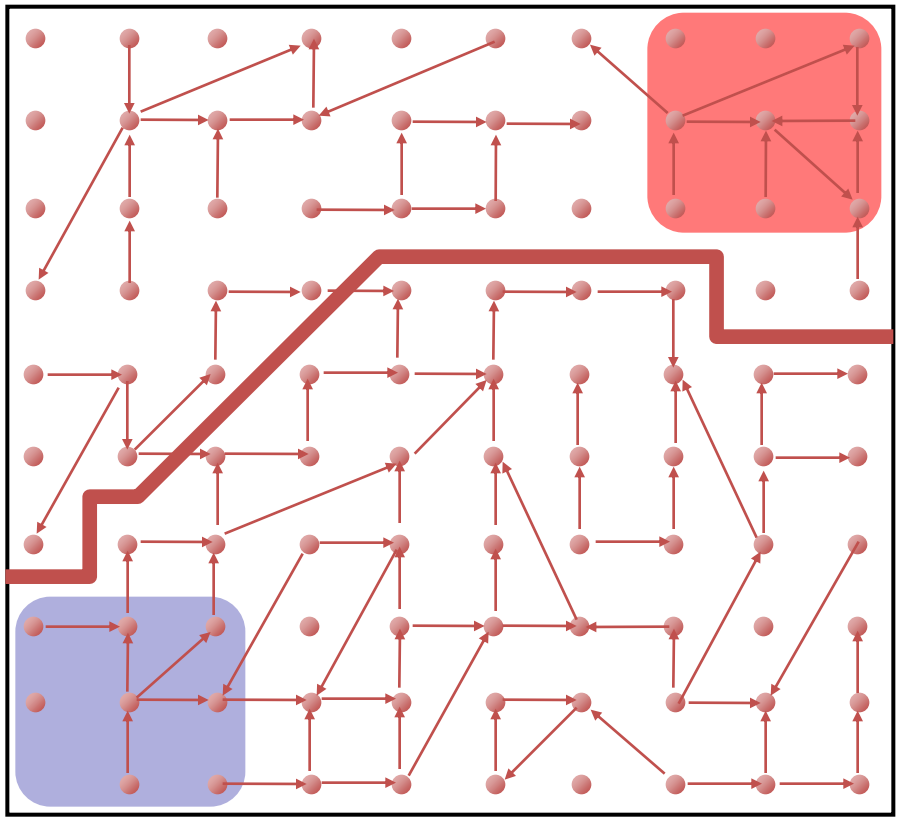
Is there a **path** from an **initial** to an **error** state ?

The Safety Verification Problem



Is there a **path** from an **initial** to an **error** state ?

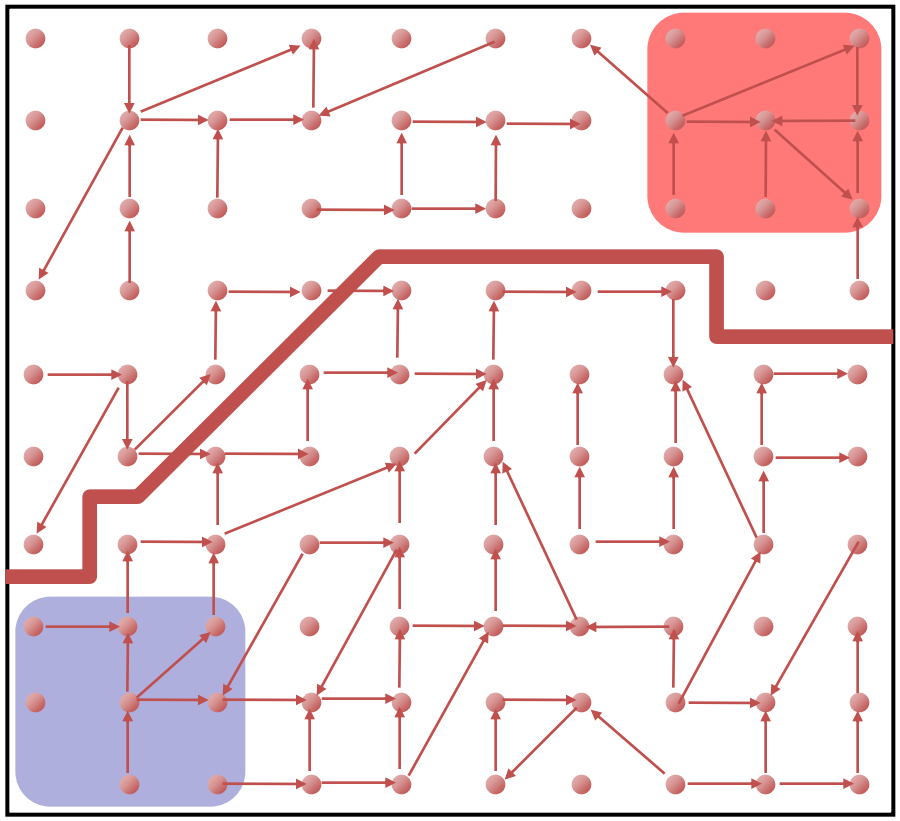
The Safety Verification Problem



Is there a **path** from an **initial** to an **error** state ?

Problem: Infinite state graph

The Safety Verification Problem



Is there a **path** from an **initial** to an **error** state ?

Problem: Infinite state graph

Solution : **Set** of states $\simeq$ logical **formula**

Representing States as *Formulas*

Representing States as *Formulas*

$[F]$	F
states satisfying F $\{s \mid s \models F\}$	FO fmla over prog. vars

Representing States as *Formulas*

$[F]$	F
states satisfying F $\{s \mid s \models F\}$	FO fmla over prog. vars
$[F_1] \cap [F_2]$	$F_1 \wedge F_2$

Representing States as *Formulas*

$[F]$	F
states satisfying F $\{s \mid s \models F\}$	FO fmla over prog. vars
$[F_1] \cap [F_2]$	$F_1 \wedge F_2$
$[F_1] \cup [F_2]$	$F_1 \vee F_2$

Representing States as *Formulas*

$[F]$	F
states satisfying F $\{s \mid s \models F\}$	FO fmla over prog. vars
$[F_1] \cap [F_2]$	$F_1 \wedge F_2$
$[F_1] \cup [F_2]$	$F_1 \vee F_2$
$\overline{[F]}$	$\neg F$

Representing States as *Formulas*

$[F]$	F
states satisfying F $\{s \mid s \models F\}$	FO fmla over prog. vars
$[F_1] \cap [F_2]$	$F_1 \wedge F_2$
$[F_1] \cup [F_2]$	$F_1 \vee F_2$
$\overline{[F]}$	$\neg F$
$[F_1] \subseteq [F_2]$	

Representing States as *Formulas*

$[F]$	F
states satisfying F $\{s \mid s \models F\}$	FO fmla over prog. vars
$[F_1] \cap [F_2]$	$F_1 \wedge F_2$
$[F_1] \cup [F_2]$	$F_1 \vee F_2$
$\overline{[F]}$	$\neg F$
$[F_1] \subseteq [F_2]$	F_1 implies F_2

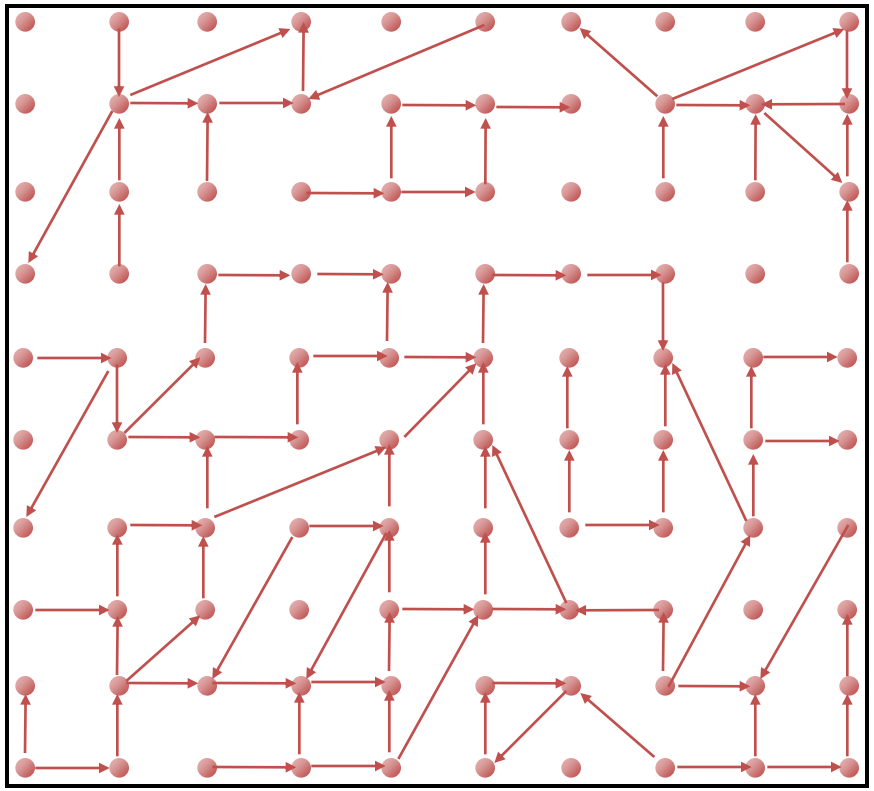
Representing States as *Formulas*

$[F]$	F
states satisfying F $\{s \mid s \models F\}$	FO fmla over prog. vars
$[F_1] \cap [F_2]$	$F_1 \wedge F_2$
$[F_1] \cup [F_2]$	$F_1 \vee F_2$
$\overline{[F]}$	$\neg F$
$[F_1] \subseteq [F_2]$	F_1 implies F_2
	i.e. $F_1 \wedge \neg F_2$ unsatisfiable

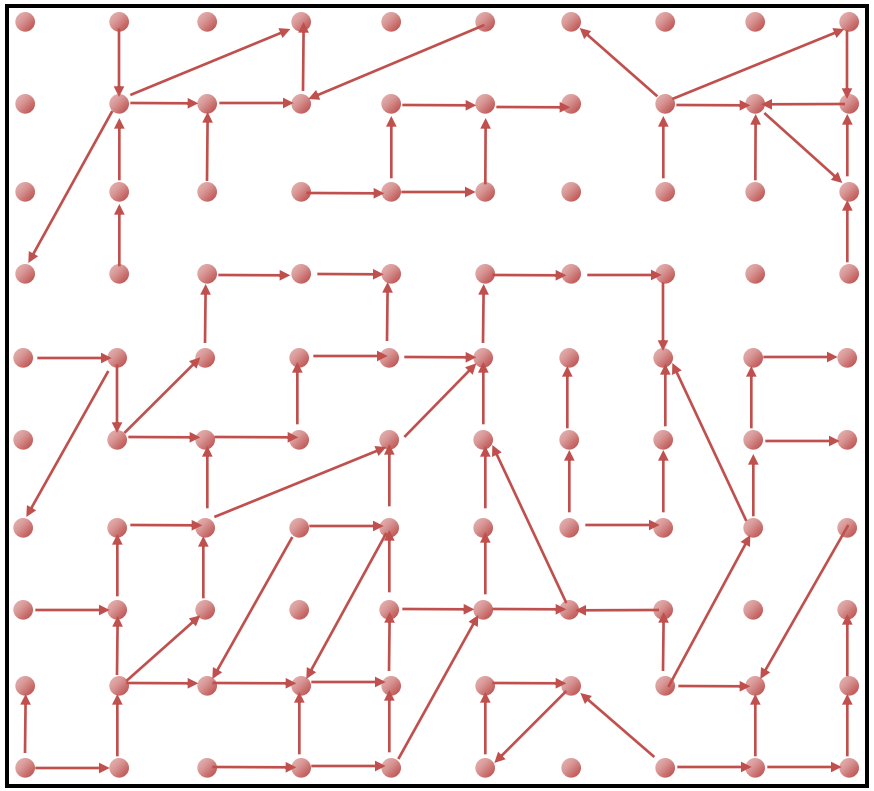
Representing States as *Formulas*

$[F]$	F
states satisfying F $\{s \mid s \models F\}$	FO fmla over prog. vars
$[F_1] \cap [F_2]$	$F_1 \wedge F_2$
$[F_1] \cup [F_2]$	$F_1 \vee F_2$
$\overline{[F]}$	$\neg F$
$[F_1] \subseteq [F_2]$	F_1 implies F_2
	i.e. $F_1 \wedge \neg F_2$ unsatisfiable

Idea 1: Predicate Abstraction



Idea 1: Predicate Abstraction

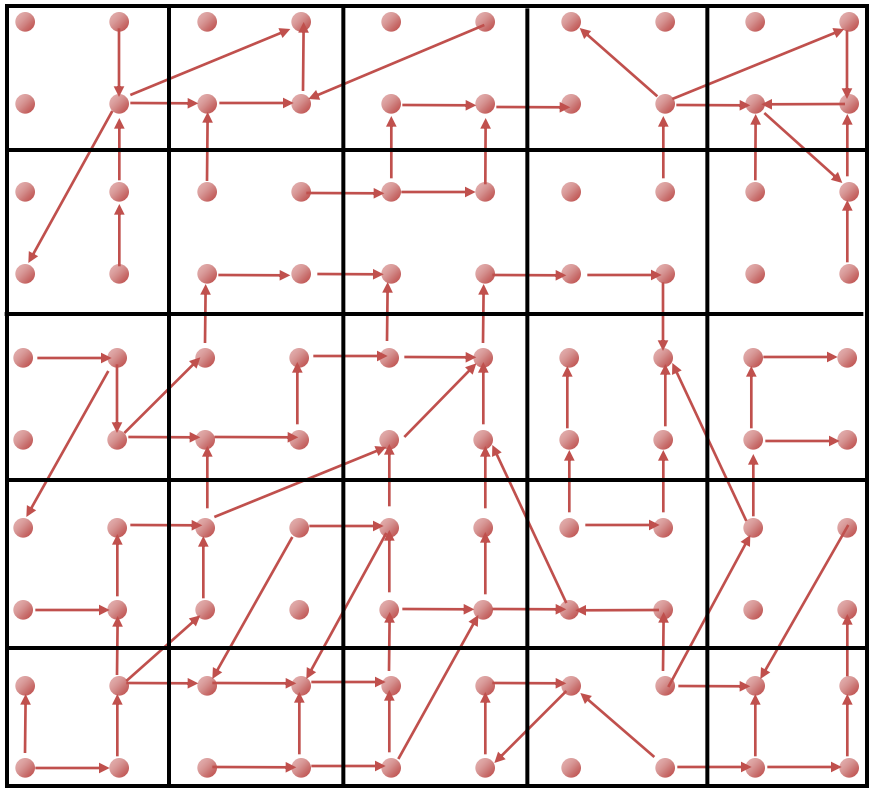


- **Predicates** on program state:

  lock

  old = new

Idea 1: Predicate Abstraction



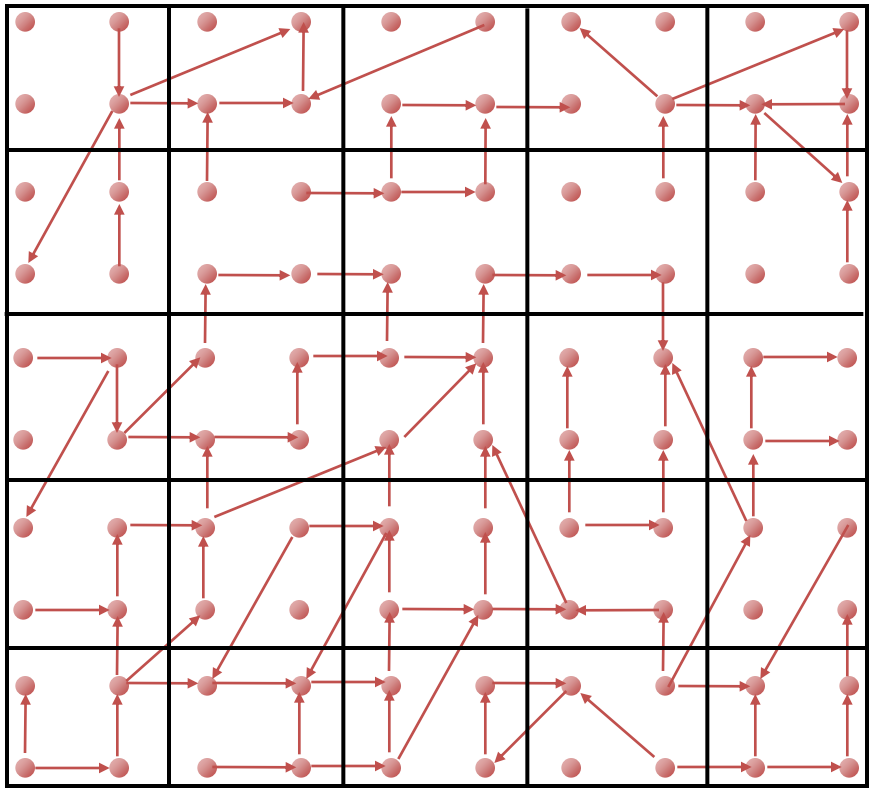
- **Predicates** on program state:

  lock

  old = new

- States satisfying **same** predicates are **equivalent**
  - **Merged** into one **abstract state**

Idea 1: Predicate Abstraction



- **Predicates** on program state:

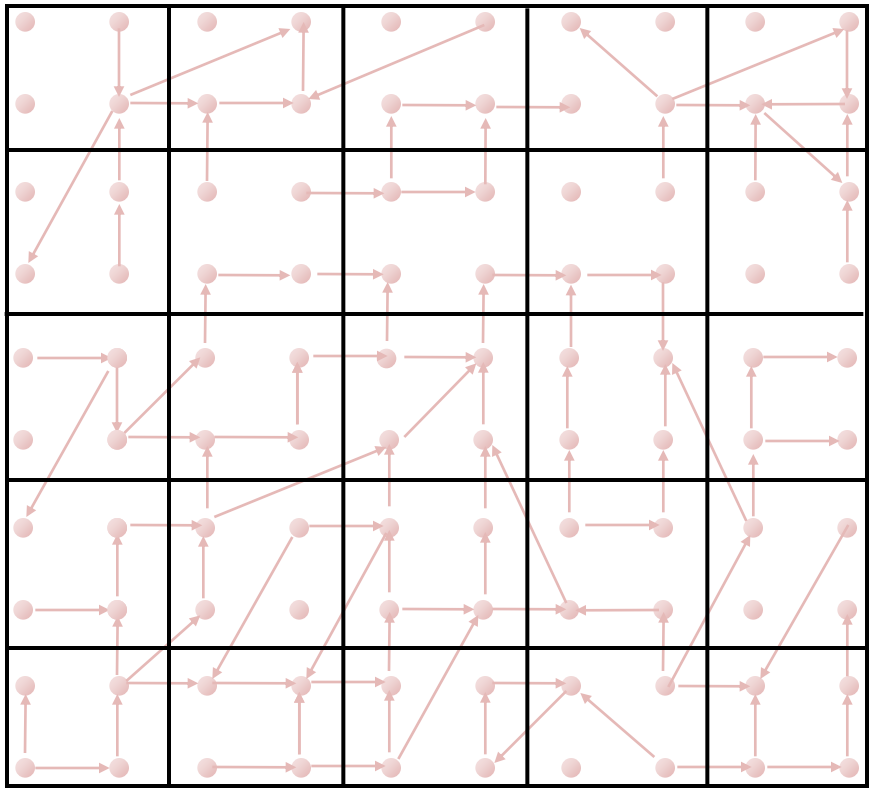
  lock

  old = new

- States satisfying **same** predicates are **equivalent**
  - **Merged** into one **abstract state**

- #abstract states is **finite**

Abstract States and Transitions



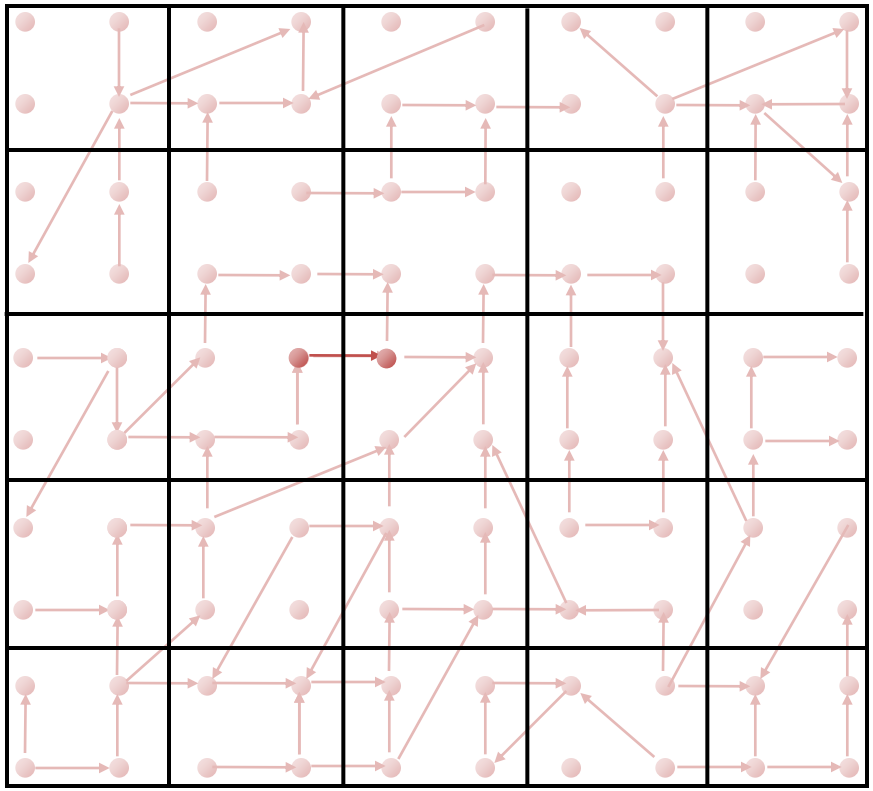
State

pc	↦ 3
lock	↦ ●
old	↦ 5
new	↦ 5
q	↦ 0x133a

```
3: lock=false;
   new++;
4:} …
```

pc	↦ 4
lock	↦ ○
old	↦ 5
new	↦ 6
q	↦ 0x133a

Abstract States and Transitions



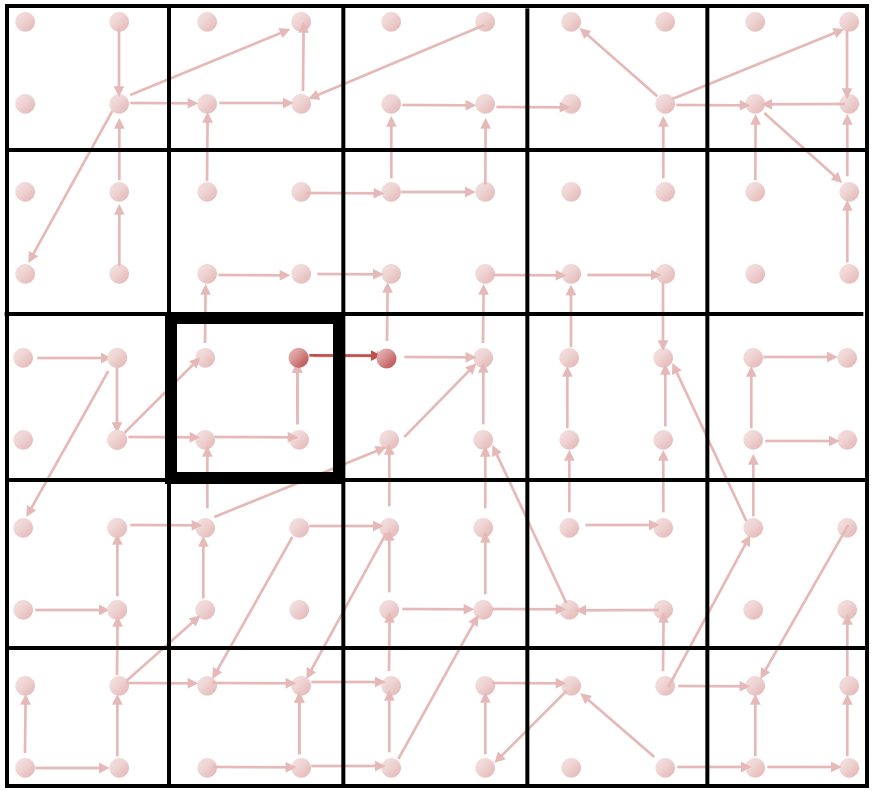
State

pc	↦ 3
lock	↦ ●
old	↦ 5
new	↦ 5
q	↦ 0x133a

```
3: lock=false;
   new++;
4:} …
```

pc	↦ 4
lock	↦ ○
old	↦ 5
new	↦ 6
q	↦ 0x133a

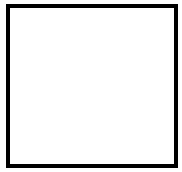
Abstract States and Transitions



State

pc	↦ 3		*pc*	↦ 4
lock	↦ ●		lock	↦ ○
old	↦ 5		old	↦ 5
new	↦ 5		new	↦ 6
q	↦ 0x133a		q	↦ 0x133a

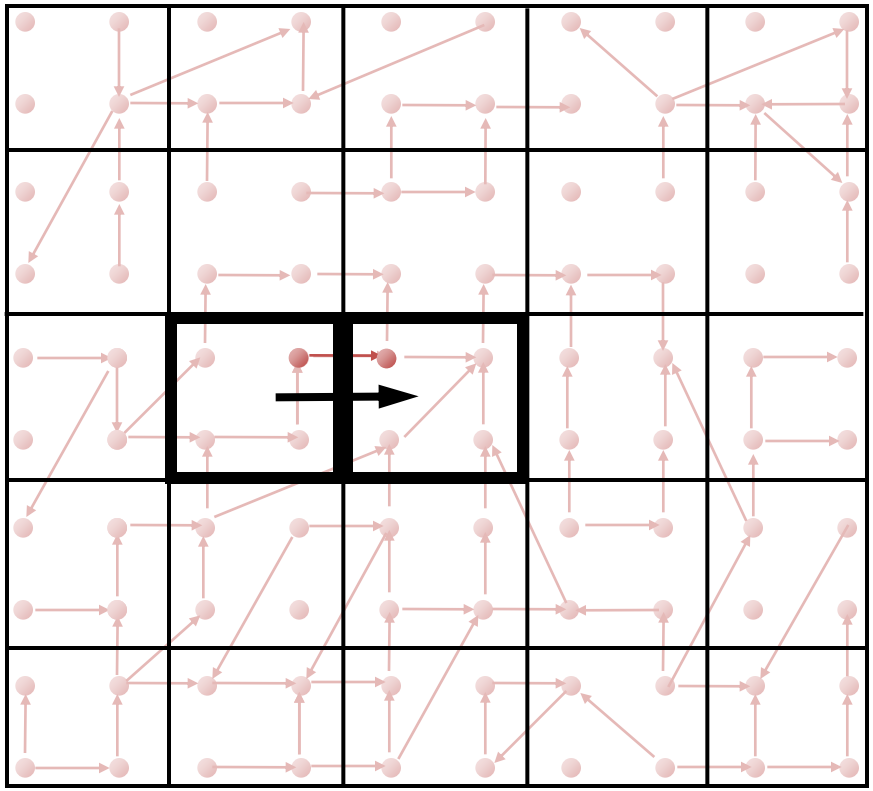
```
3: lock=false;
   new++;
4:} …
```



lock

old=new

Abstract States and Transitions

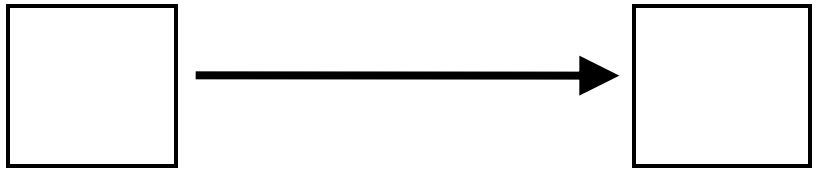


State

pc ↦ 3
lock ↦ ●
old ↦ 5
new ↦ 5
q ↦ 0x133a

```
3: lock=false;
   new++;
4: } …
```

