Automated Debugging with Error Invariants

Chanseok Oh (NYU), Martin Schäf (SRI),
Daniel Schwartz-Narbonne (NYU),
Thomas Wies (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.

```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;
         }
   } 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

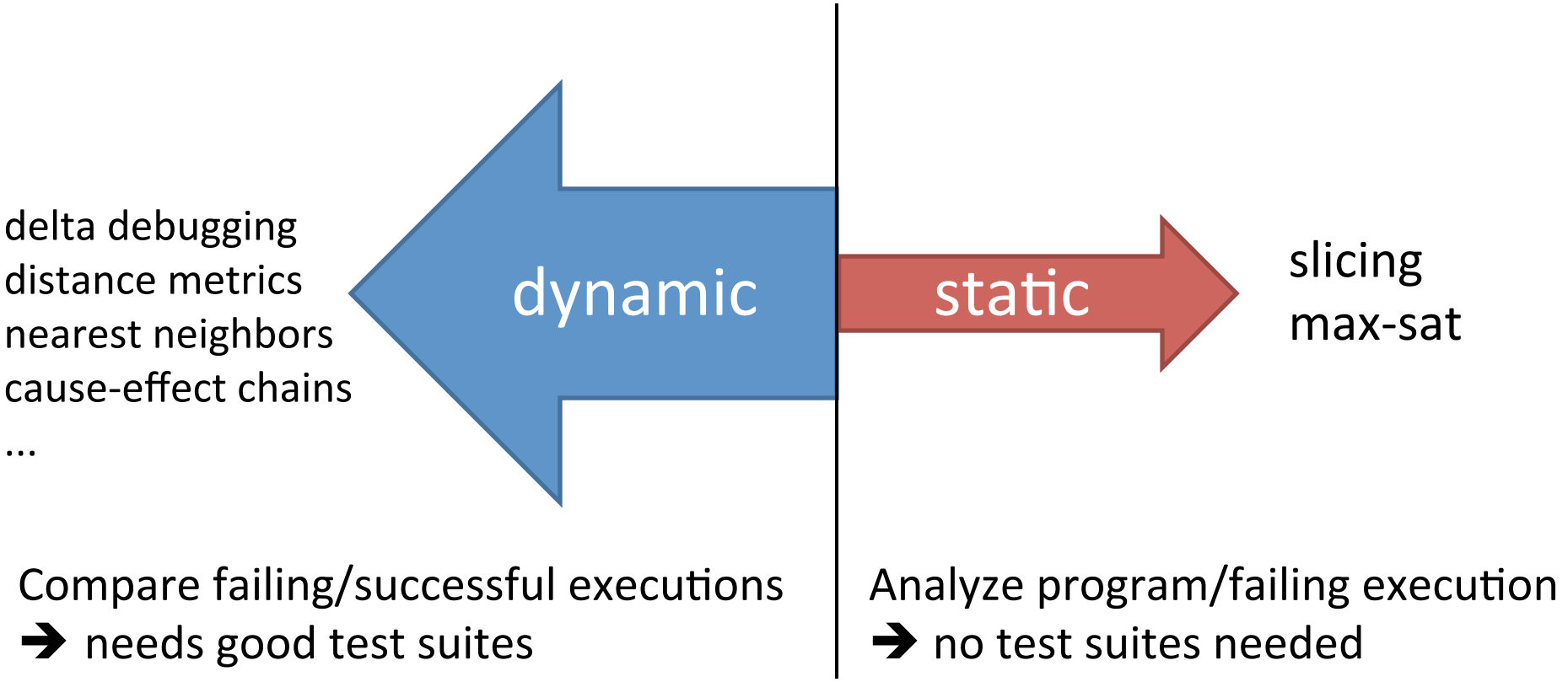
- takes a sequence of integers as input
- returns the sorted sequence.

