Automated Debugging with Error Invariants

Thomas Wies

New York University

joint work with
Jürgen Christ , Evren Ermis (Freiburg University),
Martin Schäf (SRI), Daniel Schwartz-Narbonne (NYU)

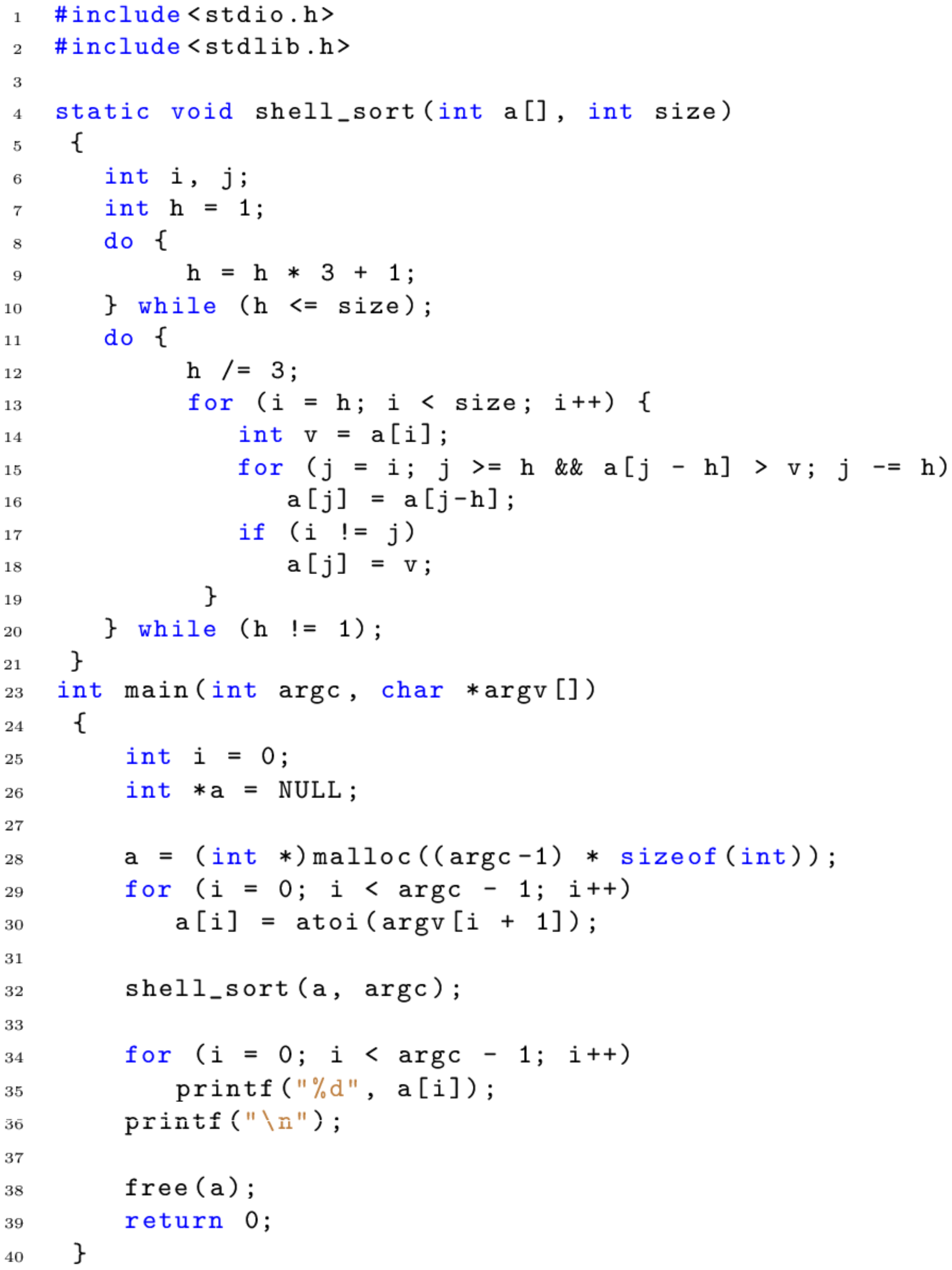
Faulty Shell Sort

[Cleve, Zeller ICSE‘05]

Program

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

On the input sequence 14, 11 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;
 }
```

Faulty Shell Sort

[Cleve, Zeller ICSE‘05]

Program

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

```c
#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;
```

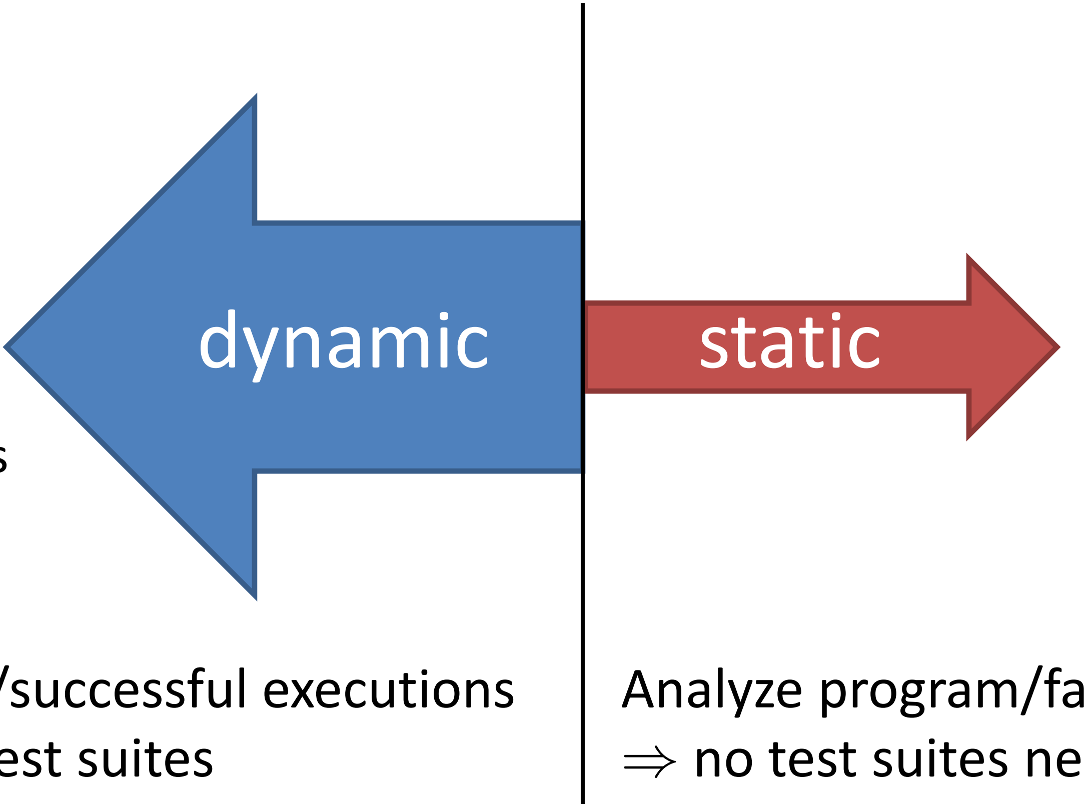
Fault Localization: given a faulty program execution, automatically identify root cause of the error.

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

```
32: shell_sort(a, argc);
```

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

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

Landscape of Fault Localization Techniques



delta debugging
distance metrics
nearest neighbors
cause-effect chains
...

slicing
max-sat

Compare failing/successful executions
⇒ needs good test suites

Analyze program/failing execution
⇒ no test suites needed

Faulty Shell Sort

[Cleve, Zeller ICSE‘05]

Program

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

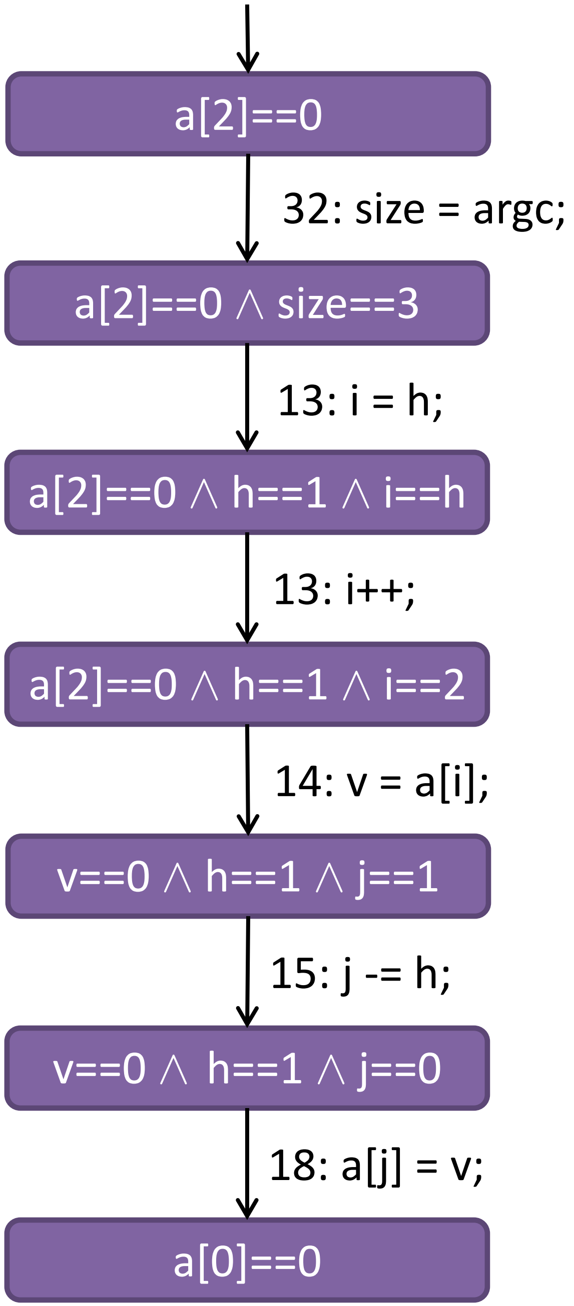
```
#include<stdio.h>
#include<stdlib.h>

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;
```

State-of-the-art **static fault localization** tool identifies **18 statements** as potential locations of the root cause.

```
       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;
 }
```

New Static Approach: Fault Abstraction

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  static void shell_sort(int a[], int size)
5   {
6     int i, j;
7     int h = 1;
8     do {
9          h = h * 3 + 1;
10    } while (h <= size);
11    do {
12         h /= 3;
13         for (i = h; i < size; i++) {
14             int v = a[i];
15             for (j = i; j >= h && a[j - h] > v; j -= h)
16                 a[j] = a[j-h];
17             if (i != j)
18                 a[j] = v;
19          }
20    } while (h != 1);
21  }
23 int main(int argc, char *argv[])
24  {
25      int i = 0;
26      int *a = NULL;
27
28      a = (int *)malloc((argc-1) * sizeof(int));
29      for (i = 0; i < argc - 1; i++)
30         a[i] = atoi(argv[i + 1]);
31
32      shell_sort(a, argc);
33
34      for (i = 0; i < argc - 1; i++)
35         printf("%d", a[i]);
36      printf("\n");
37
38      free(a);
39      return 0;
40  }
```



a[2]==0

32: size = argc;

a[2]==0 ∧ size==3

13: i = h;

a[2]==0 ∧ h==1 ∧ i==h

13: i++;

a[2]==0 ∧ h==1 ∧ i==2

14: v = a[i];

v==0 ∧ h==1 ∧ j==1

15: j -= h;

v==0 ∧ h==1 ∧ j==0

18: a[j] = v;

a[0]==0

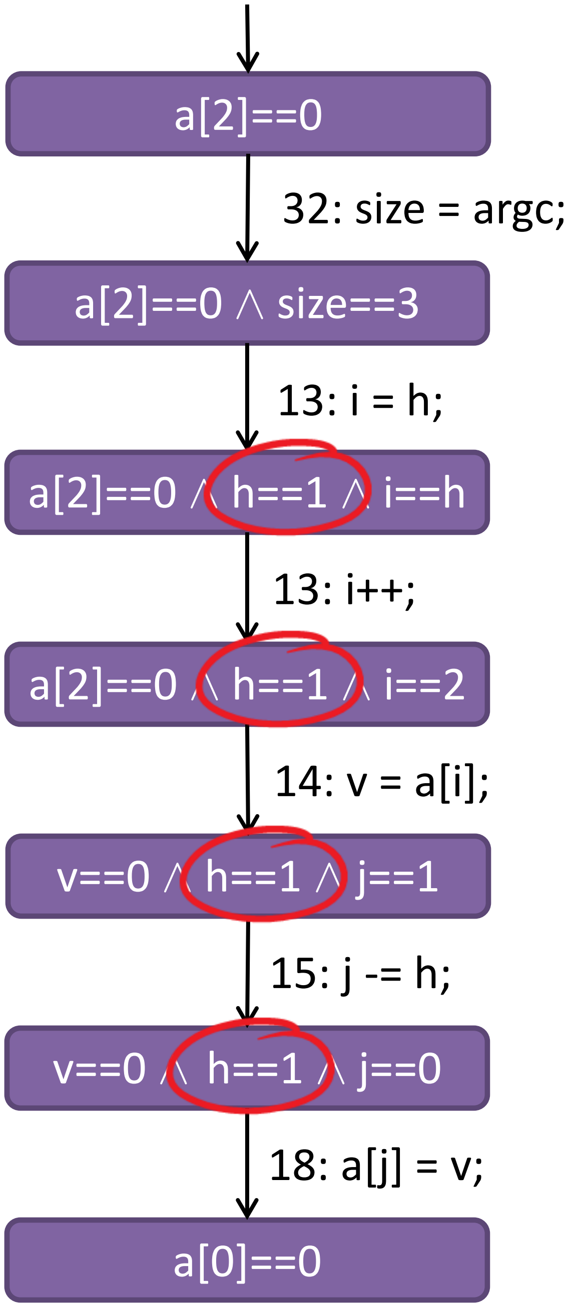
```
#include<stdio.h>
#include<stdlib.h>

static void shell_sort(int a[], int size)
 {
   int i, j;
```

```
11: do {
12:     h /= 3;
13:     for (i = h; i < size; i++) {
14:       int v = a[i];
15:       for (j = i; j >= h && a[j-h] > v;
               j -= h)
16:          a[j] = a[j-h];
17:       if (i != j)
18:          a[j] = v;
19:     }
20: } while (h != 1);
```

```
    a = (int *)malloc((argc-1) * sizeof(int));
```

Things happen in the last iteration of the do-while loop.

