Rigorous Software Development
CSCI-GA 3033-009

Instructor: Thomas Wies

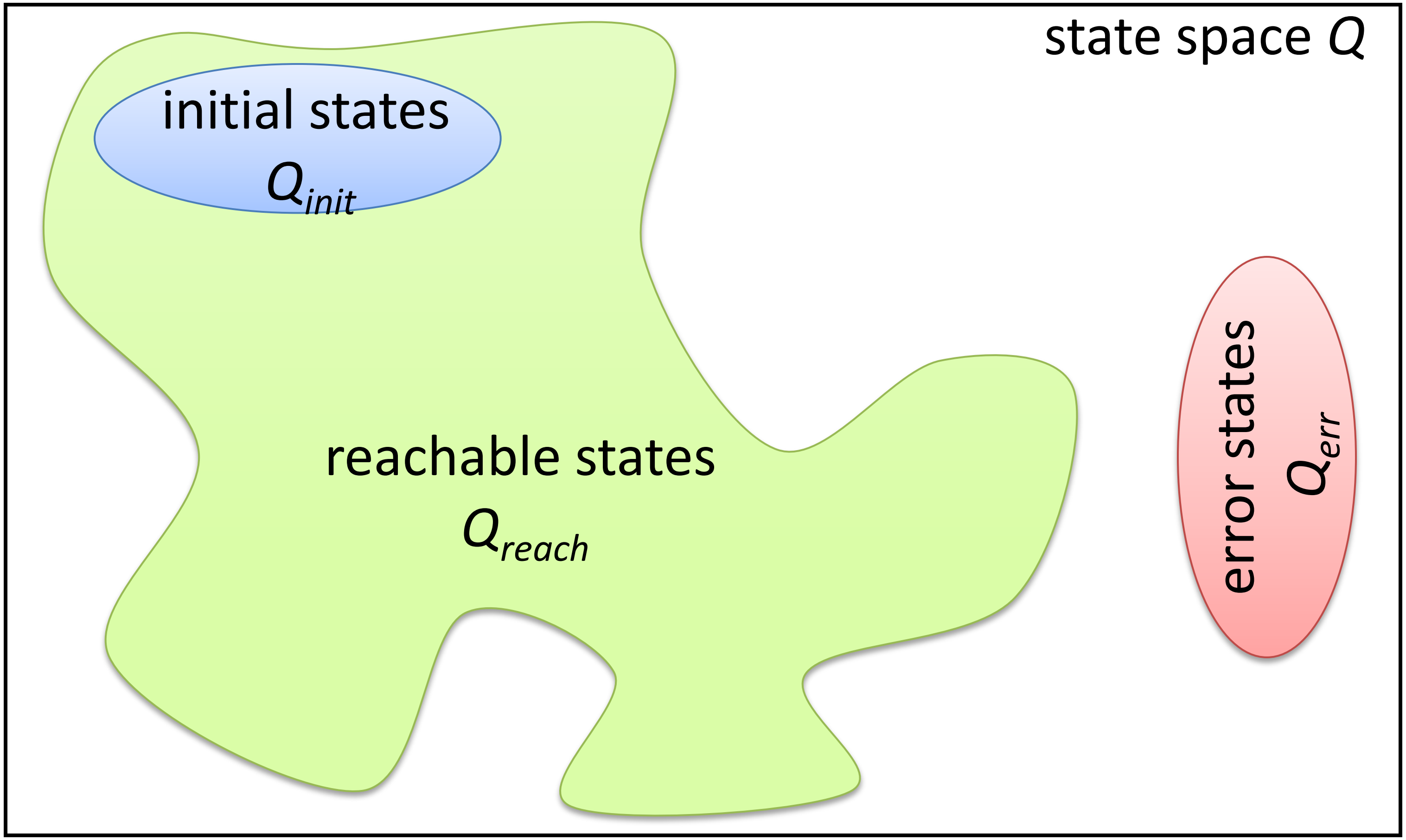
Spring 2013

Lecture 14

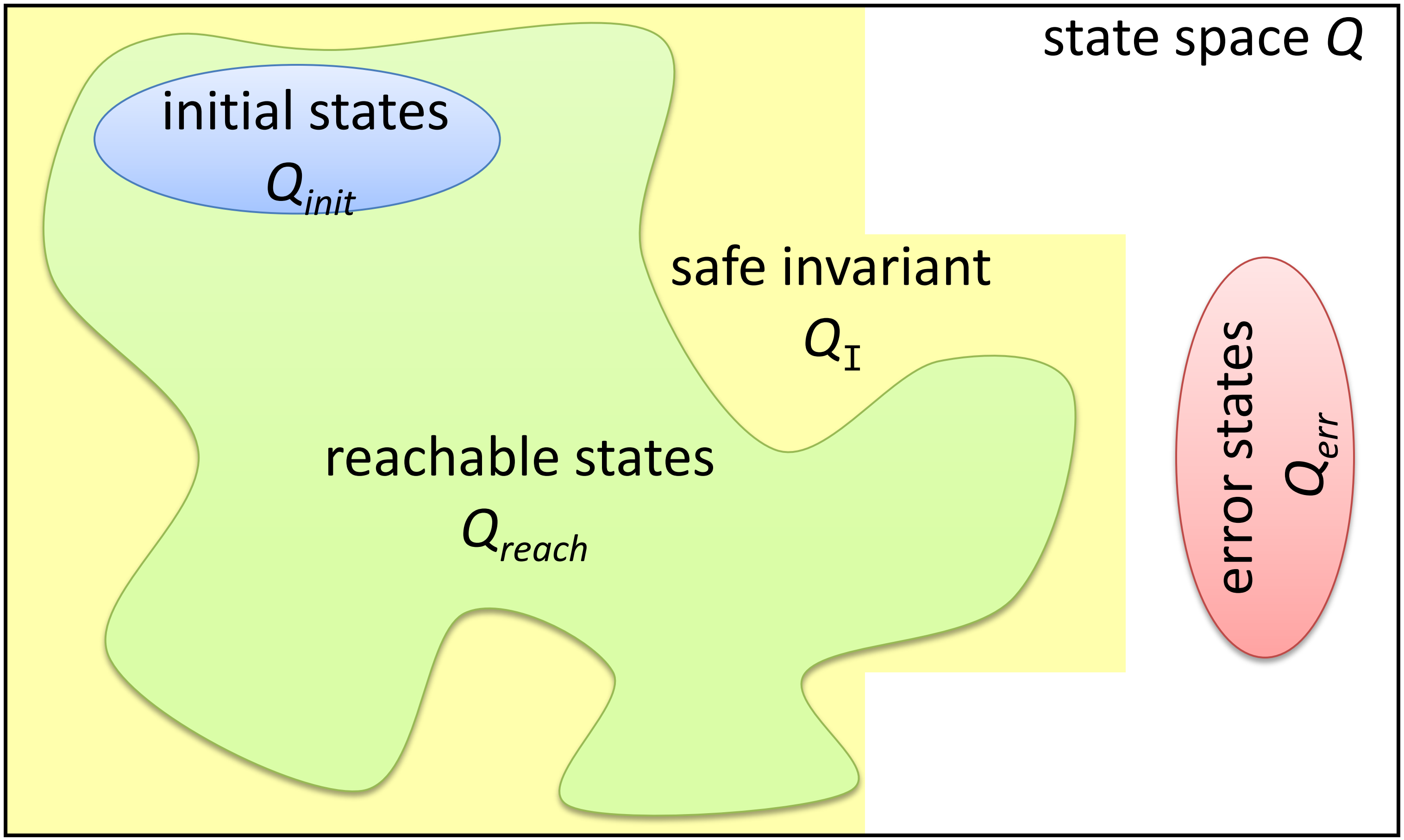
Invariant Generation

- Tools such as Dafny enable automated program verification by
  - automatically generating verification conditions and
  - automatically checking validity of the generated VCs.
- The user still needs to provide the invariants.
  - This is often the hardest part.
- Can we generate invariants automatically?

Partial Correctness of Programs

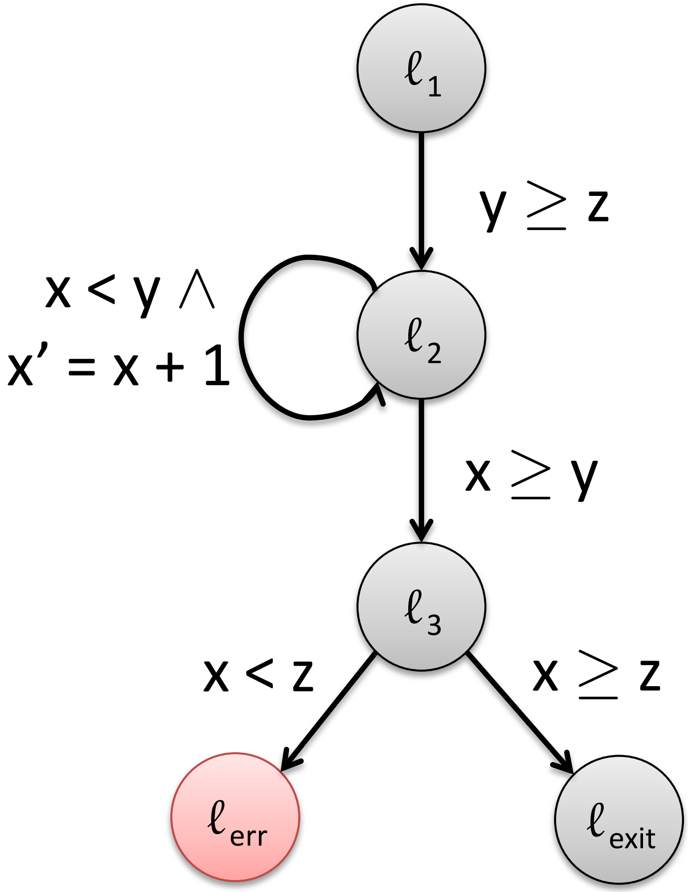


Partial Correctness of Programs



Inductive Invariants for Example Program

1: **assume** $y \geq z$;
2: **while** $x < y$ **do**
$\quad x := x + 1$;
3: **assert** $x \geq z$



Safe inductive invariant:

$pc = \ell_1 \ \vee$
$pc = \ell_2 \wedge y \geq z \ \vee$
$pc = \ell_3 \wedge y \geq z \wedge x \geq y \ \vee$
$pc = \ell_{exit}$