```
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;
```

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

```
27
28       a = (int *)malloc((argc-1) * sizeof(int));
29       for (i = 0; i < argc - 1; i++)
30          a[i] = atoi(argv[i + 1]);
32: shell_sort(a, argc);
34       for (i = 0; i < argc - 1; i++)
35          printf("%d", a[i]);
36       printf("\n");
37
38       free(a);
39       return 0;
40    }
```

Landscape of Fault Localization Techniques



Faulty Shell Sort
[Cleve, Zeller ICSE'05]

Program

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

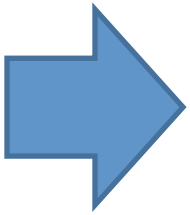
```
#include<stdio.h>
#include<stdlib.h>

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;
```

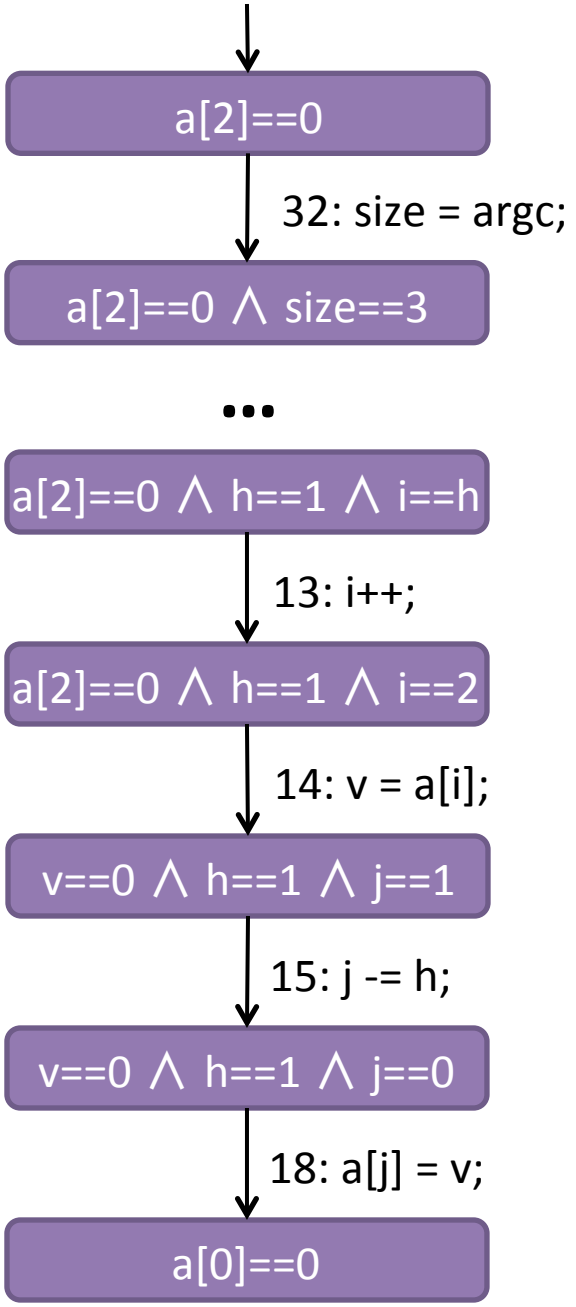
State-of-the-art **static fault localization** tool identifies **18 statements** as potential locations of the root cause.