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

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



a[2]==0

32: size = argc;

a[2]==0 ∧ size==3

13: i = h;

a[2]==0 ∧ h==1 ∧ i==h

13: i++;

a[2]==0 ∧ h==1 ∧ i==2

14: v = a[i];

v==0 ∧ h==1 ∧ j==1

15: j -= h;

v==0 ∧ h==1 ∧ j==0

18: a[j] = v;

a[0]==0

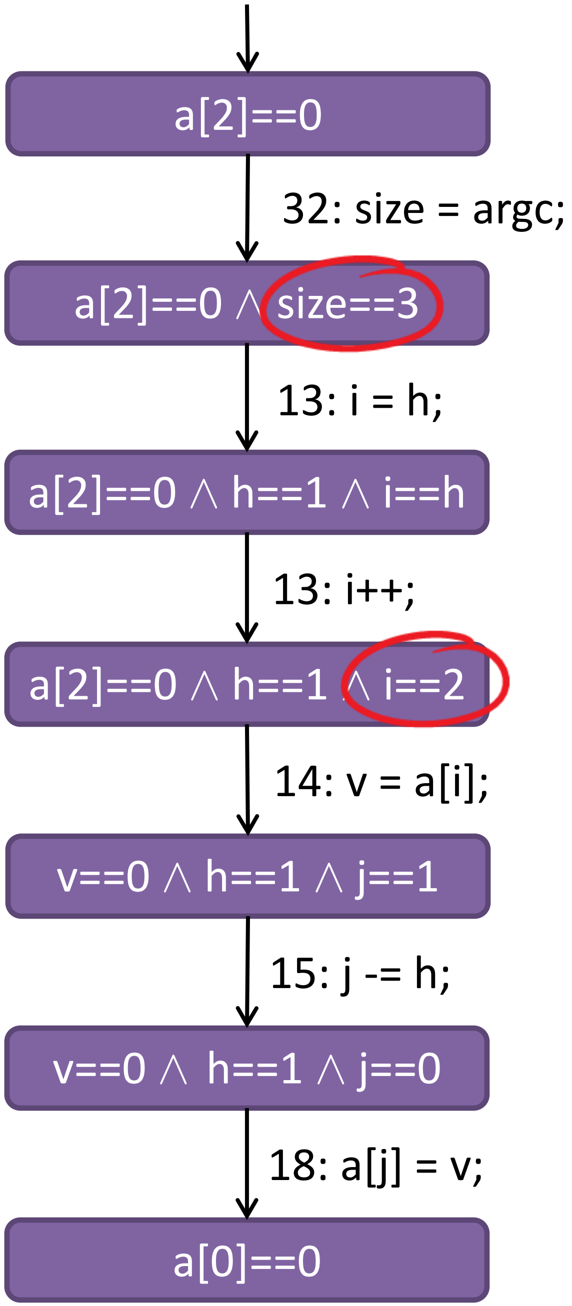
```
#include<stdio.h>
#include<stdlib.h>

static void shell_sort(int a[], int size)
 {
   int i, j;
```

```
11: do {
12:     h /= 3;
13:     for (i = h; i < size; i++) {
14:       int v = a[i];
15:       for (j = i; j >= h && a[j-h] > v;
                 j -= h)
16:         a[j] = a[j-h];
17:       if (i != j)
18:         a[j] = v;
19:     }
20: } while (h != 1);
```

```
    a = (int *)malloc((argc-1) * sizeof(int));
```

... in the last iteration of the outer for loop.

```
    shell_sort(a, argc);

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

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



a[2]==0

32: size = argc;

a[2]==0 ∧ size==3

13: i = h;

a[2]==0 ∧ h==1 ∧ i==h

13: i++;

a[2]==0 ∧ h==1 ∧ i==2

14: v = a[i];

v==0 ∧ h==1 ∧ j==1

15: j -= h;

v==0 ∧ h==1 ∧ j==0

18: a[j] = v;

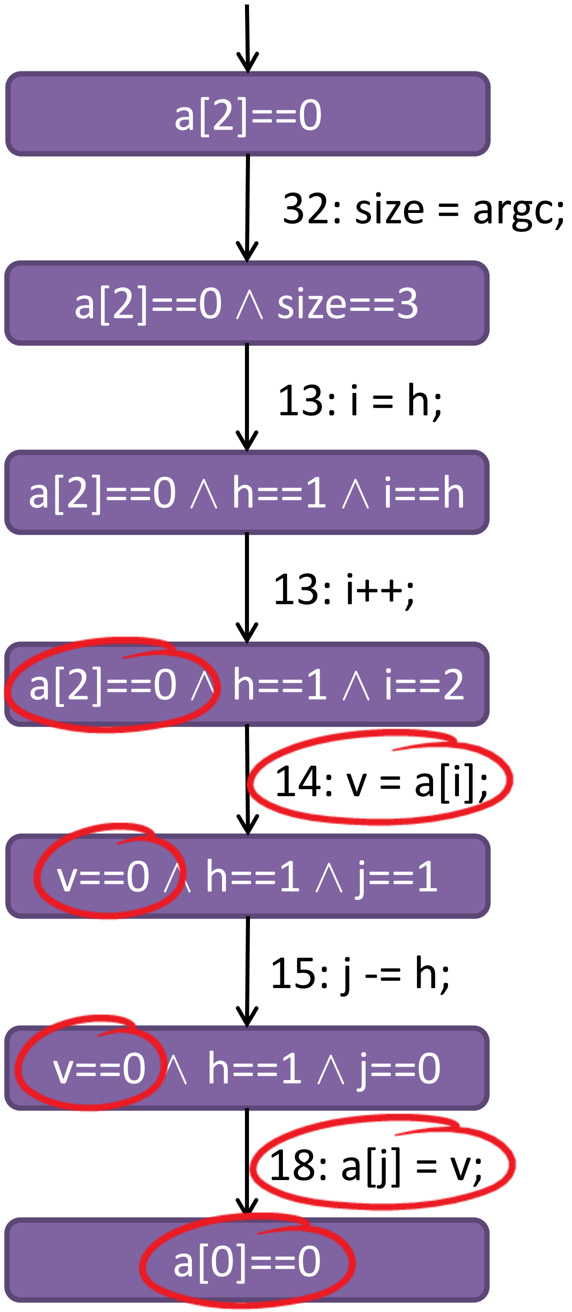
a[0]==0

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

static void shell_sort(int a[], int size)
{
   int i, j;
```

```
11: do {
12:     h /= 3;
13:     for (i = h; i < size; i++) {
14:       int v = a[i];
15:       for (j = i; j >= h && a[j-h] > v;
                j -= h)
16:          a[j] = a[j-h];
17:       if (i != j)
18:          a[j] = v;
19:     }
20: } while (h != 1);
```

The swap in the insertion sort loop is where the array gets its wrong entry.

```
      a = (int *)malloc((argc-1) * sizeof(int));

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

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



a[2]==0

32: size = argc;

a[2]==0 ∧ size==3

13: i = h;

a[2]==0 ∧ h==1 ∧ i==h

13: i++;

a[2]==0 ∧ h==1 ∧ i==2

14: v = a[i];

v==0 ∧ h==1 ∧ j==1

15: j -= h;

v==0 ∧ h==1 ∧ j==0

18: a[j] = v;

a[0]==0

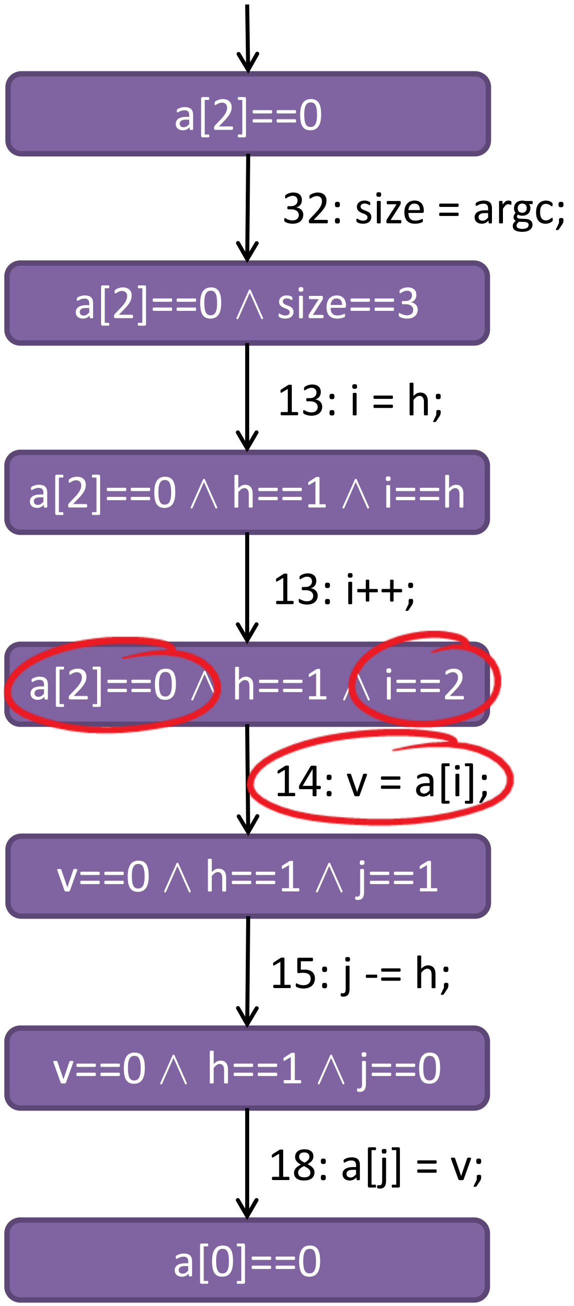
```
#include<stdio.h>
#include<stdlib.h>

static void shell_sort(int a[], int size)
 {
   int i, j;
```

```
11: do {
12:     h /= 3;
13:     for (i = h; i < size; i++) {
14:       int v = a[i];
15:       for (j = i; j >= h && a[j-h] > v;
                 j -= h)
16:         a[j] = a[j-h];
17:       if (i != j)
18:         a[j] = v;
19:     }
20: } while (h != 1);
```

```
    a = (int *)malloc((argc-1) * sizeof(int));
```

... because the array read is outside the array bounds.

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

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



a[2]==0

32: size = argc;

a[2]==0 ∧ size==3

13: i = h;

a[2]==0 ∧ h==1 ∧ i==h

13: i++;

a[2]==0 ∧ h==1 ∧ i==2

14: v = a[i];

v==0 ∧ h==1 ∧ j==1

15: j -= h;

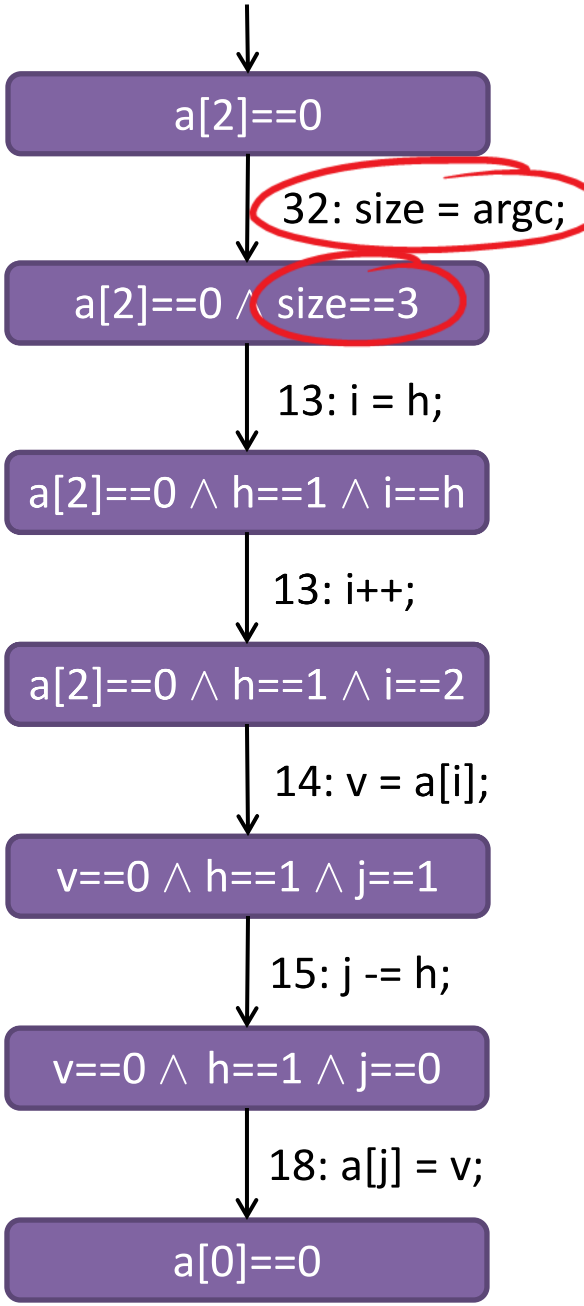
v==0 ∧ h==1 ∧ j==0

18: a[j] = v;

a[0]==0

...because argc should have been decremented by 1. Ouch!

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  static void shell_sort(int a[], int size)
5   {
10     } while (h <= size);
11     do {
12         h /= 3;
13         for (i = h; i < size; i++) {
14             int v = a[i];
15             for (j = i; j >= h && a[j - h] > v; j -= h)
16                 a[j] = a[j-h];
17             if (i != j)
18                 a[j] = v;
19         }
20     } while (h != 1);
21  }
23 int main(int argc, char *argv[])
24  {
25     int i = 0;
26     int *a = NULL;
27
28     a = (int *)malloc((argc-1) * sizeof(int));
29     for (i = 0; i < argc - 1; i++)
30         a[i] = atoi(argv[i + 1]);
31
32: shell_sort(a, argc);
33
34     for (i = 0; i < argc - 1; i++)
35         printf("%d", a[i]);
36     printf("\n");
37
38     free(a);
39     return 0;
40  }
```



a[2]==0

32: size = argc;

a[2]==0 ∧ size==3

13: i = h;

a[2]==0 ∧ h==1 ∧ i==h

13: i++;

a[2]==0 ∧ h==1 ∧ i==2

14: v = a[i];

v==0 ∧ h==1 ∧ j==1

15: j -= h;

v==0 ∧ h==1 ∧ j==0

18: a[j] = v;

a[0]==0

Fault Abstraction

Input:

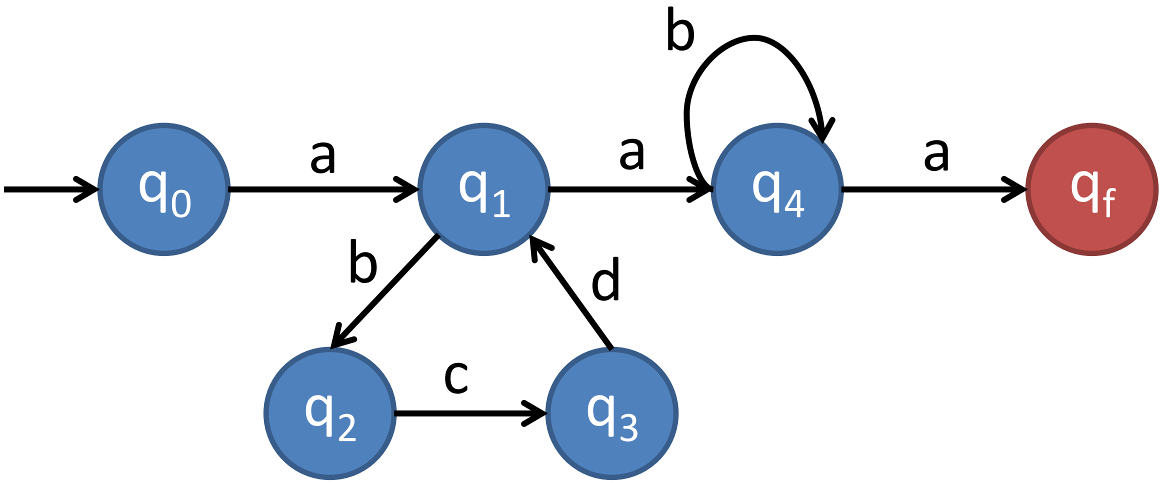
error path = program path + input state + expected output state

```
 1 size = argc;
 2 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);
```