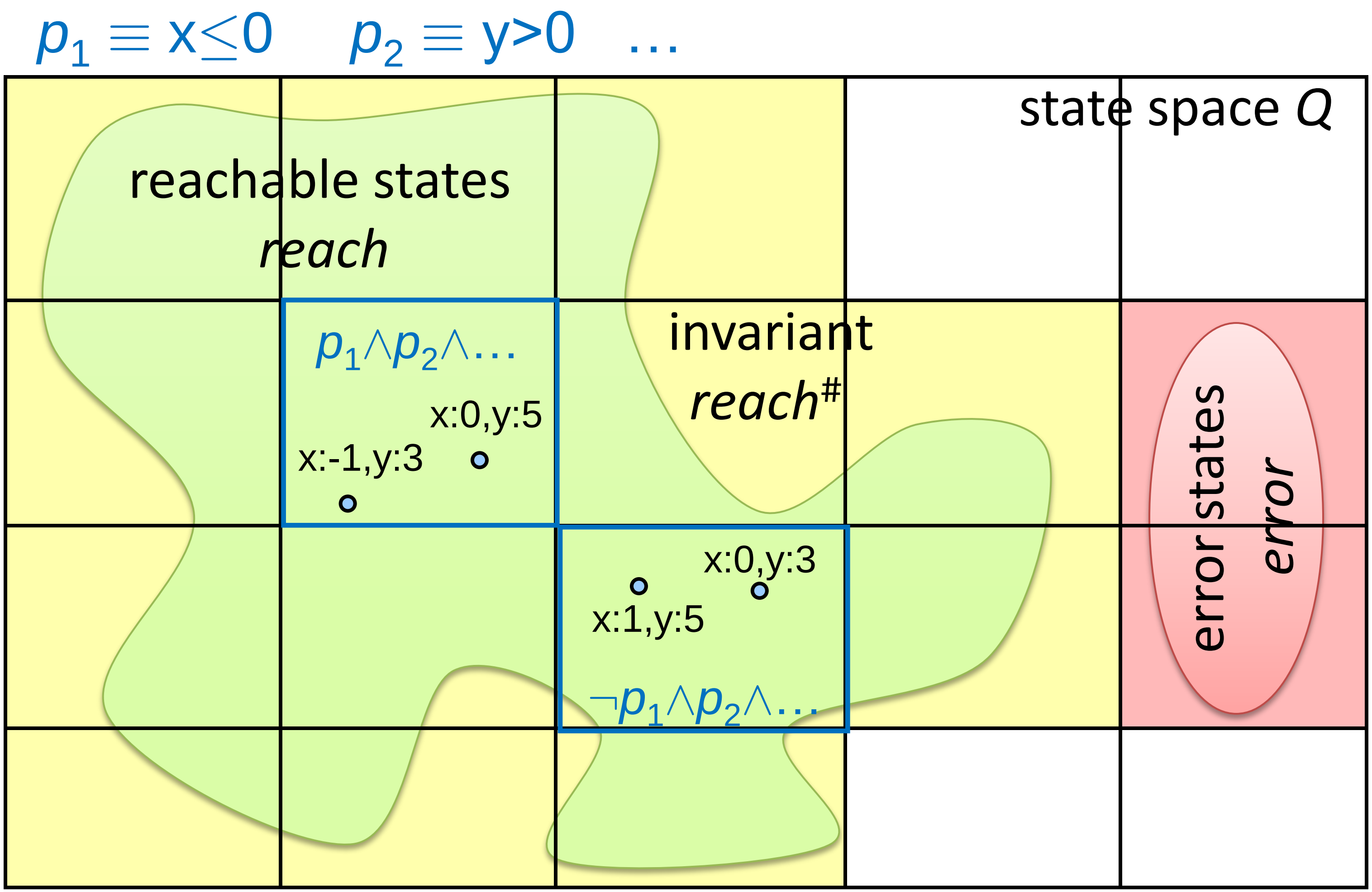
Predicate Abstraction

- construct abstraction α using a given set of building blocks, so-called predicates
- predicate = formula over program variables V
- fix finite set of predicates $Preds = \{p_1, \ldots, p_n\}$
- over-approximate F by conjunction of predicates in *Preds*

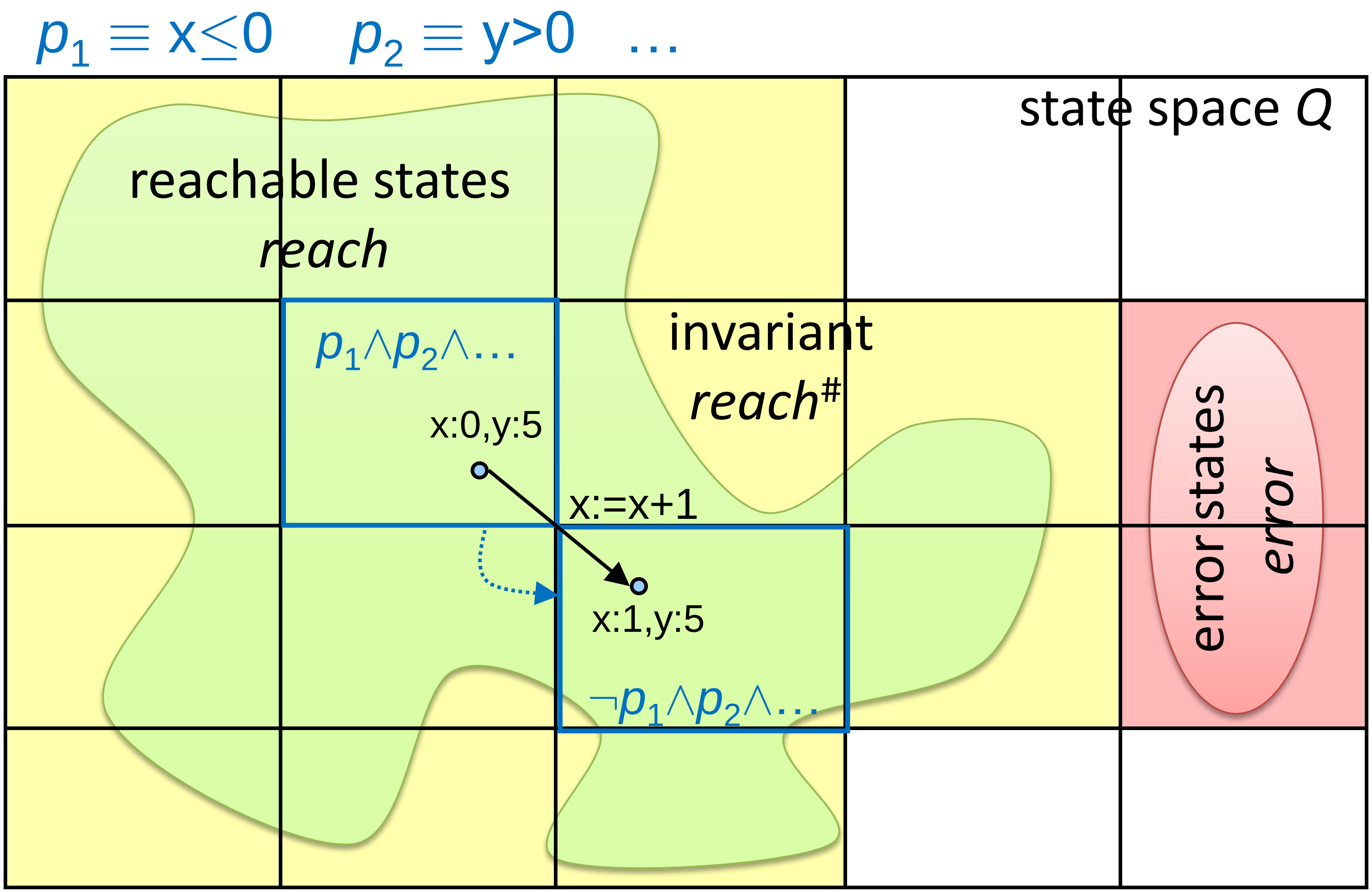
$$\alpha(F) = \bigwedge\{\, p \in Preds \mid F \vDash p \}$$

- computation of $\alpha(F)$ requires n theorem prover calls (n = number of predicates)

Predicate Abstraction



Abstract Reachability Graph



Abstract Reachability Graph

$$\begin{tikzcd}
\boxed{F_1:\ pc = \ell_1} \arrow[d, "\rho_1"] \\
\boxed{F_2:\ pc = \ell_2 \wedge y \geq z} \arrow[loop right, dashed, "\rho_2"] \arrow[d, "\rho_3"] \\
\boxed{F_3:\ pc = \ell_3 \wedge y \geq z \wedge x \geq y} \arrow[d, "\rho_5"] \\
\boxed{F_4:\ pc = \ell_{exit} \wedge y \geq z \wedge x \geq y}
\end{tikzcd}$$

$F_1 = \alpha(\mathit{init})$

$F_2 = \mathit{post}^{\#}(\rho_1, F_1)$

$\mathit{post}^{\#}(\rho_2, F_2) \models F_2$

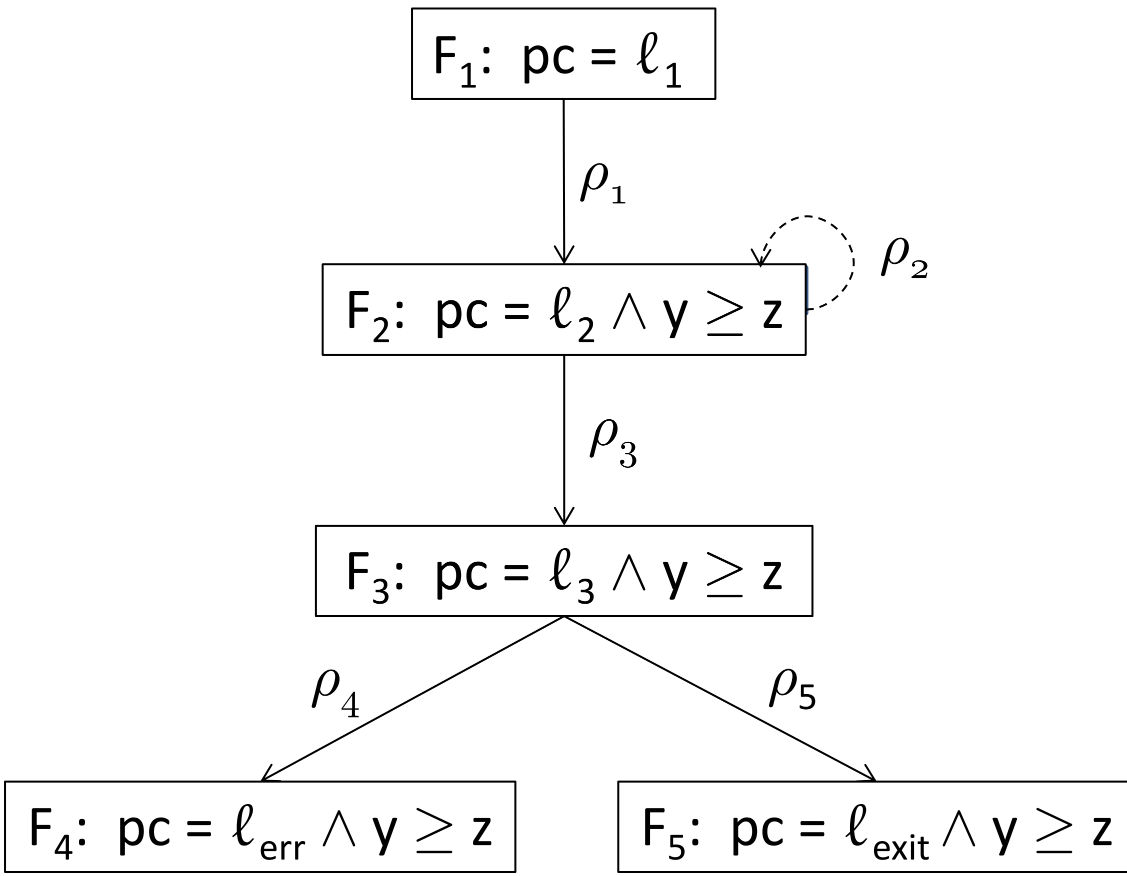
$F_3 = \mathit{post}^{\#}(\rho_3, F_2)$

$F_4 = \mathit{post}^{\#}(\rho_5, F_3)$

- *Preds* = {*false*, $pc = \ell_1$, ..., $pc = \ell_{err}$, $y \geq z$, $x \leq y$}
- nodes $F_1, \ldots, F_4 \in Q^{\#}_{reach}$
- labeled edges $\in$ *Tree*
- dotted edge: entailment relation (here: $\mathit{post}^{\#}(\rho_2, F_2) \models F_2$