```
         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

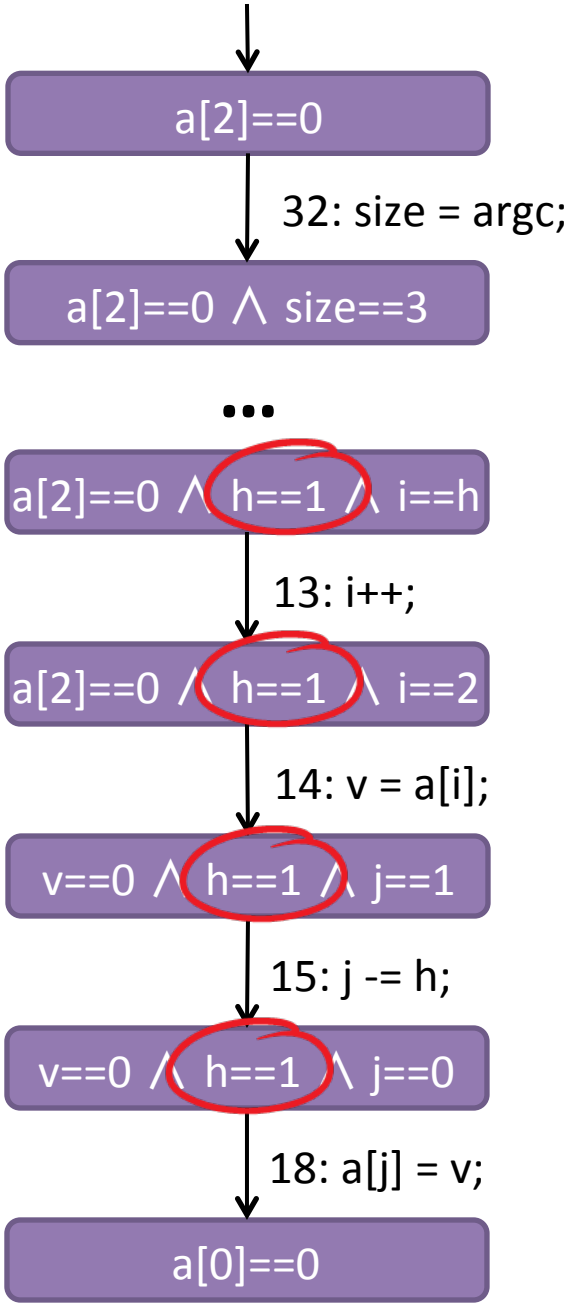
```c
#include<stdio.h>
#include<stdlib.h>

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;
         }
   } 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;
 }
```





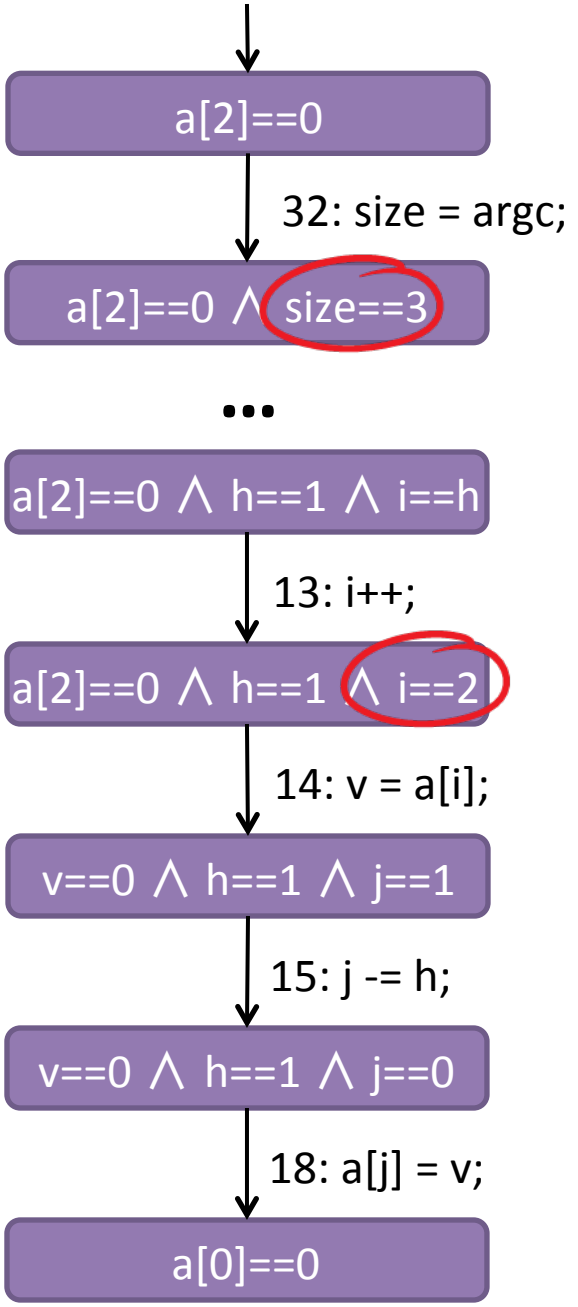
```
11: do {
12:     h /= 3;
13:     for (i = h; i < size; i++) {
14:       int v = a[i];
15:       for (j = i; j >= h && a[j-h] > v;
                j -= h)
16:          a[j] = a[j-h];
17:       if (i != j)
18:          a[j] = v;
19:     }
20: } while (h != 1);
```