Output: abstract slice of error path

Basic Idea of Fault Abstraction

- Consider a finite automaton



- A word that is accepted by the automaton:

a b c d a b b b a

Programs as Automata

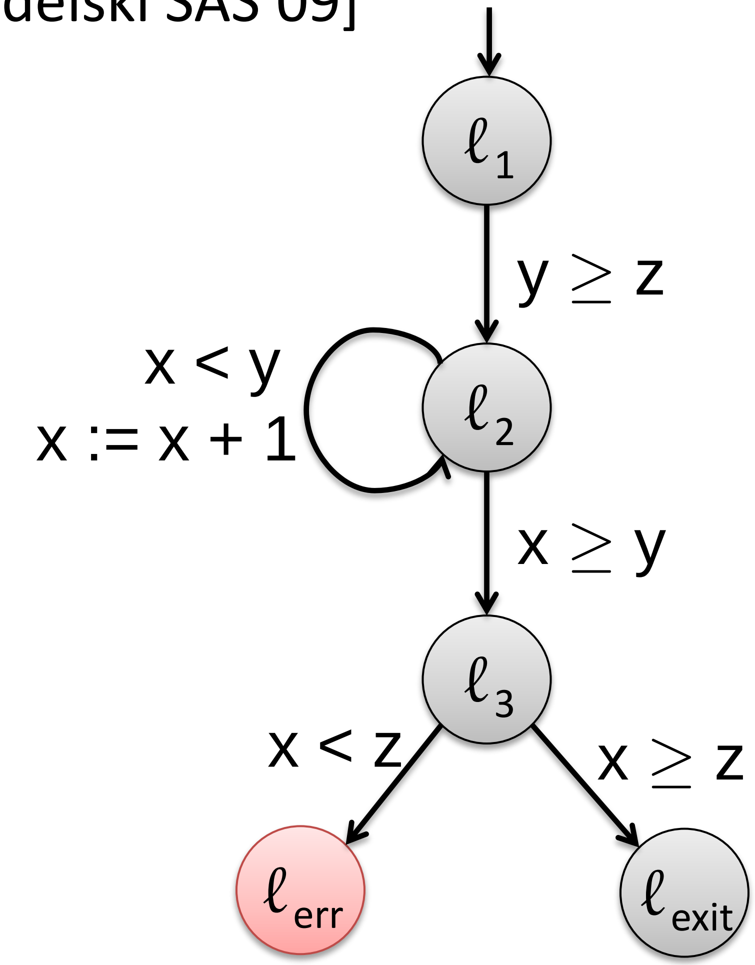
[Heizmann, Hoenicke, Podelski SAS'09]

```
1:  assume y ≥ z;
2:  while x < y do  x := x + 1;
3:  assert x ≥ z;
```

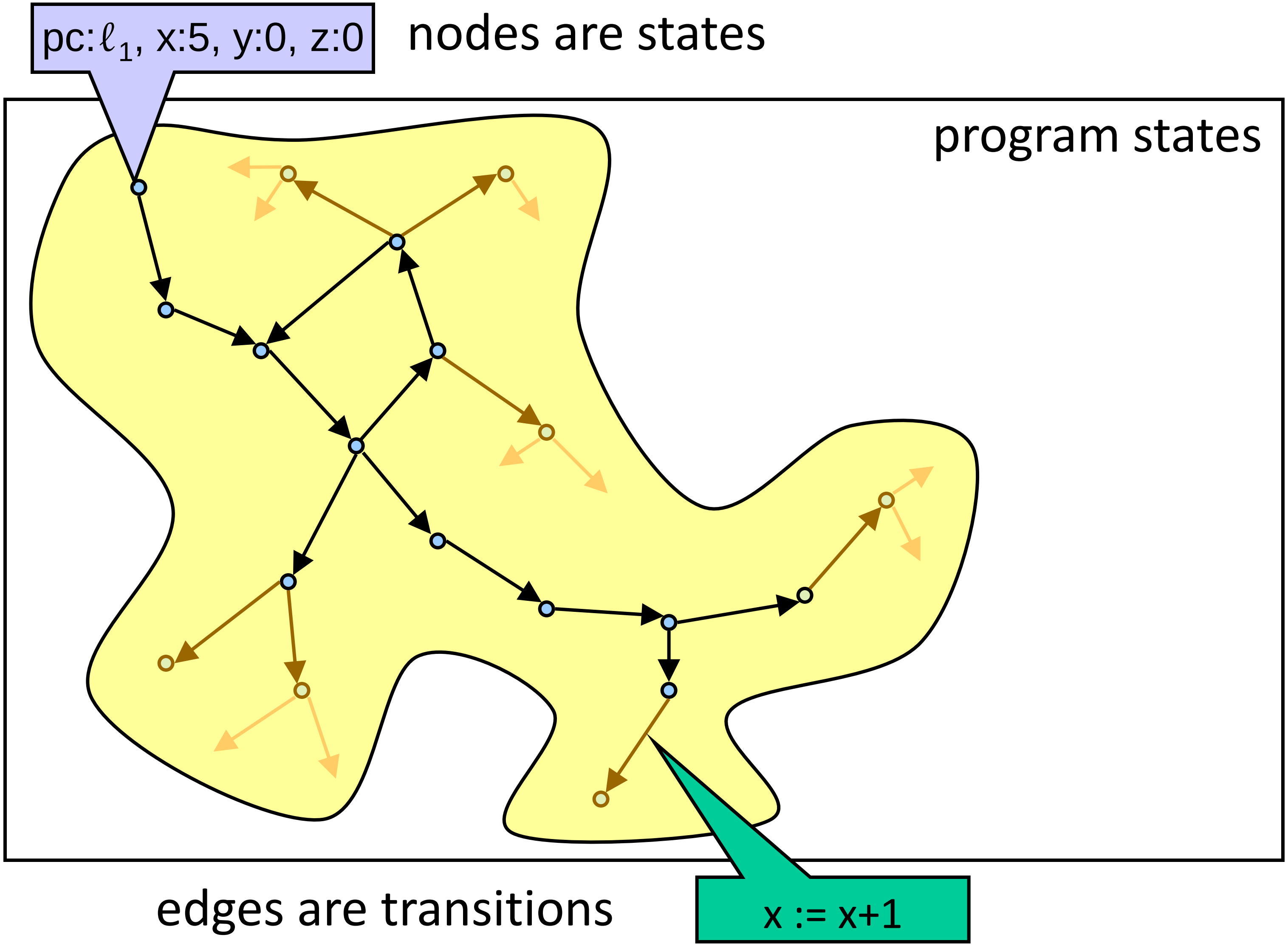
error path = finite word of program statements

program = automaton that accepts error paths

fault local. = eliminate loops in the error path



State/Transition Graph



pc:ℓ_1, x:5, y:0, z:0

nodes are states

program states

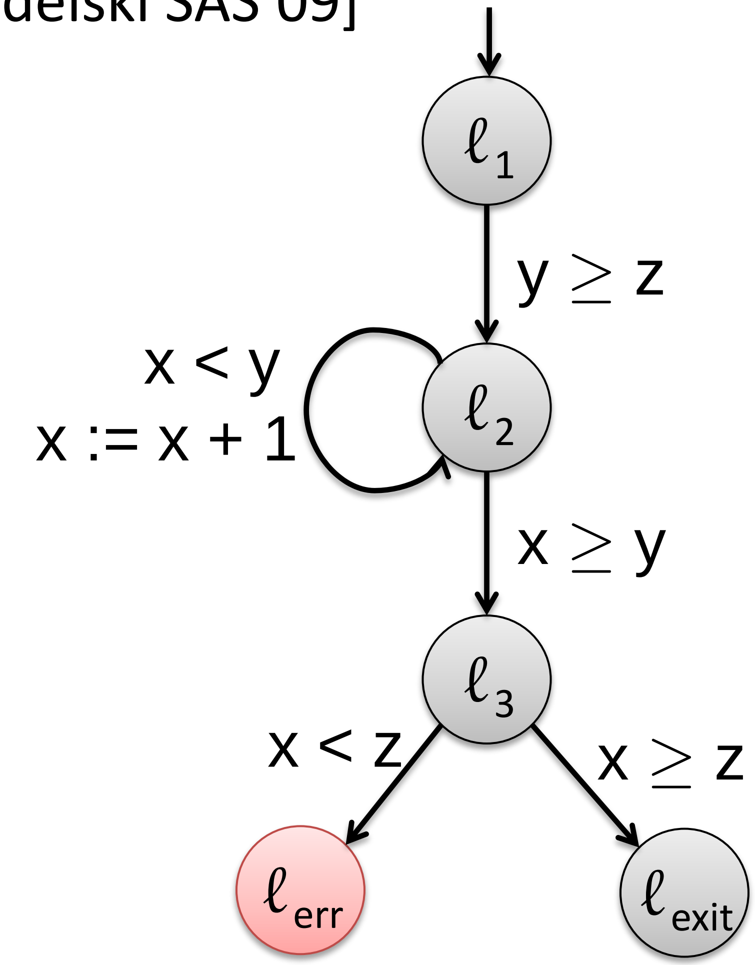
edges are transitions

x := x+1

Programs as Automata

[Heizmann, Hoenicke, Podelski SAS'09]

```
1:  assume y ≥ z;
2:  while x < y do  x := x + 1;
3:  assert x ≥ z;
```



error path = finite word of program statements

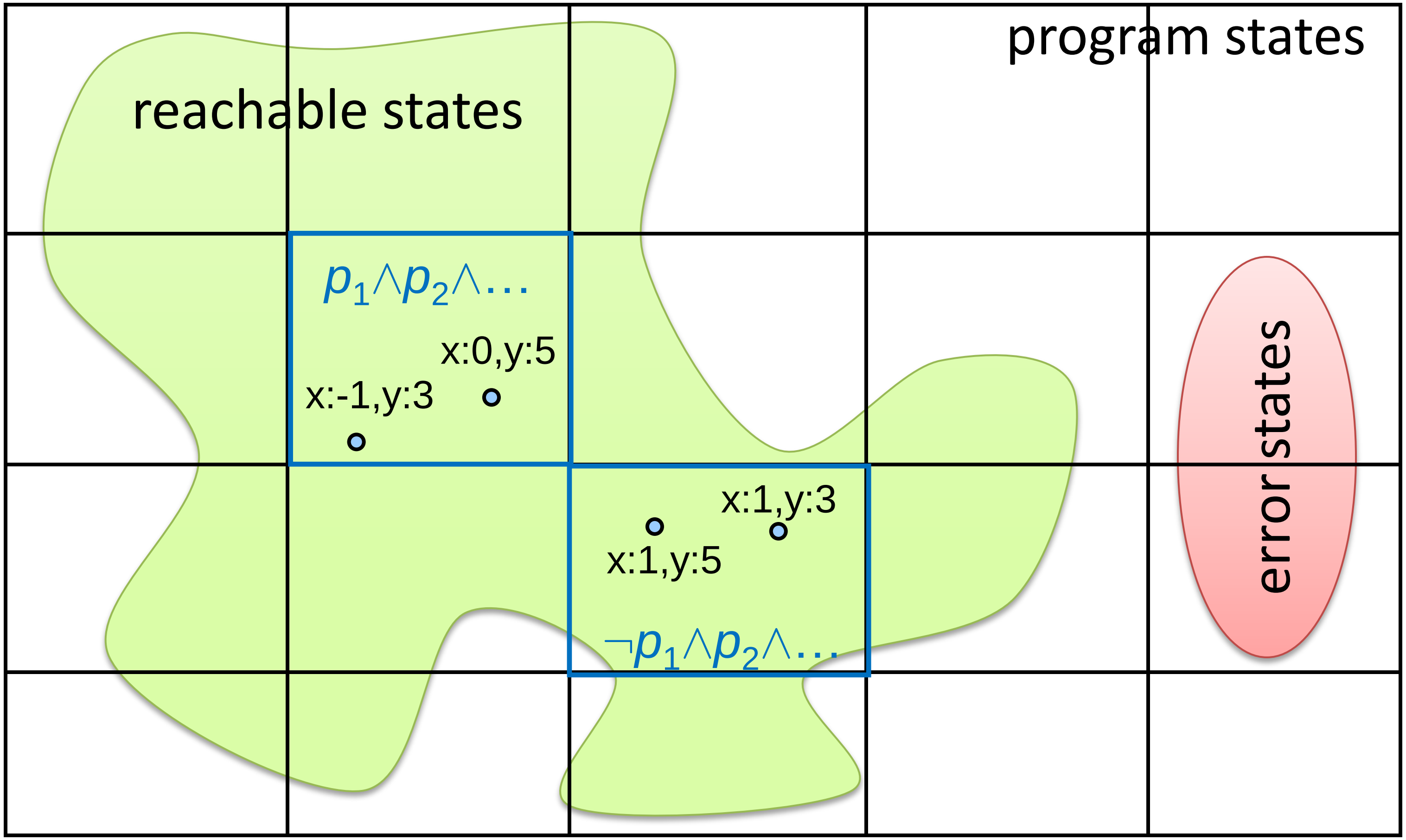
program = automaton that accepts error paths

fault local. = eliminate loops in the error path

But there are no repeating states in executions of error paths?

Predicate Abstraction

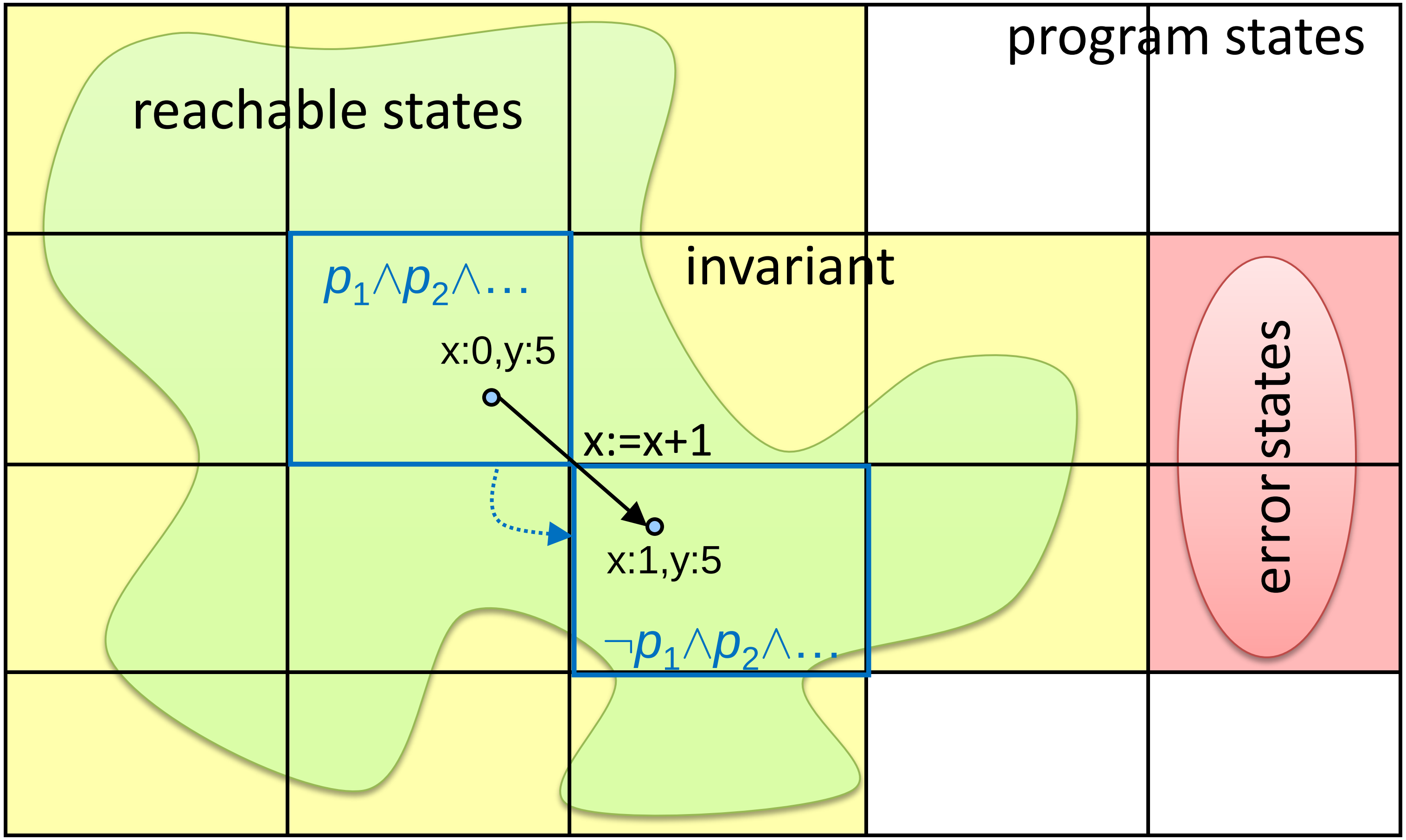
[Graf, Saïdi ‘97], [Clarke et al. ‘01], ...