Abstract Reachability Graph

with *Preds* = {*false*, $\mathrm{pc} = \ell_1$, ..., $\mathrm{pc} = \ell_{\mathrm{err}}$, $y \geq z$}



$F_1 = \alpha(\mathit{init})$

$F_2 = \mathit{post}^{\#}(\rho_1, F_1)$

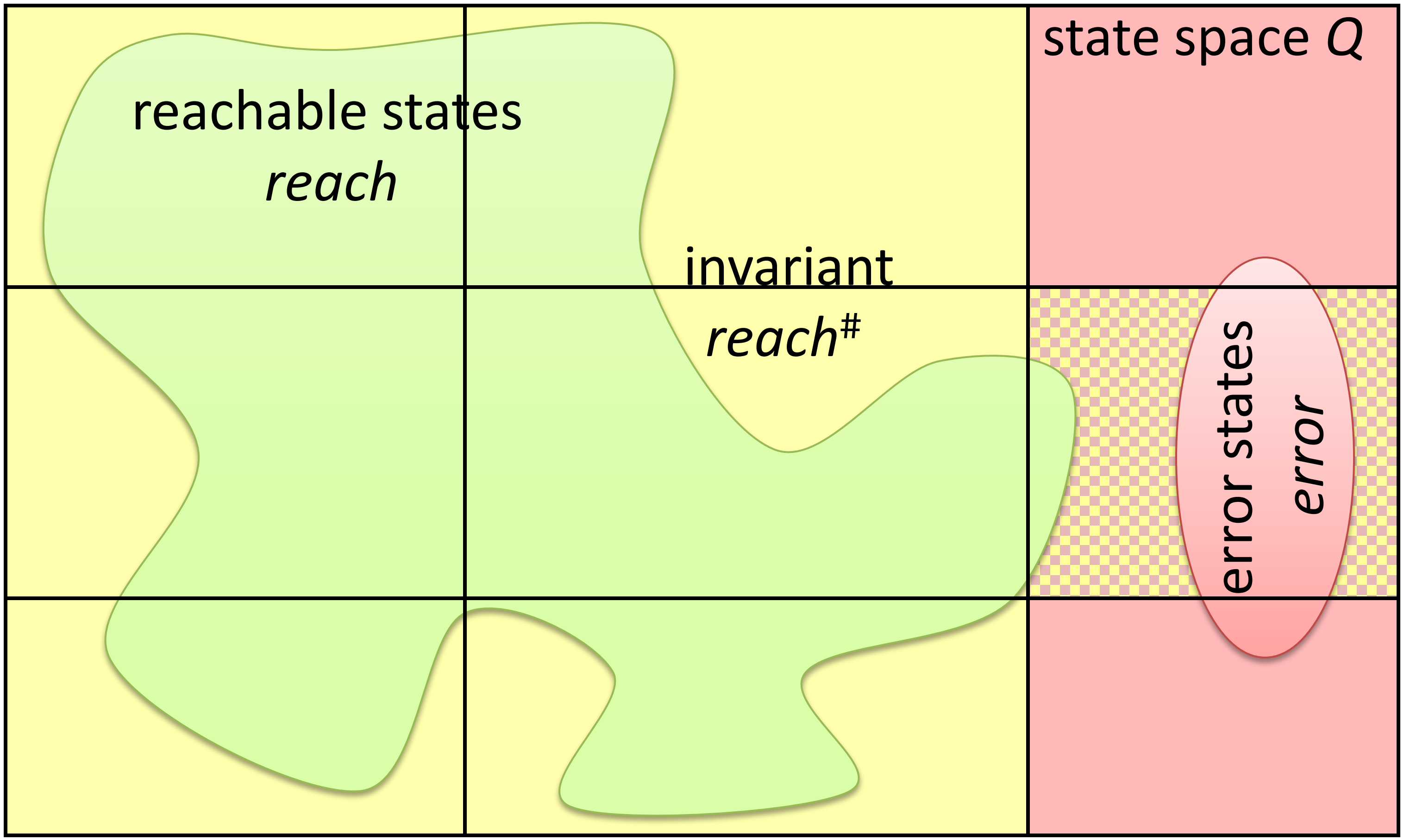
$\mathit{post}^{\#}(\rho_2, F_2) \models F_2$

$F_3 = \mathit{post}^{\#}(\rho_3, F_2)$

$F_4 = \mathit{post}^{\#}(\rho_4, F_3)$

$F_5 = \mathit{post}^{\#}(\rho_5, F_3)$

Too Coarse Abstraction



Refinement of Predicate Abstraction

- given formulas F_1, F_2, F_3, F_4 such that

$$
\begin{aligned}
&\mathit{init} \models F_1 \\
&\mathit{post}(\rho_1, F_1) \models F_2 \\
&\mathit{post}(\rho_3, F_2) \models F_3 \\
&\mathit{post}(\rho_4, F_3) \models F_4 \\
&F_4 \wedge \mathit{error} \models \mathit{false}
\end{aligned}
$$

- add atoms of F_1, ..., F_4 to *Preds.*
- refinement guarantees that counterexample path ρ_1, ρ_3, ρ_4 is eliminated.

CEGAR: Counter-Example Guided Abstraction Refinement Loop

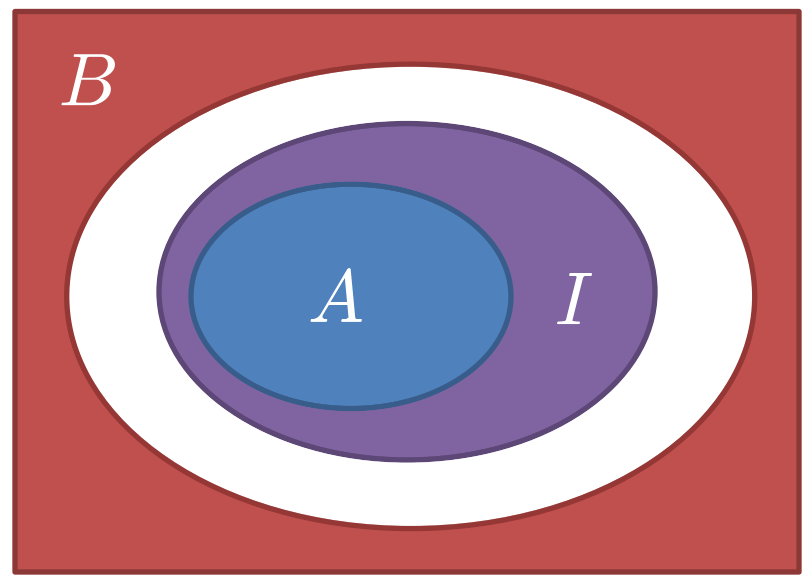
function `AbstRefineLoop`
  begin
    Preds := $\emptyset$;
    repeat
      (*reach*$^{\#}$, *Tree*) := `AbstReach`(*Preds*)
      if exists F $\in$ *reach*$^{\#}$ such that F $\wedge$ *error* $\not\models$ *false* **then**
        path := `MakePath`(F, *Tree*)
        if `FeasiblePath`(*path*) **then**
          return "counterexample path: *path*"
        else
          Preds := *Preds* $\cup$ `RefinePath`(*path*)
      else
        return "program is safe"
  end

Craig Interpolation

Given: an unsatisfiable conjunction of formulas $A \wedge B$

A Craig interpolant for $A \wedge B$ is a formula I such that

- A implies I
- $I \wedge B$ is unsatisfiable
- all free variables in I are shared between A and B



Interpolation-Based CEGAR

[McMillan’03, ...]

```
assume 0 ≤ y;
x := 0;
z := n;
while x < y do
   x := x + 1;
   z := z - 1;
assert z = n - y;
```

$$A\left\{\begin{array}{l} 0 \le y_0 \wedge \\ x_0 = 0 \wedge \\ z_0 = n_0 \wedge \\ \left.\begin{array}{l} x_0 < y_0 \wedge \\ x_1 = x_0 + 1 \wedge \\ z_1 = z_0 - 1 \wedge \end{array}\right\} \text{1. iter.} \end{array}\right.$$

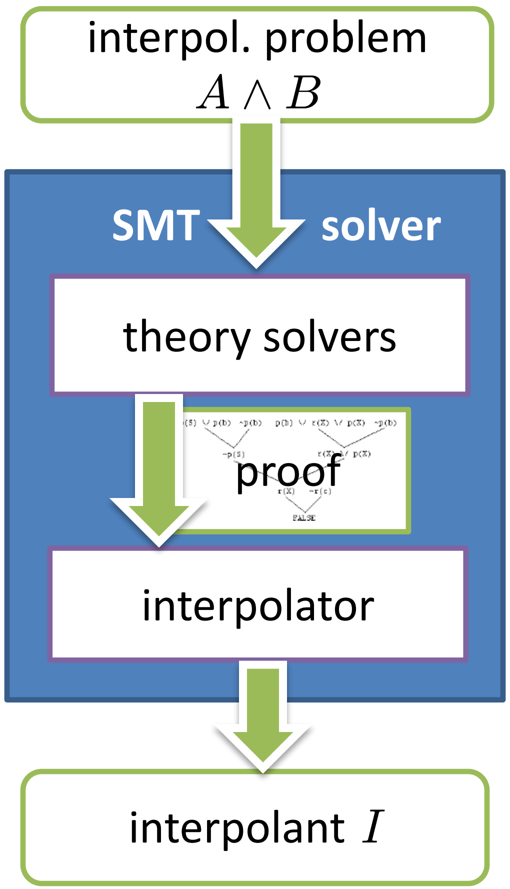
$$B\left\{\begin{array}{l} \left.\begin{array}{l} x_1 < y_0 \wedge \\ x_2 = x_1 + 1 \wedge \\ z_2 = z_1 - 1 \wedge \end{array}\right\} \text{2. iter.} \\ x_2 \ge y_0 \wedge \\ z_2 \ne n_0 - y_0 \end{array}\right.$$