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



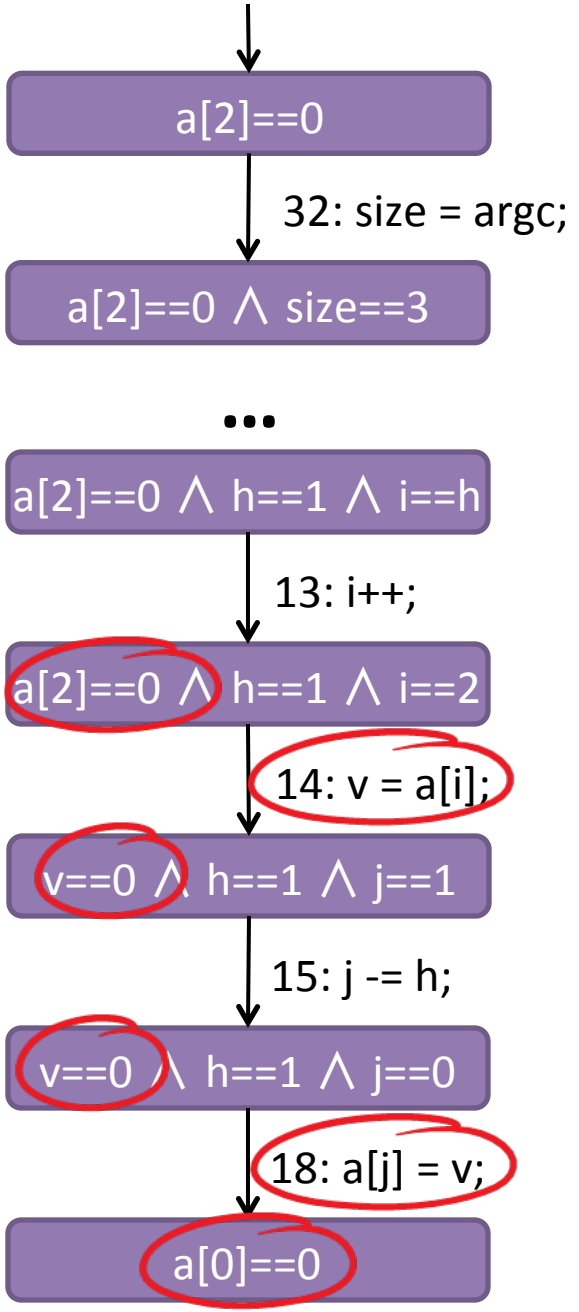
```
11: do {
12:     h /= 3;
13:     for (i = h; i < size; i++) {
14:       int v = a[i];
15:       for (j = i; j >= h && a[j-h] > v;
                j -= h)
16:          a[j] = a[j-h];
17:       if (i != j)
18:          a[j] = v;
19:     }
20: } while (h != 1);
```