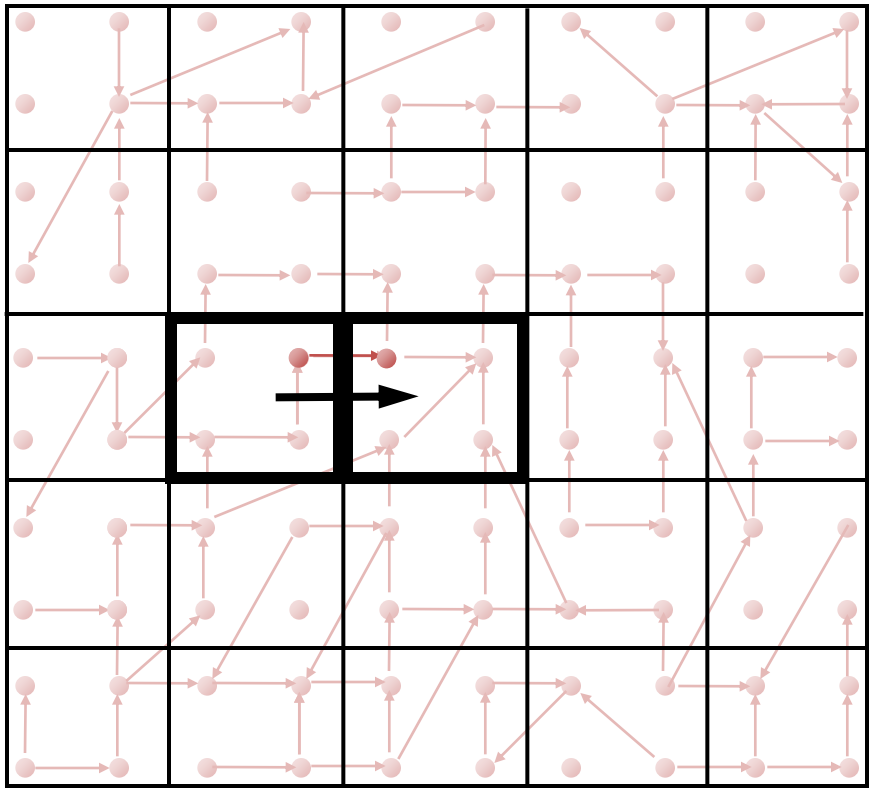
pc ↦ 4
lock ↦ ○
old ↦ 5
new ↦ 6
q ↦ 0x133a



lock
old=new

¬ lock
¬ old=new

Abstract States and Transitions



State

pc	↦ 3			*pc*	↦ 4
lock	↦ ●	`3: lock=false;`		lock	↦ ○
old	↦ 5	`new++;`		old	↦ 5
new	↦ 5	`4: }` …		new	↦ 6
q	↦ 0x133a			q	↦ 0x133a

Theorem Prover

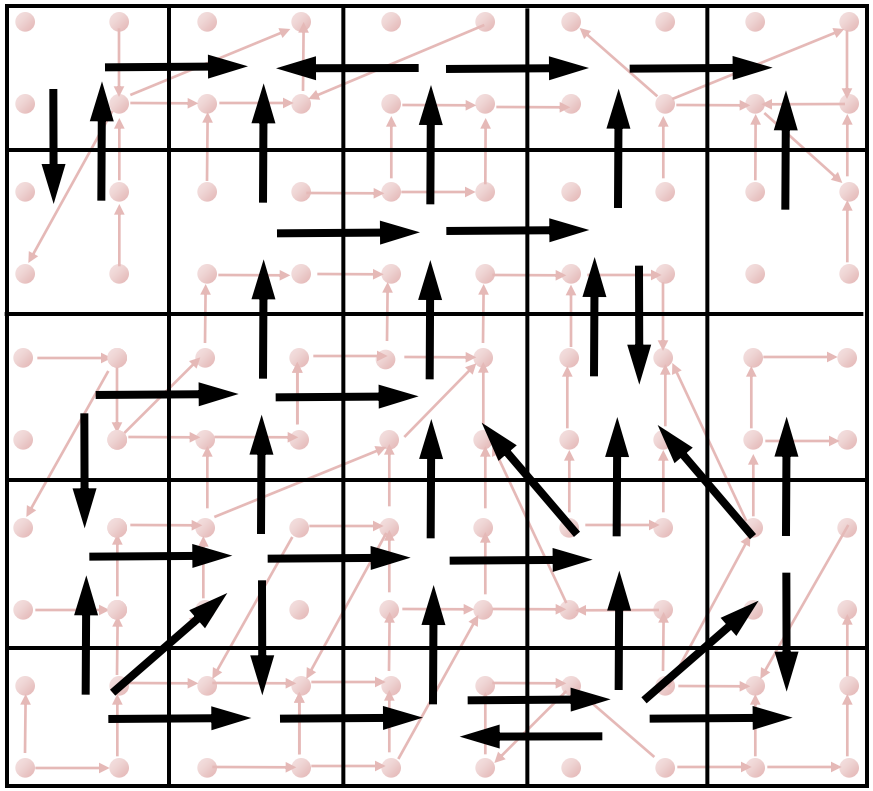
lock

old=new

¬ lock

¬ old=new

Abstraction



Existential Lifting

State

pc	↦ 3			*pc*	↦ 4
lock	↦ ●	`3: lock=false;`		lock	↦ ○
old	↦ 5	`new++;`		old	↦ 5
new	↦ 5	`4:}` …		new	↦ 6
q	↦ 0x133a			q	↦ 0x133a

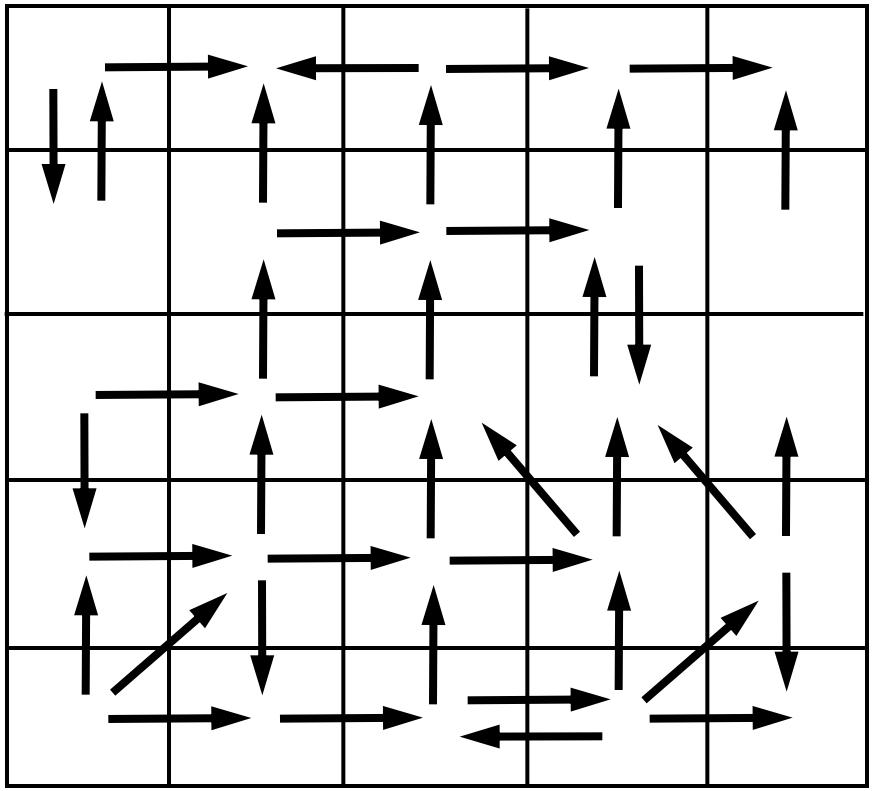
lock

old=new

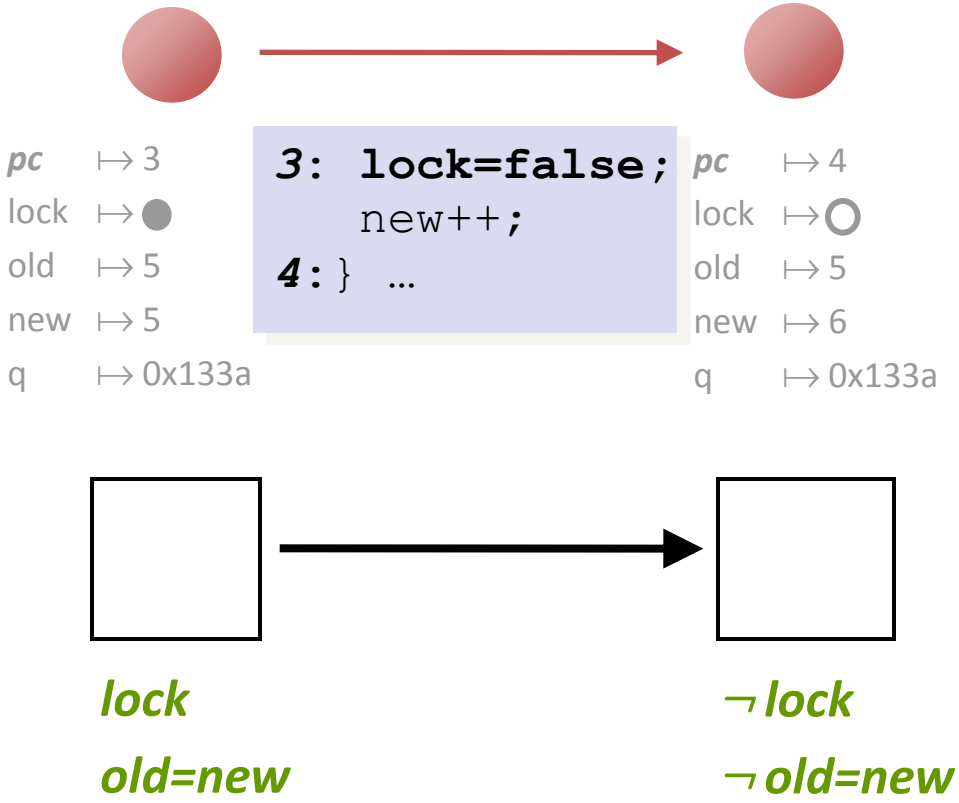
¬ lock

¬ old=new

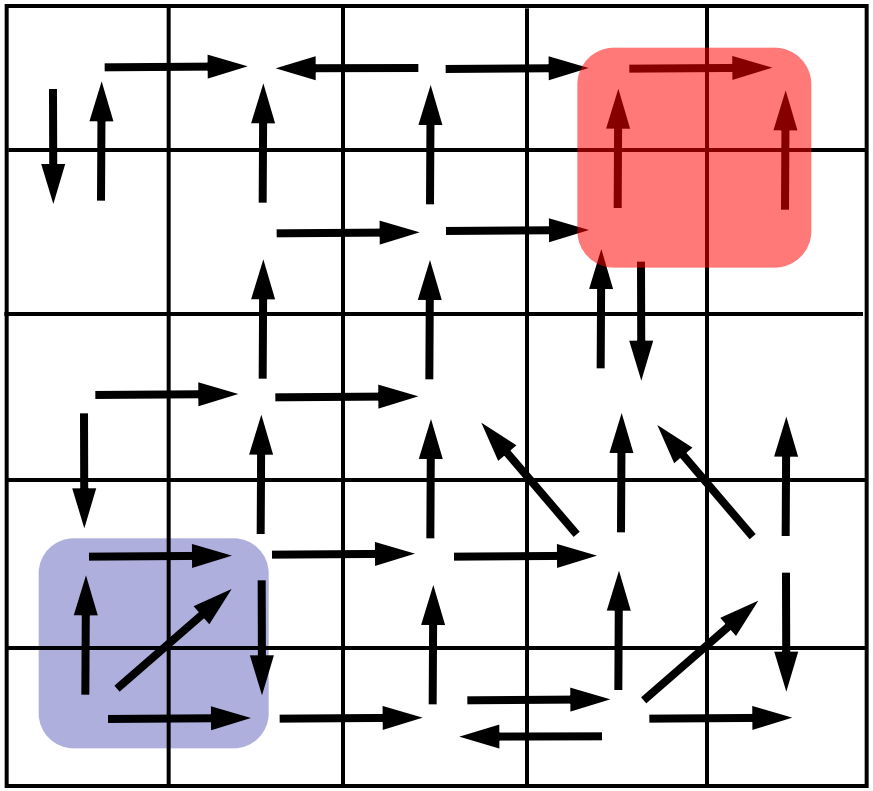
Abstraction



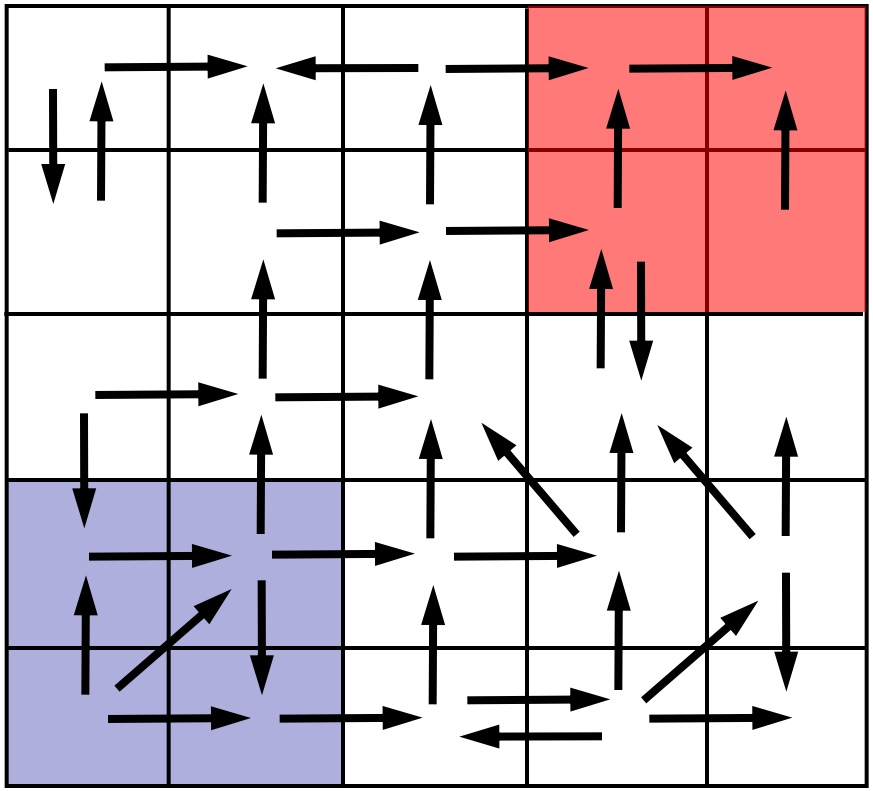
State



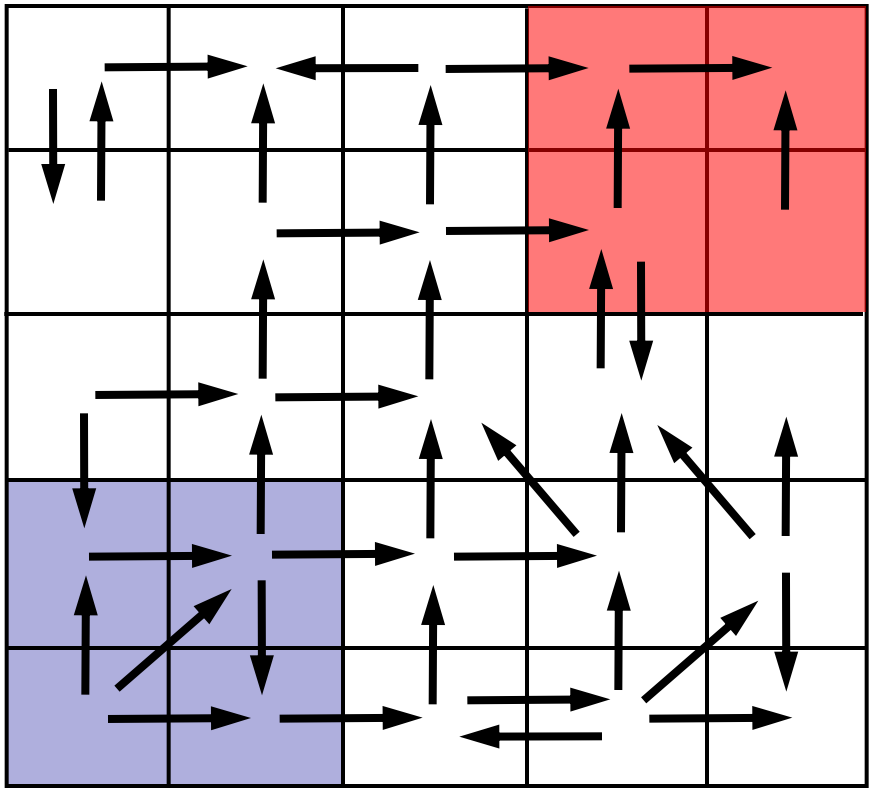
Analyze Abstraction



Analyze Abstraction

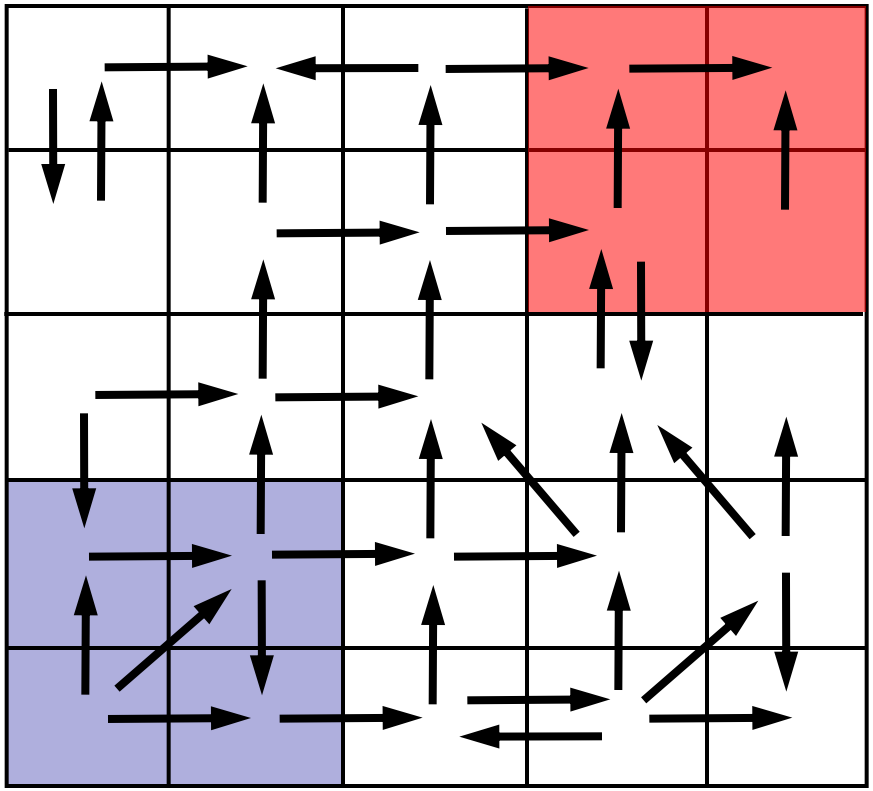


Analyze Abstraction



Analyze finite graph

Analyze Abstraction



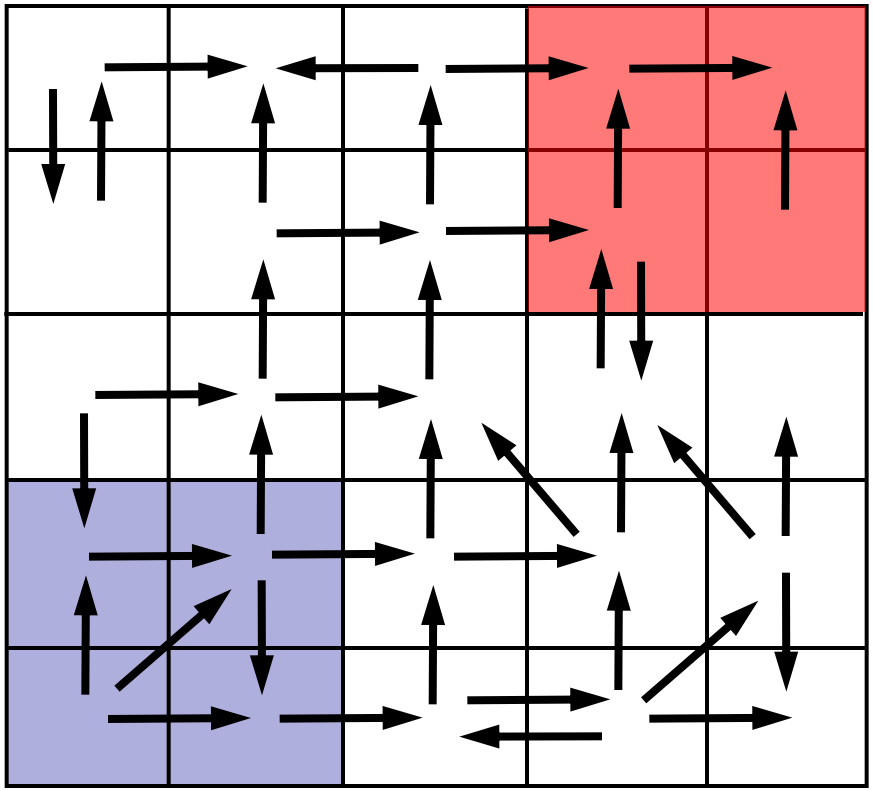
Analyze finite graph

Over Approximate:

Safe $\Rightarrow$ System Safe

No **false negatives**

Analyze Abstraction



Analyze finite graph

Over Approximate:

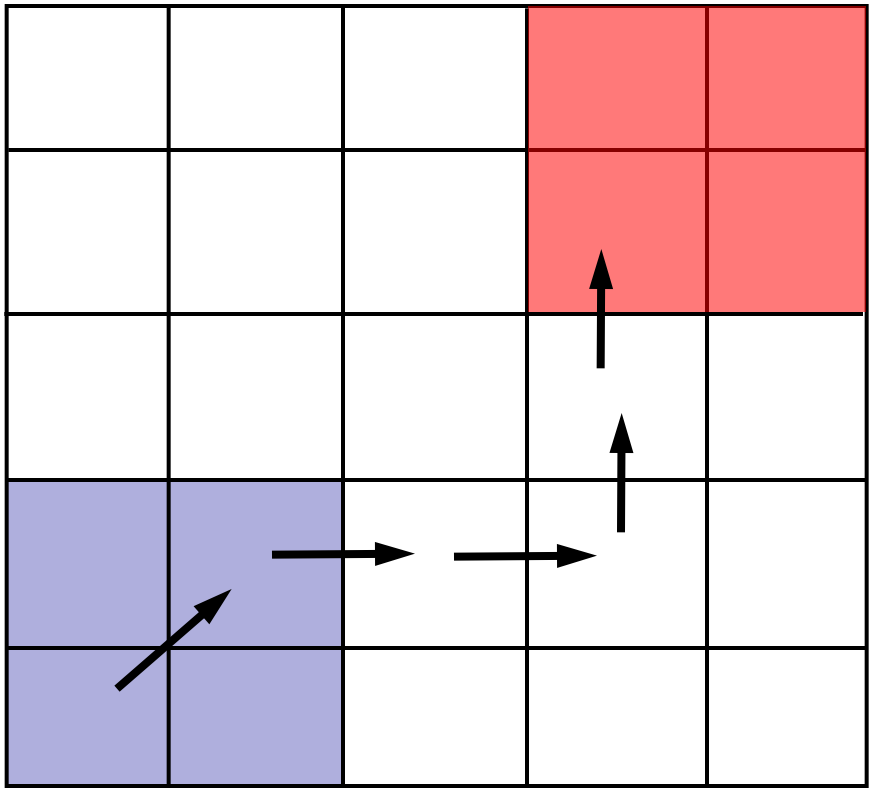
Safe $\Rightarrow$ System Safe

No **false negatives**

Problem

Spurious **counterexamples**

Analyze Abstraction



Analyze finite graph

Over Approximate:

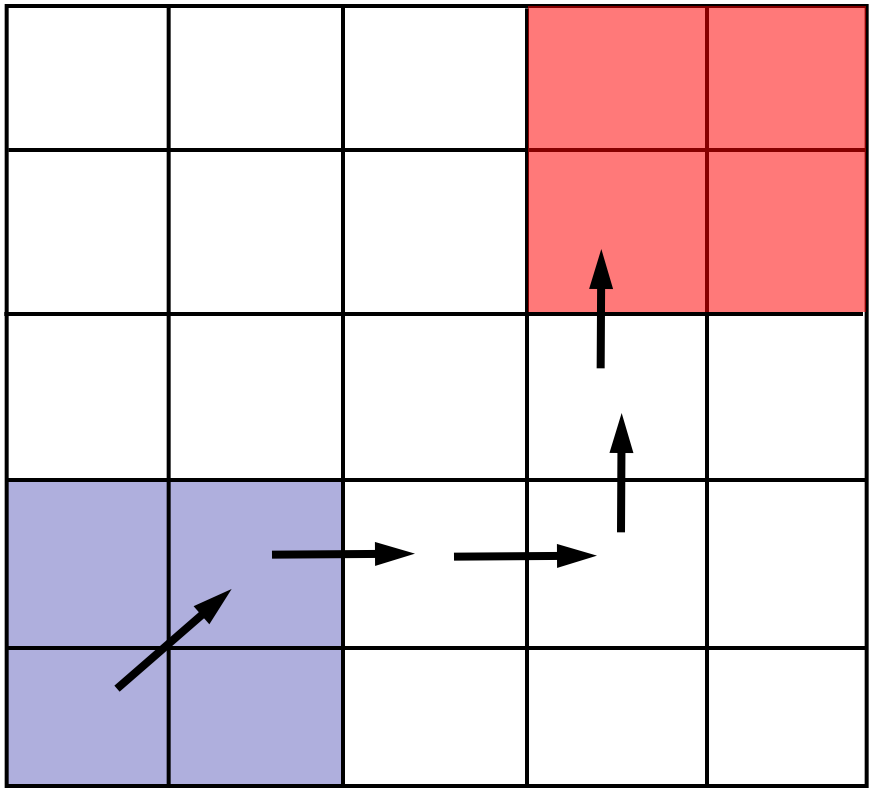
Safe $\Rightarrow$ System Safe

No **false negatives**

Problem

Spurious **counterexamples**

Idea 2: Counterex.-Guided Refinement

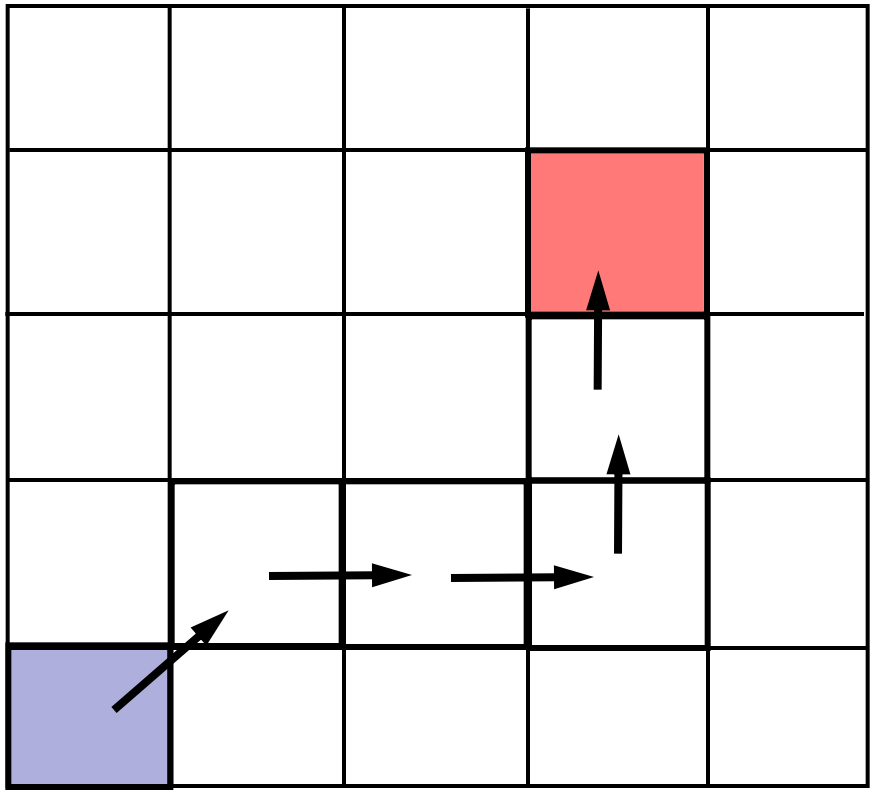


Solution

Use spurious **counterexamples**

to **refine** abstraction !