Interpolant for $A \wedge B$

$z_1 = n_0 - x_1 \wedge x_1 \le y_0$

Interpolation Procedures

- first-order predicate logic admits interpolation
- typically interested in quantifier-free interpolants
- many theories admit quantifier-free interpolation
  - linear arithmetic over $\mathbb{Z}$ and $\mathbb{Q}$
  - free function symbols with equality
  - ...
- interpolants computed from refutation proofs
- implemented in a number of Satisfiability Modulo Theory (SMT) solvers



Interpolation for Linear Arithmetic over Rationals

- Literals L : (strict) linear inequalities

  $$0 \leq c_0 + c_1x_1 + \ldots + c_nx_n$$

  where
  - $c_0, \ldots, c_n$ are constants (rational numbers)
  - $x_1, \ldots, x_n$ are variables (denoting rationals)
- Clause C : set of literals
- Sequents $C \vdash D$ where
  - C and D are clauses
  - $C \vdash D$ means $\bigwedge\{L \mid L \in C\} \models \bigvee\{L \mid L \in D\}$

A Simple Proof System

$$\text{HYP}\ \frac{}{C \vdash L}\ L \in C$$

$$\text{COMB}\ \frac{C \vdash 0 \leq \mathrm{s} \qquad C \vdash 0 \leq \mathrm{t}}{C \vdash 0 \leq \mathrm{c}_1\mathrm{s} + \mathrm{c}_2\mathrm{t}}\ \mathrm{c}_{1,2} > 0$$

$$\text{CONTRA}\ \frac{L_1, \ldots, L_n \vdash 0 \leq \mathrm{c}}{C \vdash \neg L_1, \ldots, \neg L_n}\ \mathrm{c} < 0$$

$$\text{RES}\ \frac{C \vdash L, D \qquad C \vdash \neg L, D'}{C \vdash D, D'}$$

Interpolants from Proofs

- Interpolated Sequent $(A,B) \vdash C\ [\mathcal{I}]$
  where
  - A,B,C are clauses and $\mathcal{I}$ is a formula
  - $A \vdash \mathcal{I}$
  - $B,\mathcal{I} \vdash C$
  - $fv(\mathcal{I}) \subseteq fv(A) \cap\ fv(B) \cup fv(C)$

Interpolating Proof System

$$\text{HYP-A}\ \frac{}{(A,B) \vdash L\ [L]}\ L \in \text{A}$$

$$\text{HYP-B}\ \frac{}{(A,B) \vdash L\ [\top]}\ L \in \text{B}$$

$$\text{COMB}\ \frac{(A,B) \vdash 0 \leq \text{s}\ [0 \leq \text{s}'] \qquad (A,B) \vdash 0 \leq \text{t}\ [0 \leq \text{t}']}{(A,B) \vdash 0 \leq \text{c}_1\text{s} + \text{c}_2\text{t}\ [0 \leq \text{c}_1\text{s}' + \text{c}_2\text{t}']}\ \text{c}_{1,2} > 0$$

Interpolating Proof System

$$\text{CONTRA}\ \frac{(\{a_1, \ldots, a_n\}, \{b_1, \ldots, b_m\}) \vdash 0 \leq c\ [I]}{(A,B) \vdash \neg a_1, \ldots, \neg a_n, \neg b_1, \ldots, \neg b_m\ [I \vee \neg a_1 \vee \ldots \vee \neg a_n]}\ c < 0$$

$$\text{RES-A}\ \frac{(A,B) \vdash L, D\ [I] \quad (A,B) \vdash \neg L, D'\ [I']}{(A,B) \vdash D, D'\ [I \vee I']}\ L\ \textit{not}\ \text{occurs in}\ B$$

$$\text{RES-B}\ \frac{(A,B) \vdash L, D\ [I] \quad (A,B) \vdash \neg L, D'\ [I']}{(A,B) \vdash D, D'\ [I \wedge I']}\ L\ \text{occurs in}\ B$$

Example

- Computing an interpolant for (*A*,*B*) where
  $A \equiv 0 \leq y - x \wedge 0 \leq z - y$
  $B \equiv 0 \leq x - z - 1$

Automated Debugging

Faulty Shell Sort

Program

- takes a sequence of integers as input
- returns the sorted sequence.

On the input sequence 11, 14 the program returns 0, 11 instead of 11,14.

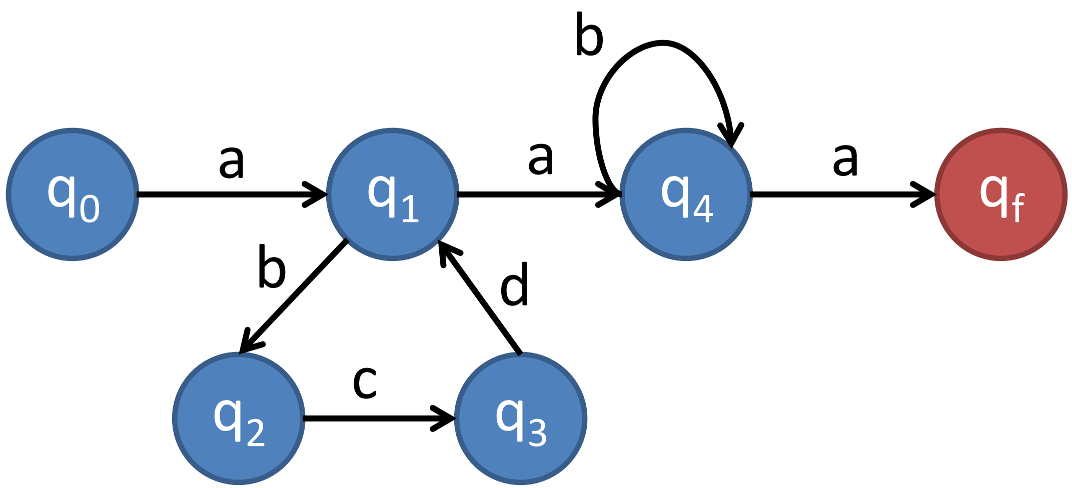
```
#include<stdio.h>
#include<stdlib.h>

static void shell_sort(int a[], int size)
 {
   int i, j;
   int h = 1;
   do {
        h = h * 3 + 1;
   } while (h <= size);
   do {
        h /= 3;
        for (i = h; i < size; i++) {
           int v = a[i];
           for (j = i; j >= h && a[j - h] > v; j -= h)
              a[j] = a[j-h];
           if (i != j)
              a[j] = v;
         }
   } while (h != 1);
 }
int main(int argc, char *argv[])
 {
     int i = 0;
     int *a = NULL;

     a = (int *)malloc((argc-1) * sizeof(int));
     for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);

     shell_sort(a, argc);

     for (i = 0; i < argc - 1; i++)
        printf("%d", a[i]);
     printf("\n");

     free(a);
     return 0;
 }
```

Error Trace

```
0 int i,j, a[];
1 int size=3;
2 int h=1;
3 h = h*3+1;
4 assume !(h<=size);
5 h/=3;
6 i=h;
7 assume (i<size);
8 v=a[i];
9 j=i;
10 assume !(j>=h && a[j-h]>v);
11 i++;
12 assume (i<size);
13 v=a[i];
14 j=i;
15 assume (j>=h && a[j-h]>v);
16 a[j]=a[j-h];
17 j-=h;
18 assume (j>=h && a[j-h]>v);
19 a[j]=a[j-h];
20 j-=h;
21 assume !(j>=h && a[j-h]>v);
22 assume (i!=j);
23 a[j]=v;
24 i++;
25 assume !(i<size);
26 assume (h==1);
27 assert a[0] == 11 && a[1] == 14;
```

Idea of our Approach

- Consider reachability in a finite automaton



- A word that witnesses the reachability of q_f:

  a b c d a b b b a

- We can eliminate loops to obtain a simpler witness:

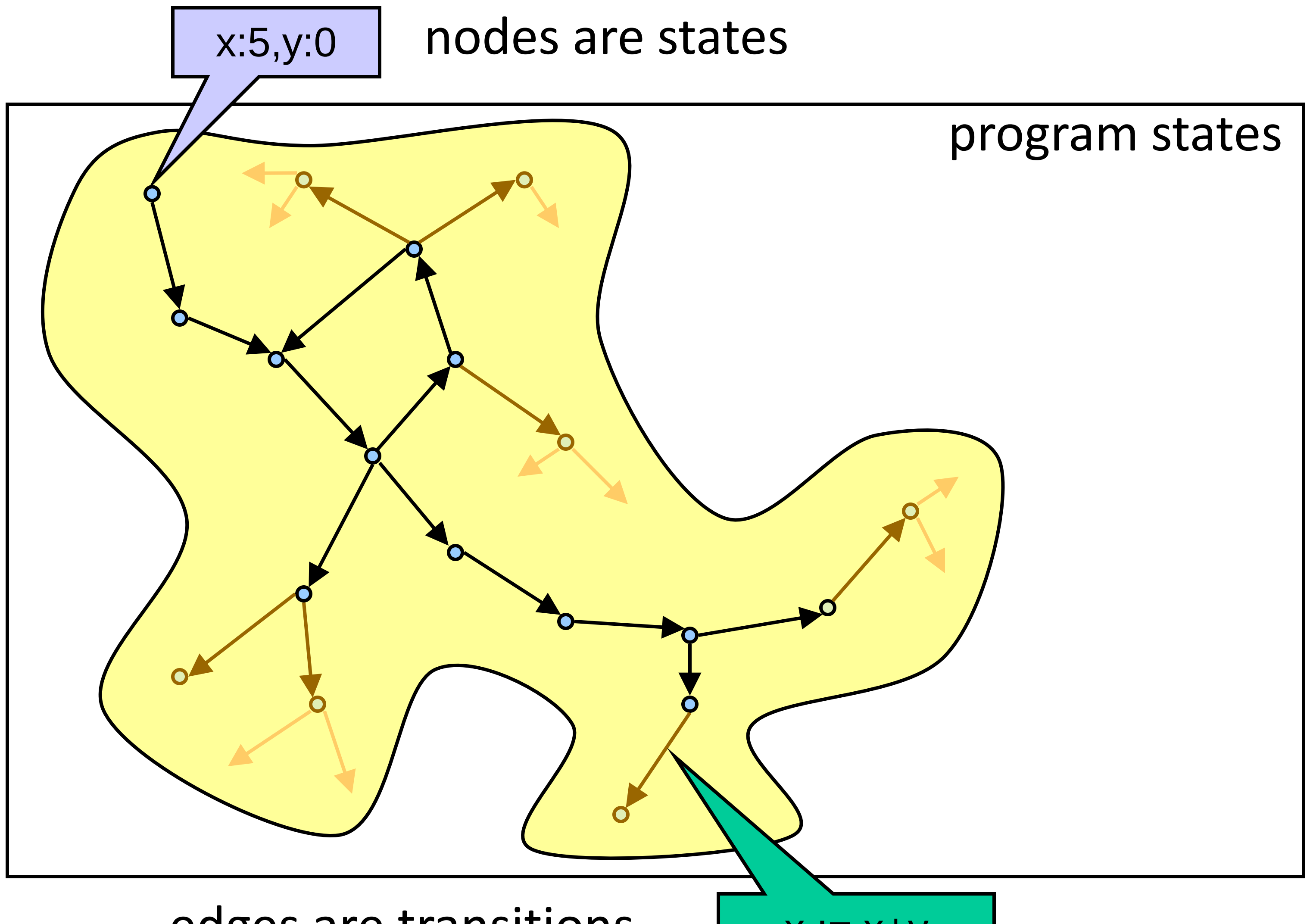
  a a a

Fault Localization in Programs

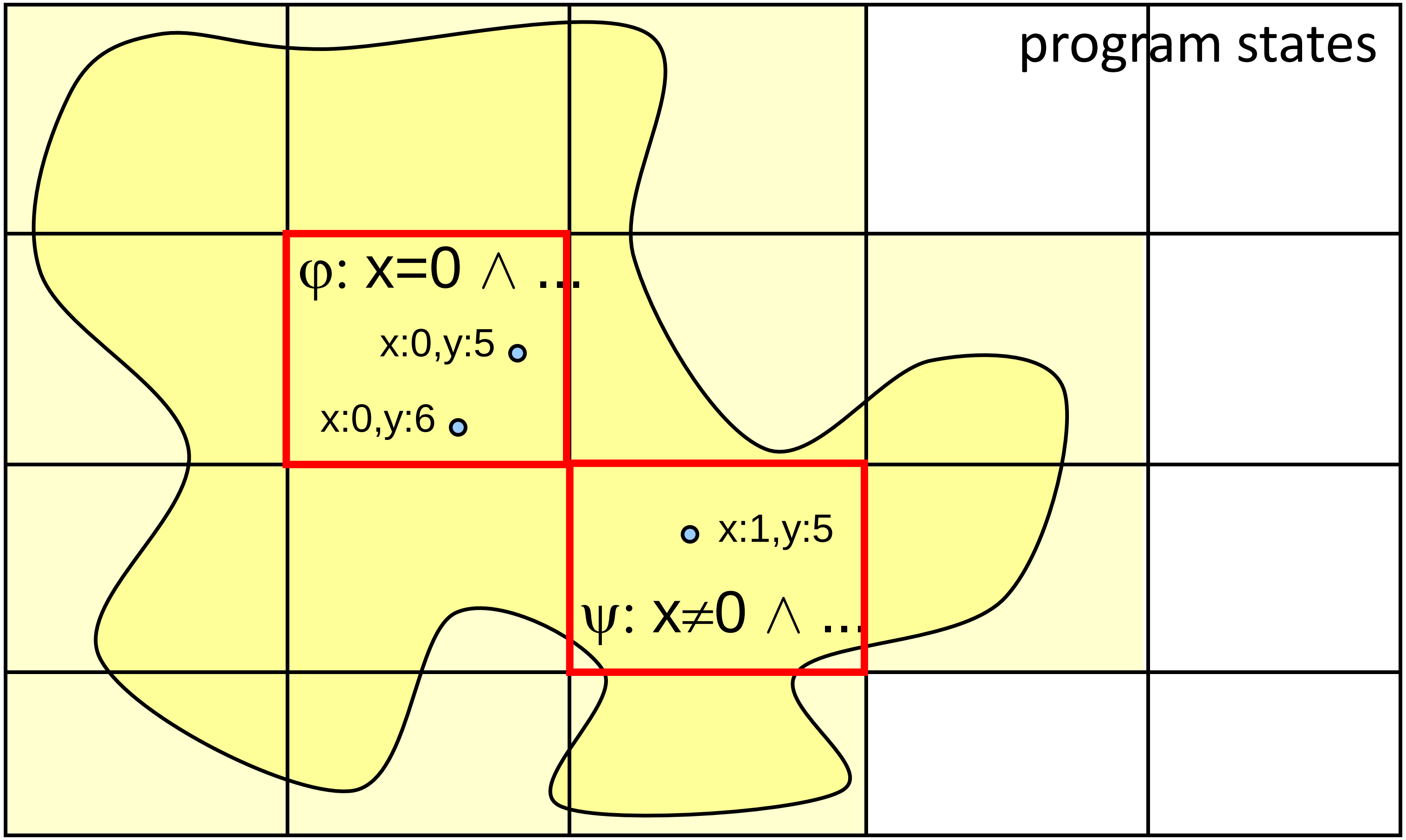
Apply this idea to programs:

- error trace = finite word of program statements
- program = automaton that accepts error traces
  (state/transition graph)
- fault localization = eliminate loops in the
  error trace