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



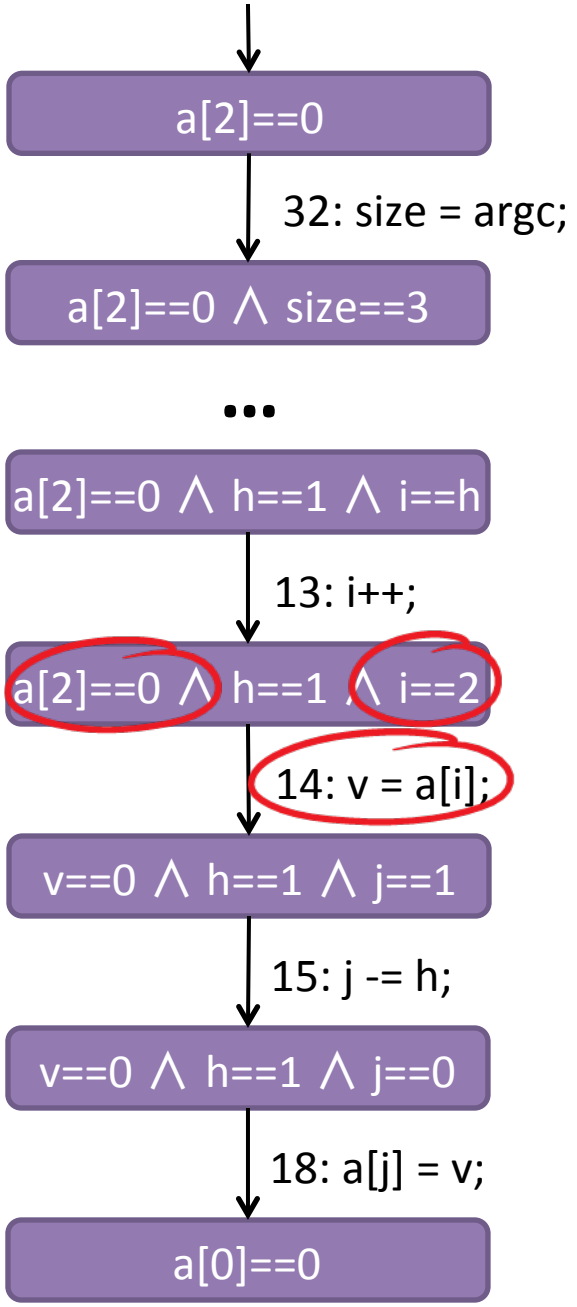
```
11: do {
12:     h /= 3;
13:     for (i = h; i < size; i++) {
14:       int v = a[i];
15:       for (j = i; j >= h && a[j-h] > v;
                j -= h)
16:          a[j] = a[j-h];
17:       if (i != j)
18:          a[j] = v;
19:     }
20: } while (h != 1);
```