Idea 2: Counterex.-Guided Refinement

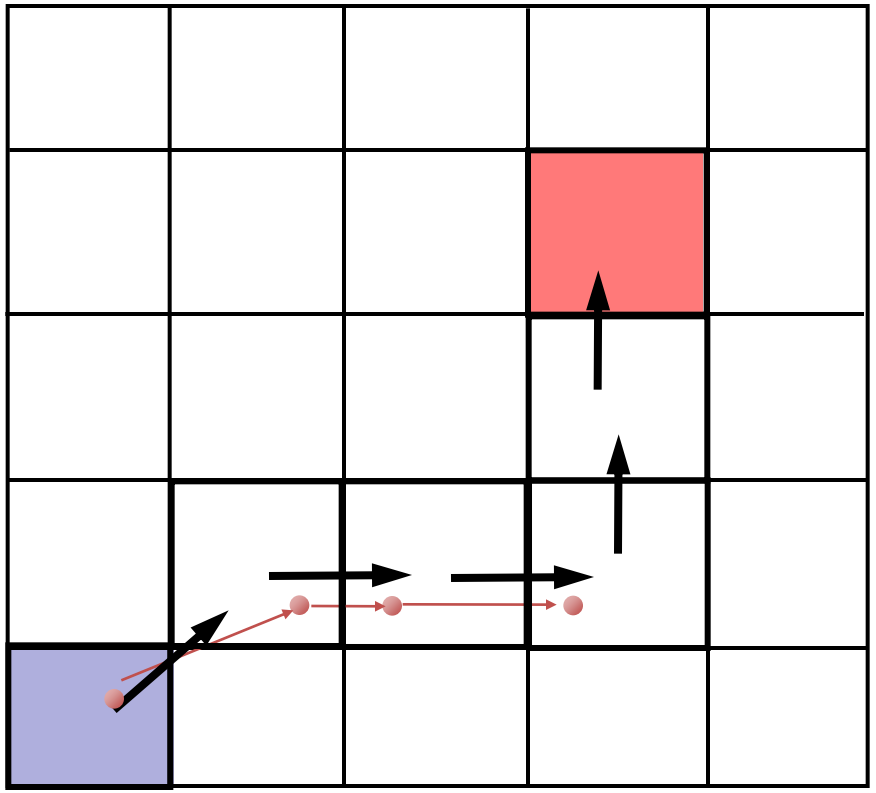


Solution

Use spurious **counterexamples**

to **refine** abstraction

Idea 2: Counterex.-Guided Refinement

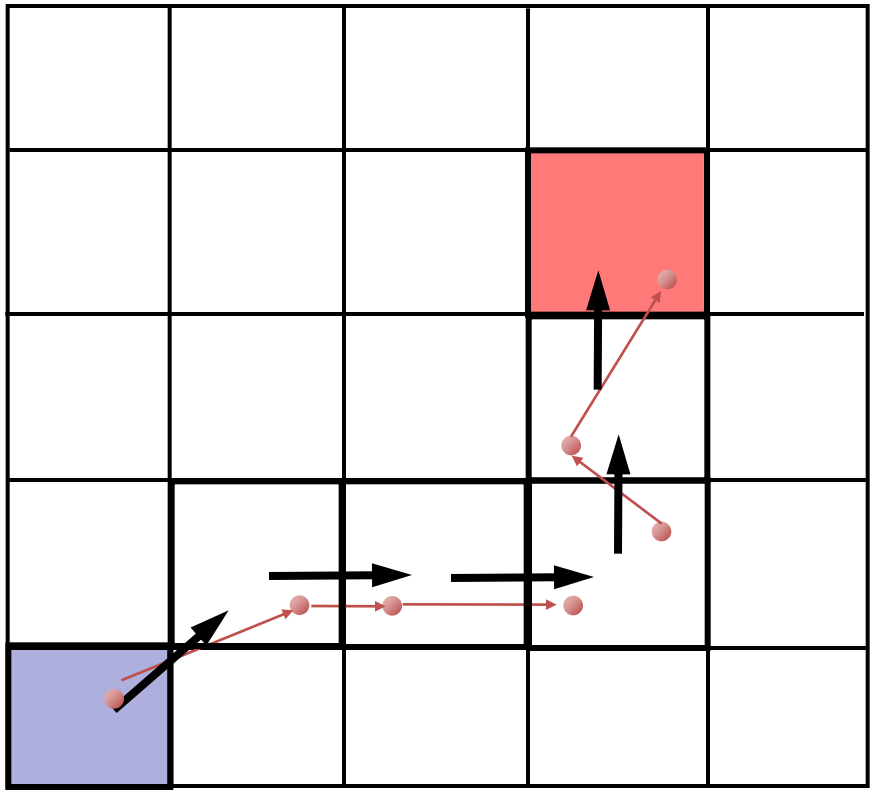


Solution

Use spurious **counterexamples**

to **refine** abstraction

Idea 2: Counterex.-Guided Refinement

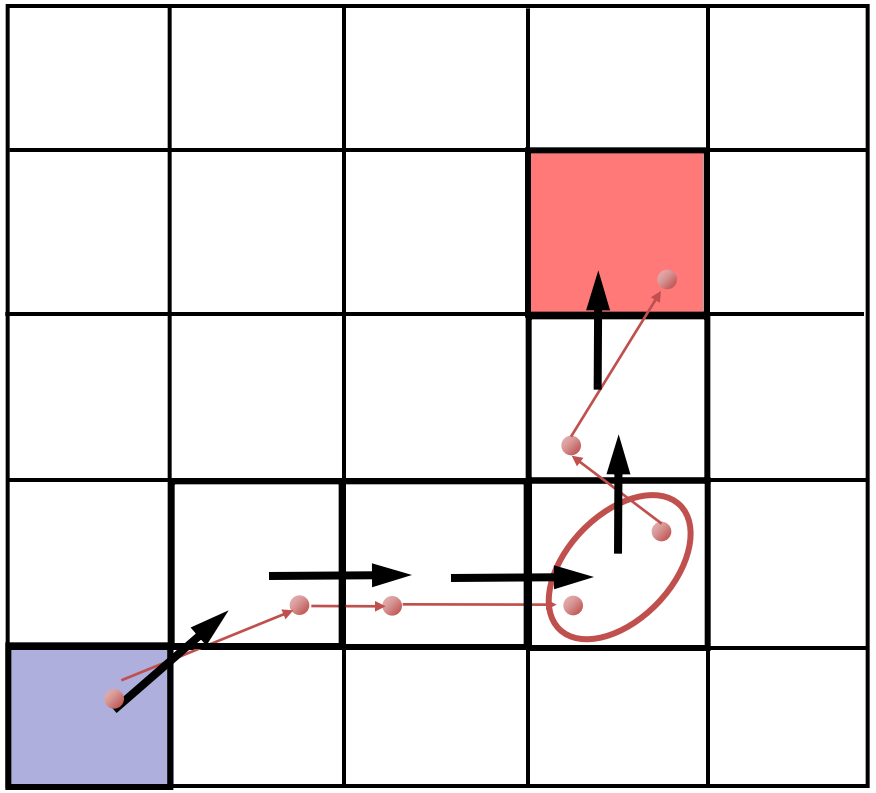


Solution

Use spurious **counterexamples**

to **refine** abstraction

Idea 2: Counterex.-Guided Refinement



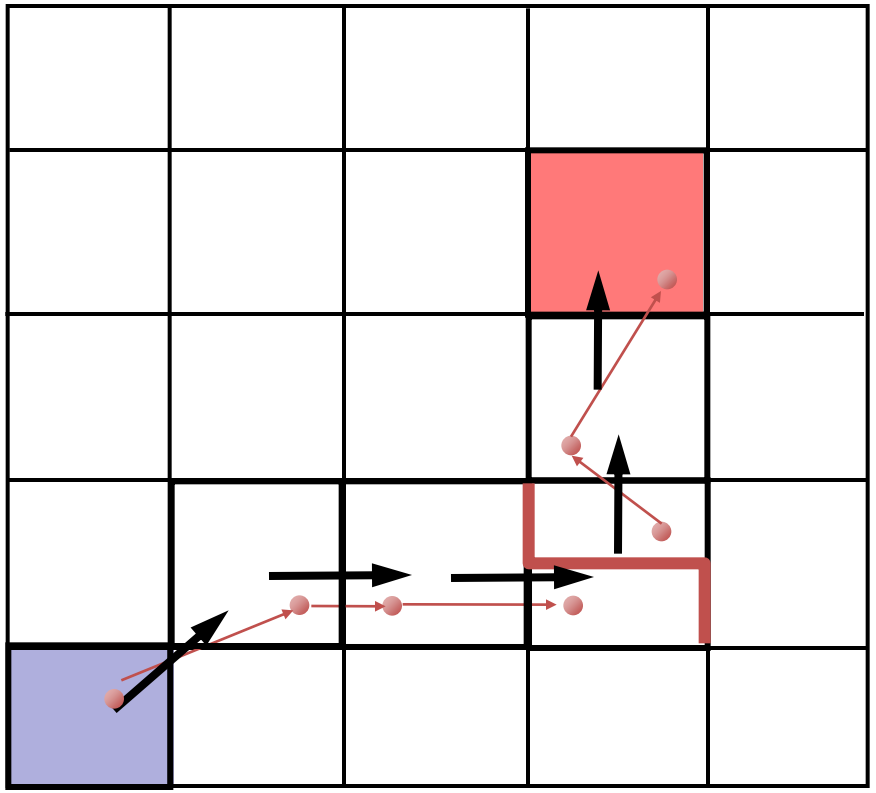
Solution

Use spurious **counterexamples**

to **refine** abstraction

Imprecision due to **merge**

Idea 2: Counterex.-Guided Refinement

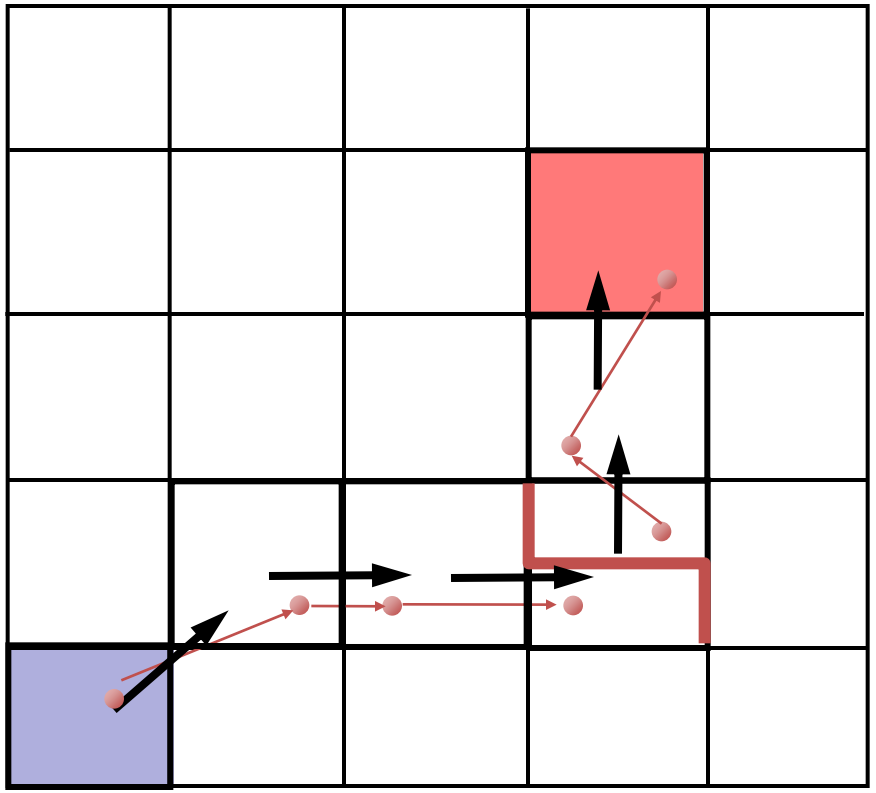


Solution

Use spurious **counterexamples** to **refine** abstraction

1. **Add predicates** to distinguish states across **cut**

Idea 2: Counterex.-Guided Refinement

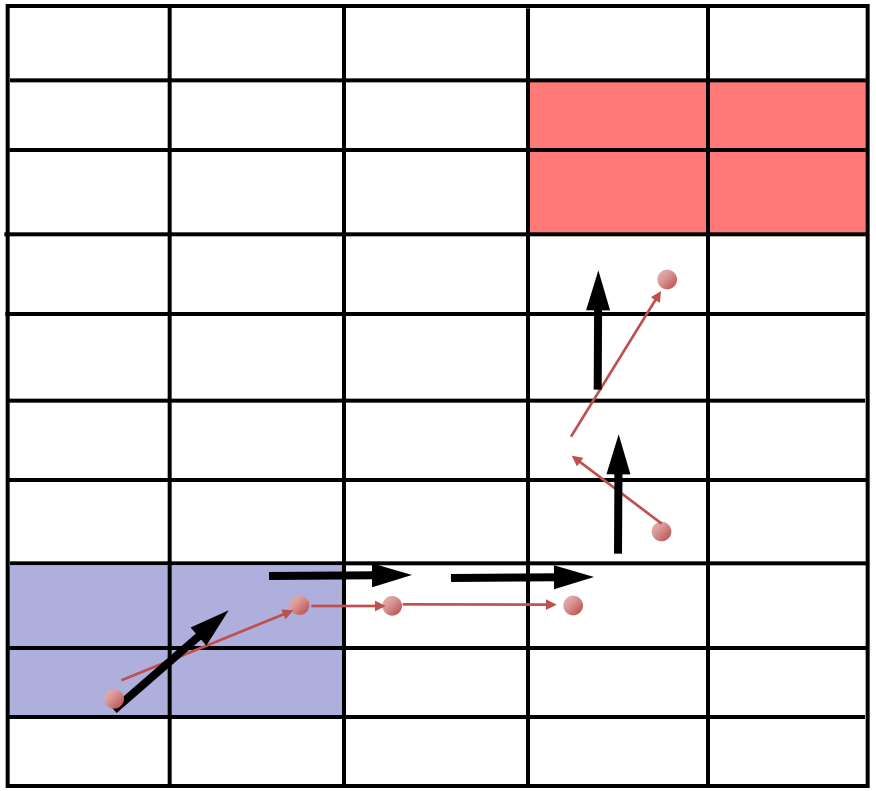


Solution

Use spurious **counterexamples** to **refine** abstraction

1. **Add predicates** to distinguish states across **cut**
2. Build **refined** abstraction

Iterative Abstraction-Refinement



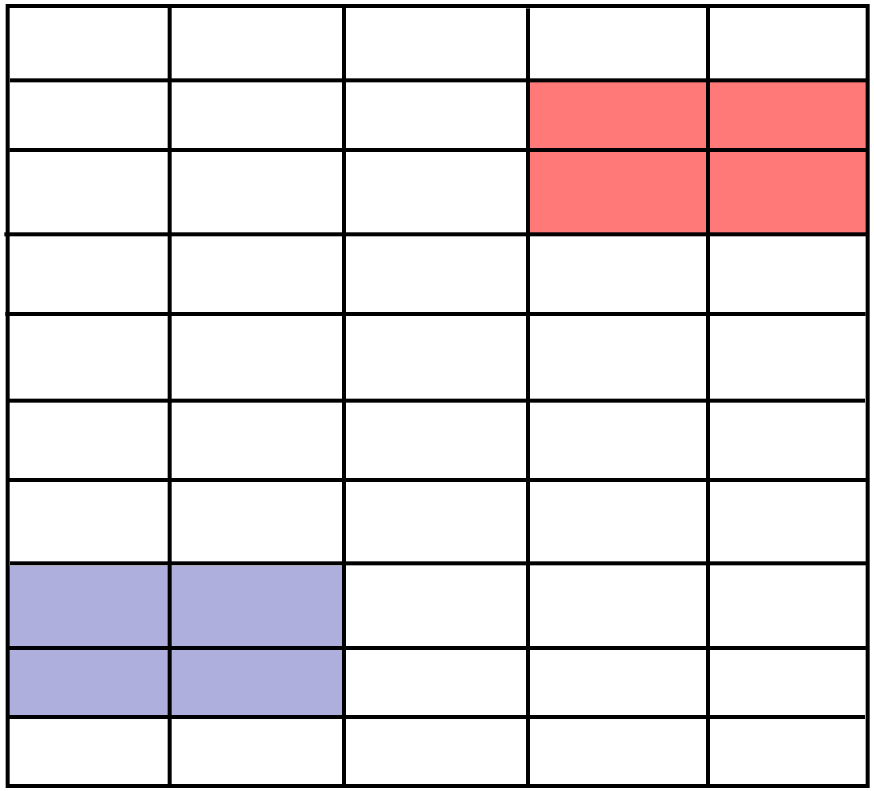
[Kurshan et al 93] [Clarke et al 00]
[Ball-Rajamani 01]

Solution

Use spurious **counterexamples** to **refine** abstraction

1. Add predicates to distinguish states across **cut**
2. Build **refined** abstraction
   -eliminates counterexample

Iterative Abstraction-Refinement



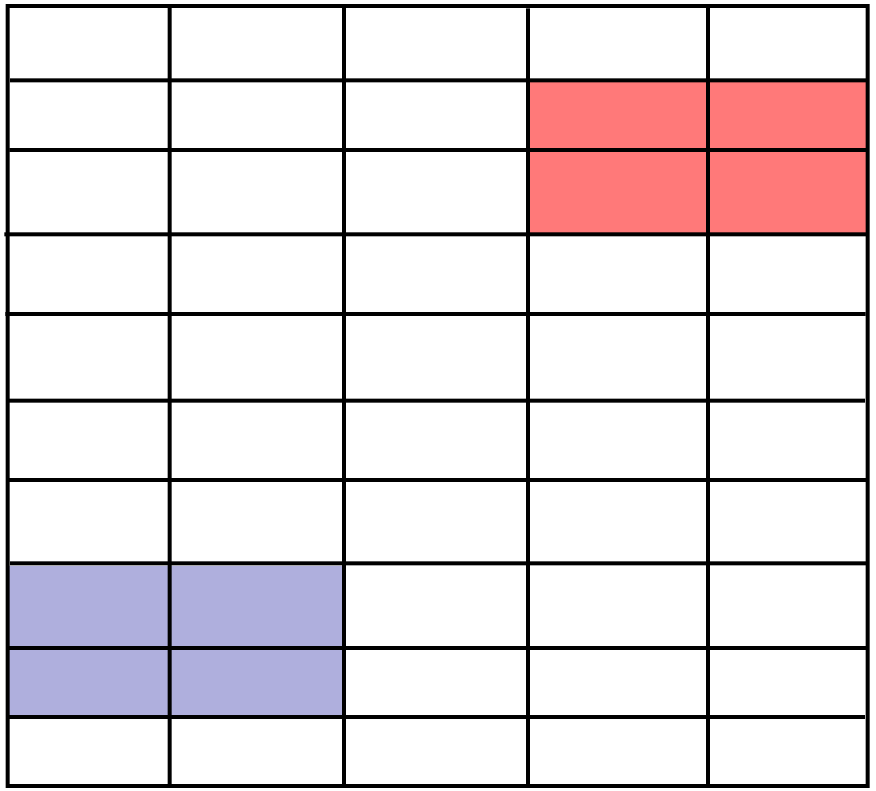
Solution

Use spurious **counterexamples** to **refine** abstraction

1. Add predicates to distinguish states across **cut**
2. Build **refined** abstraction
   -eliminates counterexample

[Kurshan et al 93] [Clarke et al 00]
[Ball-Rajamani 01]

Iterative Abstraction-Refinement



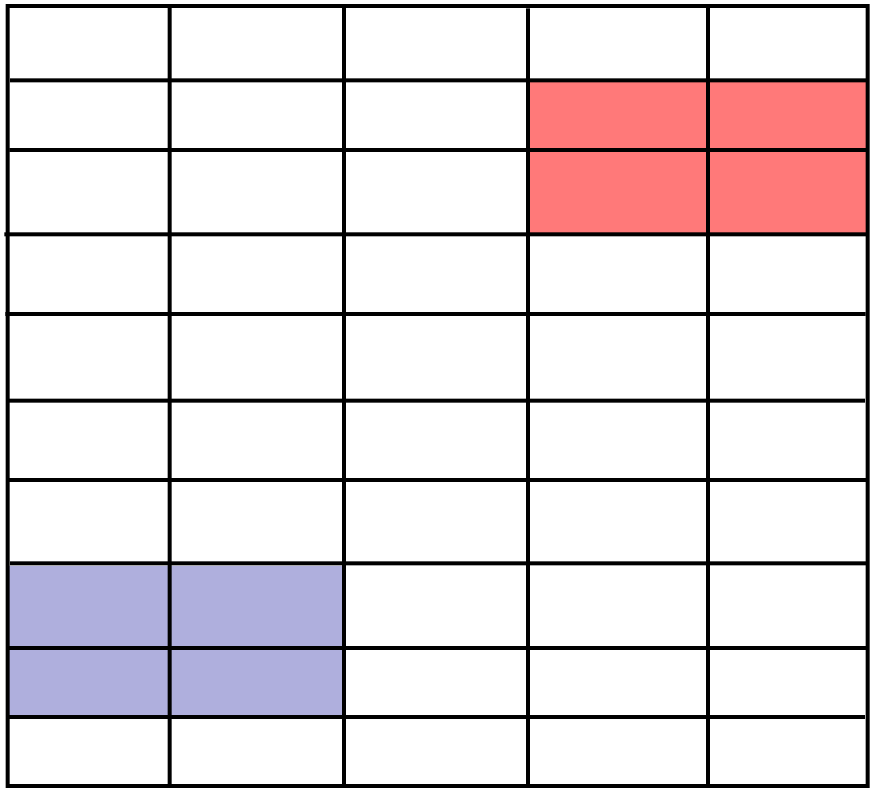
Solution

Use spurious **counterexamples** to **refine** abstraction

1. Add predicates to distinguish states across **cut**
2. Build **refined** abstraction
   -eliminates counterexample
3. **Repeat** search

[Kurshan et al 93] [Clarke et al 00]
[Ball-Rajamani 01]

Iterative Abstraction-Refinement



[Kurshan et al 93] [Clarke et al 00]
[Ball-Rajamani 01]

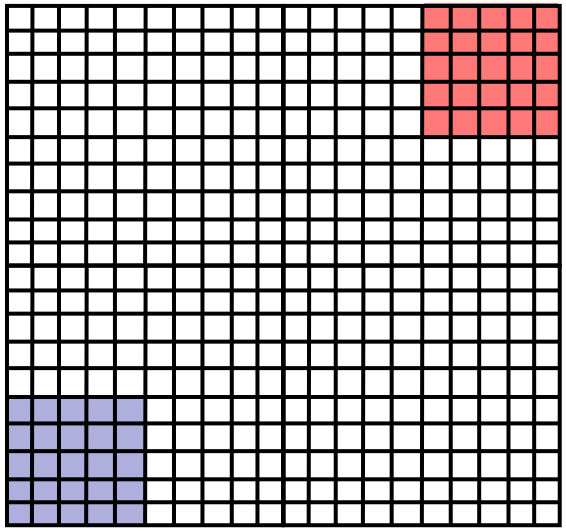
Solution

Use spurious **counterexamples**
to **refine** abstraction

1. Add predicates to distinguish states across **cut**
2. Build **refined** abstraction
   -eliminates counterexample
3. **Repeat** search
   Till real counterexample
   or system proved safe