$p_1 \equiv x \leq 0$ $p_2 \equiv y > 0$ $\ldots$

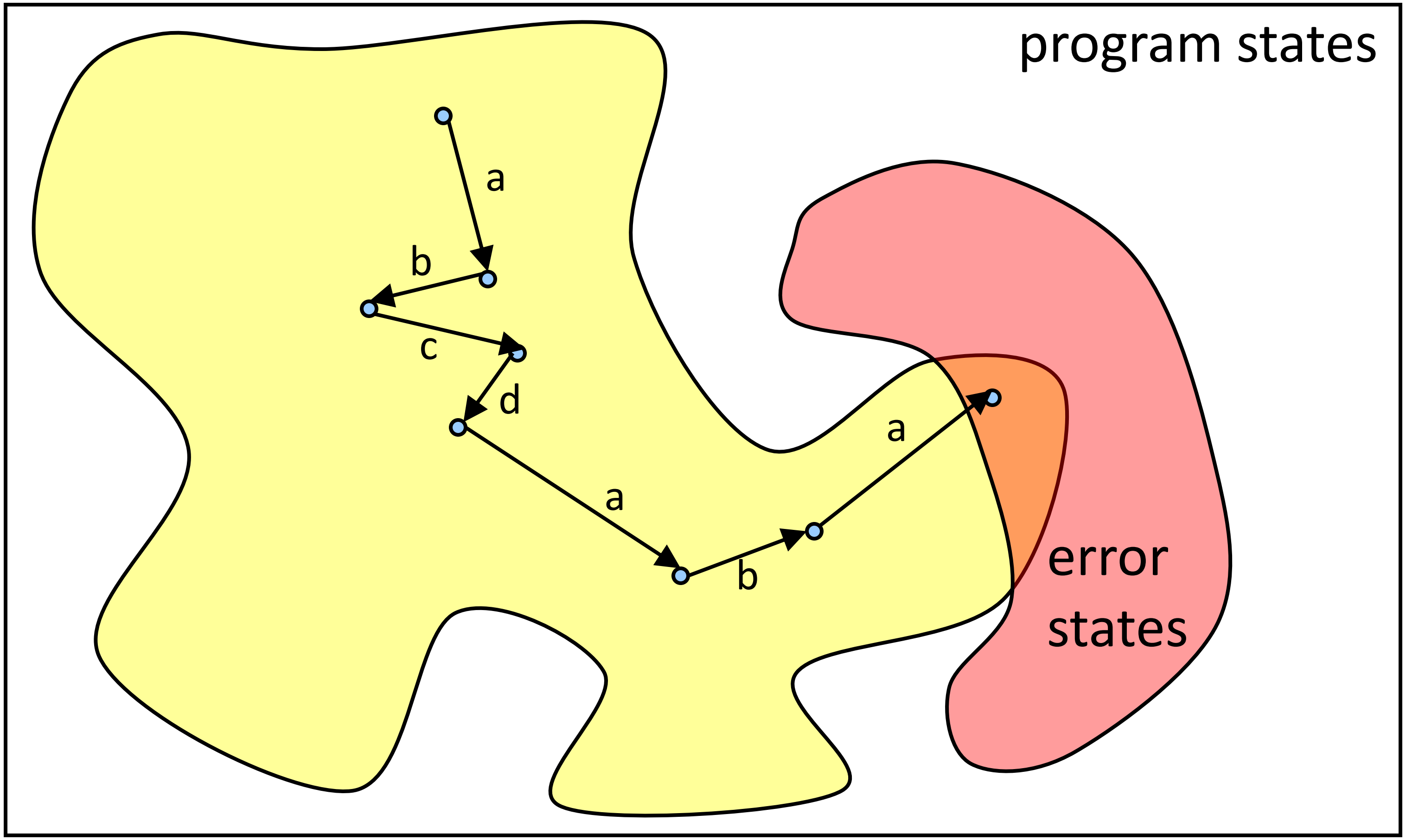
Predicate Abstraction

[Graf, Saïdi '97], [Clarke et al. '01], ...

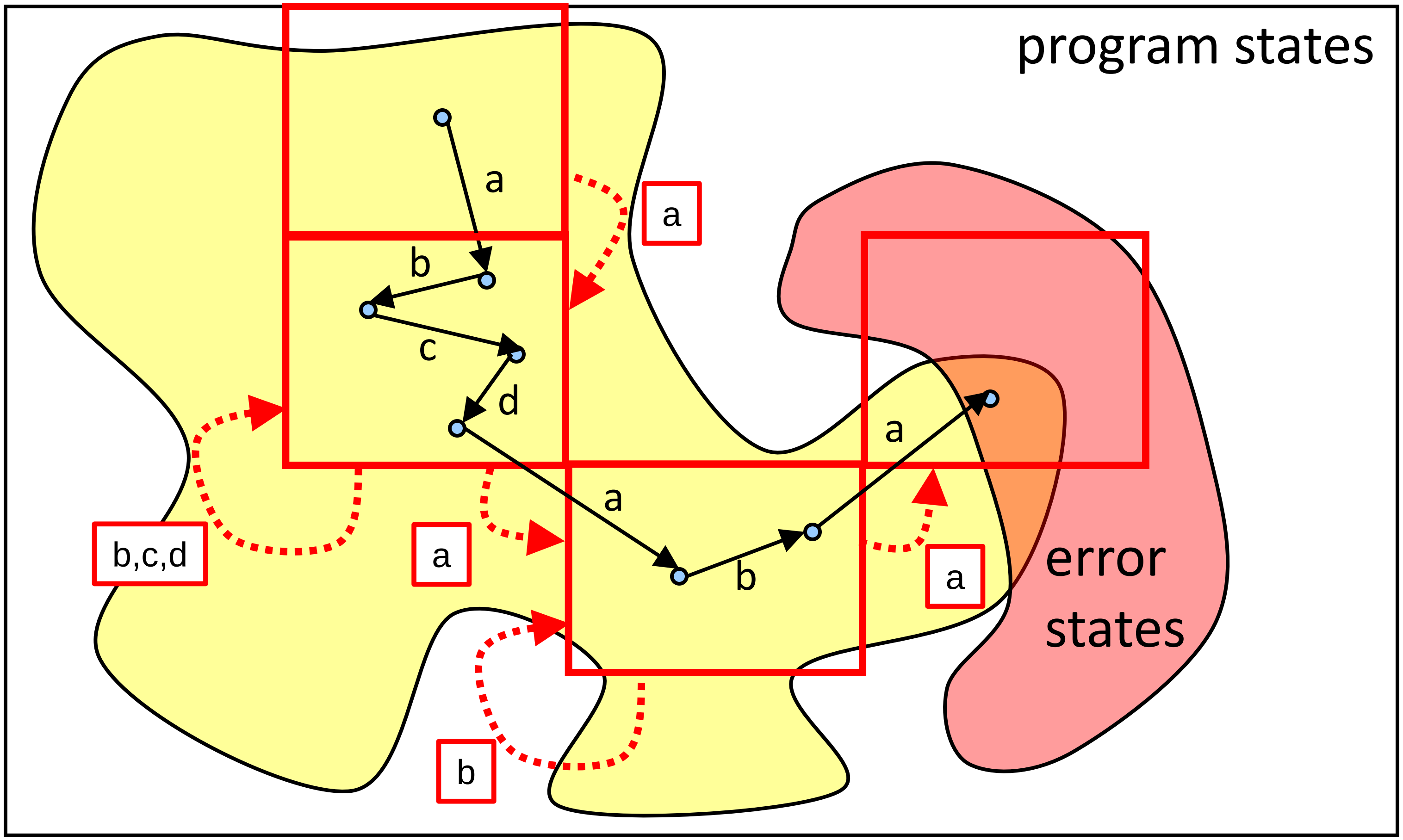


$p_1 \equiv x \leq 0 \quad p_2 \equiv y > 0 \quad \ldots$

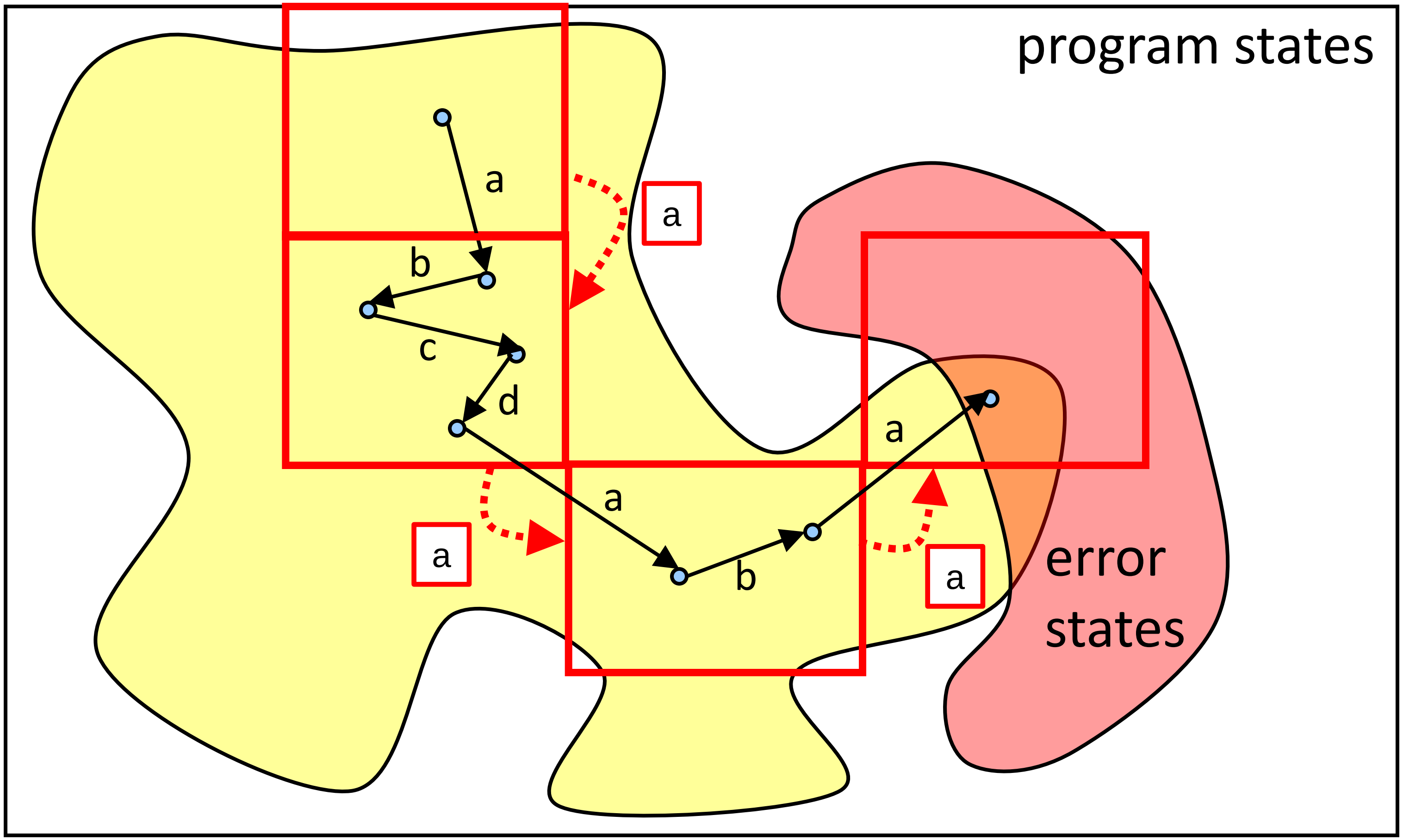
Fault Abstraction



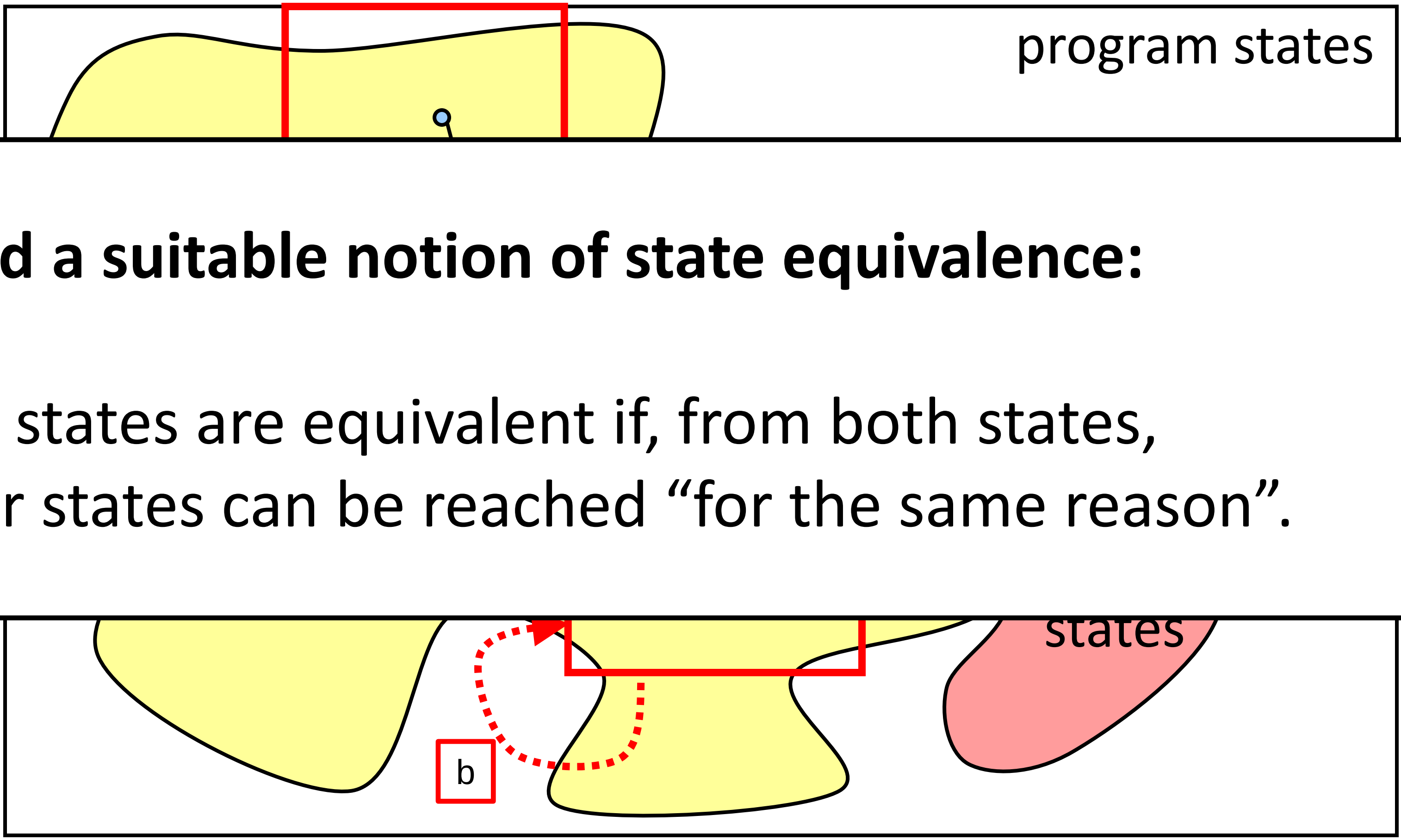
Fault Abstraction



Fault Abstraction



Fault Abstraction



Need a suitable notion of state equivalence:

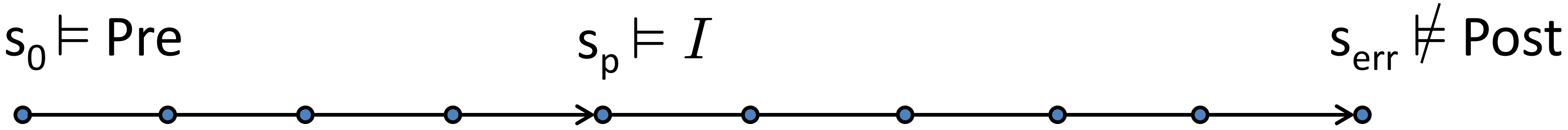
Two states are equivalent if, from both states, error states can be reached “for the same reason”.