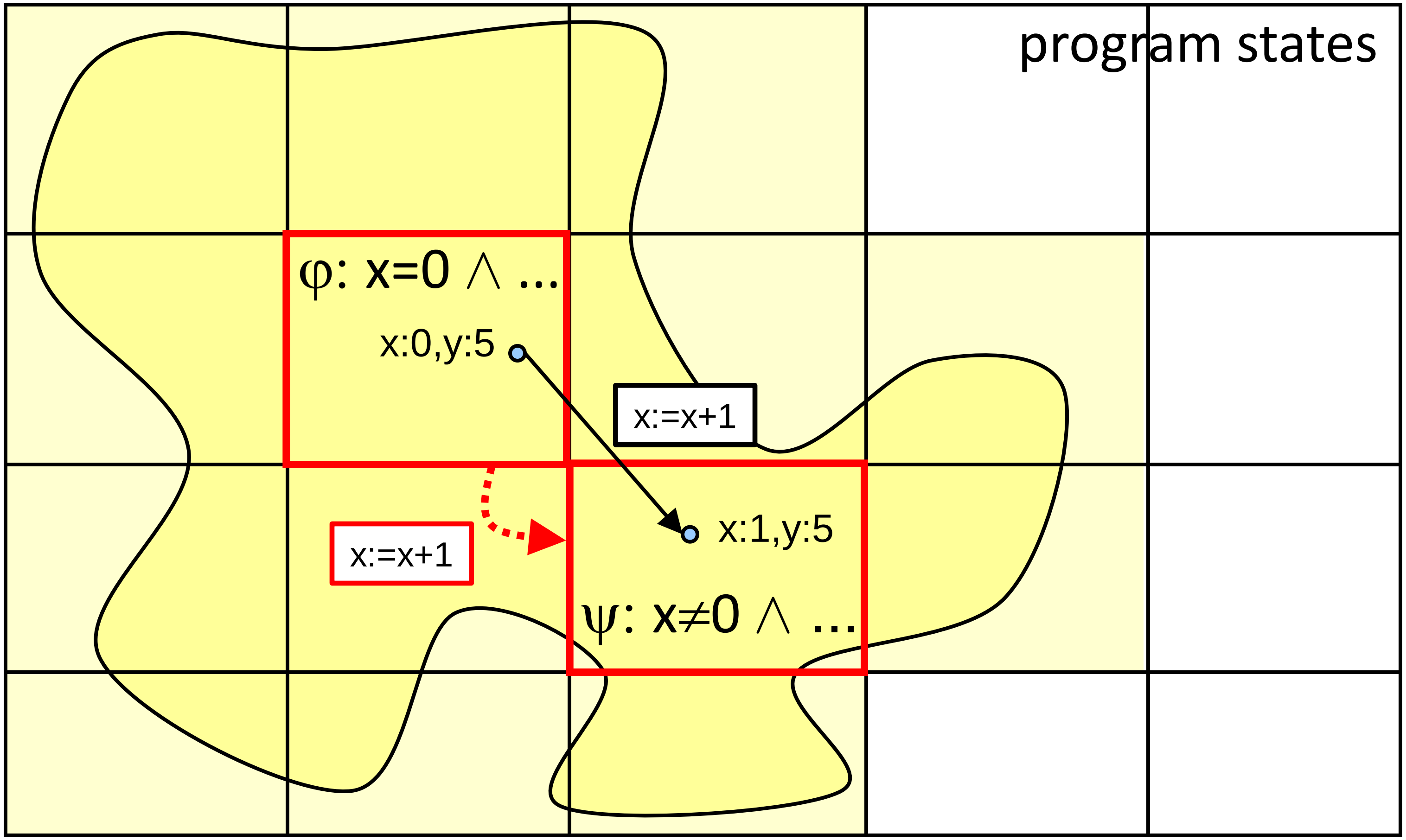
State/Transition Graph



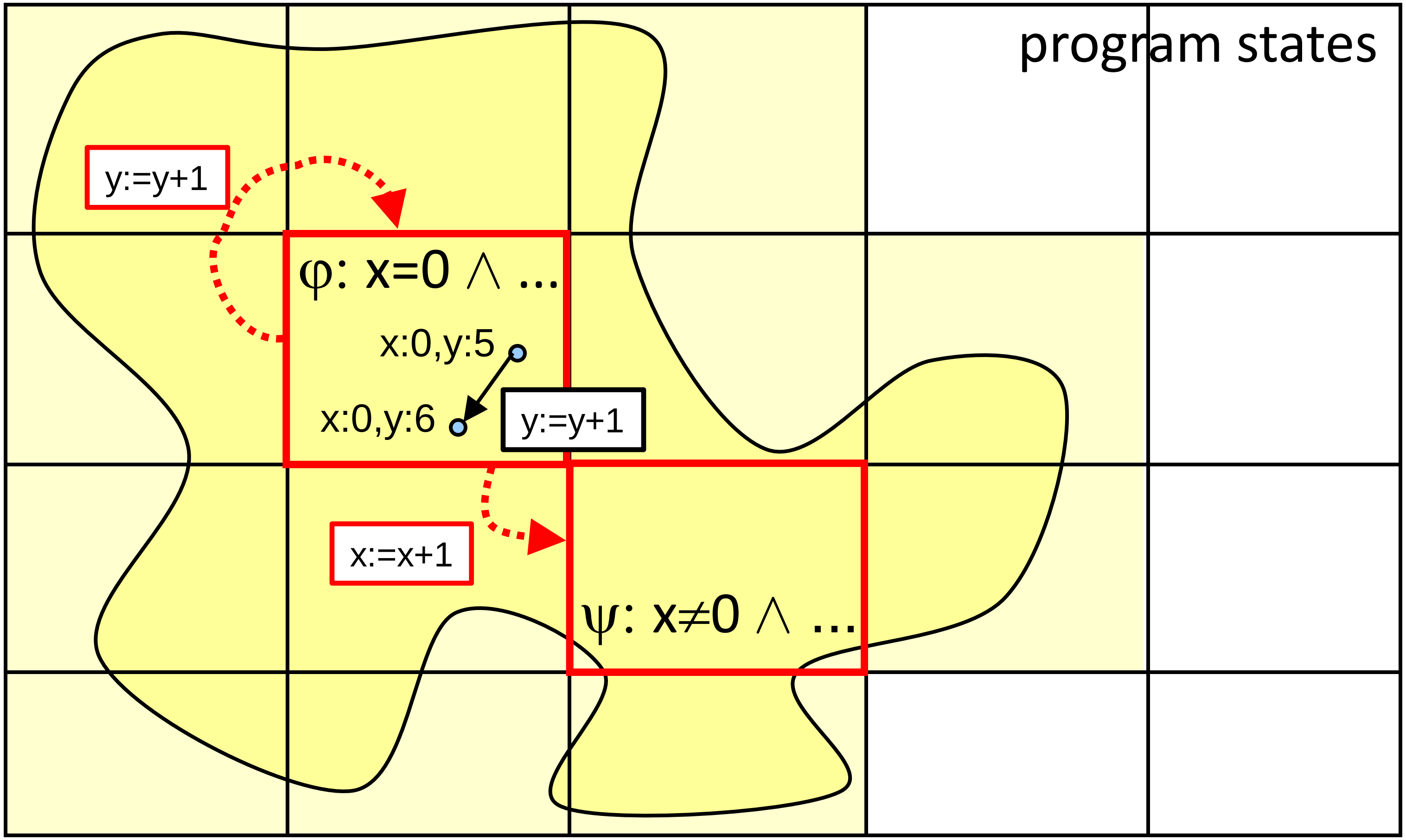
Predicate Abstraction



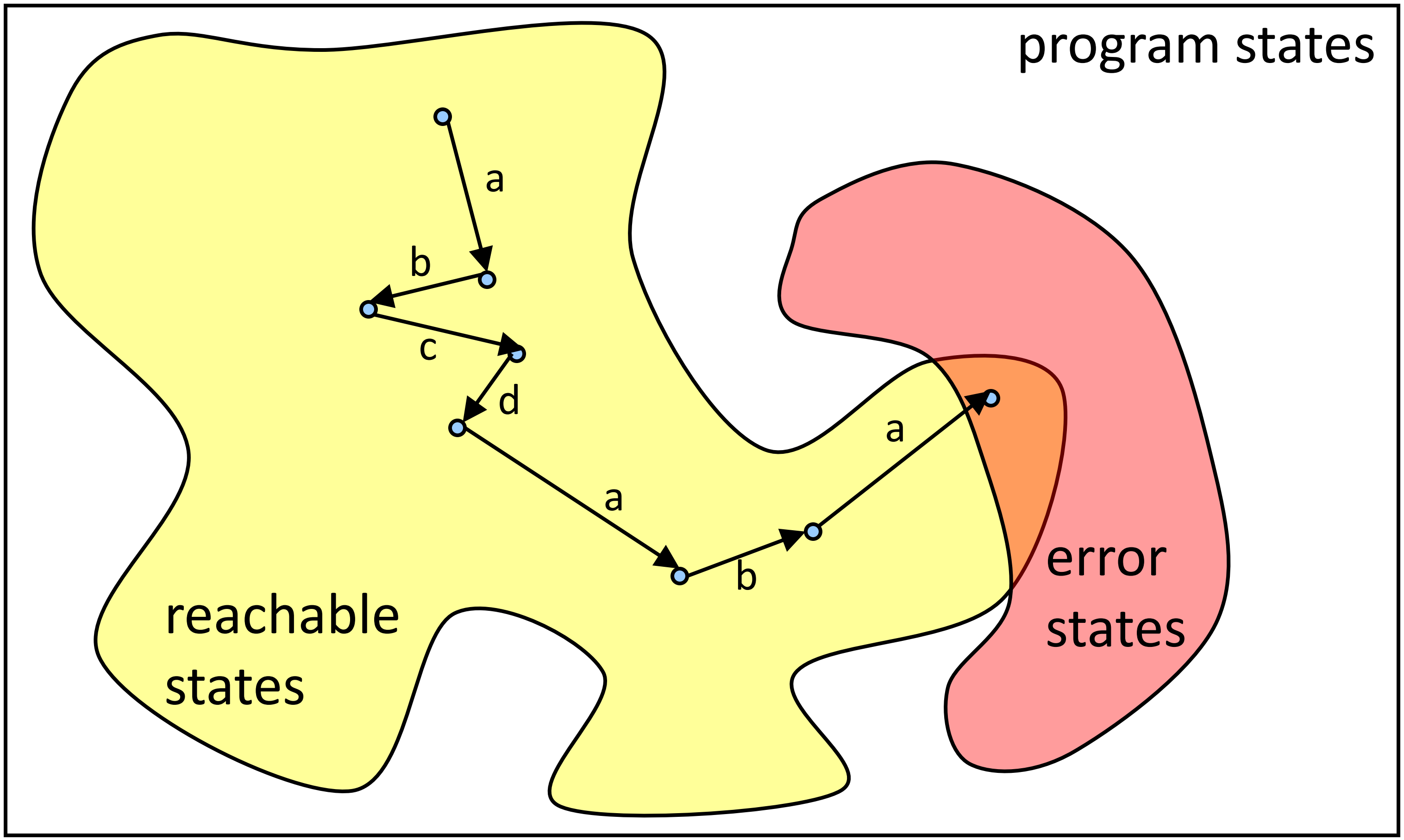
Predicate Abstraction



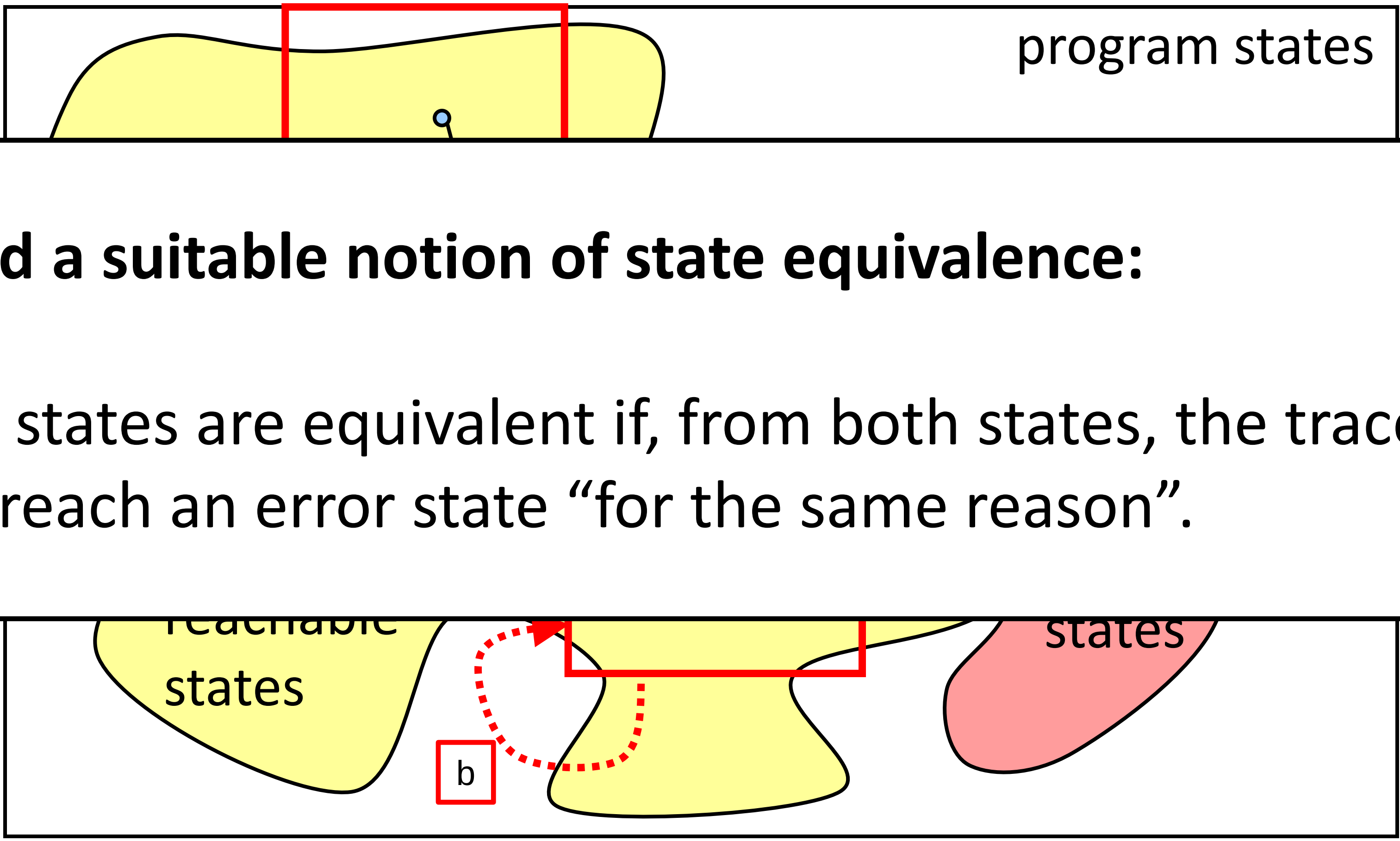
Predicate Abstraction



Abstraction-Based Fault Localization



Abstraction-Based Fault Localization



Need a suitable notion of state equivalence:

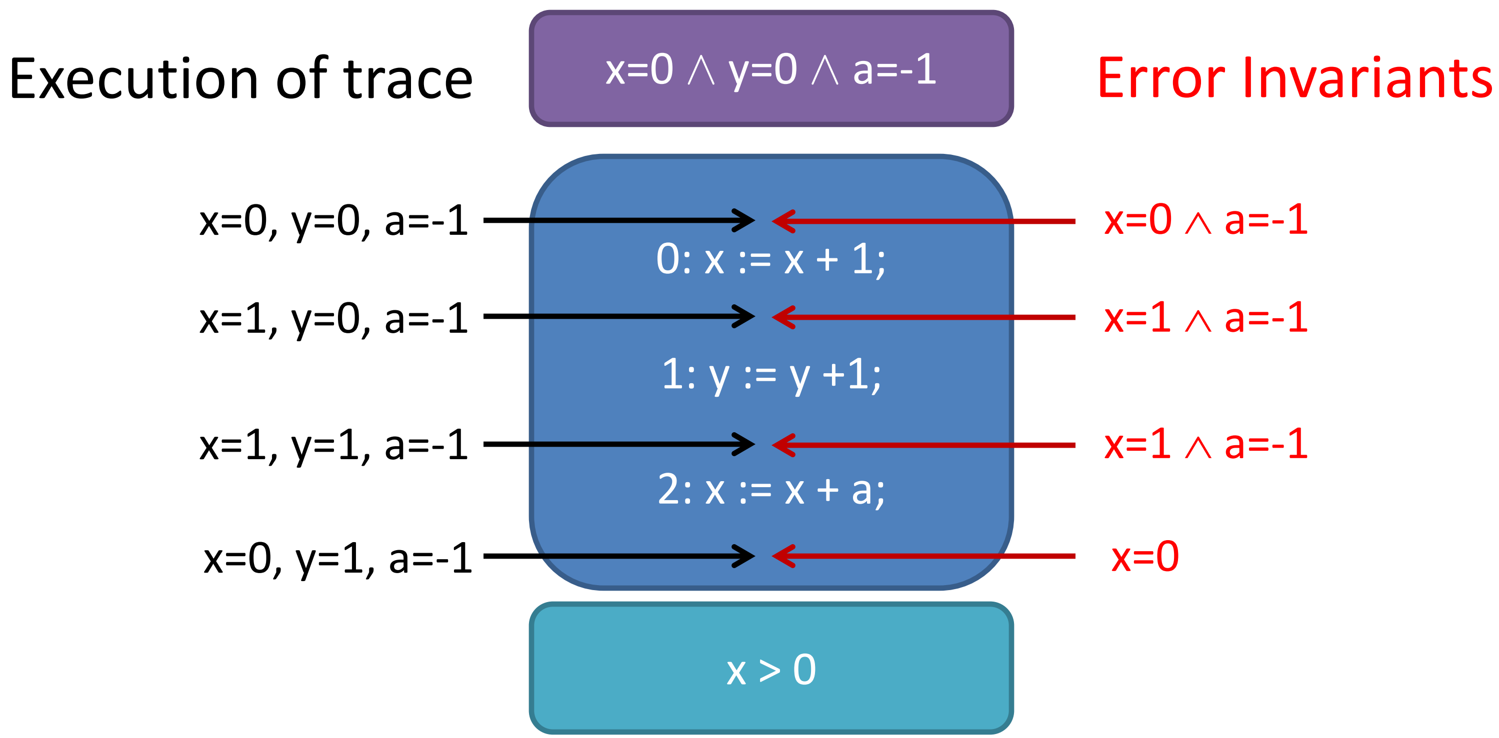
Two states are equivalent if, from both states, the trace can reach an error state “for the same reason”.

Error Invariants

An error invariant I for a position i in an error trace τ is a formula over program variables s.t.

- all states reachable by executing the prefix of τ up to position i satisfy I
- all executions of the suffix of τ that start from i in a state that satisfies I, still lead to the error.

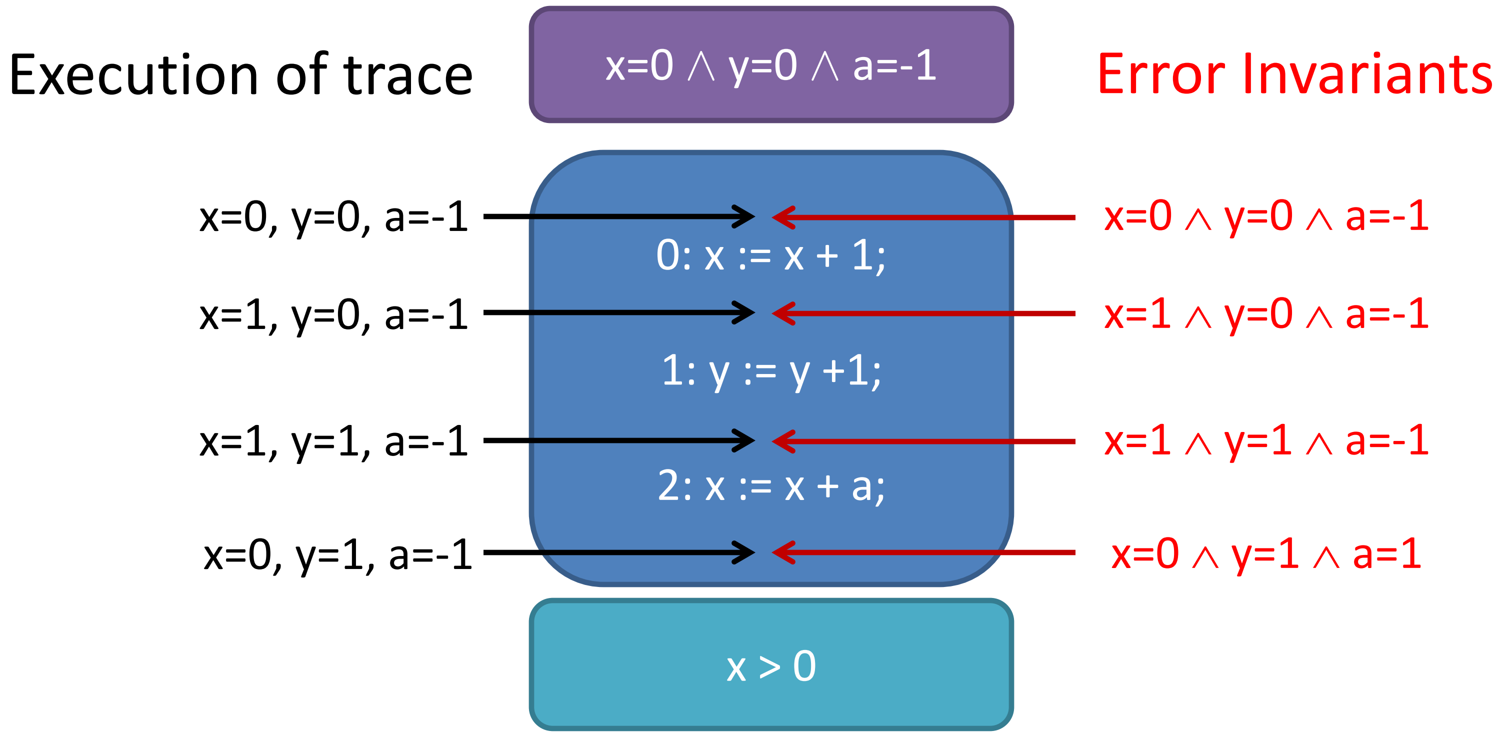
Error Invariants



Information provided by the error invariants

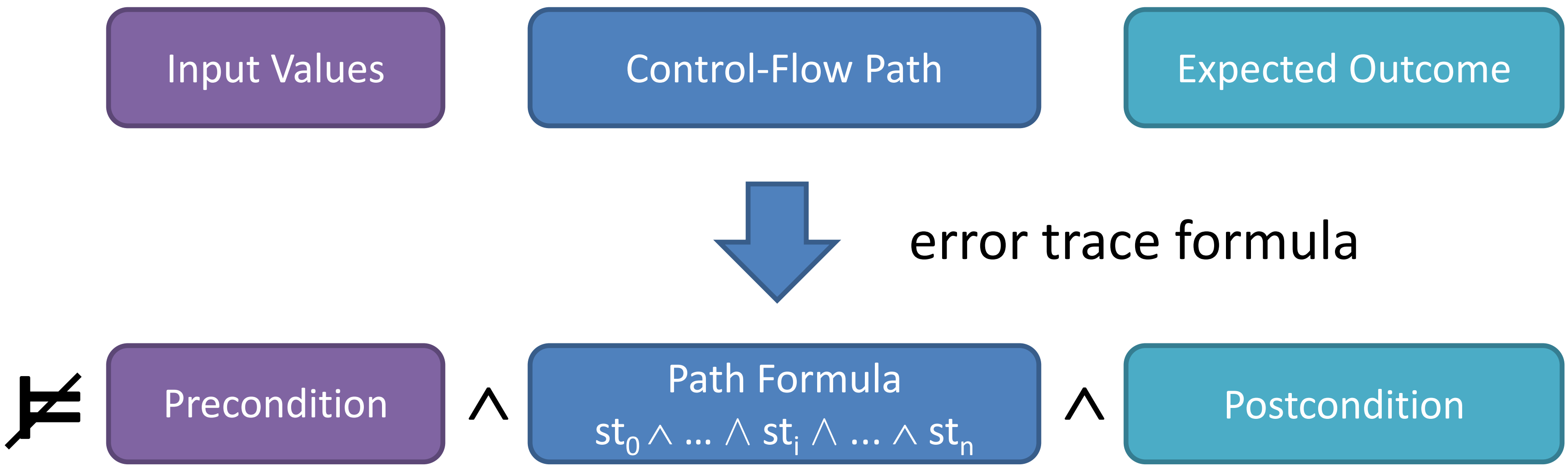
- Statement y := y + 1 is irrelevant
- Variable y is irrelevant
- Variable a is irrelevant after position 2

Error invariants are not unique



We are interested in **inductive** error invariants!

Checking Error Invariants



Error Trace Formula

Example

$x=0 \wedge y=0 \wedge a=-1$

0: x := x + 1;

1: y := y +1;

2: x := x + a;