Problem: Abstraction is Expensive

Problem: Abstraction is Expensive

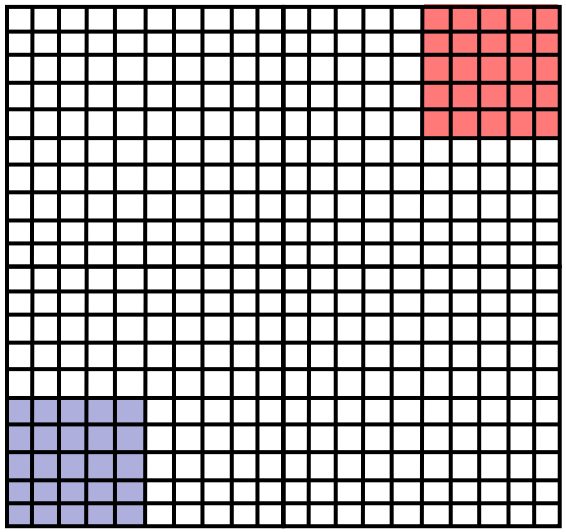


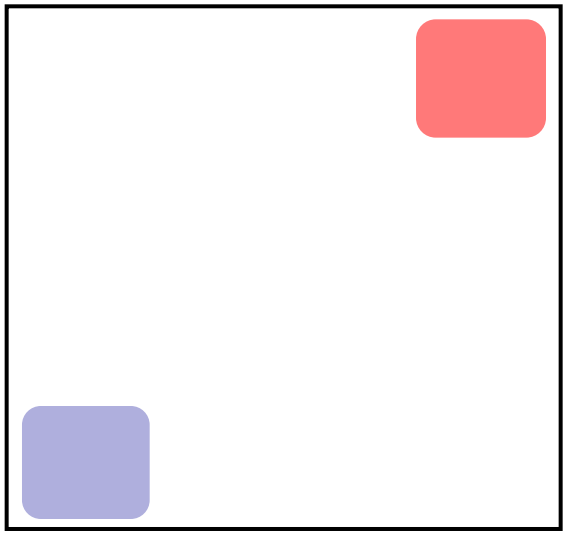
Problem

#abstract states = $2^{\#\text{predicates}}$

Exponential Thm. Prover queries

Problem: Abstraction is Expensive



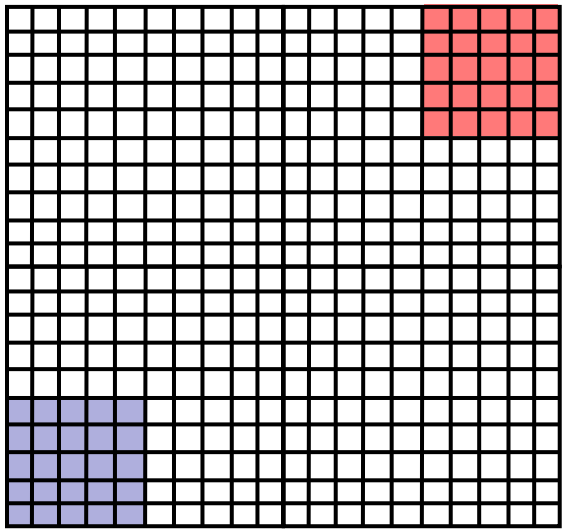


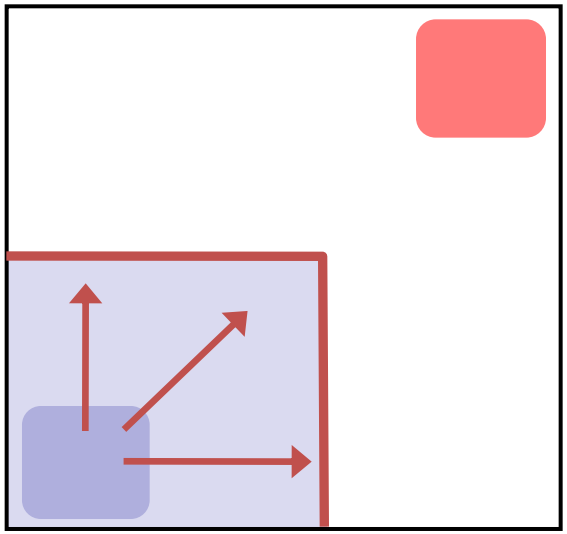
Problem

#abstract states = $2^{\#predicates}$

Exponential Thm. Prover queries

Problem: Abstraction is Expensive





Reachable

Problem

#abstract states = $2^{\#predicates}$

Exponential Thm. Prover queries

Observe

Fraction of state space reachable

#Preds ~ 100's, #States ~ 2^{100} ,

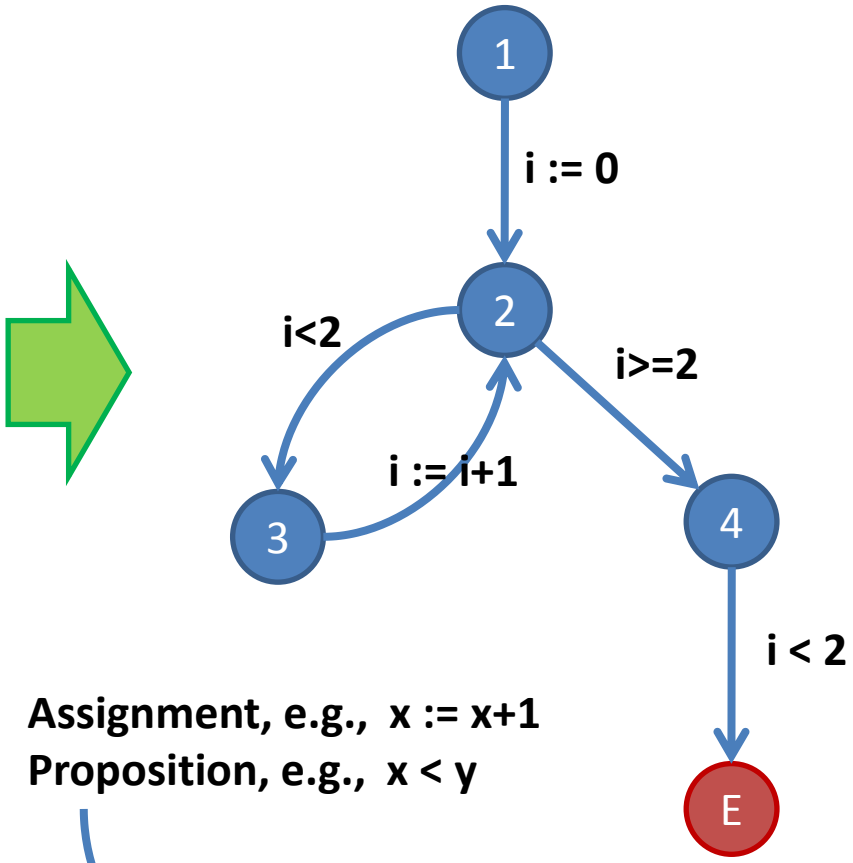
A Concrete Example

Control Flow Automata

Program

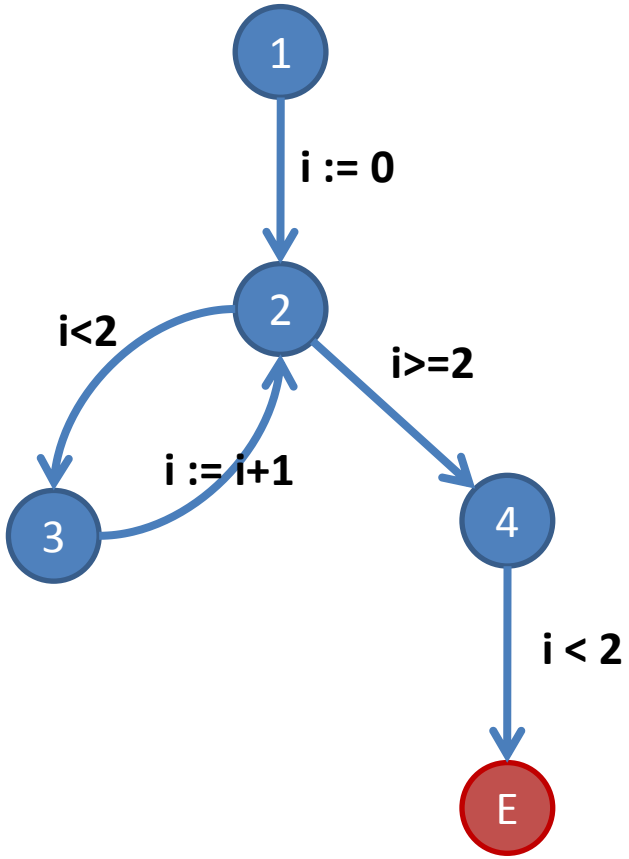
```
void main() {
    int i = 0;
    while (i < 2) {
        i++;  }
    assert (i >= 2);
}
```

Control Flow Automata

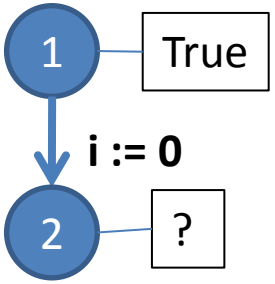


Abstract Reachability Tree (ART)

Control Flow Automata



ART



Assume that each node is associated with predicate (i < 2).

How to compute “successors” ?

```
Example ( ) {
1: do{
     lock();
     old = new;
     q = q->next;
2:   if (q != NULL){
3:     q->data = new;
       unlock();
       new ++;
     }
4:}while(new != old);
5: unlock ();
}
```


Refinement

Predicates: LOCK, new==old

Post(ϕ, op)

- Let ϕ be a formula describes a set of current states.
- Post(ϕ, op) is a formula that describes the set of next states via operation **op**.
- Recall that two types of op are considered:
  - Assignment x:=e, e.g., x:= x + 1
  - Proposition e1 < e2 or e1 == e2, e.g., x<y

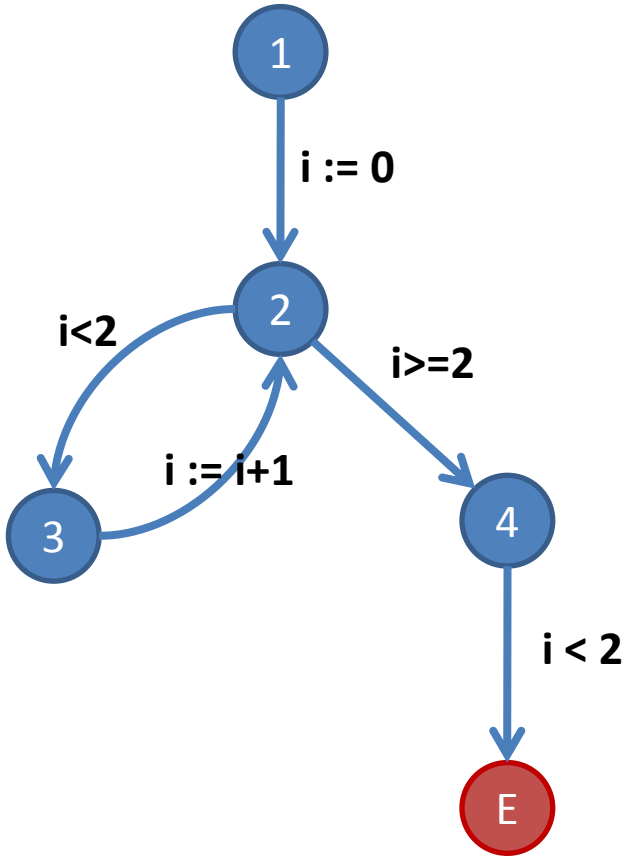
Post(ϕ, op)

- Recall that two types of op are considered:
  - Assignment x:=e, e.g., x:= x + 1
  - Proposition p, e.g., x<y
- Post(ϕ, x:=e) = ϕ[x/x'] $\wedge$ x=e[x/x']
- Post(ϕ, p) = $\phi \wedge$ p

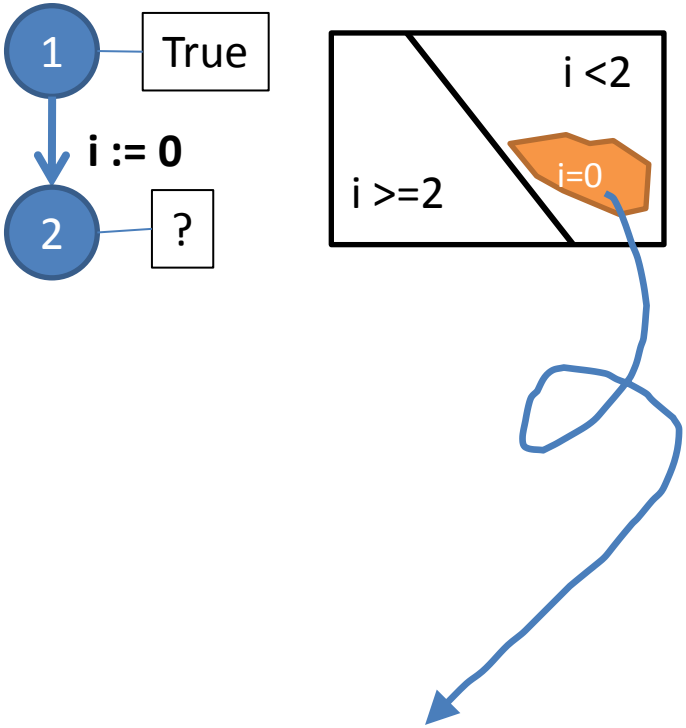
- Try Post(x>3, x:=x+3)

Abstract Reachability Tree (ART)

Control Flow Automata



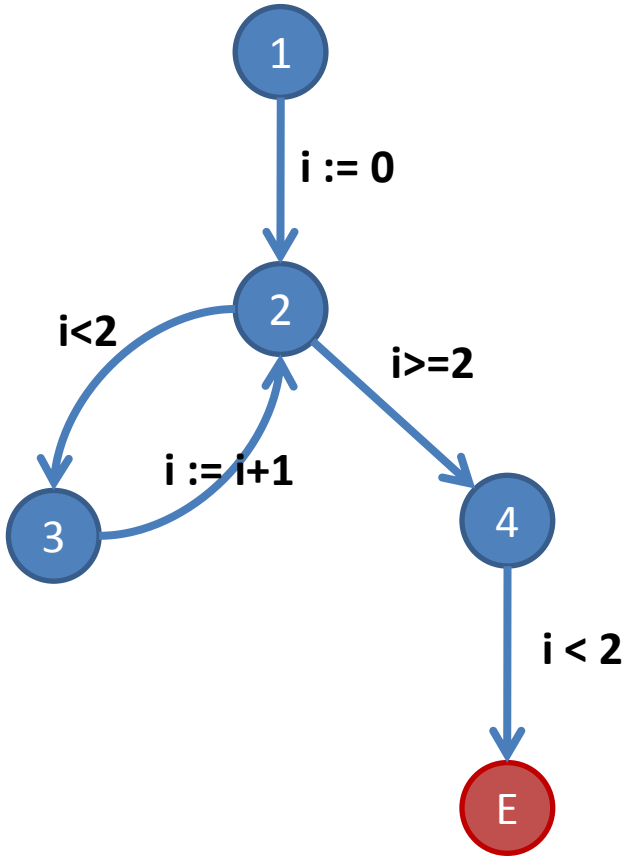
ART



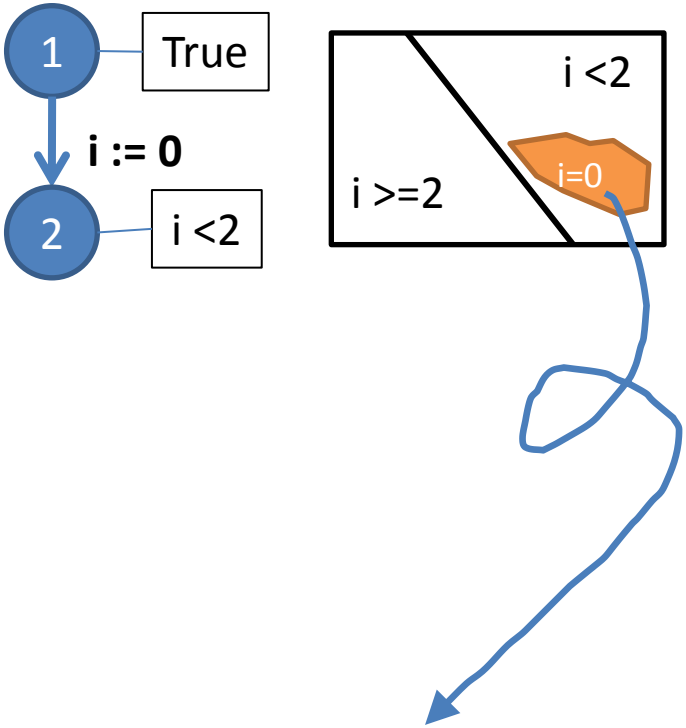
Assume that each node is associated with predicate (i < 2).

Abstract Reachability Tree (ART)

Control Flow Automata



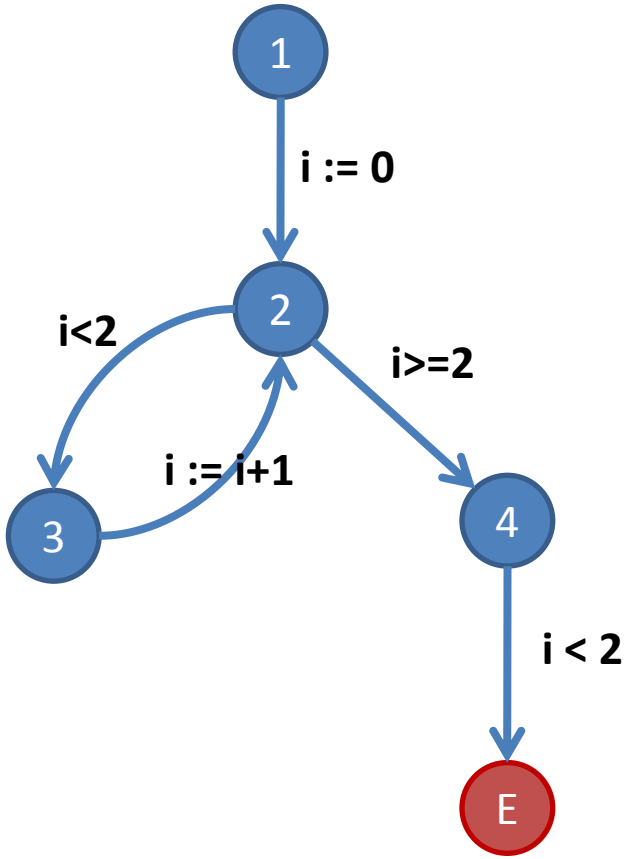
ART



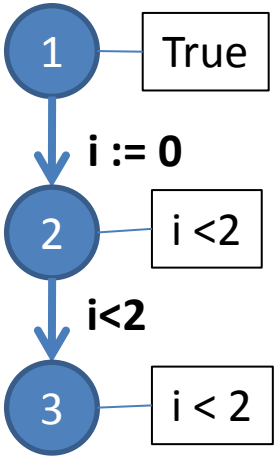
Assume that each node is associated with predicate (i < 2).

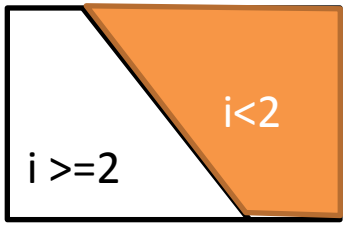
Abstract Reachability Tree (ART)

Control Flow Automata



ART



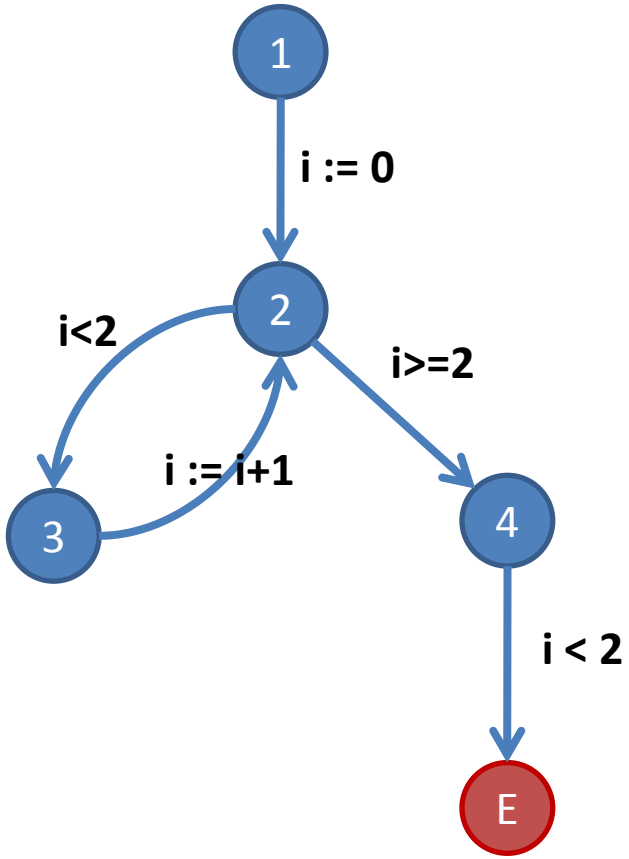


Post(i<2, i<2) = i<2 ∧ i<2

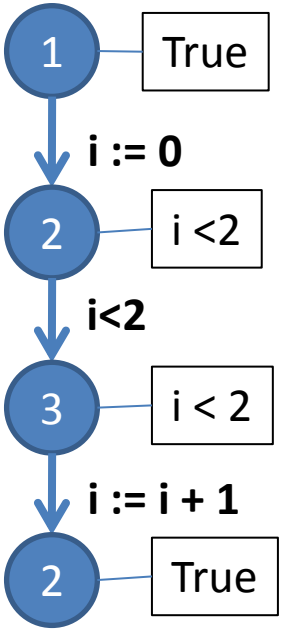
Assume that each node is associated with predicate (i < 2).

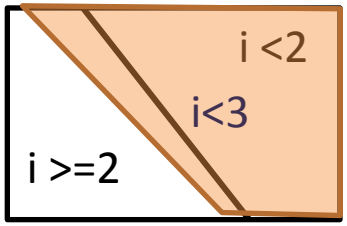
Abstract Reachability Tree (ART)

Control Flow Automata



ART



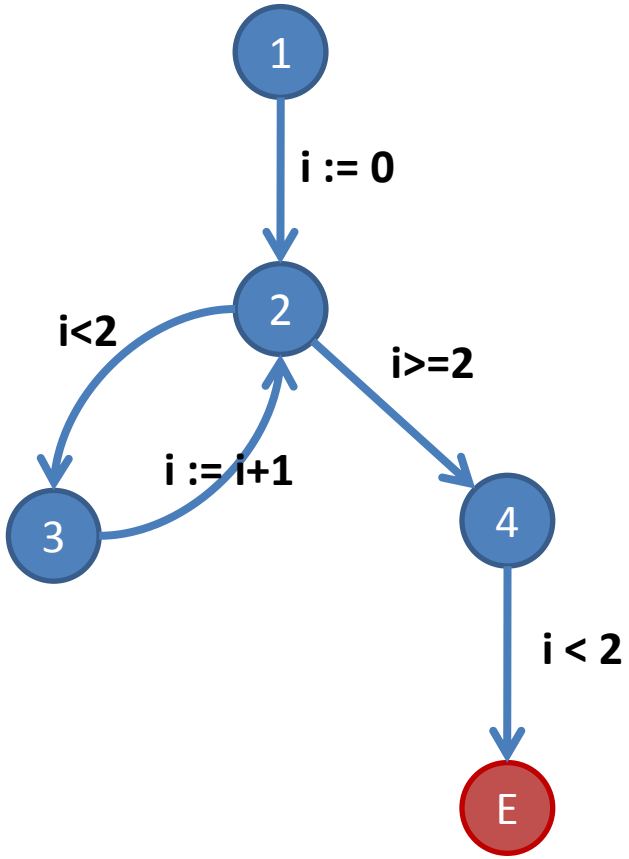


Post(i<2, i:=i+1) = i< 2[i/i'] ∧ i=i+1[i/i']
= l'<2 ∧ i=l'+1

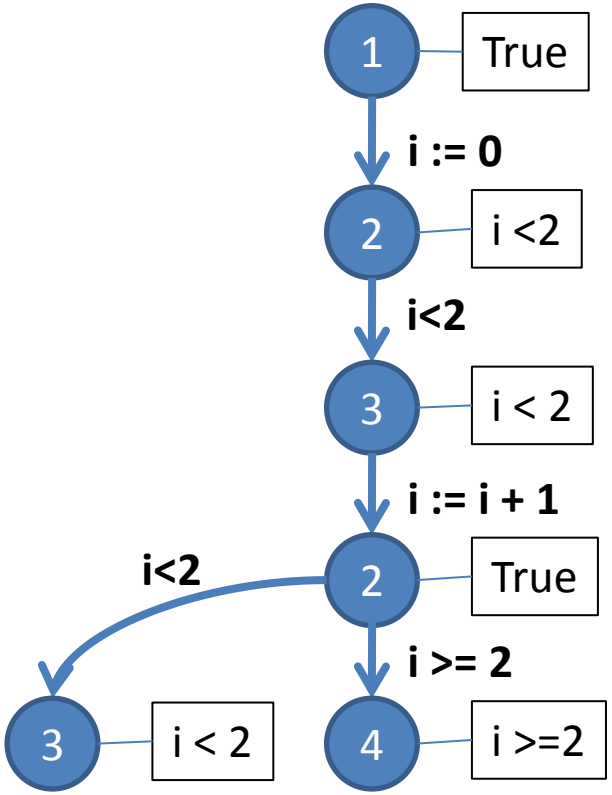
Assume that each node is associated with predicate (i < 2).

Abstract Reachability Tree (ART)

Control Flow Automata

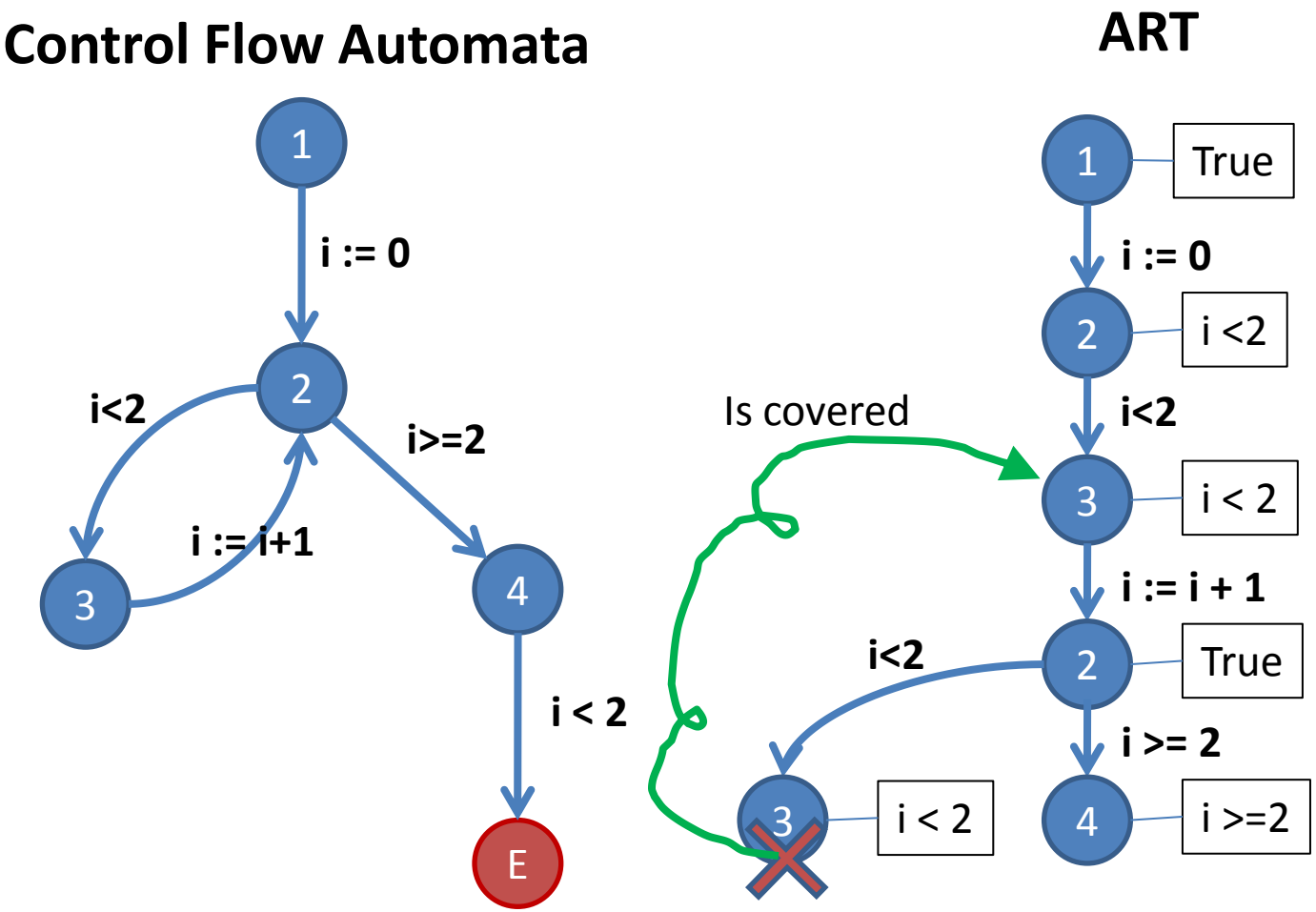


ART



Assume that each node is associated with predicate (i < 2).