Error Invariants

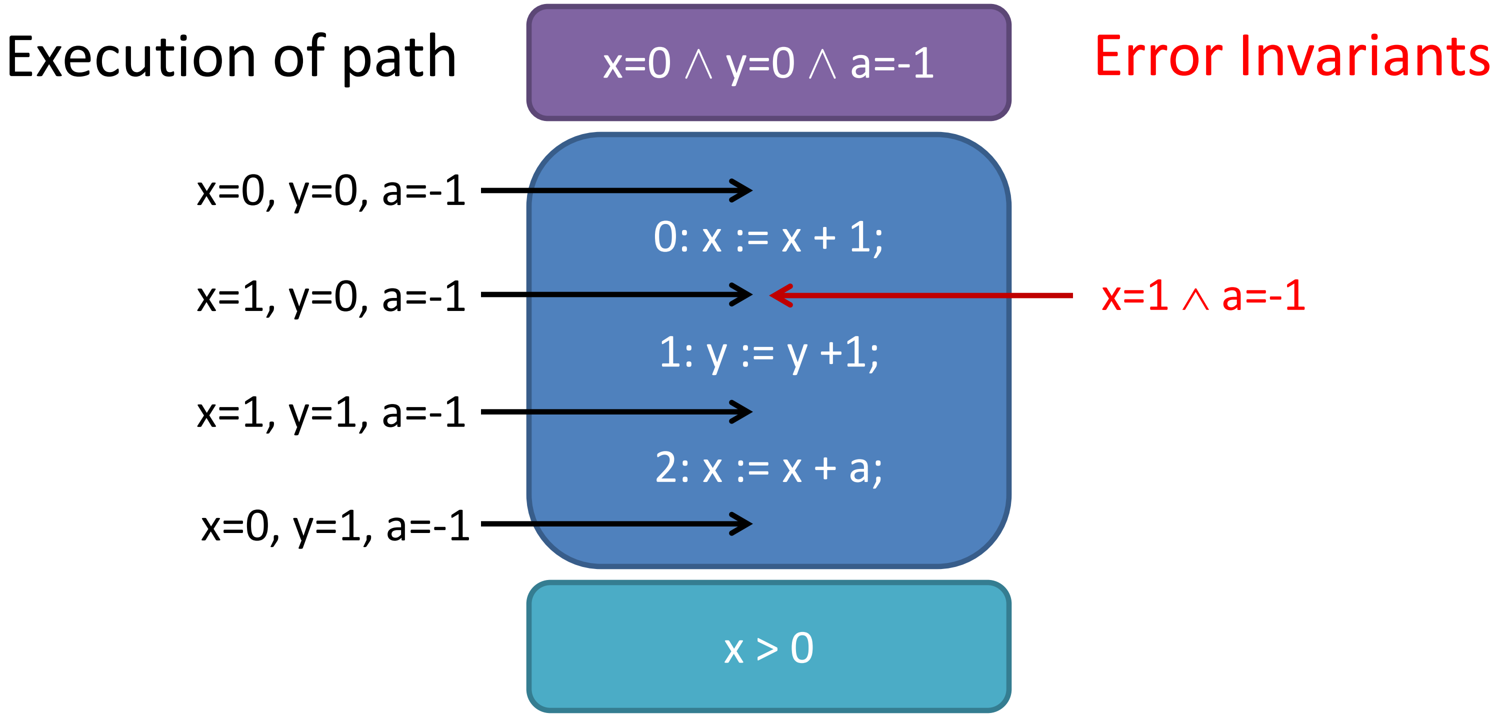
[Ermis, Schäf, Wies FM'12]

An error invariant I for a position p in an error trace τ is a formula over program variables such that

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



Error Invariants



Execution of path

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

Error Invariants

$x=0, y=0, a=-1$

0: x := x + 1;

$x=1, y=0, a=-1$

$x=1 \land a=-1$

1: y := y +1;

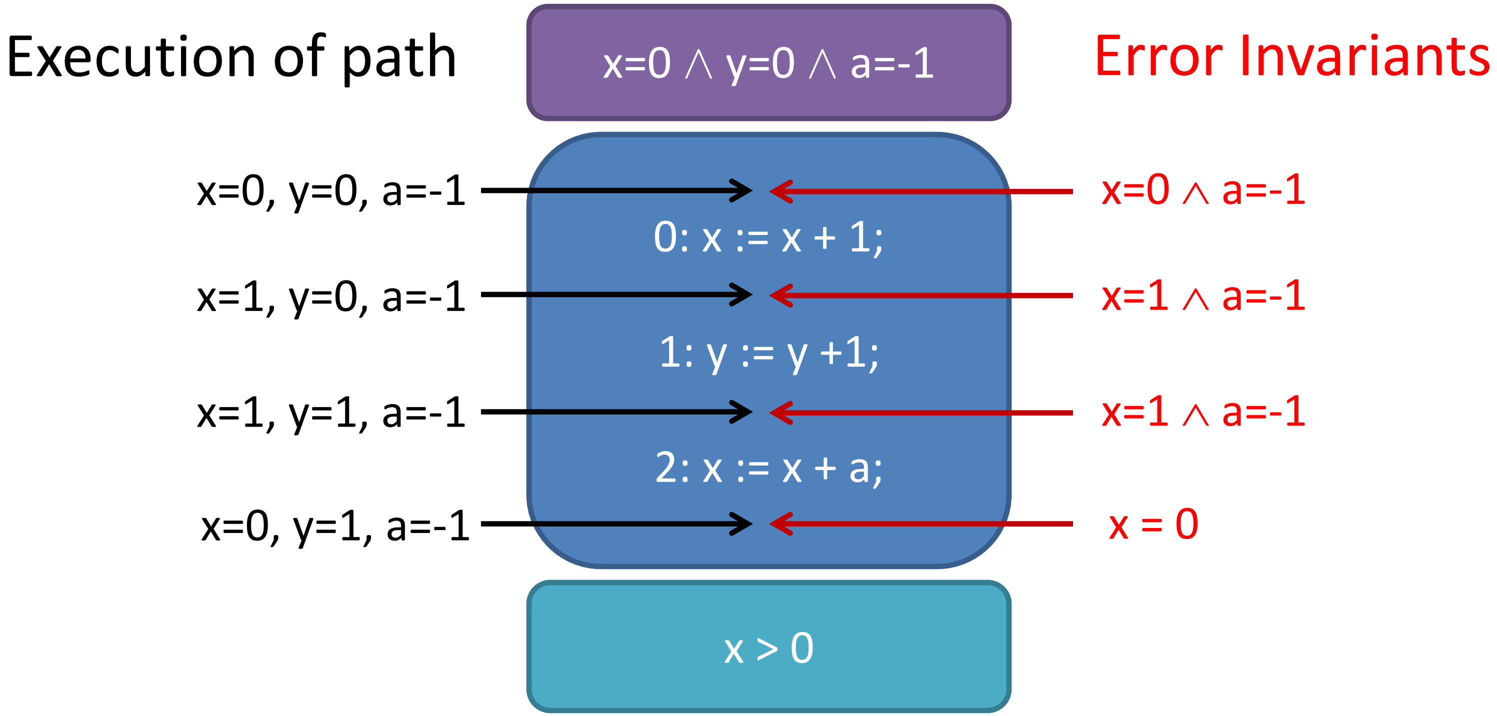
$x=1, y=1, a=-1$

2: x := x + a;

$x=0, y=1, a=-1$

$x > 0$

Error Invariants



Execution of path

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

Error Invariants

x=0, y=0, a=-1 → ← $x=0 \wedge a=-1$

0: x := x + 1;

x=1, y=0, a=-1 → ← $x=1 \wedge a=-1$

1: y := y +1;

x=1, y=1, a=-1 → ← $x=1 \wedge a=-1$

2: x := x + a;

x=0, y=1, a=-1 → ← x = 0

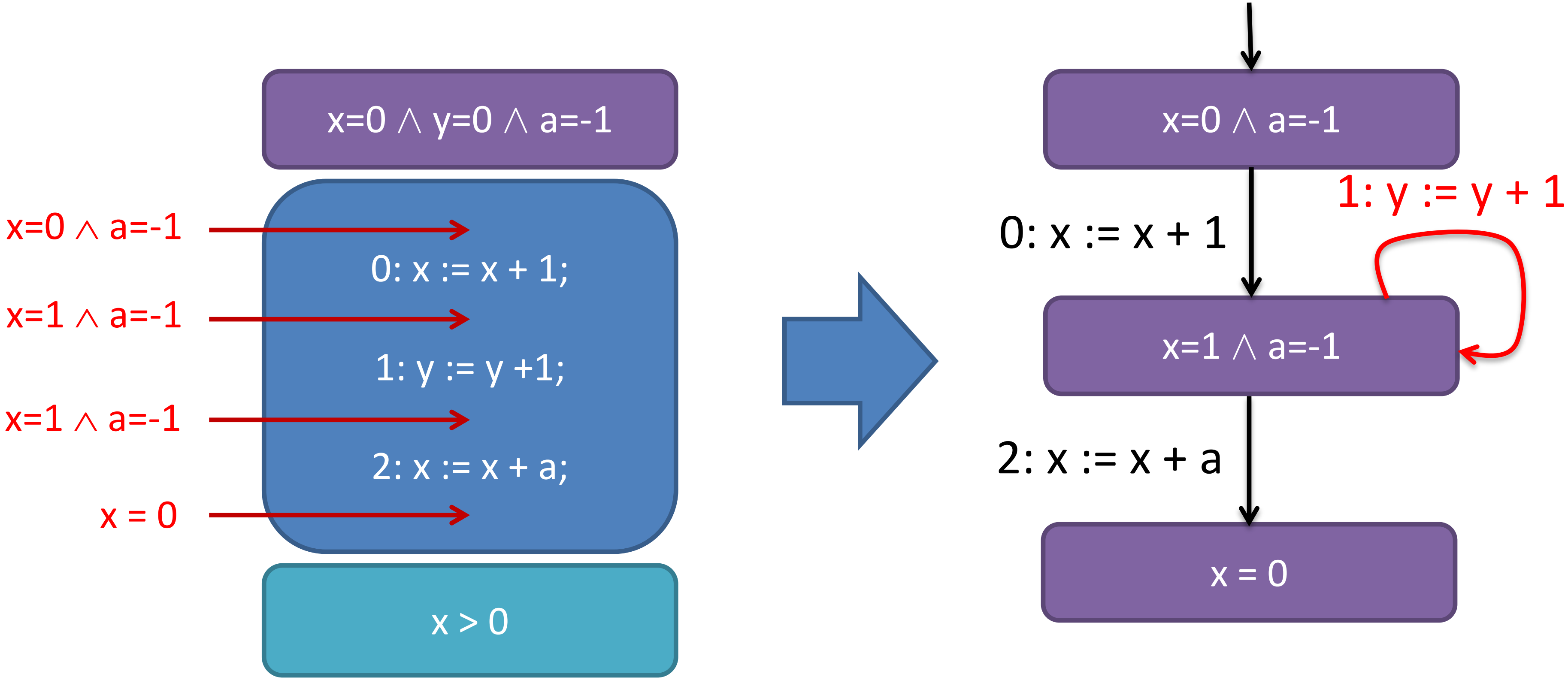
$x > 0$

Information provided by the error invariants

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

Fault Abstraction

[Ermis, Schaef, W. FM’12], [Christ, Ermis, Schaef, W. VMCAI’13]



x=0 ∧ y=0 ∧ a=-1

x=0 ∧ a=-1

0: x := x + 1;

x=1 ∧ a=-1

1: y := y +1;

x=1 ∧ a=-1

2: x := x + a;

x = 0

x > 0

x=0 ∧ a=-1

0: x := x + 1

1: y := y + 1

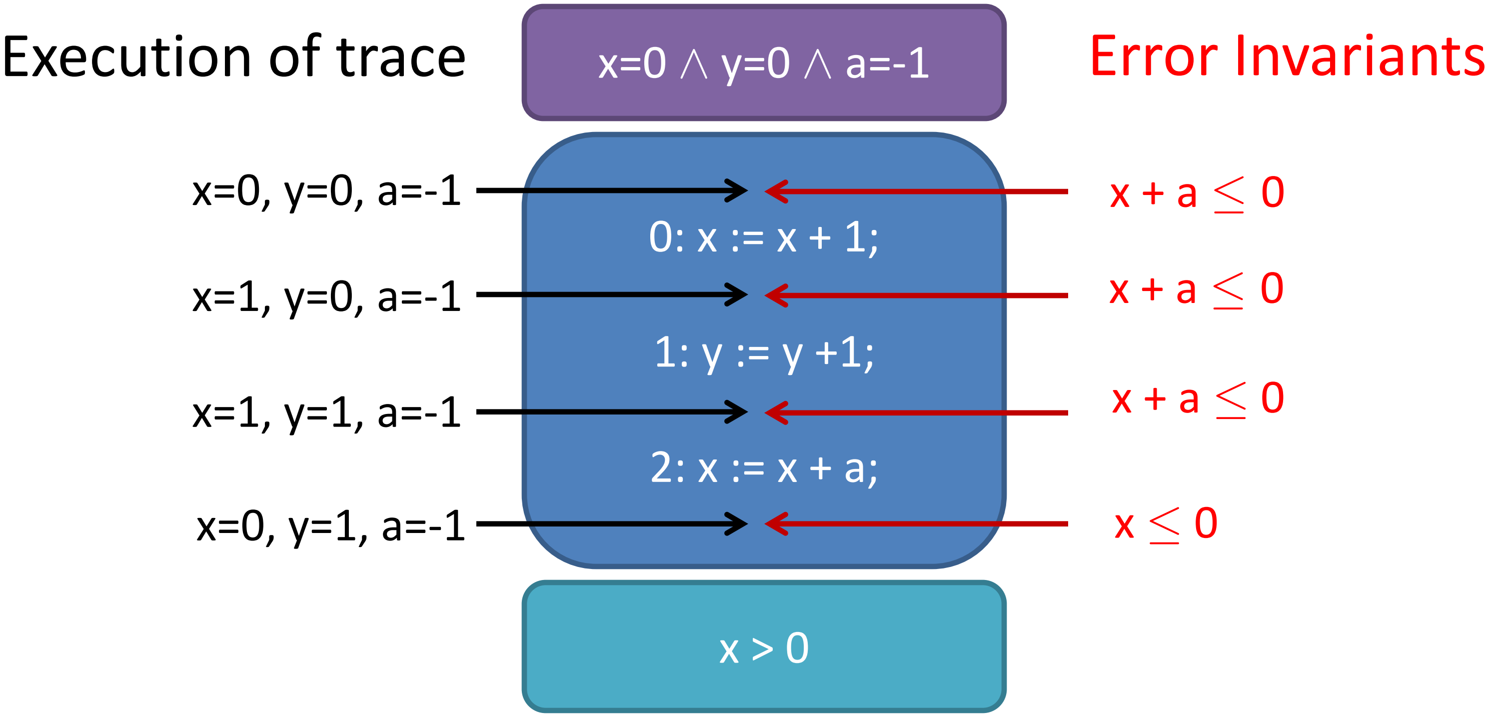
x=1 ∧ a=-1

2: x := x + a

x = 0

Error invariants and error path define a finite automaton that abstracts the error path.