$x > 0$

$x_0=0 \wedge y_0=0 \wedge a_0=-1$

$\wedge$

$x_1 = x_0 + 1 \wedge$

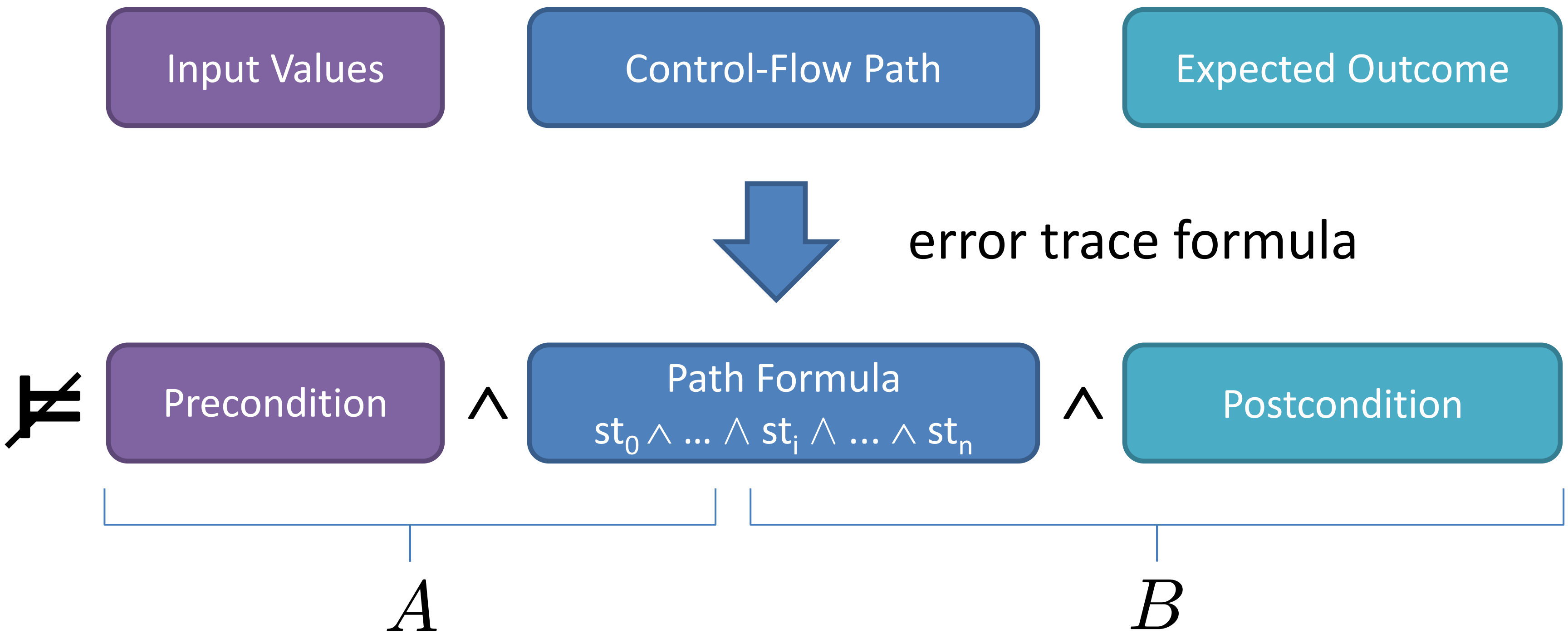
$y_1 = y_0 + 1 \wedge$

$x_2 = x_1 + a_0$

$\wedge$

$x_2 > 0$

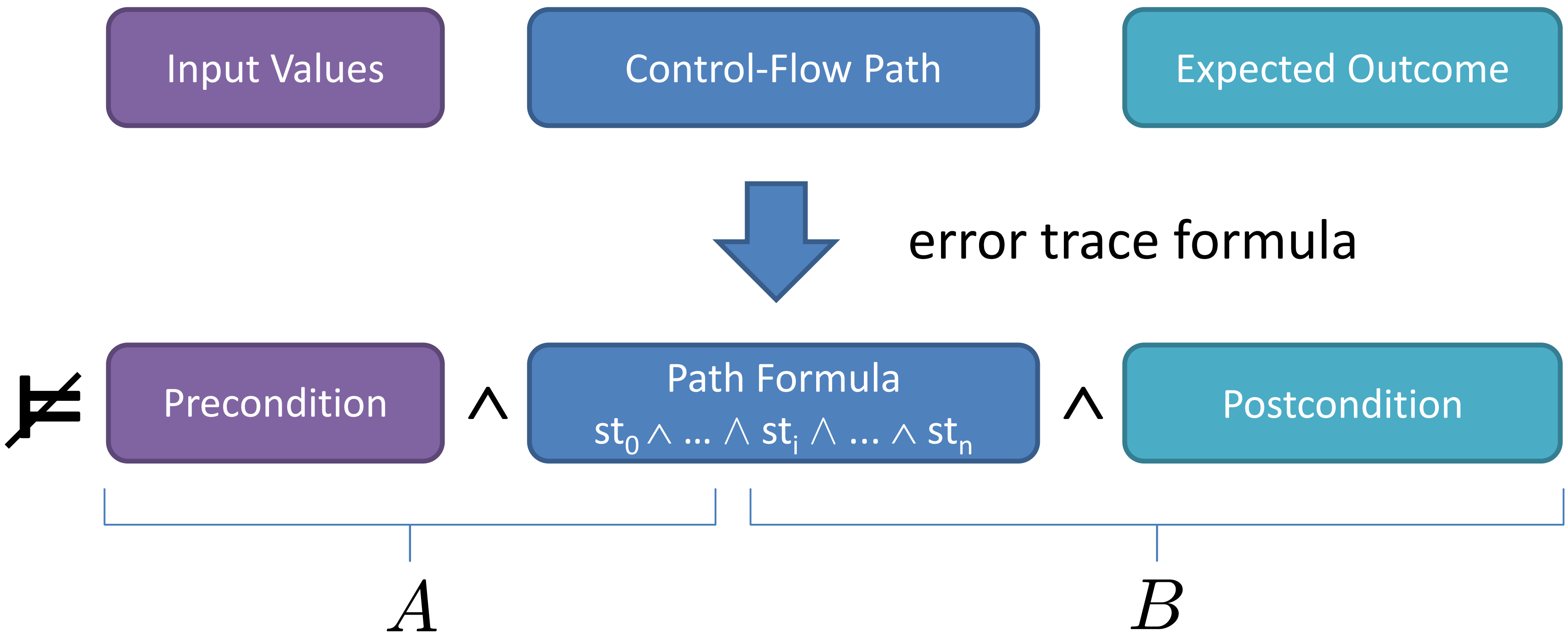
Checking Error Invariants



I is an error invariant for position i iff

$$A \models I \quad \text{and} \quad I \wedge B \models \bot$$

Craig Interpolants are Error Invariants



Craig interpolant for $A \wedge B$ is an error invariant for position i

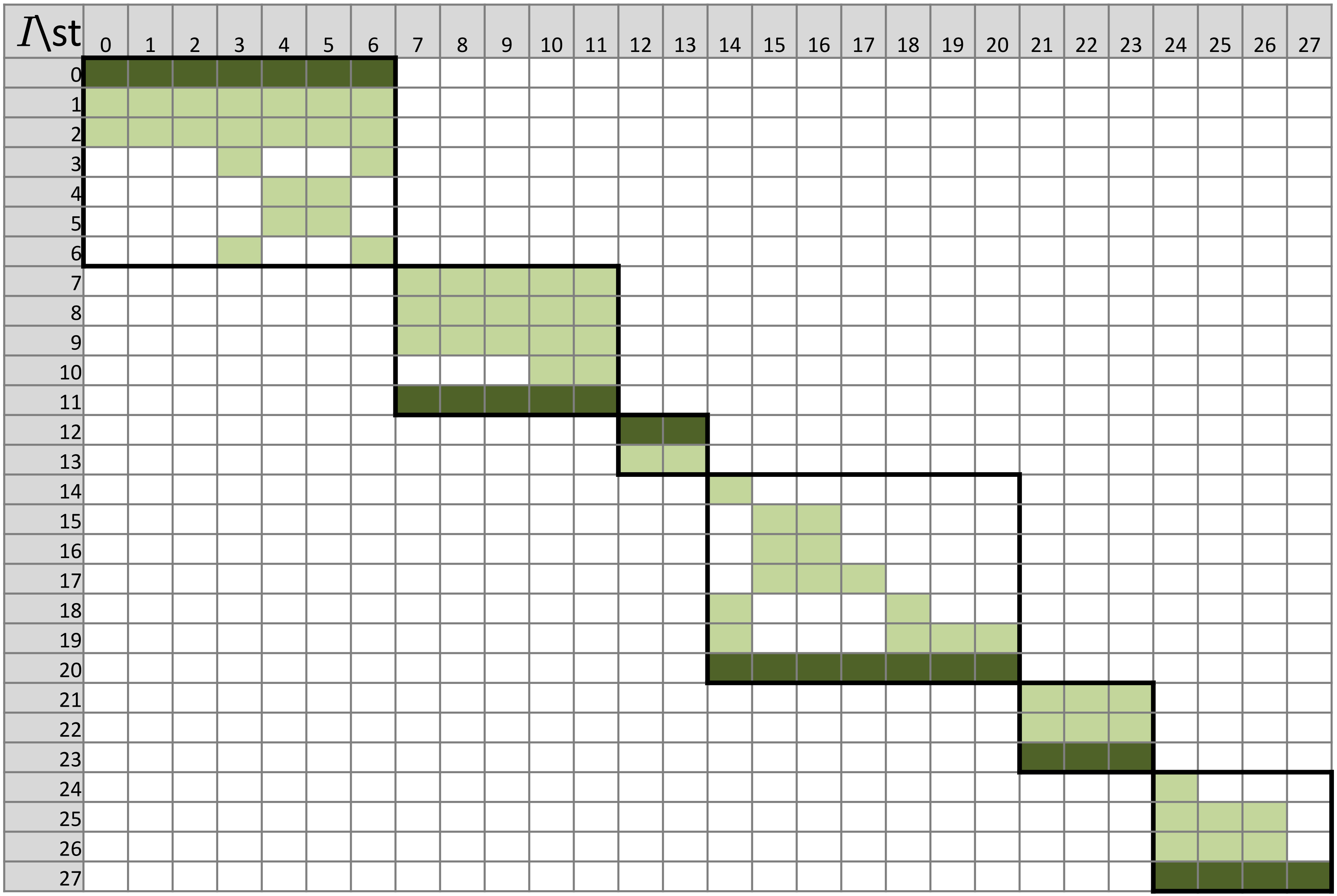
$\Rightarrow$ use Craig interpolation to compute candidates for inductive error invariants.

Computing Abstract Error Traces

Basic Algorithm:

1. Compute the error trace formula from the error trace.
2. Compute a Craig interpolant I_i for each position i in the error trace.
3. Compute the error invariant matrix:
   - for each I_i and j, check whether I_i is an error invariant for j.
4. Choose minimal covering of error trace with inductive error invariants.
5. Output abstract error trace.

Error Invariant Matrix for Faulty Shell Sort



Abstract Error Trace for Faulty Shell Sort

```
 0 int i,j, a[];
 1 int size=3;
 2 int h=1;
 3 h = h*3+1;
 4 assume !(h<=size);
 5 h/=3;
 6 i=h;
 7 assume (i<size);
 8 v=a[i];
 9 j=i;
10 assume !(j>=h && a[j-h]>v);
11 i++;
12 assume (i<size);
13 v=a[i];
14 j=i;
15 assume (j>=h && a[j-h]>v);
16 a[j]=a[j-h];
17 j-=h;
18 assume (j>=h && a[j-h]>v);
19 a[j]=a[j-h];
20 j-=h;
21 assume !(j>=h && a[j-h]>v);
22 assume (i!=j);
23 a[j]=v;
24 i++;
25 assume !(i<size);
26 assume (h==1);
27 assert a[0] == 11 && a[1] == 14;
```

$a[2]=0$

6: i := h;

$a[2]=0 \wedge h=1 \wedge i=h$

11: i := i+1;

$a[2]=0 \wedge h=1 \wedge i=2$

13: v := a[i];

$h=1 \wedge i=2 \wedge v=0 \wedge h \leq j \wedge j \geq 1$

20: j := j - h;

$h=1 \wedge i=2 \wedge v=0 \wedge j=0$

23: a[j] := v;

$a[0]=0$

$a[0] = 11 \wedge a[1] = 14$