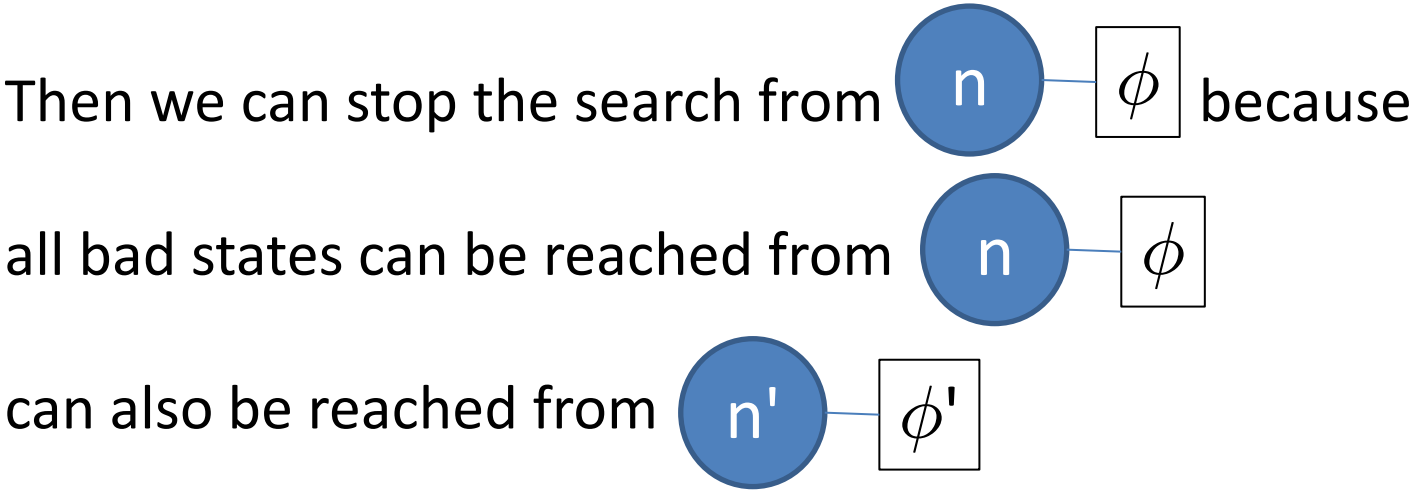
Abstract Reachability Tree (ART)



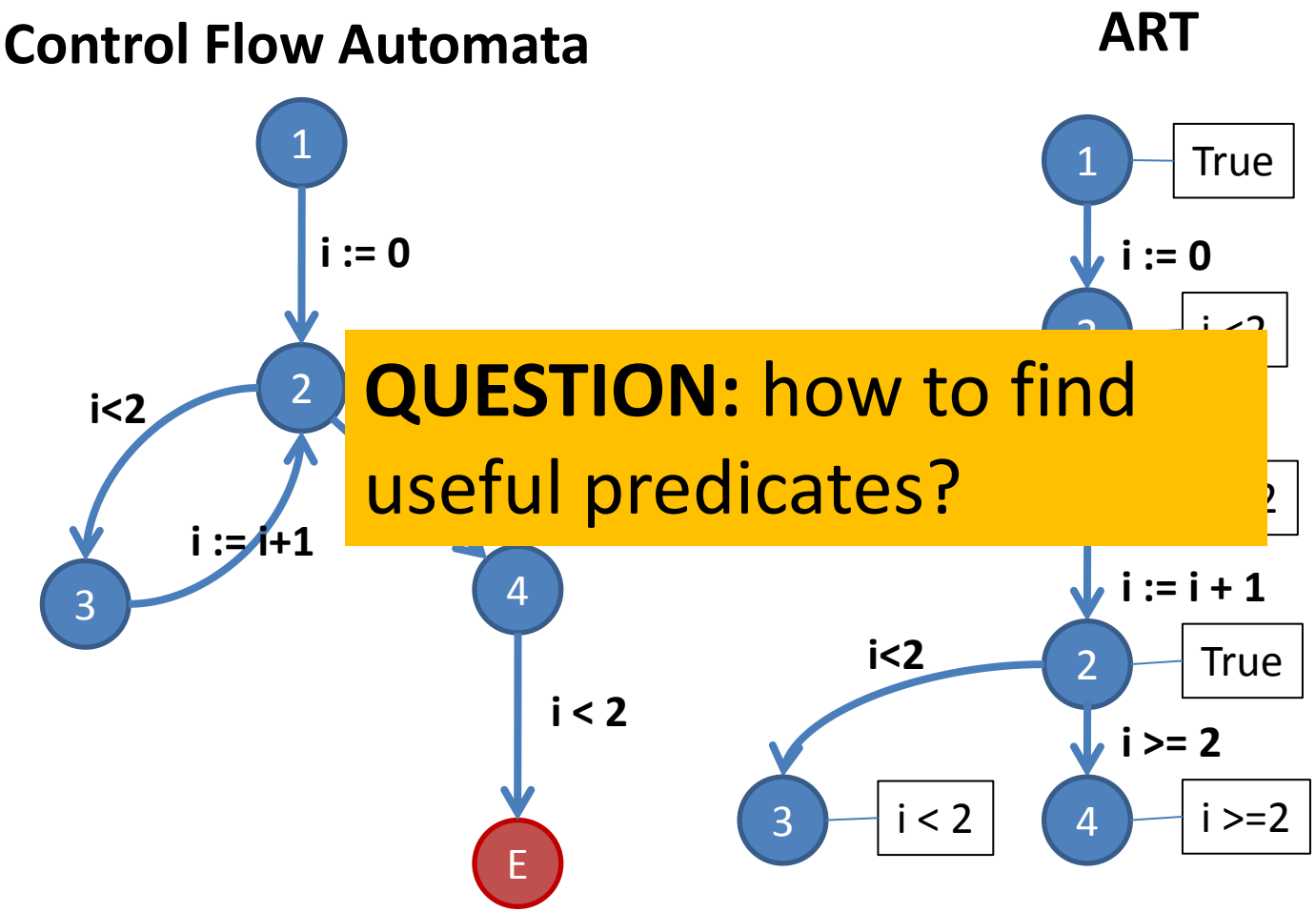
Assume that each node is associated with predicate (i < 2).

Cover

iff (1) n=n' (2) $\phi \rightarrow \phi'$



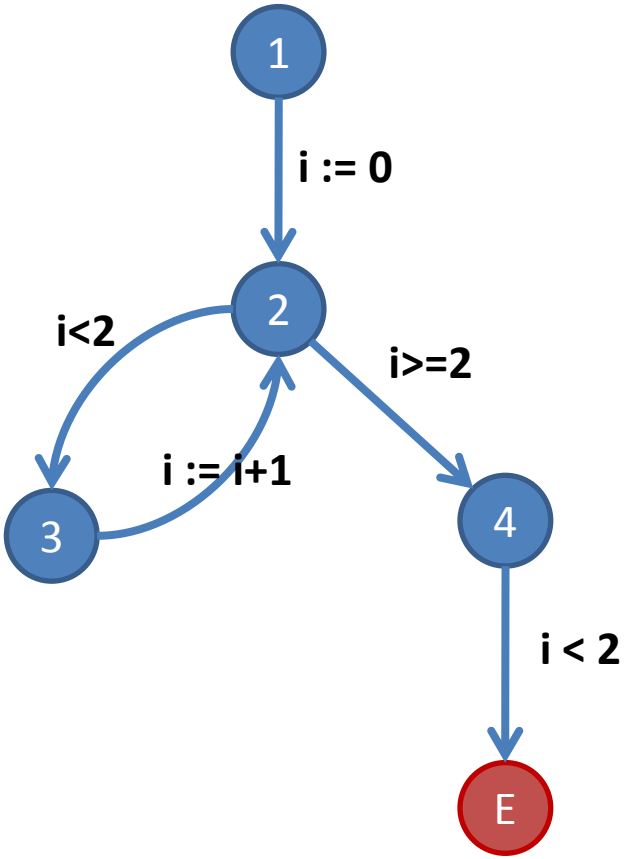
Abstract Reachability Tree (ART)



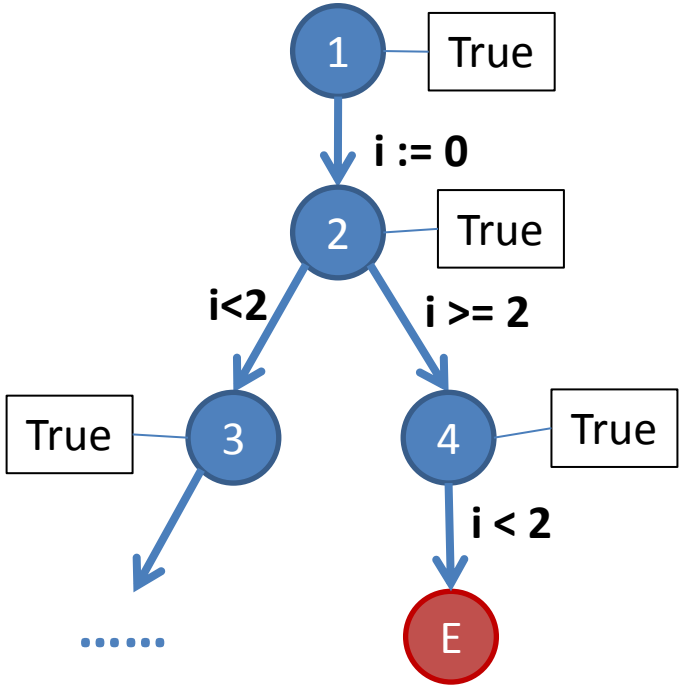
Assume that each node is associated with predicate (i < 2).

Lazy Abstraction (1)

Control Flow Automata



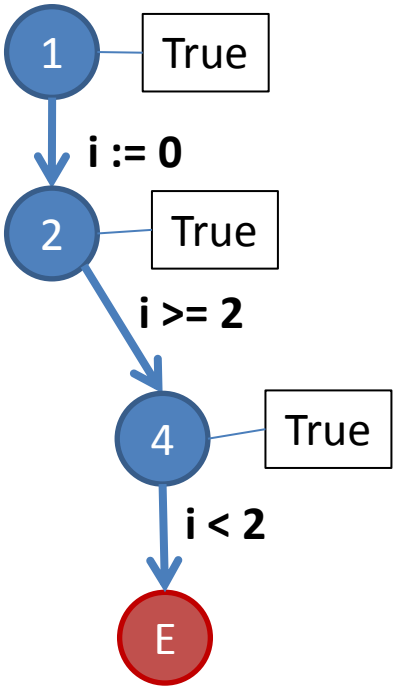
ART



Initially, we do not have any predicates

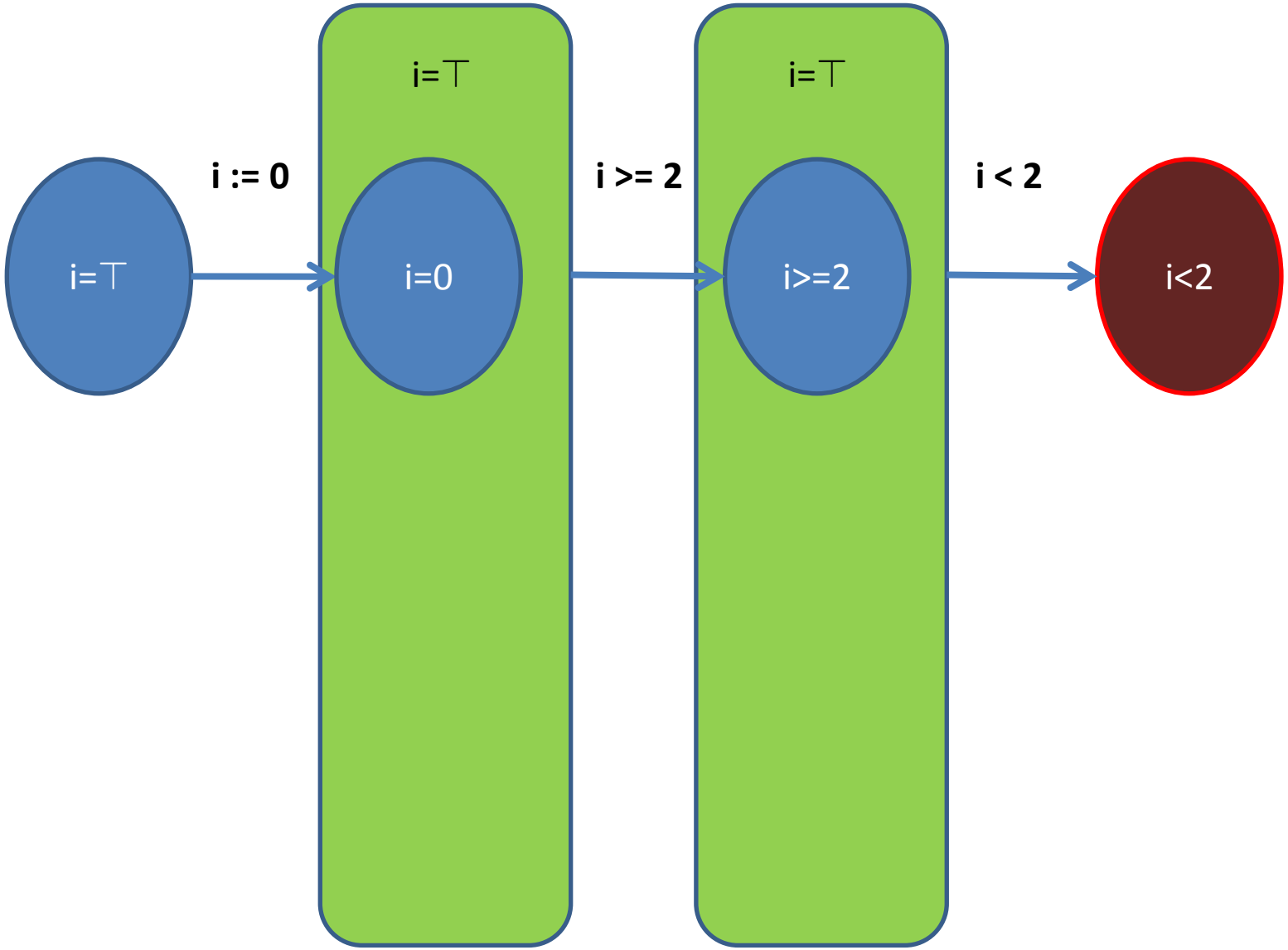
Lazy Abstraction (2)

False Counterexample

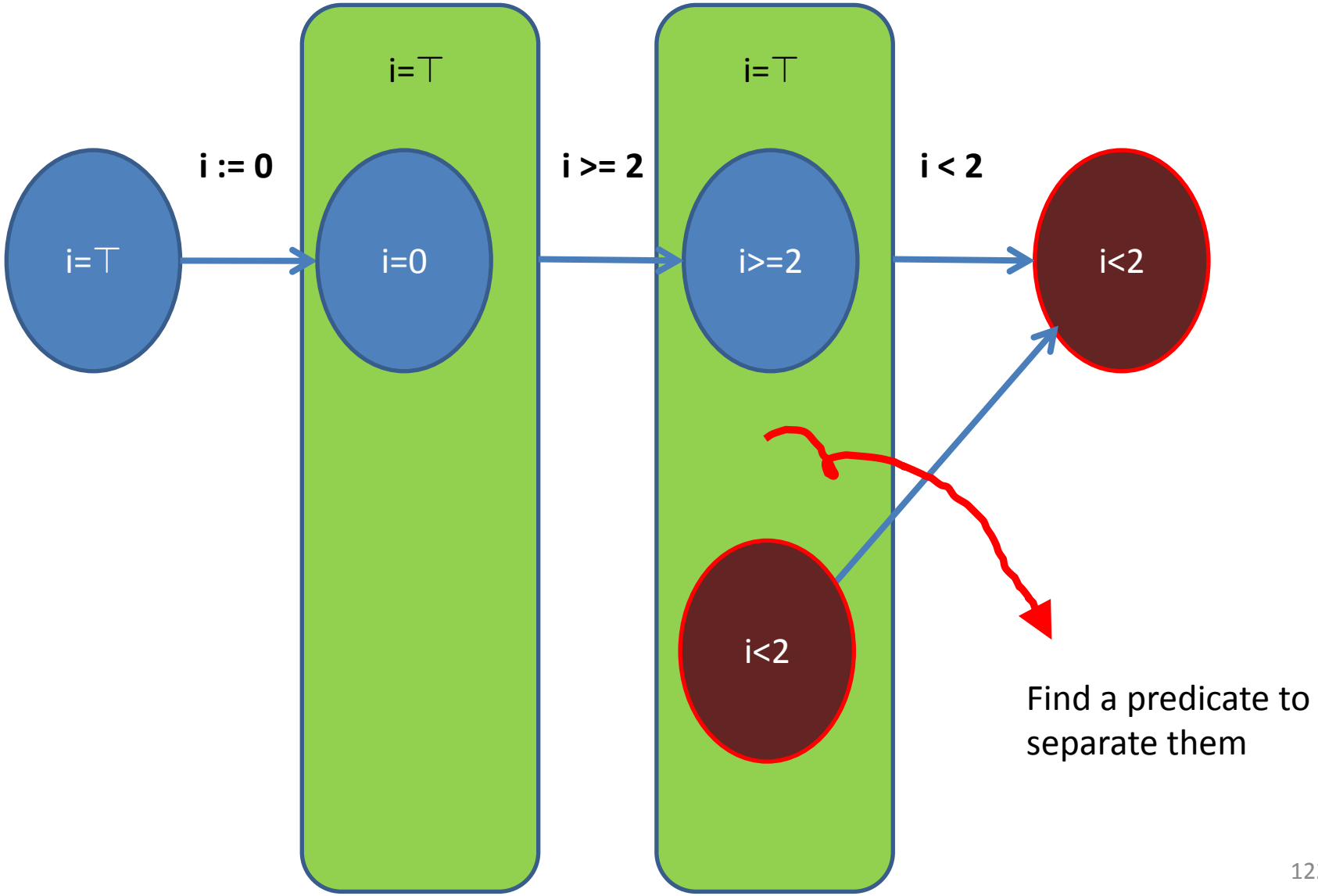


We want to find a predicate that will remove this false counterexample

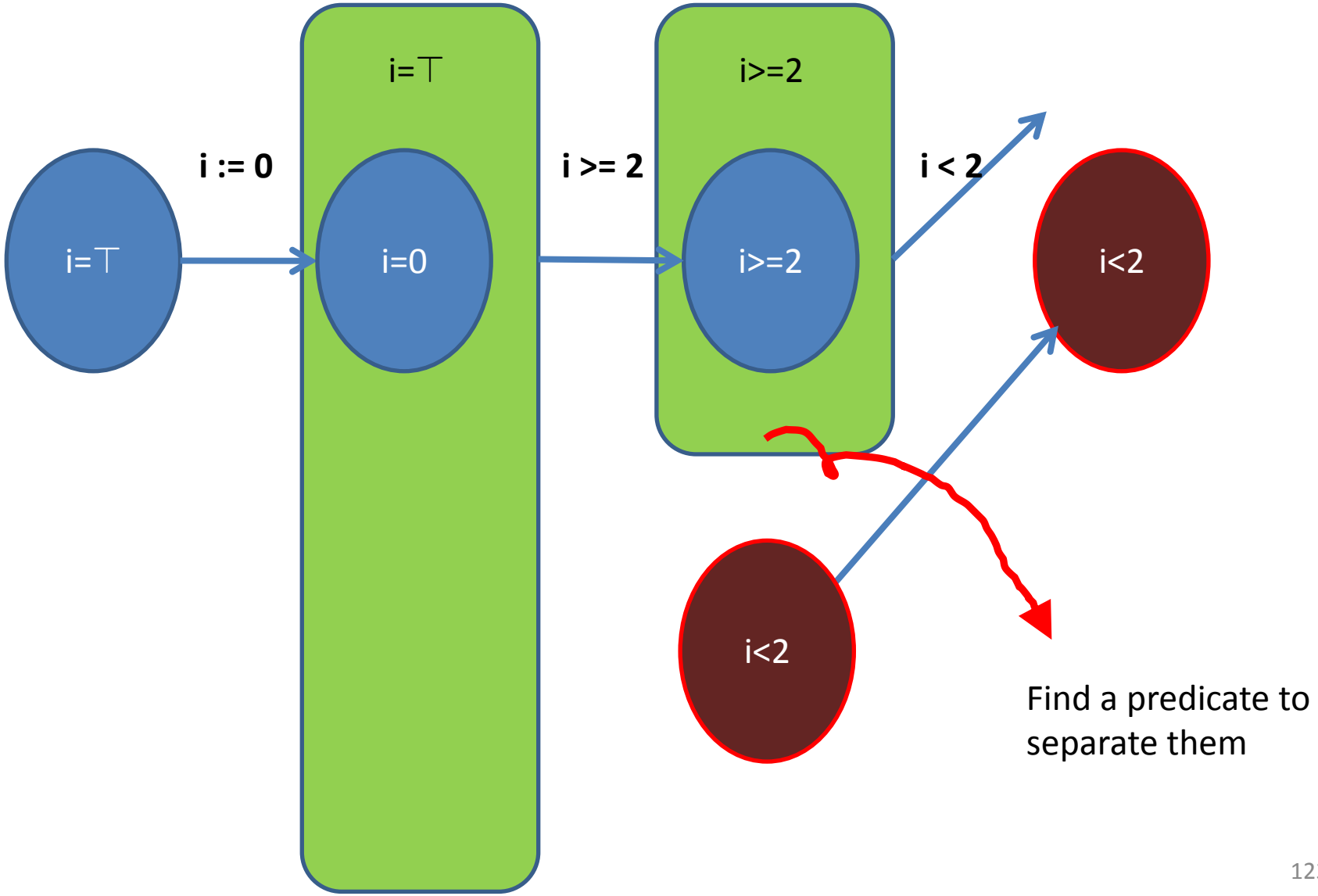
A More Detailed Illustration



A More Detailed Illustration

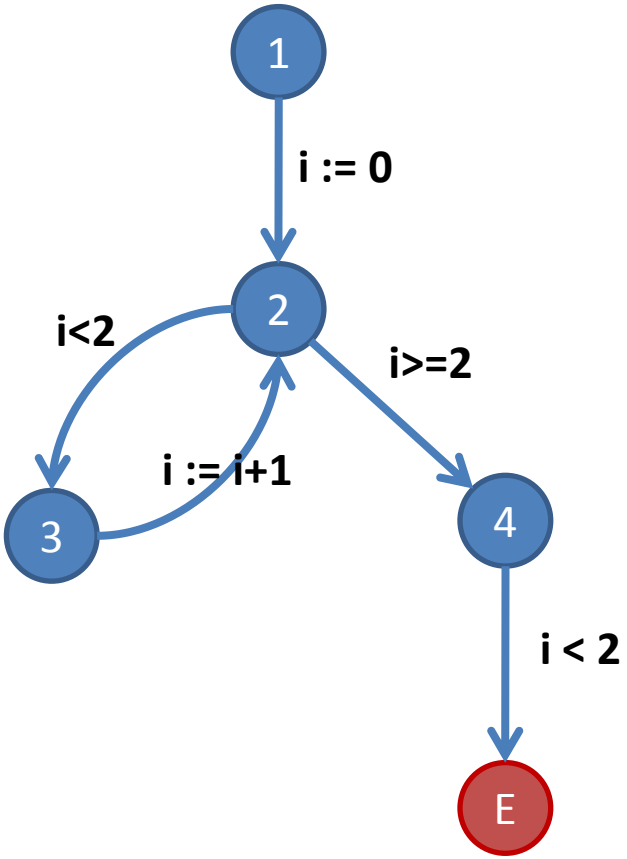


A More Detailed Illustration

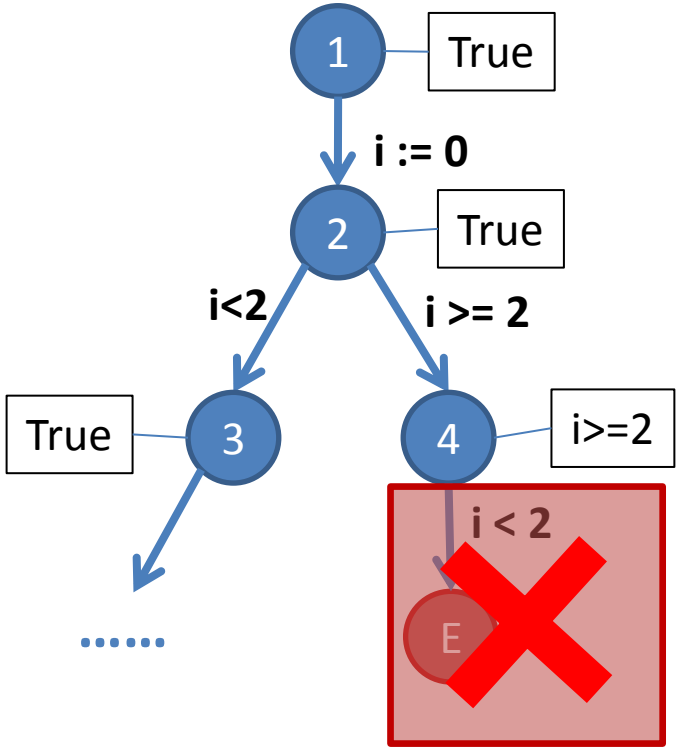


Lazy Abstraction

Control Flow Automata



ART



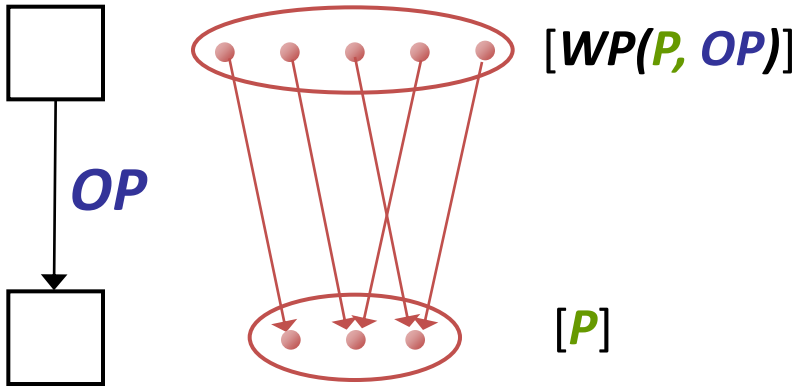
Weakest Preconditions

WP(P,OP)