Error Invariants are not unique



Execution of trace

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

Error Invariants

x=0, y=0, a=-1 → ← $x + a \leq 0$

0: x := x + 1;

x=1, y=0, a=-1 → ← $x + a \leq 0$

1: y := y +1;

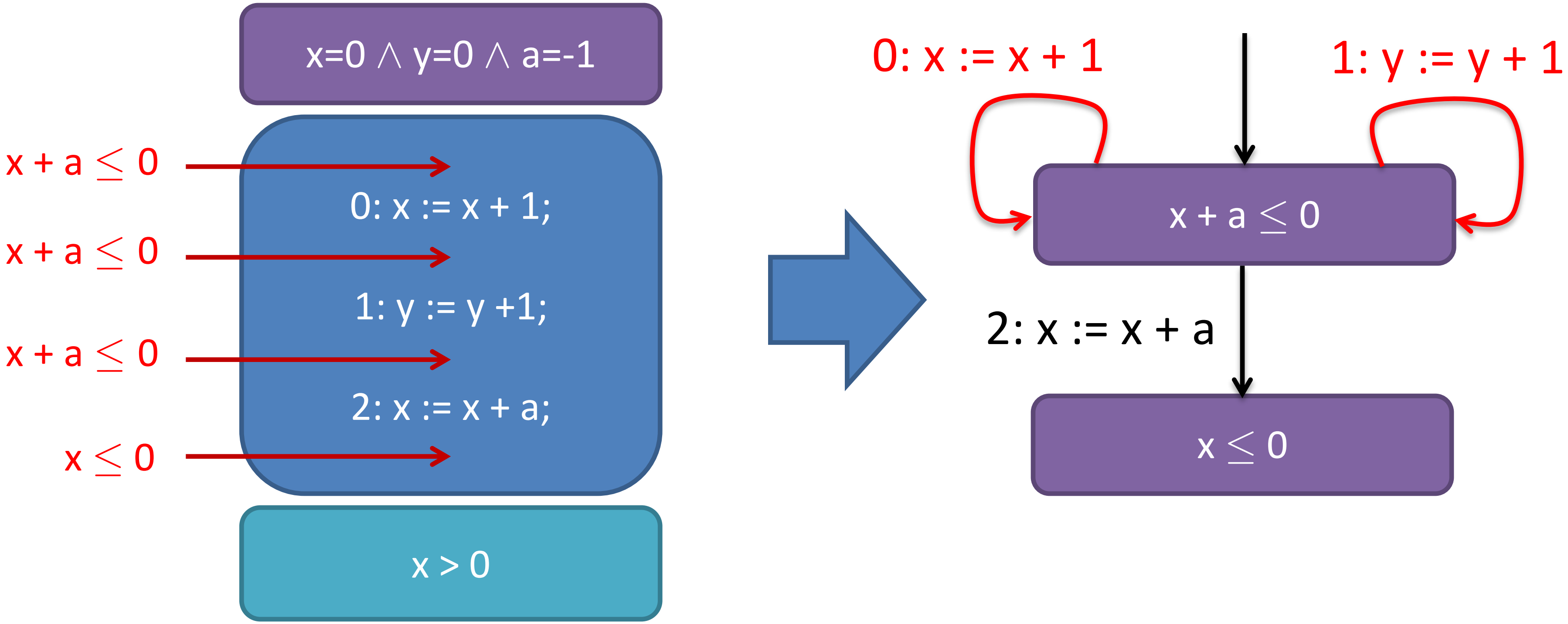
x=1, y=1, a=-1 → ← $x + a \leq 0$

2: x := x + a;

x=0, y=1, a=-1 → ← $x \leq 0$

$x > 0$

Error Invariants are not unique



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

$x + a \leq 0$

0: x := x + 1;

$x + a \leq 0$

1: y := y +1;

$x + a \leq 0$

2: x := x + a;

$x \leq 0$

$x > 0$

0: x := x + 1

1: y := y + 1

$x + a \leq 0$

2: x := x + a

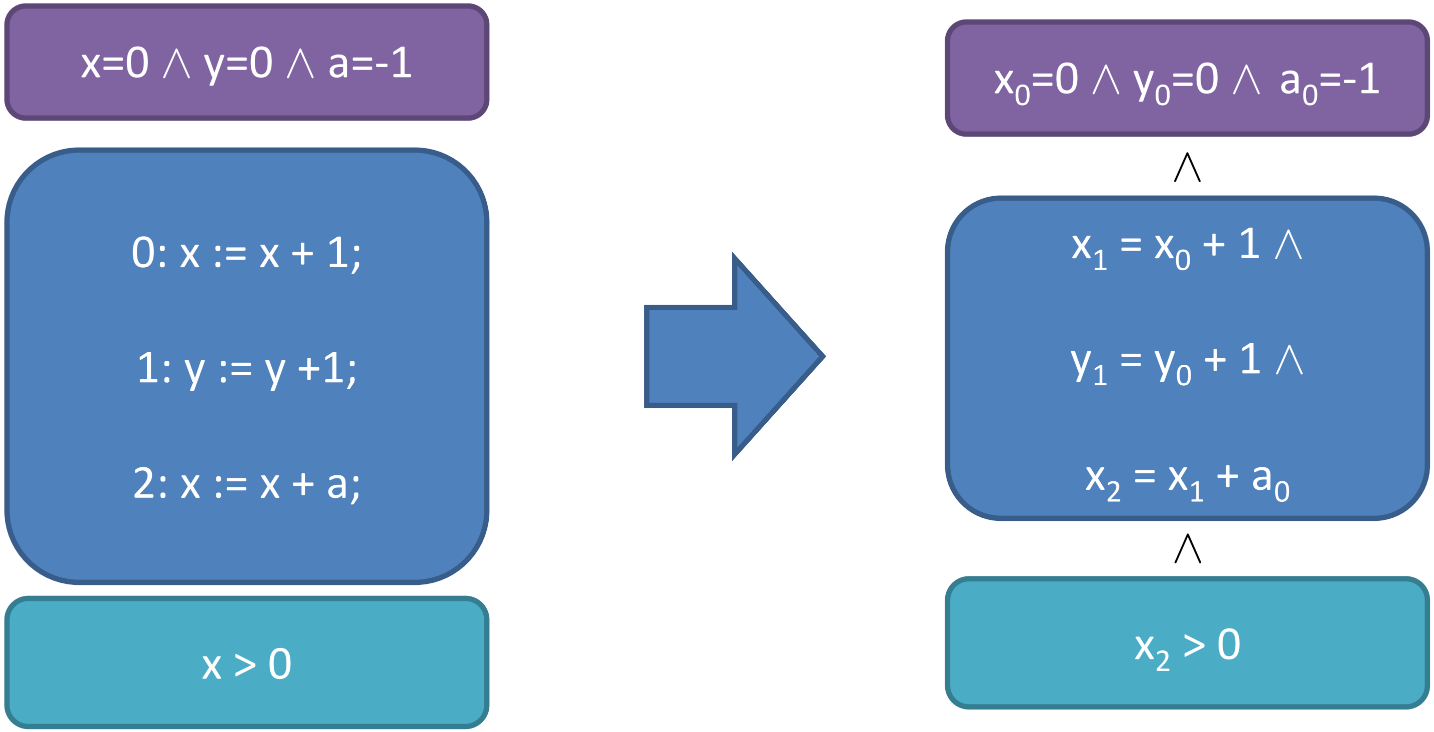
$x \leq 0$

- Can we automatically compute error invariants, given an error path?

- How do we obtain *good* error invariants?

Error Path Formula

Example



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

```
0: x := x + 1;
1: y := y +1;
2: x := x + a;
```

$x > 0$

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

$\wedge$

$x_1 = x_0 + 1 \wedge$

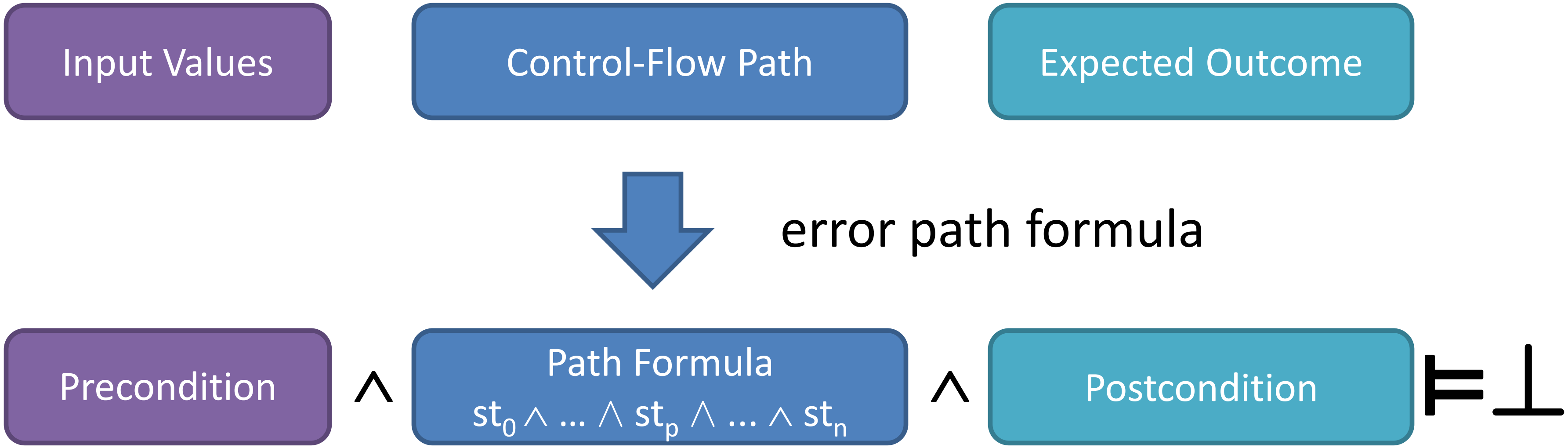
$y_1 = y_0 + 1 \wedge$

$x_2 = x_1 + a_0$

$\wedge$

$x_2 > 0$

Checking Error Invariants



Input Values

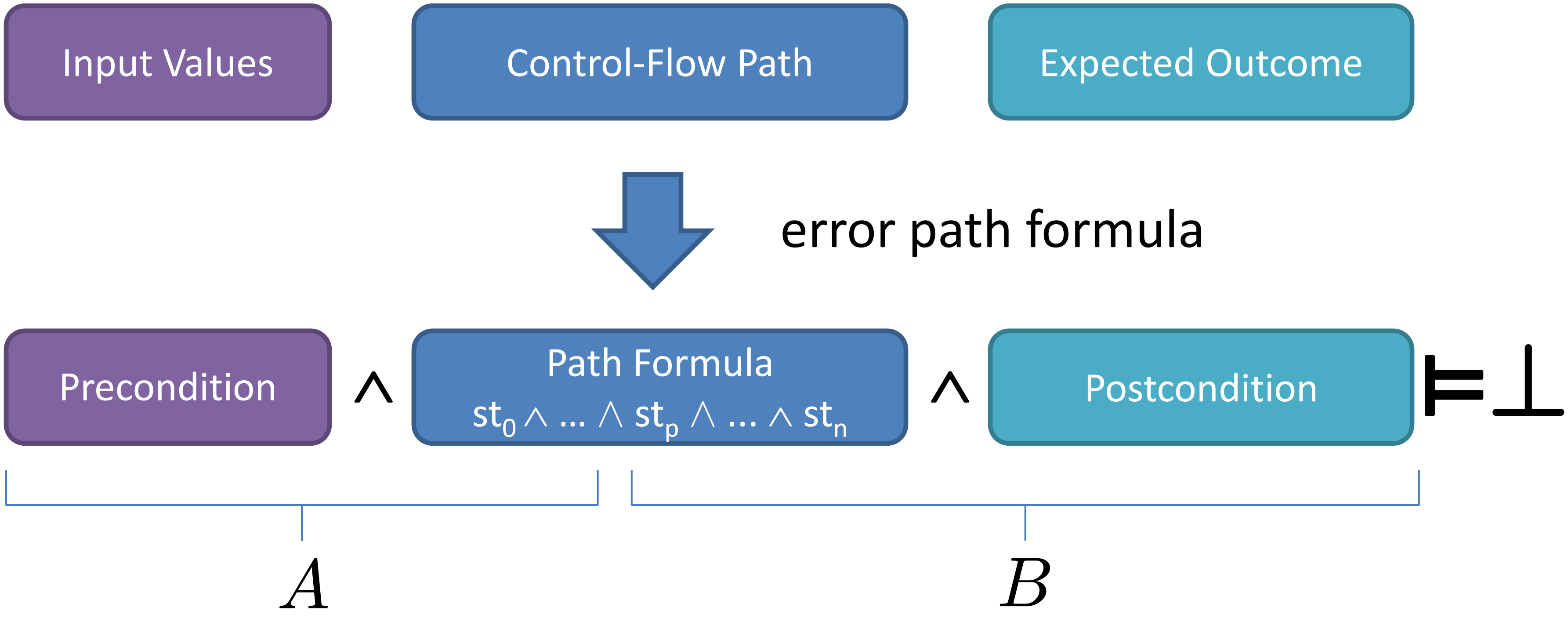
Control-Flow Path

Expected Outcome

error path formula

Precondition $\wedge$ Path Formula $st_0 \wedge \ldots \wedge st_p \wedge \ldots \wedge st_n$ $\wedge$ Postcondition $\models \bot$

Checking Error Invariants



Input Values

Control-Flow Path

Expected Outcome

error path formula

Precondition $\wedge$ Path Formula $st_0 \wedge \ldots \wedge st_p \wedge \ldots \wedge st_n$ $\wedge$ Postcondition $\models \bot$

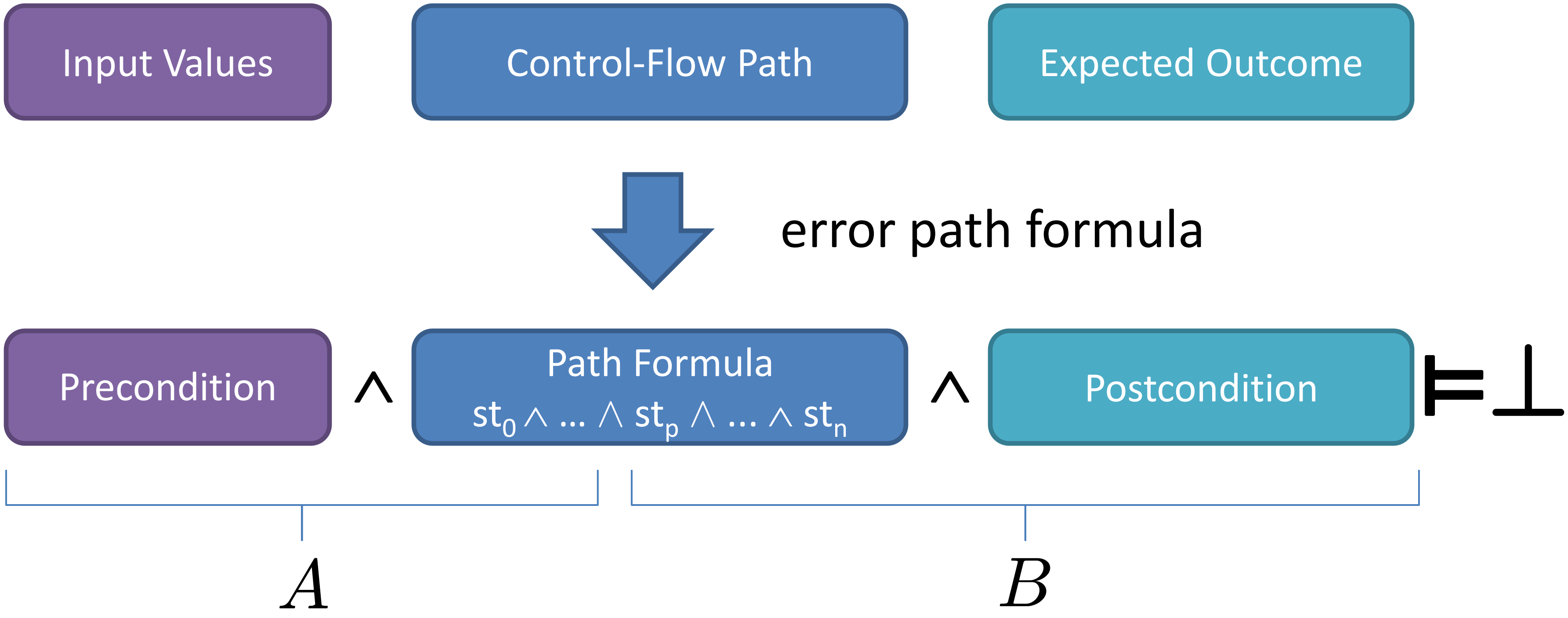
A

B

I is an error invariant for position p iff

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

Craig Interpolants -> Error Invariants



Input Values

Control-Flow Path

Expected Outcome

error path formula

Precondition ∧ Path Formula $st_0 \wedge \ldots \wedge st_p \wedge \ldots \wedge st_n$ ∧ Postcondition $\models \bot$

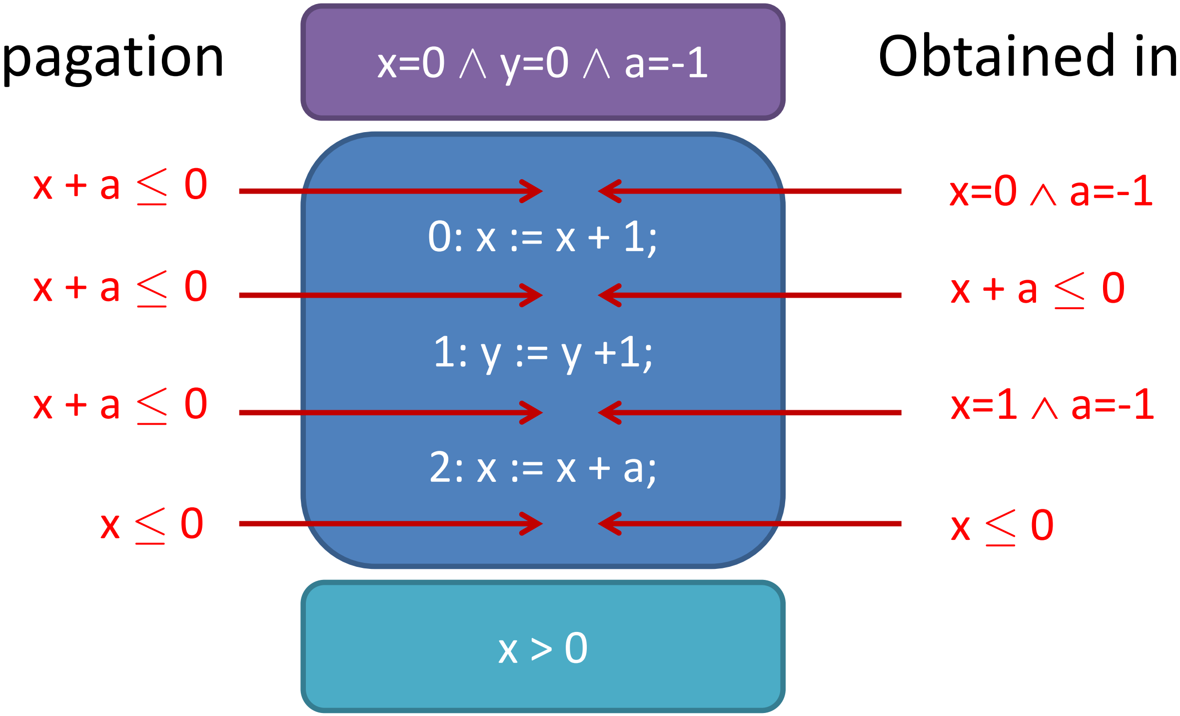
A

B

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

$\Rightarrow$ use interpolating SMT solver to compute candidate error invariants

Computing Abstractions of Error Traces



After propagation

Obtained interpolants

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

$x + a \leq 0$

$x=0 \wedge a=-1$

0: x := x + 1;

$x + a \leq 0$

$x + a \leq 0$

1: y := y +1;

$x + a \leq 0$

$x=1 \wedge a=-1$

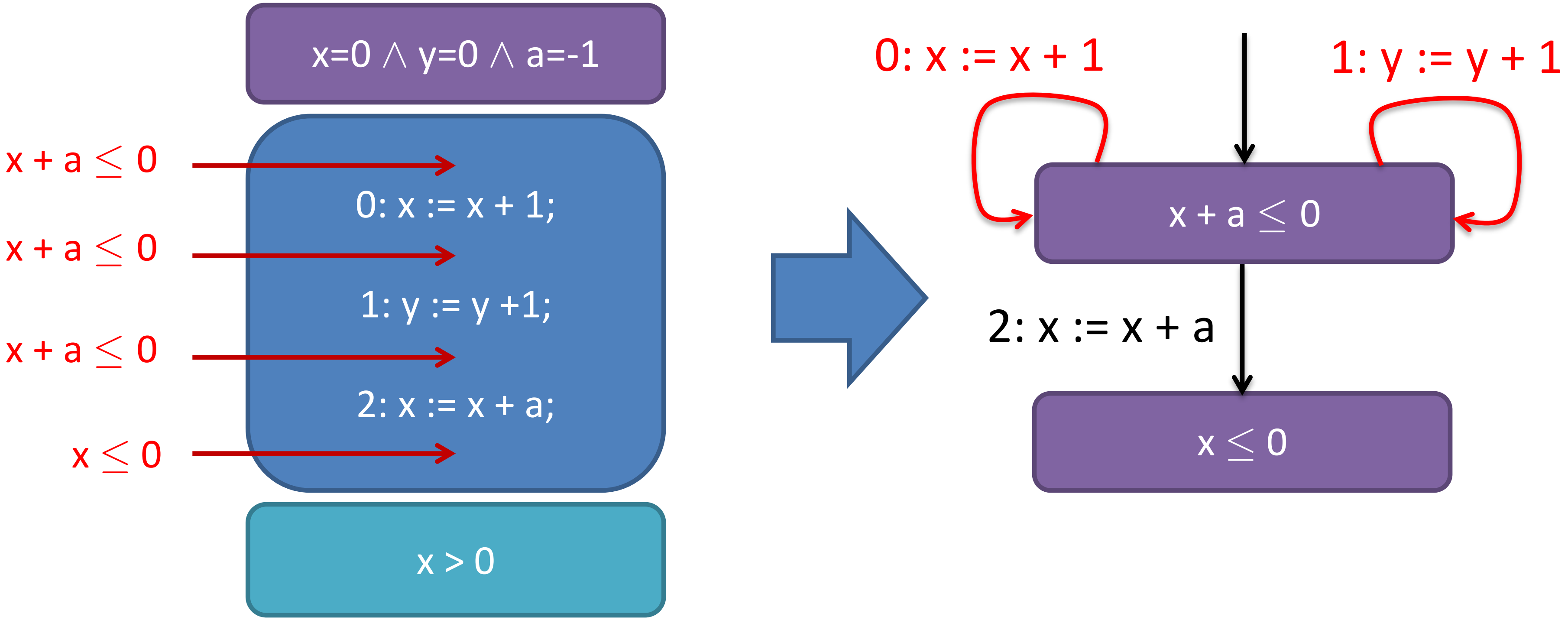
2: x := x + a;

$x \leq 0$

$x \leq 0$

$x > 0$

Computing Abstractions of Error Traces



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

$x + a \leq 0$

0: x := x + 1;

$x + a \leq 0$

1: y := y +1;

$x + a \leq 0$

2: x := x + a;

$x \leq 0$

$x > 0$

0: x := x + 1

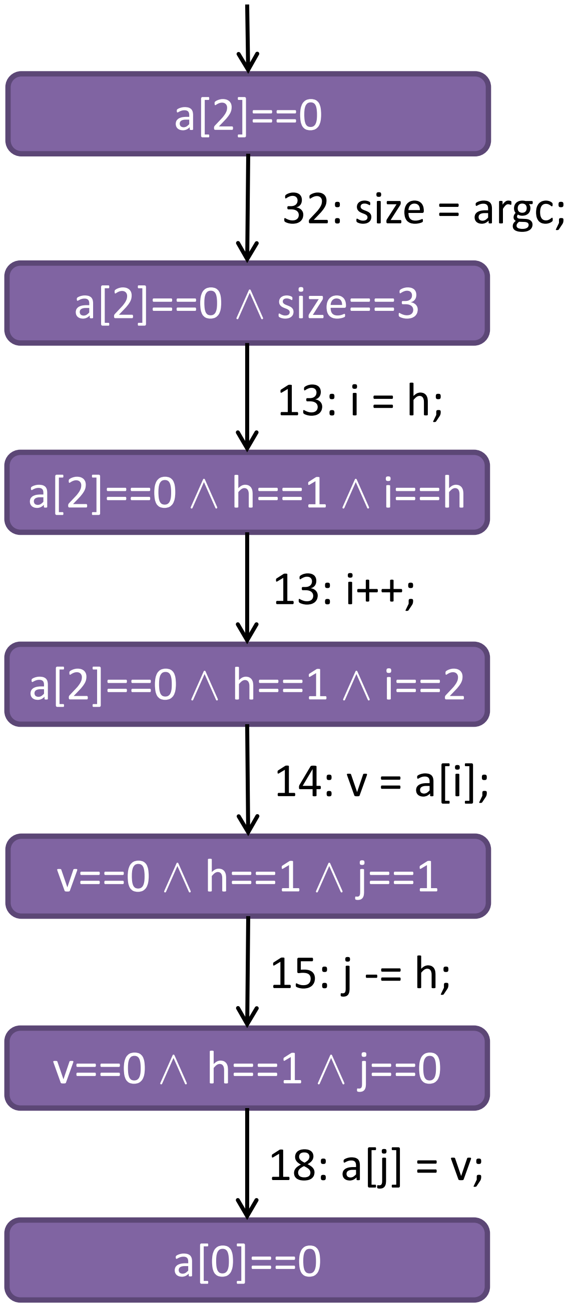
1: y := y + 1

$x + a \leq 0$

2: x := x + a

$x \leq 0$

```
 1  #include<stdio.h>
 2  #include<stdlib.h>
 3
 4  static void shell_sort(int a[], int size)
 5   {
 6     int i, j;
 7     int h = 1;
 8     do {
 9          h = h * 3 + 1;
10     } while (h <= size);
11     do {
12          h /= 3;
13          for (i = h; i < size; i++) {
14              int v = a[i];
15              for (j = i; j >= h && a[j - h] > v; j -= h)
16                 a[j] = a[j-h];
17              if (i != j)
18                 a[j] = v;
19           }
20     } while (h != 1);
21   }
23  int main(int argc, char *argv[])
24   {
25      int i = 0;
26      int *a = NULL;
27
28      a = (int *)malloc((argc-1) * sizeof(int));
29      for (i = 0; i < argc - 1; i++)
30         a[i] = atoi(argv[i + 1]);
31
32      shell_sort(a, argc);
33
34      for (i = 0; i < argc - 1; i++)
35         printf("%d", a[i]);
36      printf("\n");
37
38      free(a);
39      return 0;
40   }
```



a[2]==0

32: size = argc;

a[2]==0 ∧ size==3

13: i = h;

a[2]==0 ∧ h==1 ∧ i==h

13: i++;

a[2]==0 ∧ h==1 ∧ i==2

14: v = a[i];

v==0 ∧ h==1 ∧ j==1

15: j -= h;

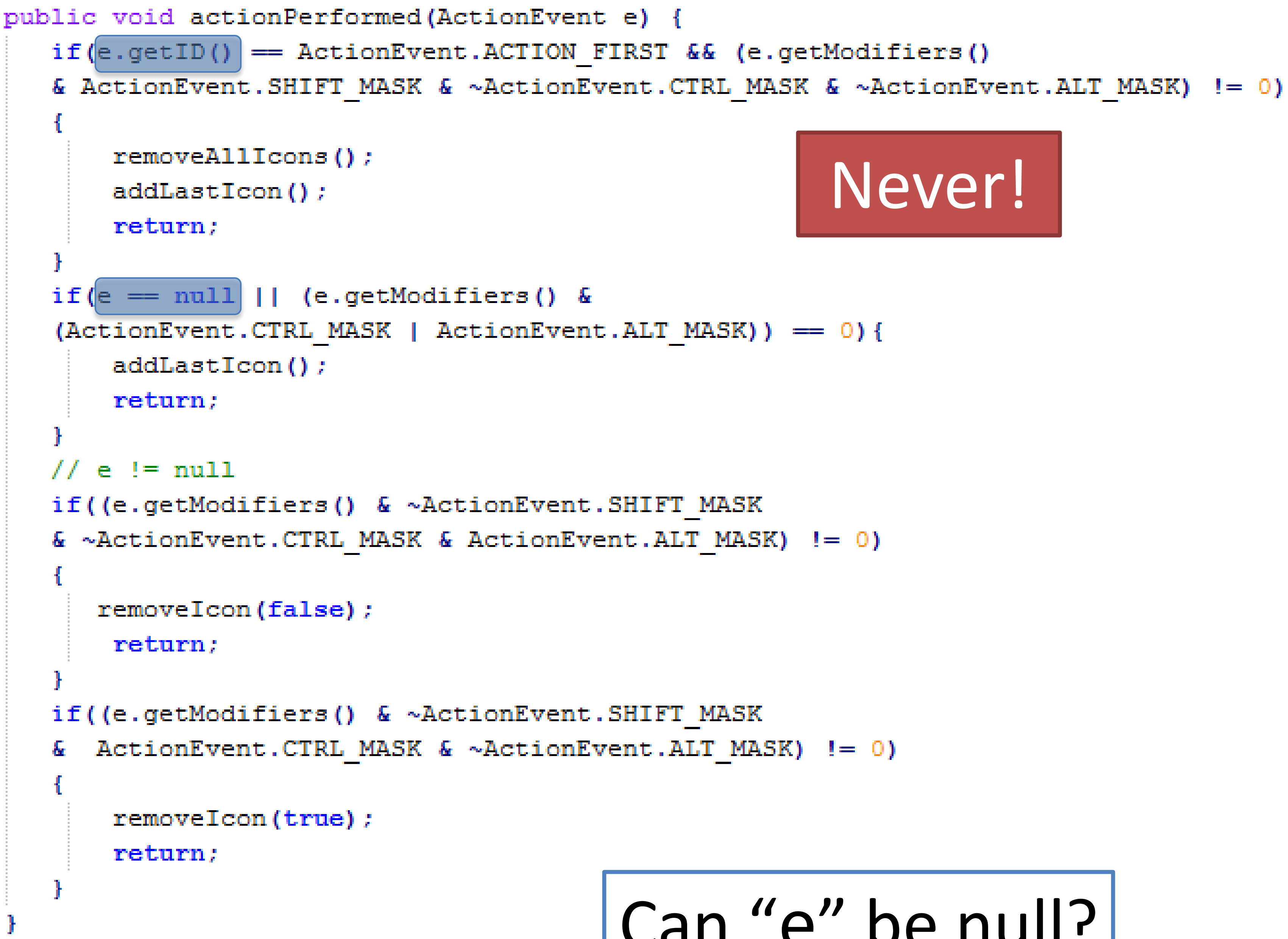
v==0 ∧ h==1 ∧ j==0

18: a[j] = v;

a[0]==0

Explaining Inconsistent Code

[Schäf, Schwartz-Narbonne, Wies, ESEC/FSE’13]



```
public void actionPerformed(ActionEvent e) {
    if(e.getID() == ActionEvent.ACTION_FIRST && (e.getModifiers()
    & ActionEvent.SHIFT_MASK & ~ActionEvent.CTRL_MASK & ~ActionEvent.ALT_MASK) != 0)
    {
        removeAllIcons();
        addLastIcon();
        return;
    }
    if(e == null || (e.getModifiers() &
    (ActionEvent.CTRL_MASK | ActionEvent.ALT_MASK)) == 0){
        addLastIcon();
        return;
    }
    // e != null
    if((e.getModifiers() & ~ActionEvent.SHIFT_MASK
    & ~ActionEvent.CTRL_MASK & ActionEvent.ALT_MASK) != 0)
    {
       removeIcon(false);
        return;
    }
    if((e.getModifiers() & ~ActionEvent.SHIFT_MASK
    &  ActionEvent.CTRL_MASK & ~ActionEvent.ALT_MASK) != 0)
    {
        removeIcon(true);
        return;
    }
}
```

Never!