Weakest formula *P'* s.t.

if *P'* is true before *OP*

then *P* is true after *OP*



Assign

`x = e`

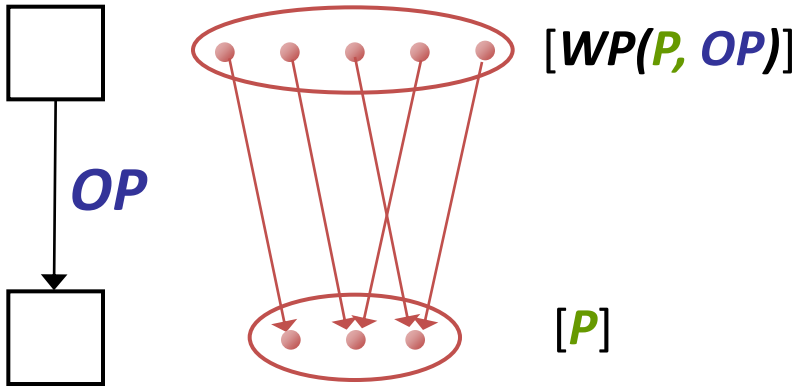
Weakest Preconditions

WP(P,OP)

Weakest formula *P'* s.t.

if *P'* is true before *OP*

then *P* is true after *OP*



□

Assign

`x = e`

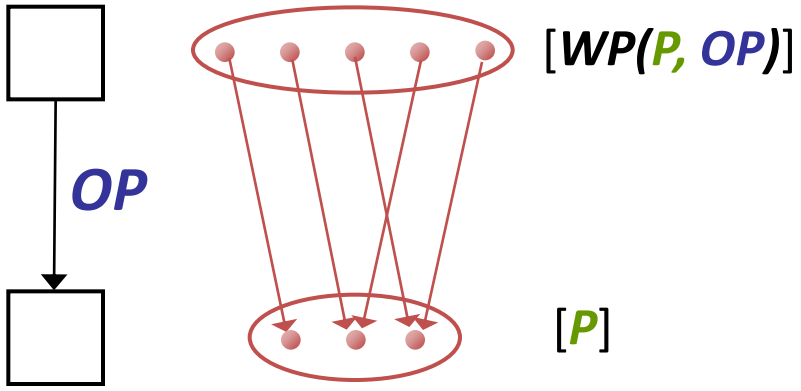
Weakest Preconditions

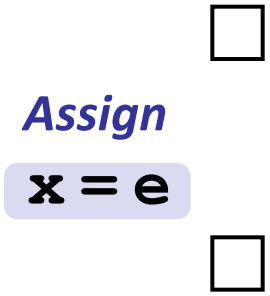
WP(P,OP)

Weakest formula *P'* s.t.

if *P'* is true before *OP*

then *P* is true after *OP*





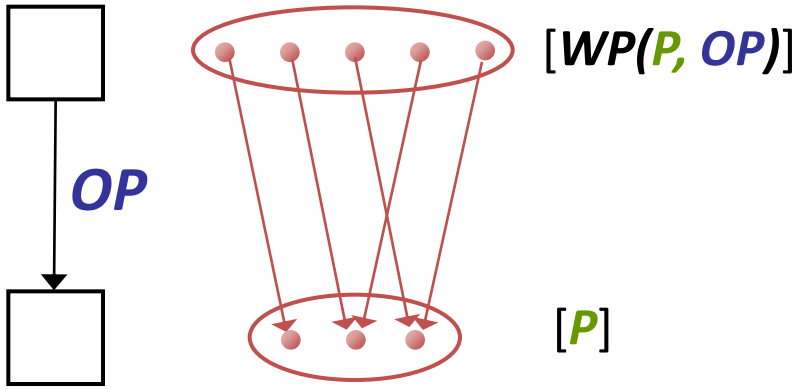
Weakest Preconditions

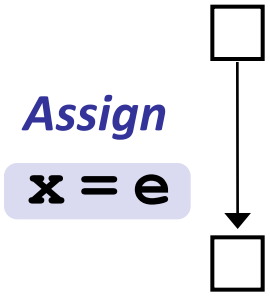
WP(P,OP)

Weakest formula *P'* s.t.

if *P'* is true before *OP*

then *P* is true after *OP*





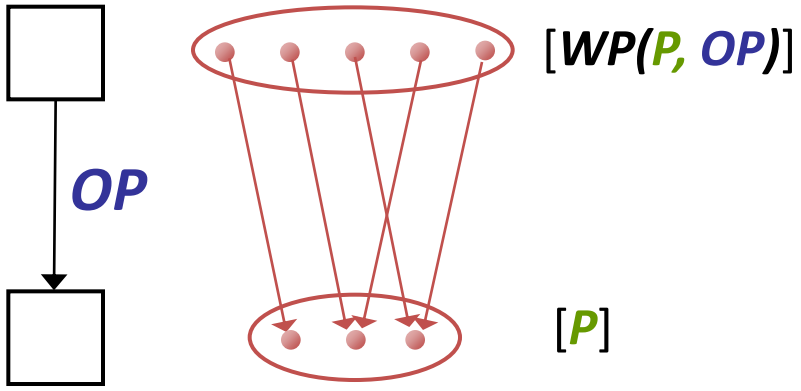
Weakest Preconditions

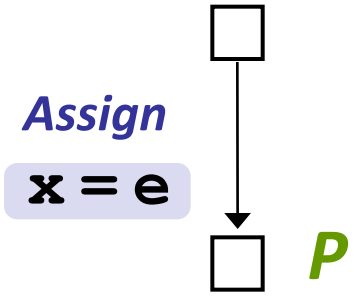
WP(P,OP)

Weakest formula *P'* s.t.

if *P'* is true before *OP*

then *P* is true after *OP*





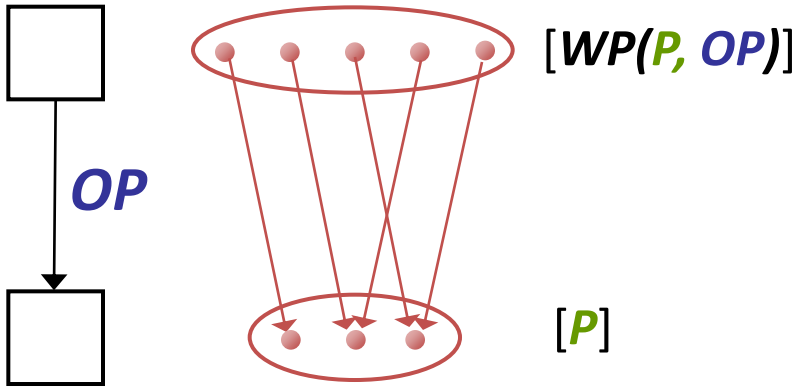
Weakest Preconditions

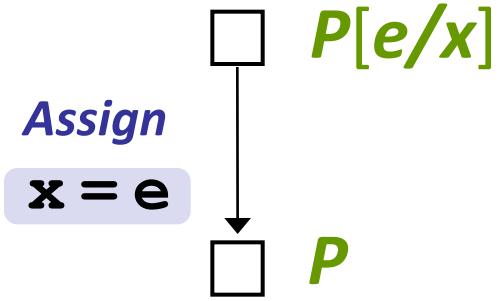
WP(P,OP)

Weakest formula *P'* s.t.

if *P'* is true before *OP*

then *P* is true after *OP*





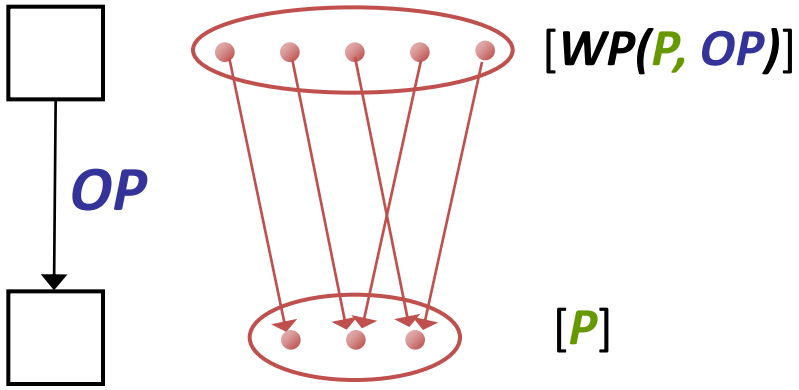
Weakest Preconditions

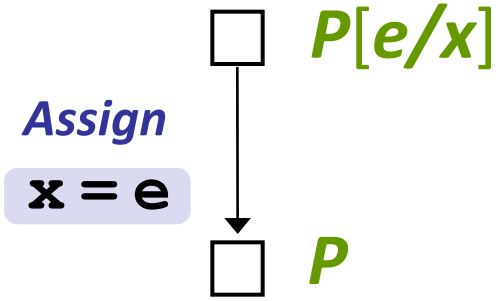
WP(P,OP)

Weakest formula *P'* s.t.

if *P'* is true before *OP*

then *P* is true after *OP*





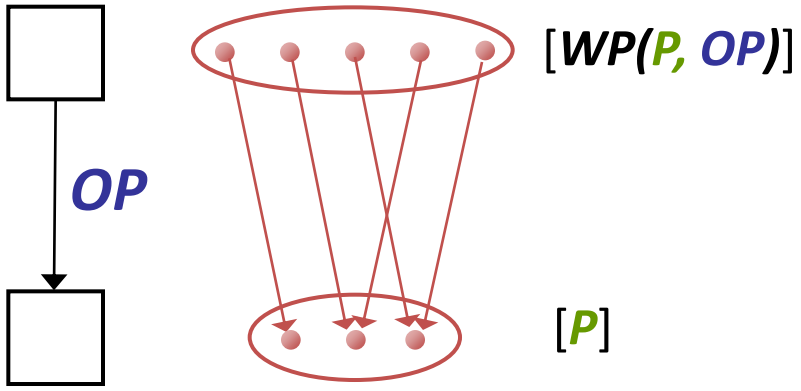
Weakest Preconditions

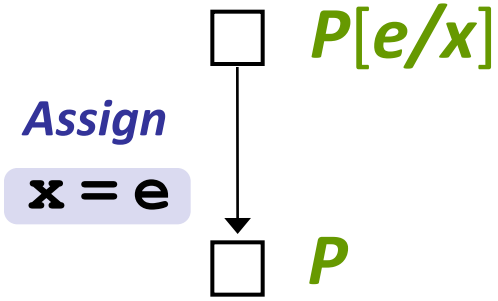
WP(P,OP)

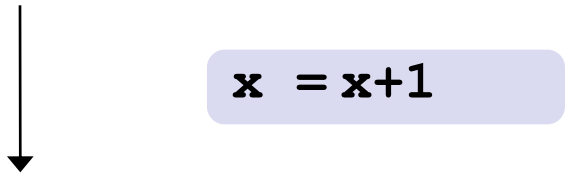
Weakest formula *P'* s.t.

if *P'* is true before *OP*

then *P* is true after *OP*







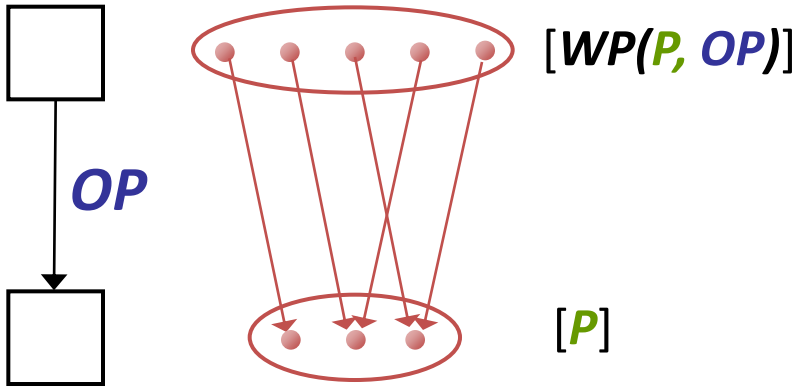
Weakest Preconditions

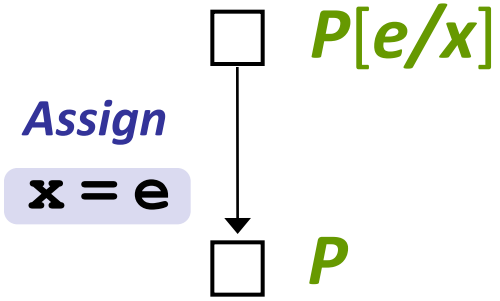
WP(P,OP)

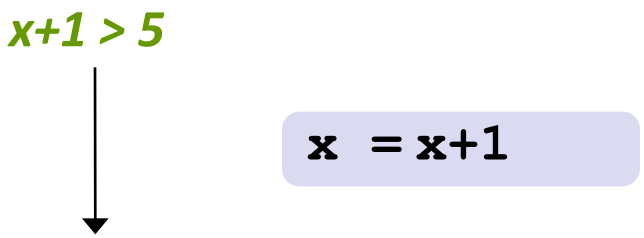
Weakest formula *P'* s.t.

if *P'* is true before *OP*

then *P* is true after *OP*







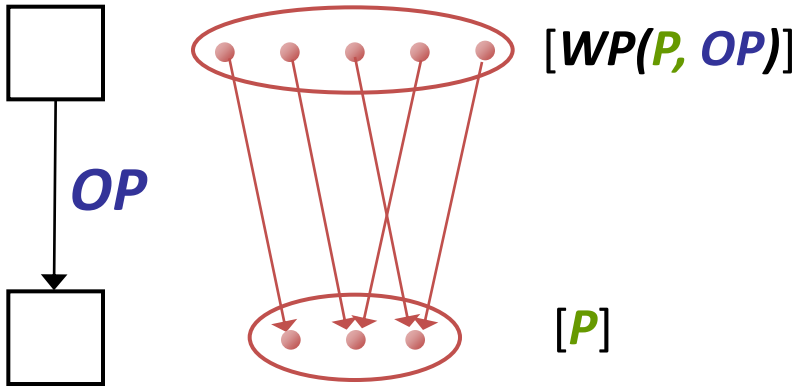
Weakest Preconditions

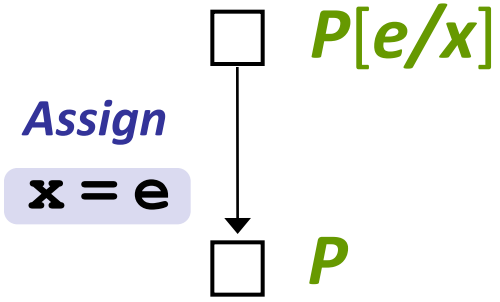
WP(P,OP)

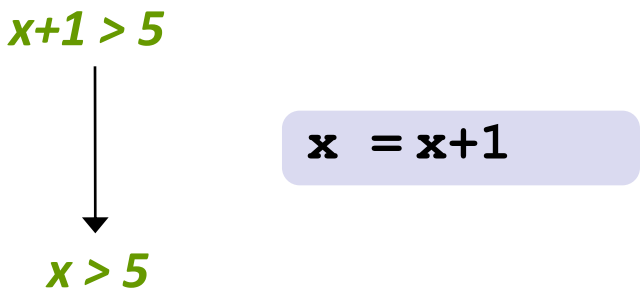
Weakest formula *P'* s.t.

if *P'* is true before *OP*

then *P* is true after *OP*







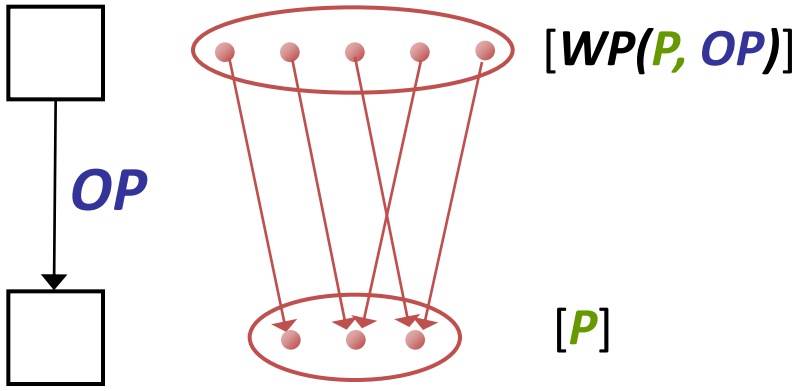
Weakest Preconditions

WP(P,OP)

Weakest formula *P'* s.t.

if *P'* is true before *OP*

then *P* is true after *OP*



Assign

`x = e`

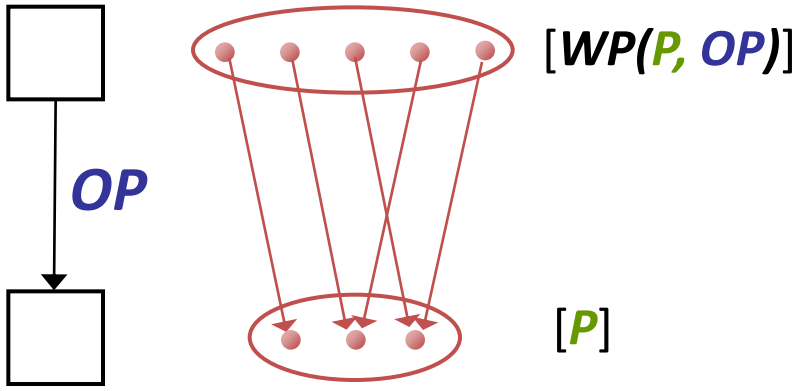
Weakest Preconditions

WP(P,OP)

Weakest formula *P'* s.t.

if *P'* is true before *OP*

then *P* is true after *OP*



□

Assign

`x = e`

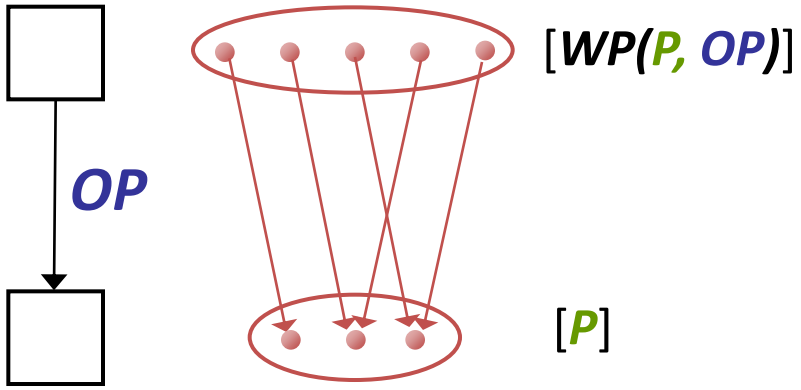
Weakest Preconditions

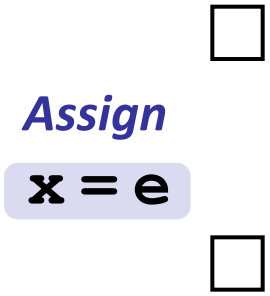
WP(P,OP)

Weakest formula *P'* s.t.

if *P'* is true before *OP*

then *P* is true after *OP*





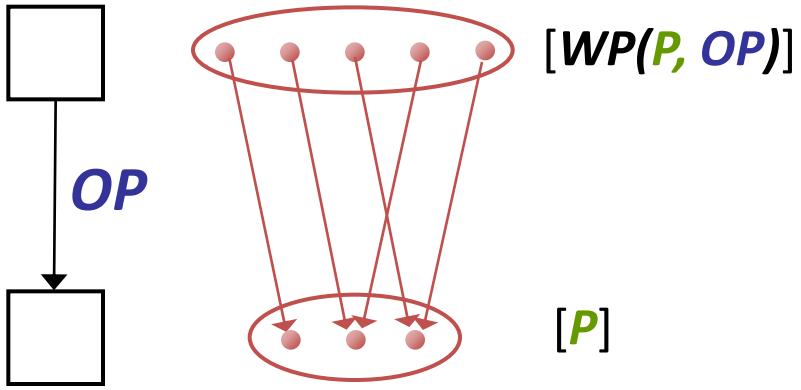
Weakest Preconditions

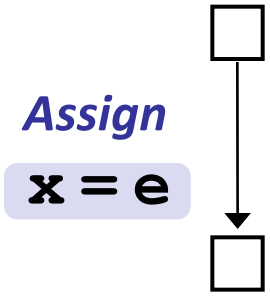
WP(P,OP)

Weakest formula *P'* s.t.

if *P'* is true before *OP*

then *P* is true after *OP*





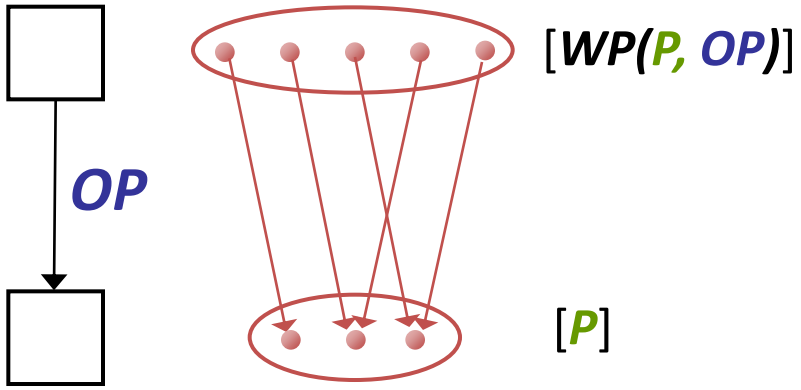
Weakest Preconditions

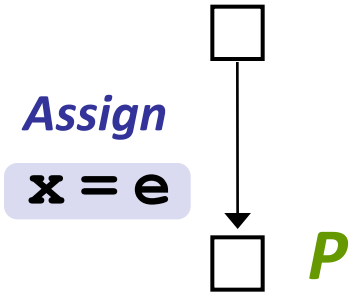
WP(P,OP)

Weakest formula *P'* s.t.

if *P'* is true before *OP*

then *P* is true after *OP*





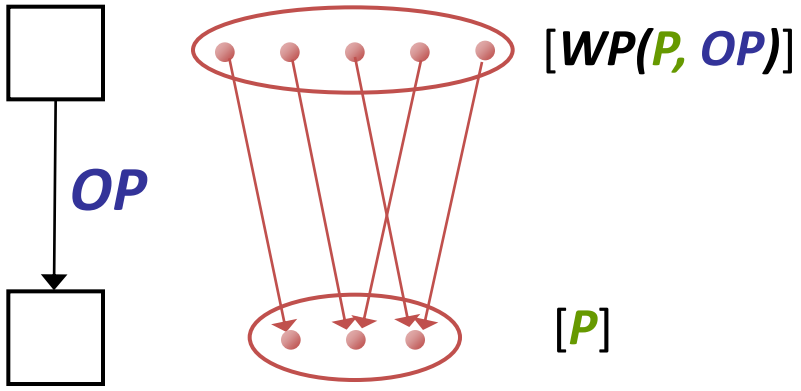
Weakest Preconditions

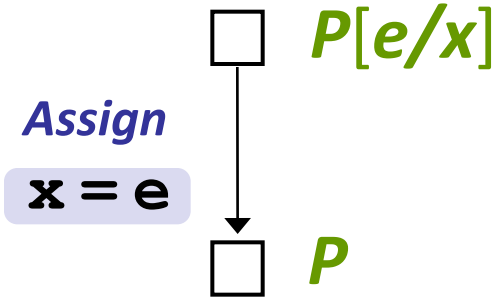
WP(P,OP)

Weakest formula *P'* s.t.

if *P'* is true before *OP*

then *P* is true after *OP*





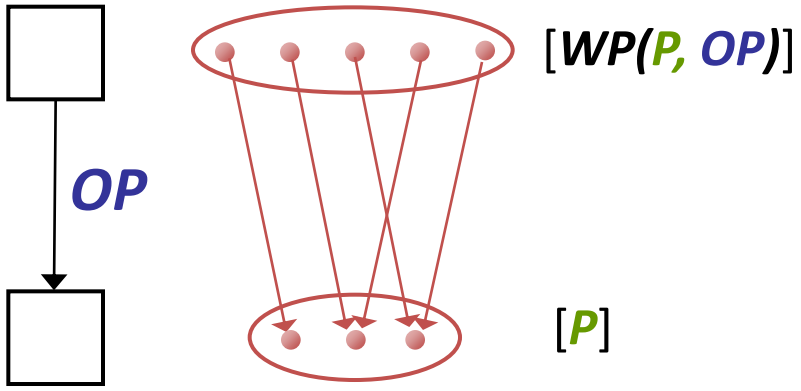
Weakest Preconditions

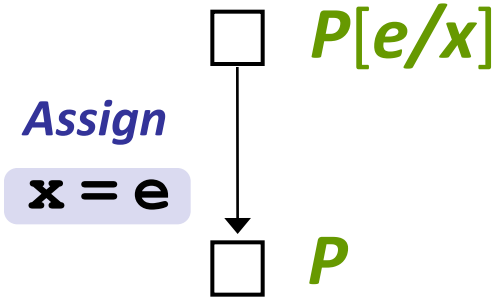
WP(P,OP)

Weakest formula *P'* s.t.

if *P'* is true before *OP*

then *P* is true after *OP*





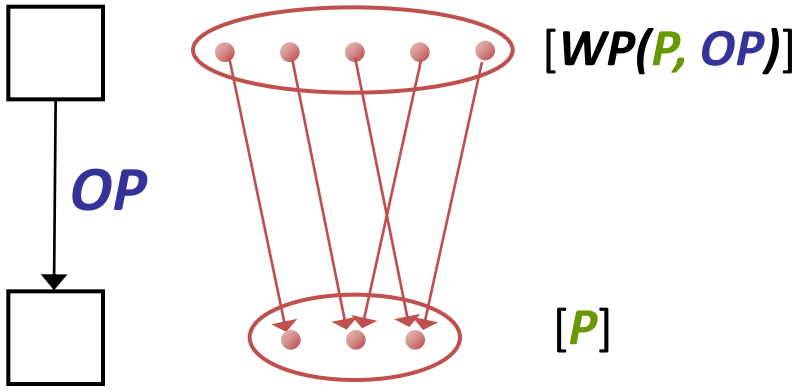
Weakest Preconditions

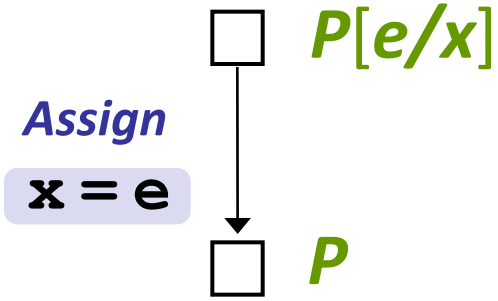
WP(P,OP)

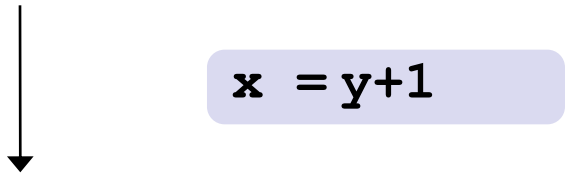
Weakest formula *P'* s.t.