Can “e” be null?

Inconsistent Code Reflects Programmer Confusion

```
assert(x);
if (!x) {…}
```

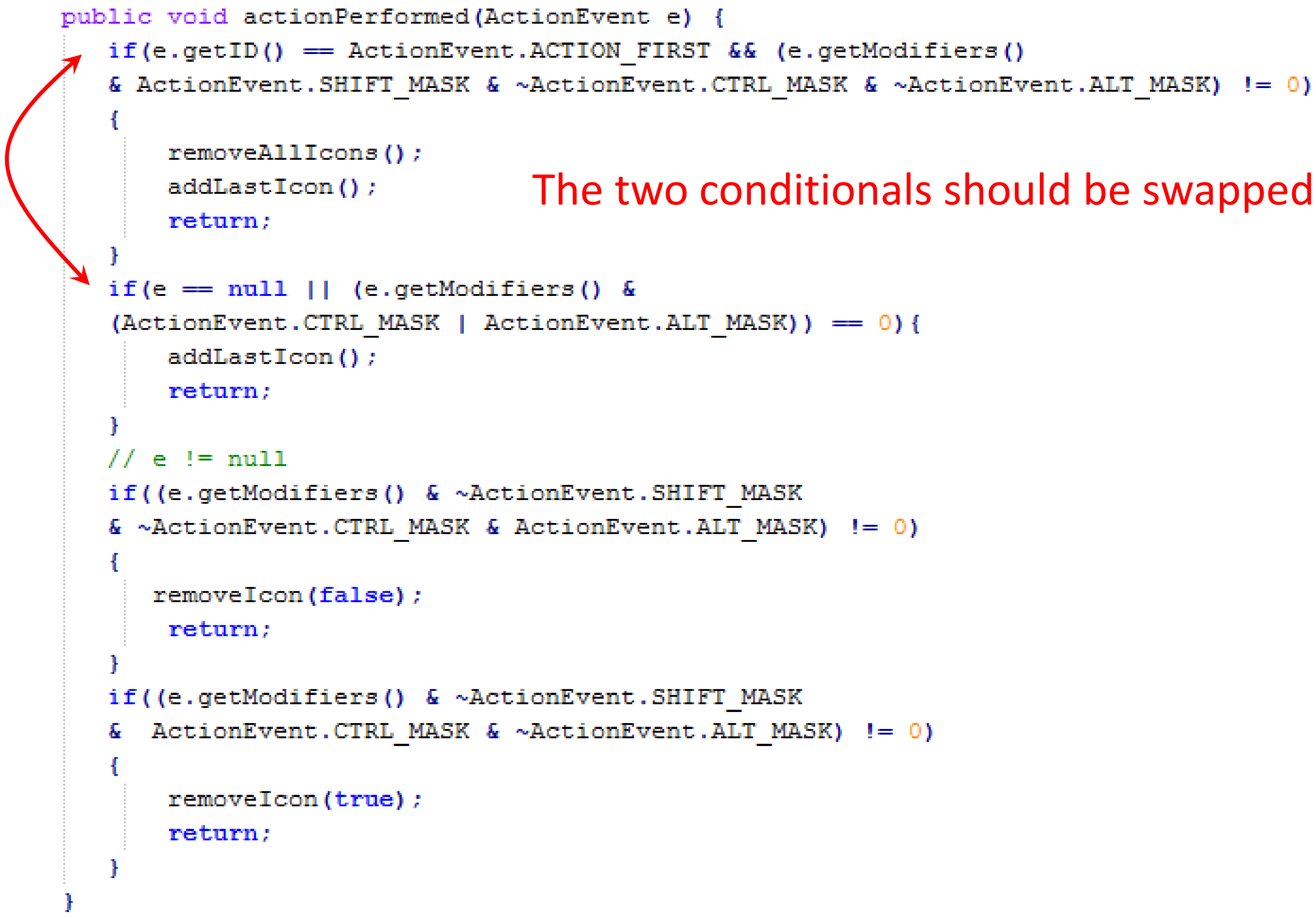
```
p.DoSomething();
if (p == null) {…}
```

```
p = null;
p.DoSomething();
```

```
int v = 4 * a;
if (isOdd(v)) {…}
```

Inconsistent code has no feasible executions

Highly correlated with real world bugs



```
public void actionPerformed(ActionEvent e) {
    if(e.getID() == ActionEvent.ACTION_FIRST && (e.getModifiers()
    & ActionEvent.SHIFT_MASK & ~ActionEvent.CTRL_MASK & ~ActionEvent.ALT_MASK) != 0)
    {
        removeAllIcons();
        addLastIcon();
        return;
    }
    if(e == null || (e.getModifiers() &
    (ActionEvent.CTRL_MASK | ActionEvent.ALT_MASK)) == 0){
        addLastIcon();
        return;
    }
    // e != null
    if((e.getModifiers() & ~ActionEvent.SHIFT_MASK
    & ~ActionEvent.CTRL_MASK & ActionEvent.ALT_MASK) != 0)
    {
       removeIcon(false);
        return;
    }
    if((e.getModifiers() & ~ActionEvent.SHIFT_MASK
    &  ActionEvent.CTRL_MASK & ~ActionEvent.ALT_MASK) != 0)
    {
        removeIcon(true);
        return;
    }
}
```

The two conditionals should be swapped.

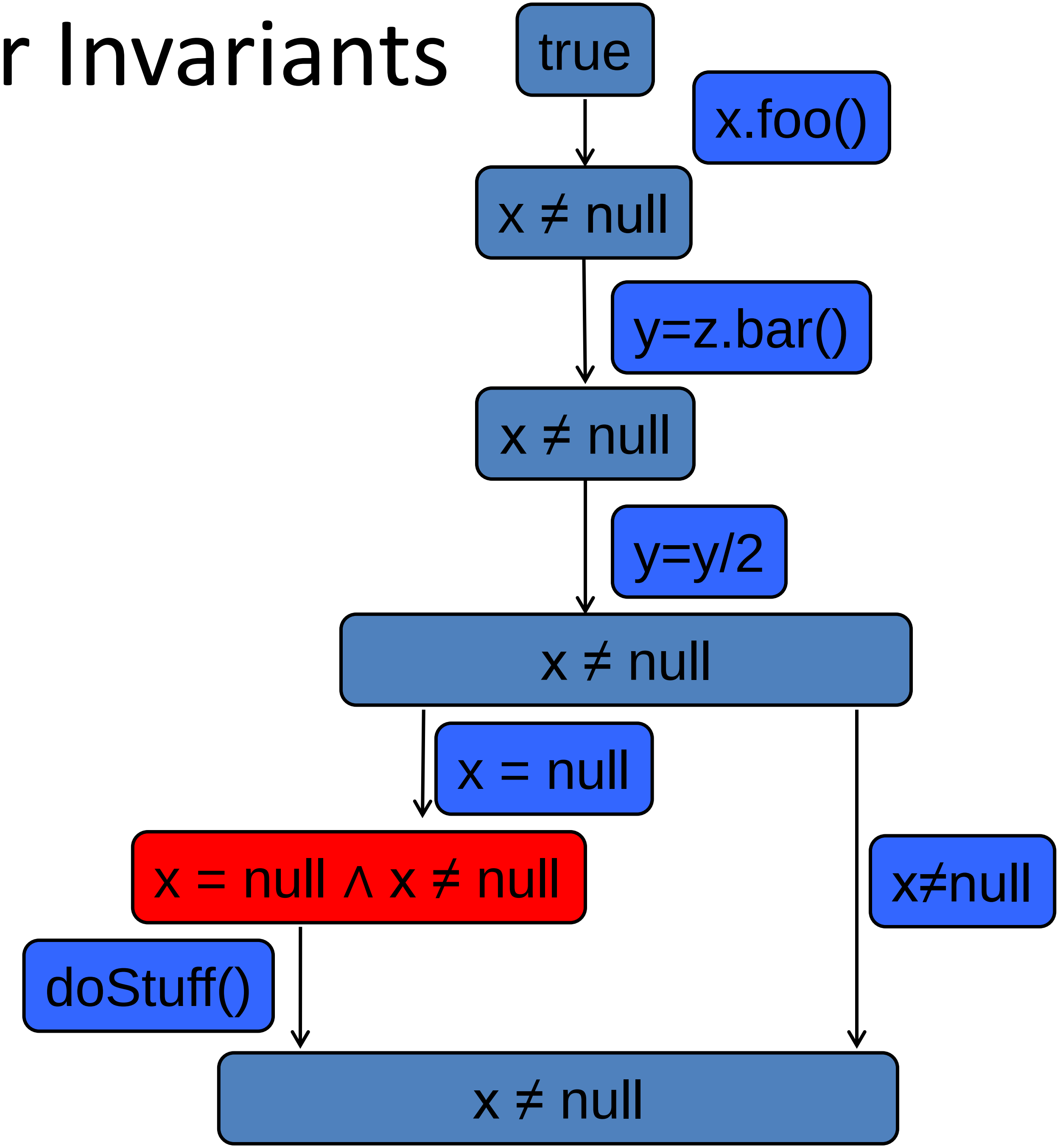
Explaining Inconsistent Code

```
1: x.foo();
2: y = z.bar();
3: y = (y/2);
4: if(x == null){
5:   doStuff();
6: }
7: y = x.baz();
```

Static analysis tool (e.g. Joogie) reports inconsistency on line 4

Using Error Invariants

```
1: x.foo();
2: y = z.bar();
3: y = (y/2);
4: if(x == null){
5:   doStuff();
6: }
7: y = x.baz();
```



true

x.foo()

x ≠ null

y=z.bar()

x ≠ null

y=y/2

x ≠ null

x = null

x = null ∧ x ≠ null

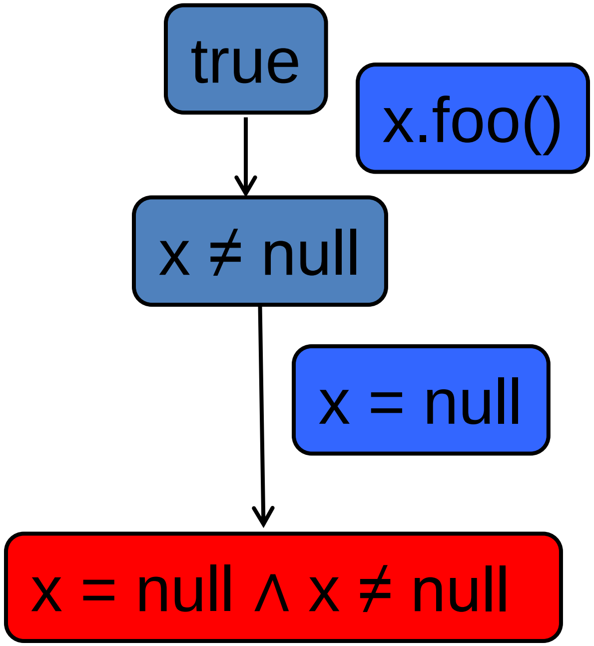
x≠null

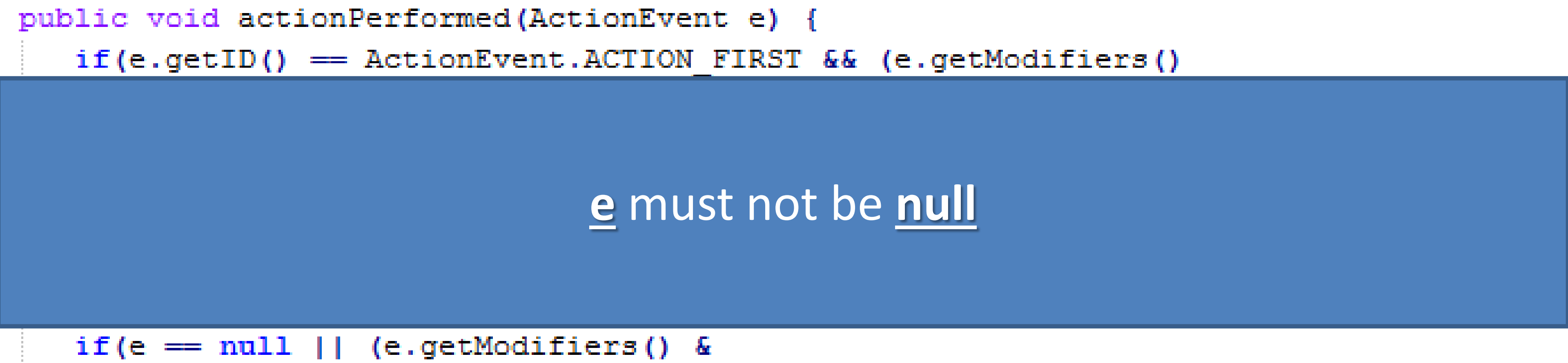
doStuff()

x ≠ null

Computed Abstract Slice

```
1: x.foo();
2: y = z.bar();
3: y = (y/2);
4: if(x == null){
5:   doStuff();
6: }
7: y = x.baz();
```





```
public void actionPerformed(ActionEvent e) {
    if(e.getID() == ActionEvent.ACTION_FIRST && (e.getModifiers()
```

e must not be **null**

```
    if(e == null || (e.getModifiers() &
```

Human study: with abstract slices, programmers need **60% less time** to understand inconsistencies.

Conclusions

Fault abstraction

- new static approach to fault localization
  - no need to compare failing and successful executions
  - enables computation of concise error explanations
  - key concept: error invariants
- relies on verification/static analysis technology
  - abstraction (as in software model checking)
  - interpolation