if *P'* is true before *OP*

then *P* is true after *OP*







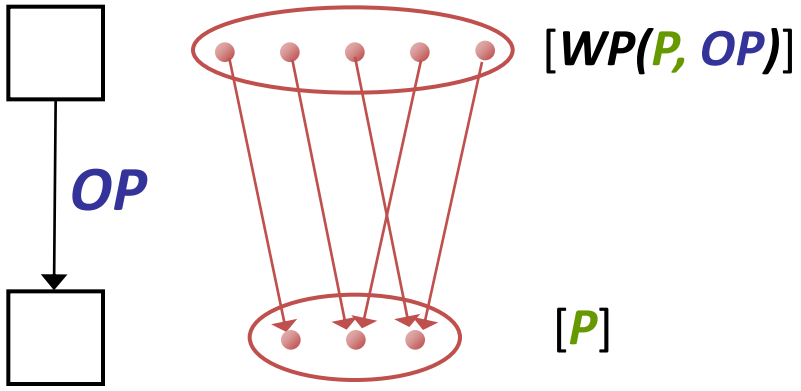
Weakest Preconditions

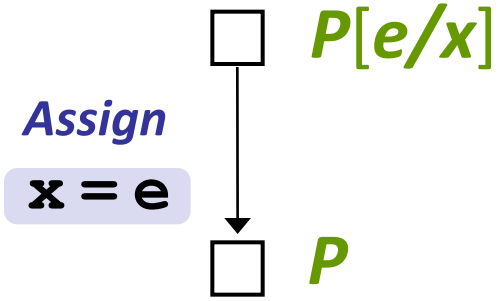
WP(P,OP)

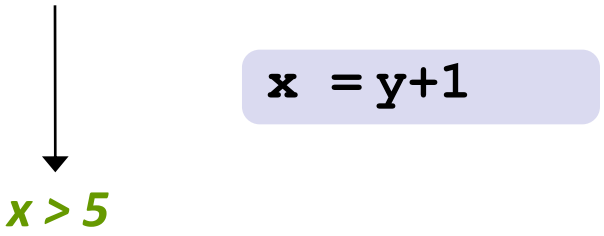
Weakest formula *P'* s.t.

if *P'* is true before *OP*

then *P* is true after *OP*







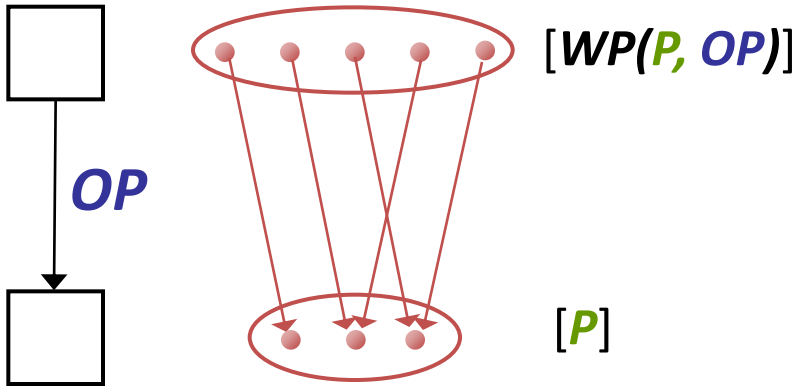
Weakest Preconditions

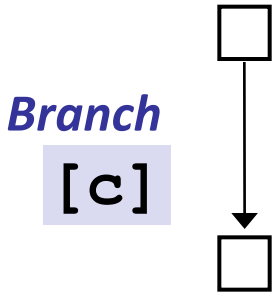
WP(P,OP)

Weakest formula *P'* s.t.

if *P'* is true before *OP*

then *P* is true after *OP*





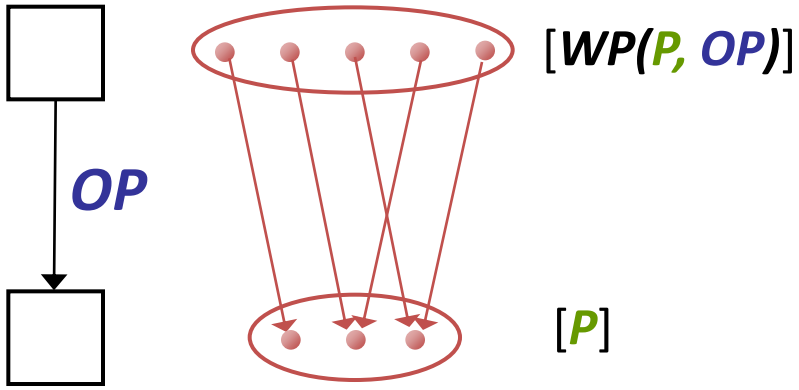
Weakest Preconditions

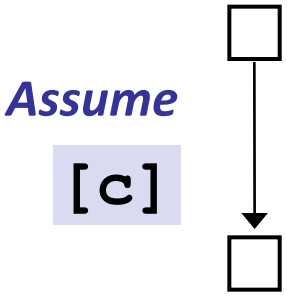
WP(P,OP)

Weakest formula *P'* s.t.

if *P'* is true before *OP*

then *P* is true after *OP*





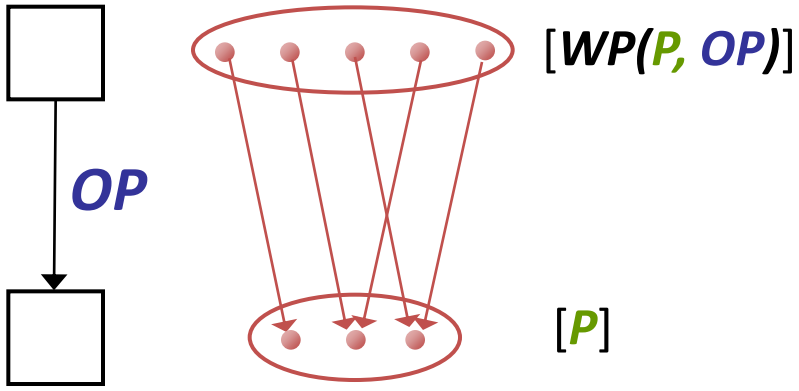
Weakest Preconditions

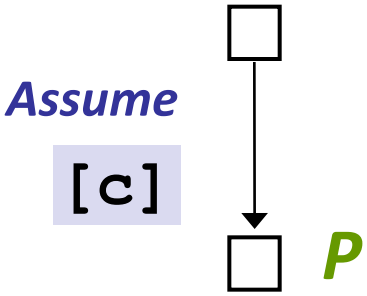
WP(P,OP)

Weakest formula *P'* s.t.

if *P'* is true before *OP*

then *P* is true after *OP*





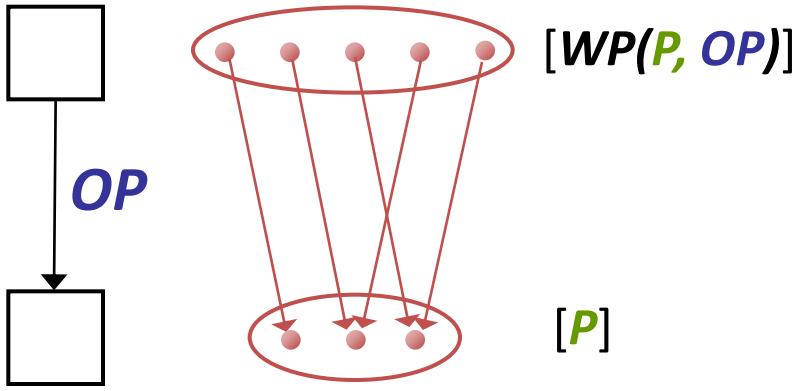
Weakest Preconditions

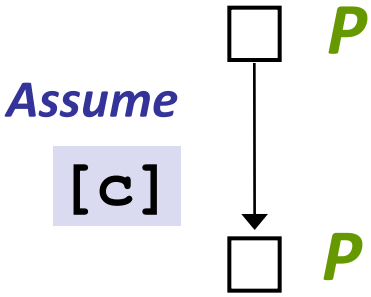
WP(P,OP)

Weakest formula *P'* s.t.

if *P'* is true before *OP*

then *P* is true after *OP*





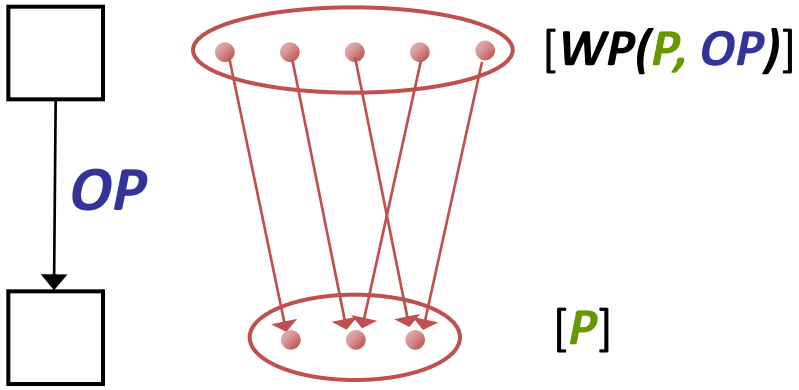
Weakest Preconditions

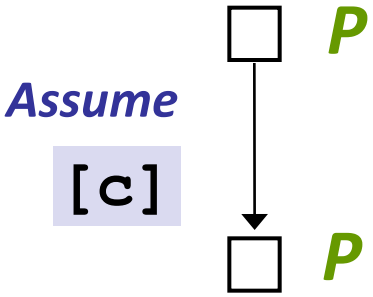
WP(P,OP)

Weakest formula *P'* s.t.

if *P'* is true before *OP*

then *P* is true after *OP*





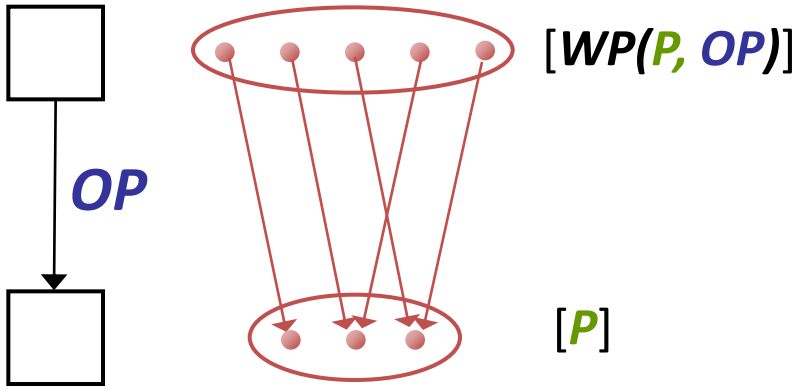
Weakest Preconditions

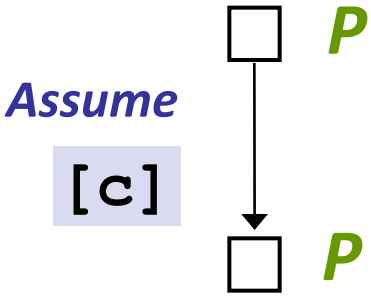
WP(P,OP)

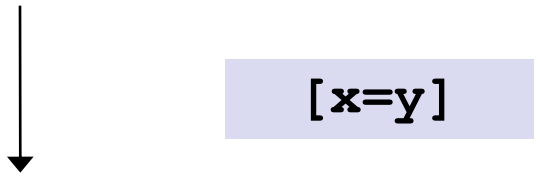
Weakest formula *P'* s.t.

if *P'* is true before *OP*

then *P* is true after *OP*







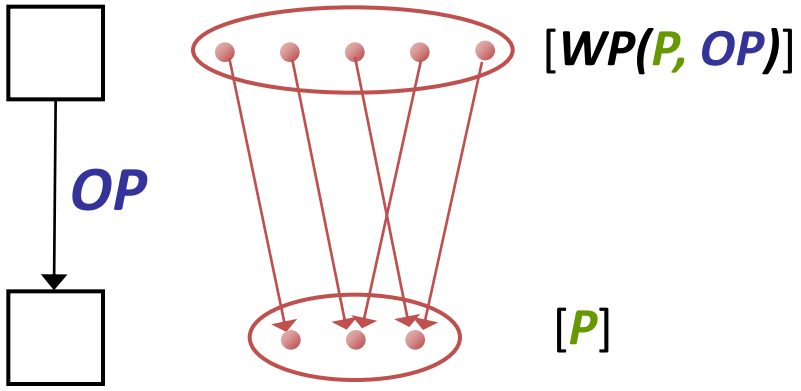
Weakest Preconditions

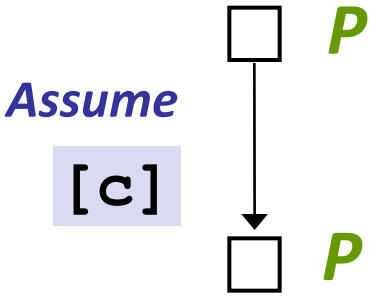
WP(P,OP)

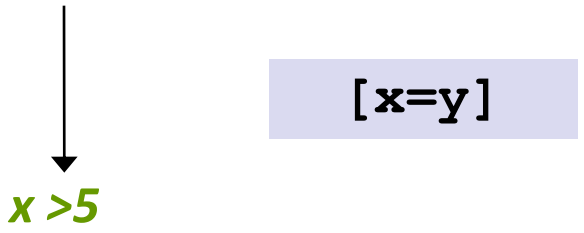
Weakest formula *P'* s.t.

if *P'* is true before *OP*

then *P* is true after *OP*







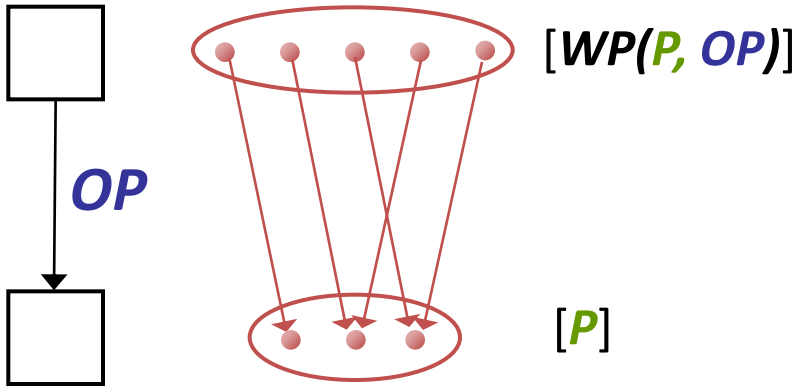
Weakest Preconditions

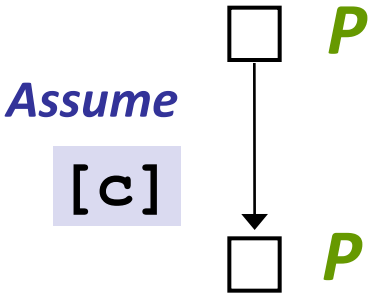
WP(P,OP)

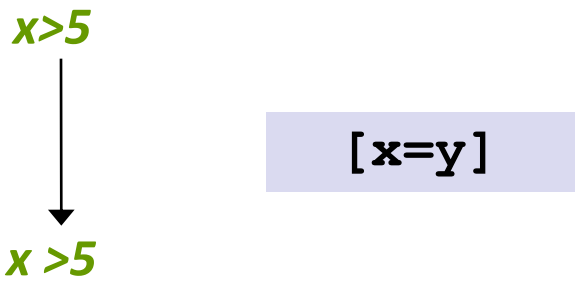
Weakest formula *P'* s.t.

if *P'* is true before *OP*

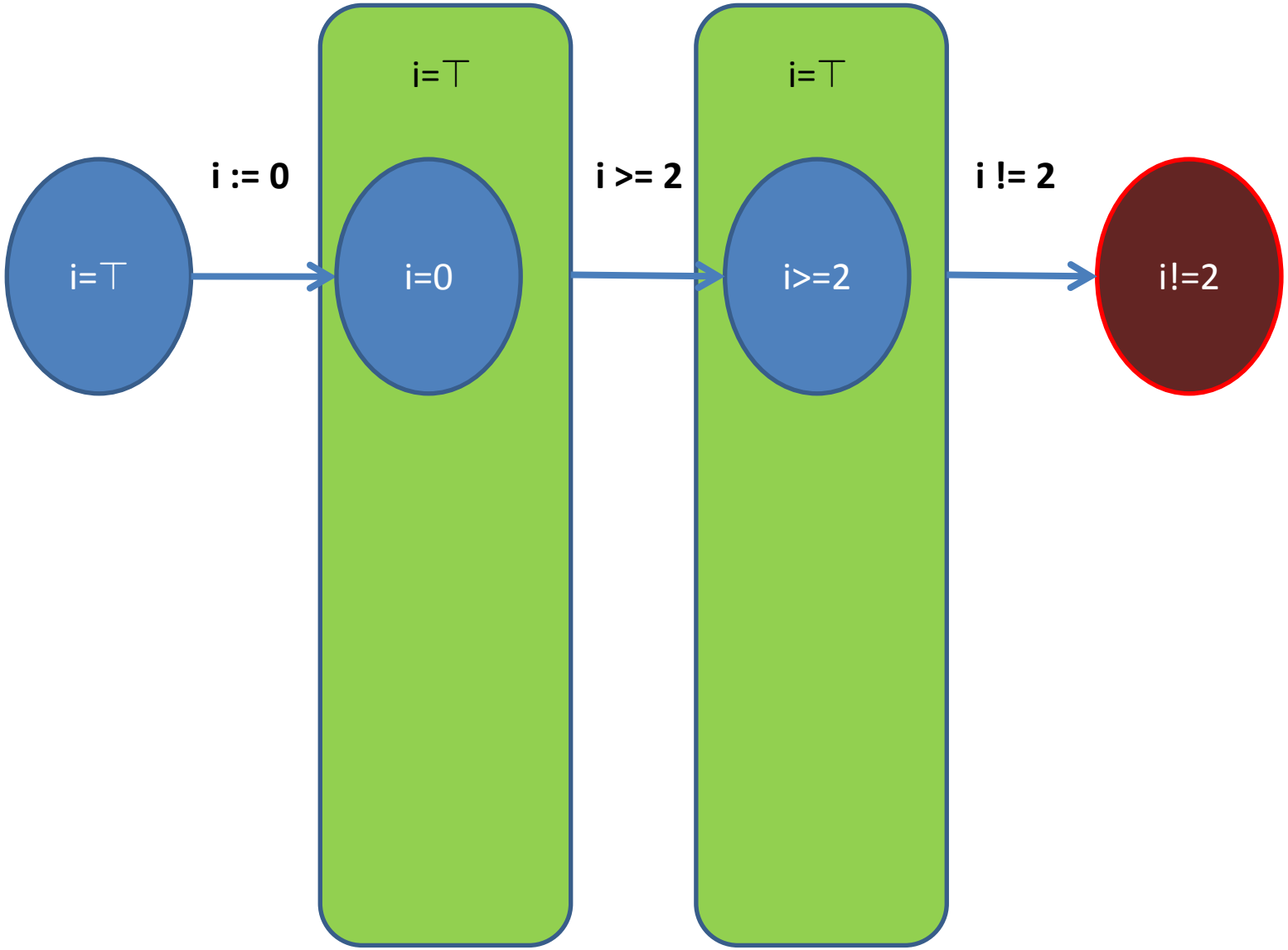
then *P* is true after *OP*



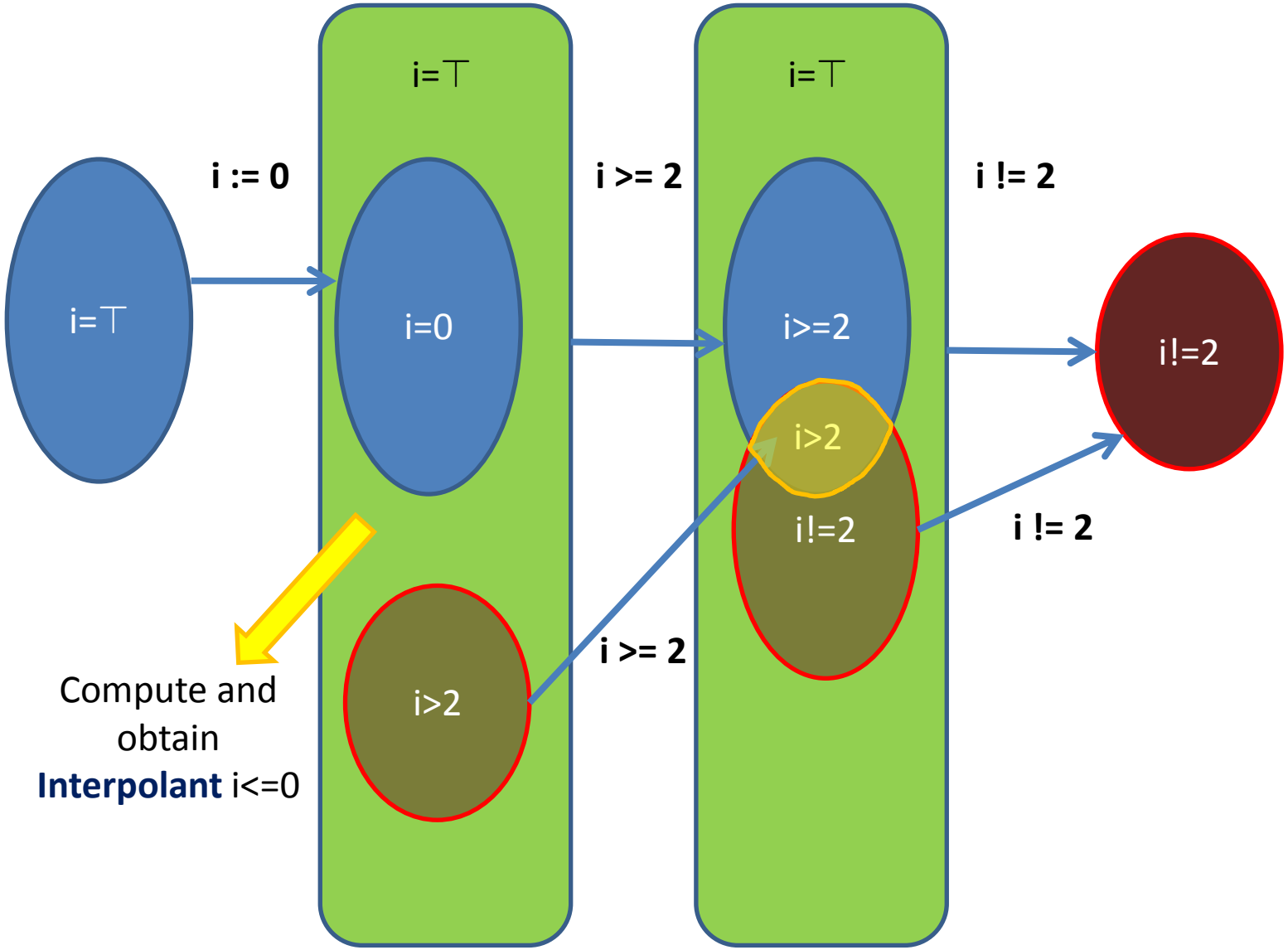




Another Example

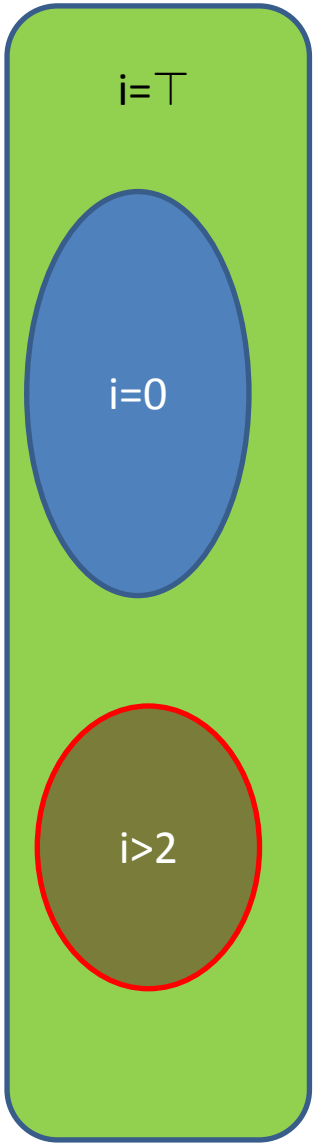


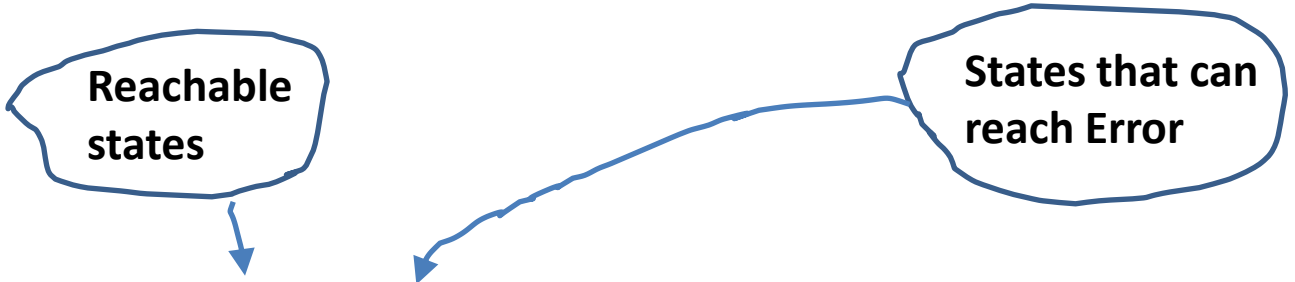
Another Example



Craig Interpolant

(Craig,57)





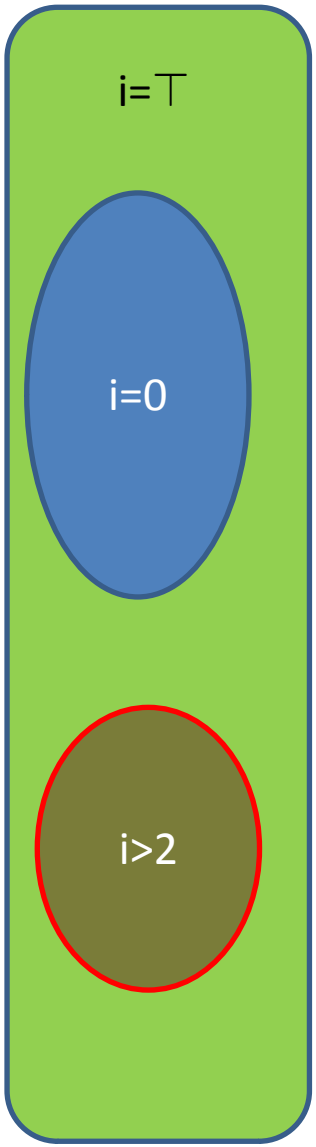
Let **R** and **E** be FOL formulae, if **R** ∧ **E** Is UNSAT., then there exists a first order formula **I** (interpolant) such that .

(a) **R** ⇒ **I**

(b) **E** ∧ **I** is UNSAT

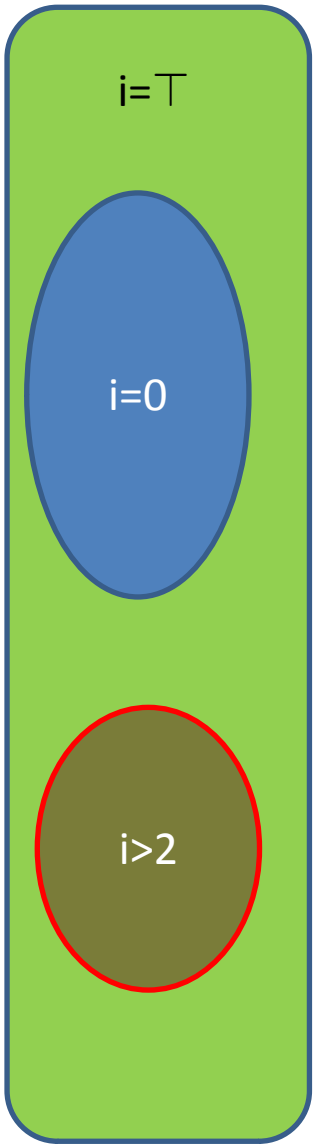
(c) Var(**I**) = Var (**R**) ∩ Var(**E**)

Compute Interpolant using Princess



- (i=0) ∧ (i>2) is UNSAT

Compute Interpolant using Princess

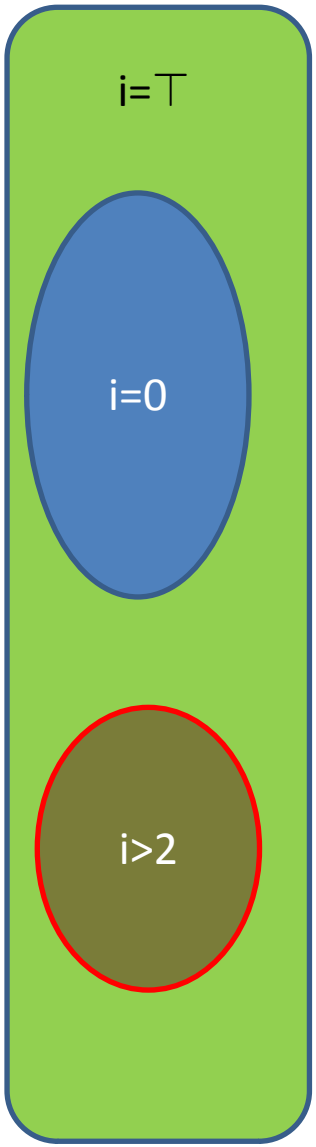


- (i=0) ∧ (i>2) is UNSAT

```
\functions {
  int i;
}

\problem {
  \part[p0] (i=0)
 &
  \part[p1] (i>2)
-> false
}
\interpolant {p0; p1}
```

Compute Interpolant using Princess



- (i=0) ∧ (i>2) is UNSAT

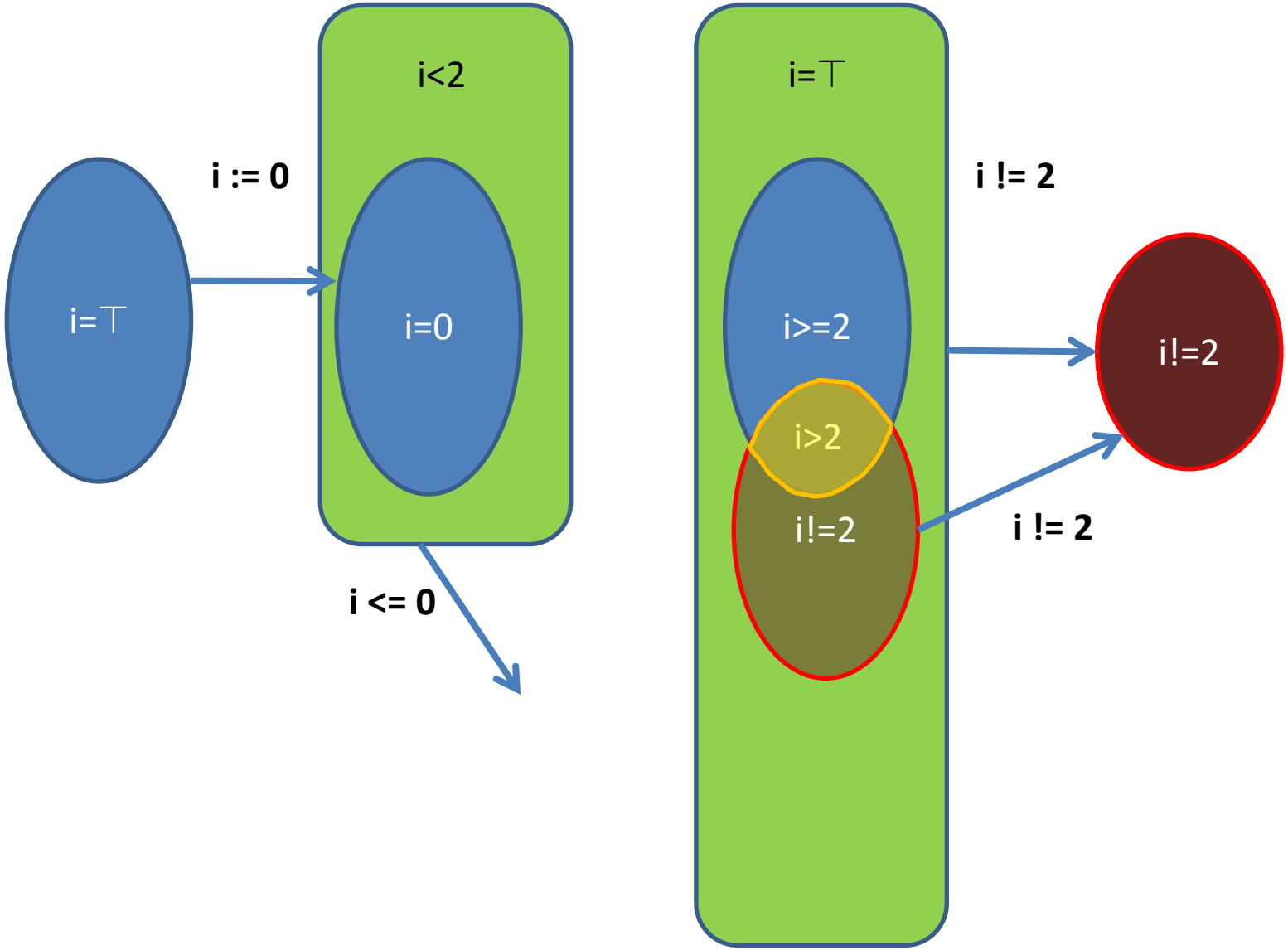
```
\functions {
  int i;
}

\problem {
  \part[p0] (i=0)
 &
  \part[p1] (i>2)
-> false
}
\interpolant {p0; p1}
```

No countermodel exists, formula is valid

Interpolants:
(-1*i >= 0)

A More Detailed Illustration



RHS Algorithm (Modified)

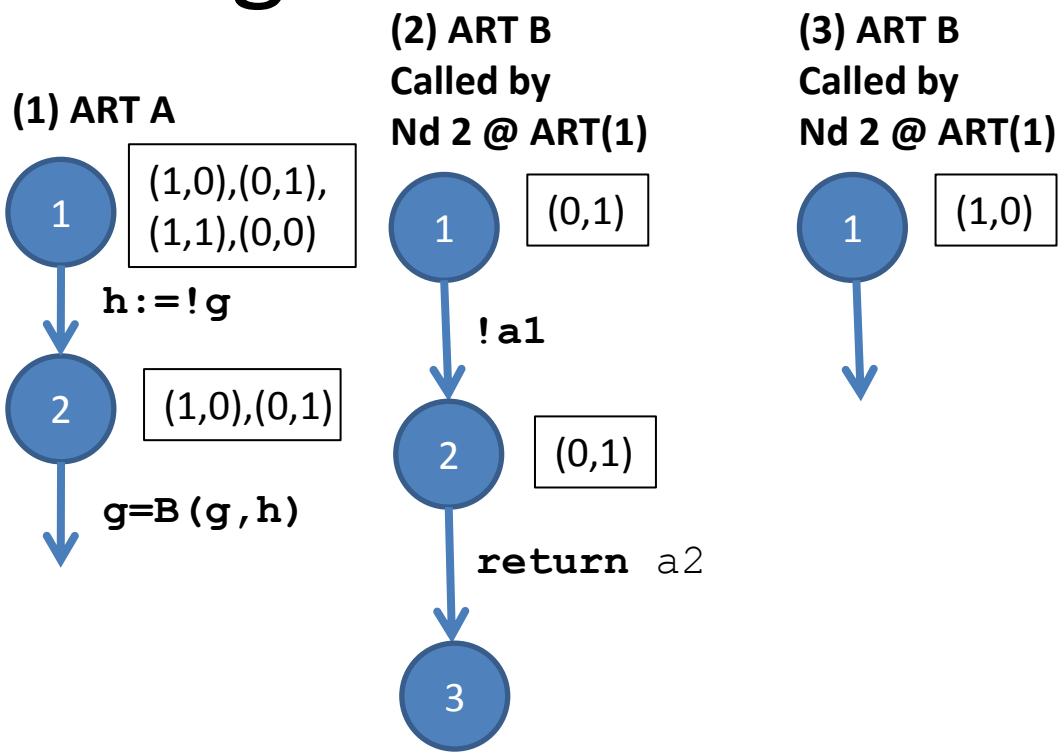
```
void A(){
  h := !g;
  g=B(g,h);
  g=B(g,h);
  assert(g);
}
```

```
void B(a1,a2){
  if (a1)
    return B(a2,a1);
  else
    return a2;
}
```

```
void A(){
  h := !g;
  g=B(g,h);
  g=B(g,h);
  assert(g);
}
```

```
void B(a1,a2){
  if (a1)
    return B(a2,a1);
  else
    return a2;
}
```

RHS Algorithm

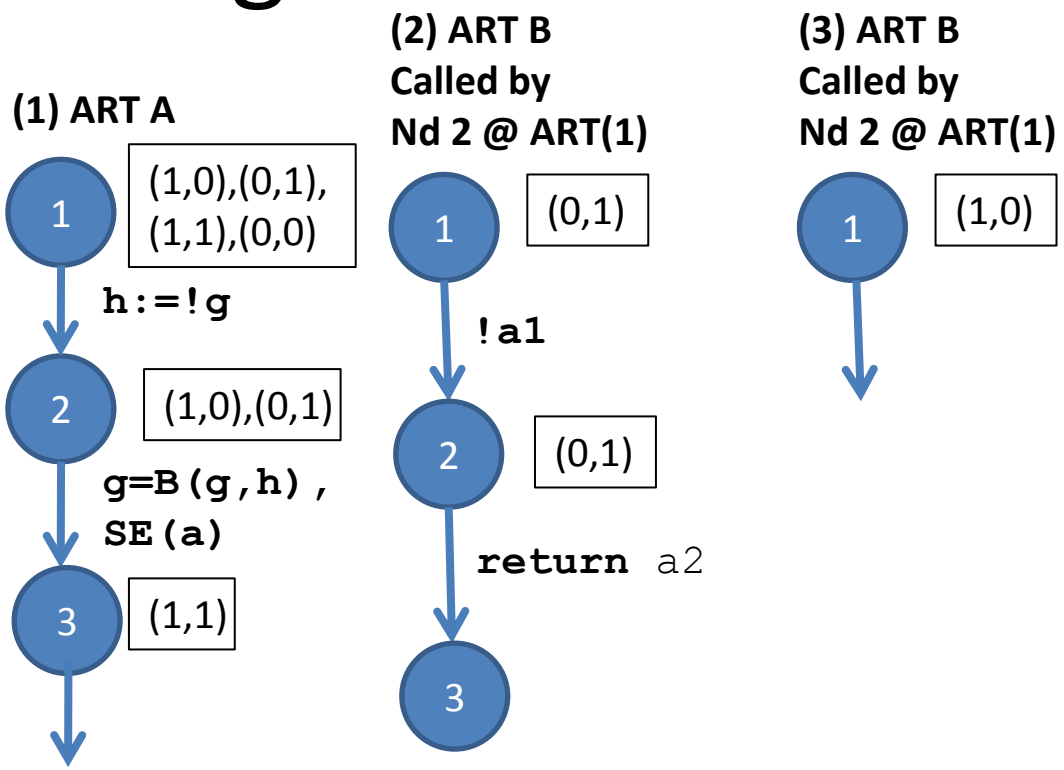


Establish summary edge (0,1)→1
Add it to node 2 @ ART (1)

```
void A(){
  h := !g;
  g=B(g,h);
  g=B(g,h);
  assert(g);
}
```

```
void B(a1,a2){
  if (a1)
    return B(a2,a1);
  else
    return a2;
}
```

RHS Algorithm

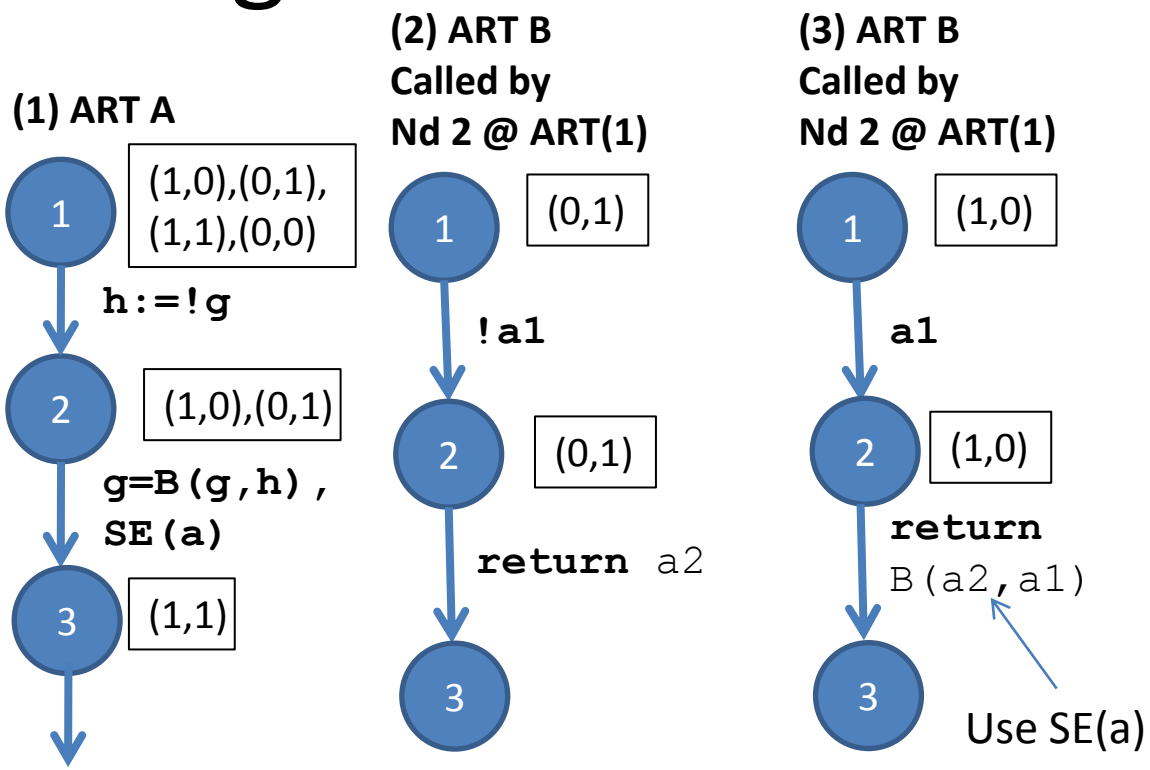


Summary edges
(a) (0,1)→1

```
void A(){
  h := !g;
  g=B(g,h);
  g=B(g,h);
  assert(g);
}
```

```
void B(a1,a2){
  if (a1)
    return B(a2,a1);
  else
    return a2;
}
```

RHS Algorithm



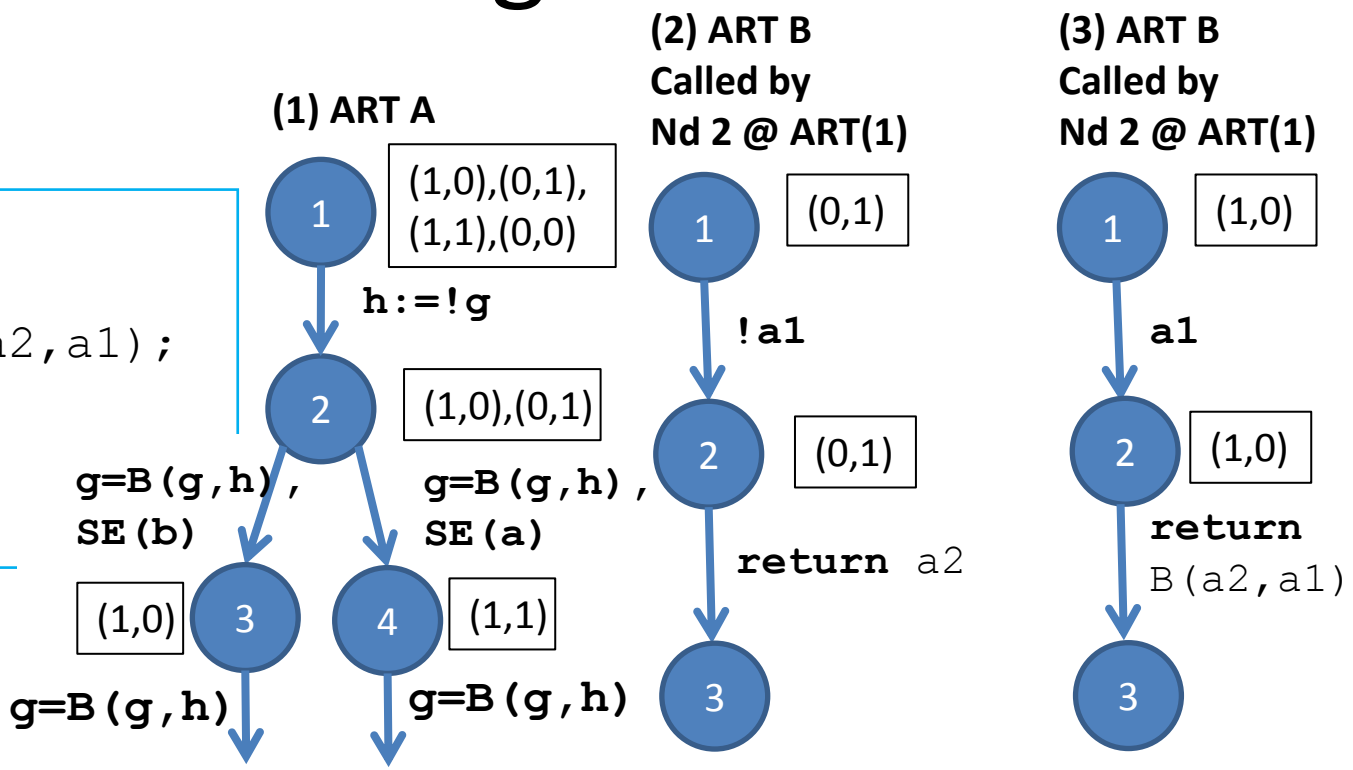
Summary edges
(a) (0,1)→1

Establish summary edge (1,0)→1
Add it to node 2 @ ART (1)

RHS Algorithm

```
void A(){
  h := !g;
  g=B(g,h);
  g=B(g,h);
  assert(g);
}
```

```
void B(a1,a2){
  if (a1)
    return B(a2,a1);
  else
    return a2;
}
```

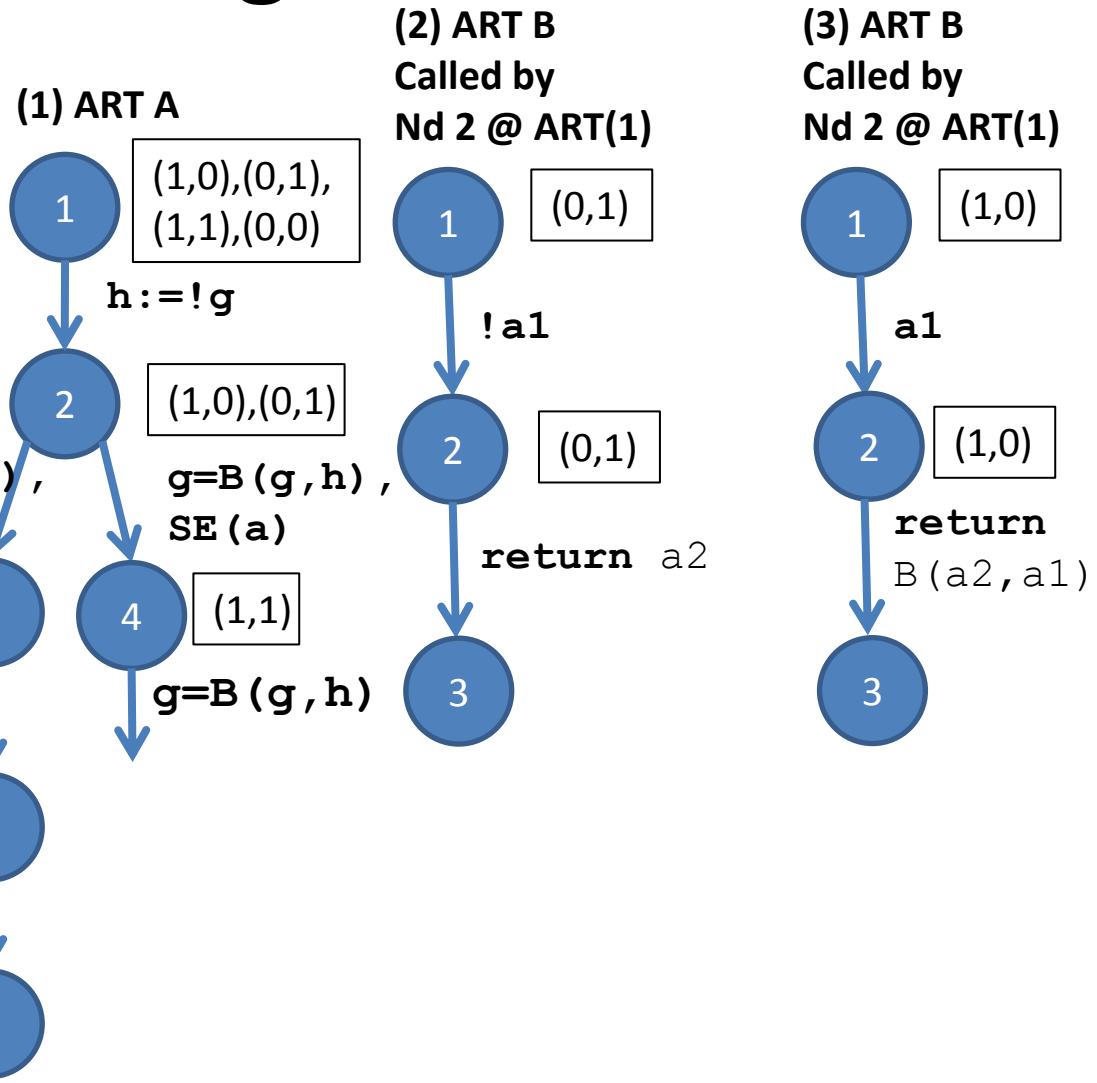


Summary edges
(a) (0,1)→1
(b) (1,0)→1

RHS Algorithm

```
void A(){
  h := !g;
  g=B(g,h);
  g=B(g,h);
  assert(g);
}
```

```
void B(a1,a2){
  if (a1)
    return B(a2,a1);
  else
    return a2;
}
```

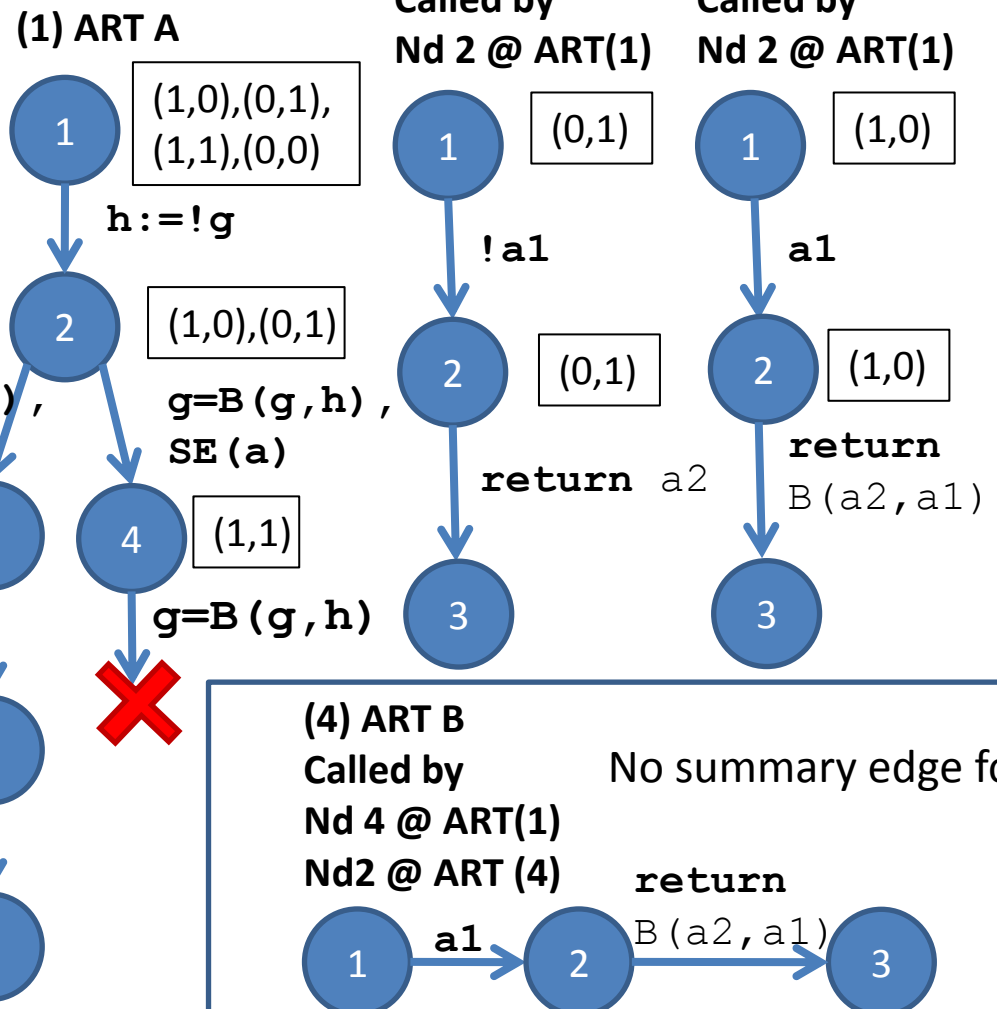


Summary edges
(a) (0,1)→1
(b) (1,0)→1

RHS Algorithm

```
void A(){
  h := !g;
  g=B(g,h);
  g=B(g,h);
  assert(g);
}
```

```
void B(a1,a2){
  if (a1)
    return B(a2,a1);
  else
    return a2;
}
```



Summary edges
(a) (0,1)→1
(b) (1,0)→1

Lazy abstraction + RHS alg. ?

- Major problems
  - For each function call, we enumerate all combinations of predicates.
  - When generate predicates, we hope it only contains **local** variables of a function.