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



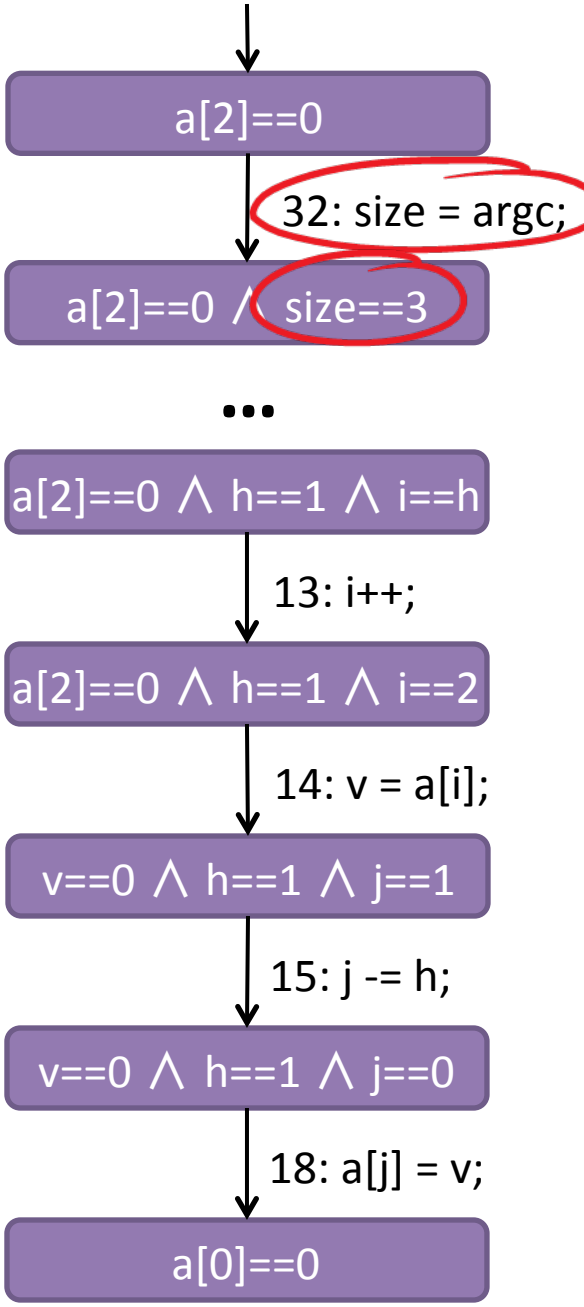
```
11: do {
12:     h /= 3;
13:     for (i = h; i < size; i++) {
14:       int v = a[i];
15:       for (j = i; j >= h && a[j-h] > v;
                j -= h)
16:           a[j] = a[j-h];
17:       if (i != j)
18:           a[j] = v;
19:     }
20: } while (h != 1);
```

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



```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  static void shell_sort(int a[], int size)
5  {
6      int h, i, j;
7      h = 1;
8      do {
9          h = 3 * h + 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 }
```

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



Fault Abstraction

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

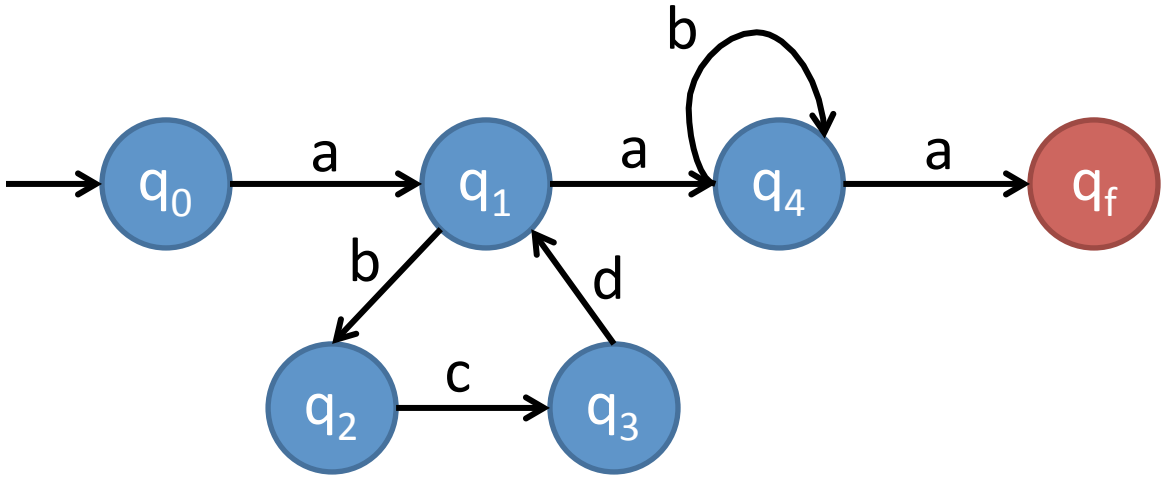
```
 1 size = argc;
 2 h = 1;
 3 h = h*3+1;
 4 assume !(h <= size);
 5 h /= 3;
 6 i = h;
 7 assume (i < size);
 8 v = a[i];
 9 j = i;
10 assume !(j >= h && a[j-h] > v);
11 i++;
12 assume (i < size);
13 v = a[i];
```

```
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



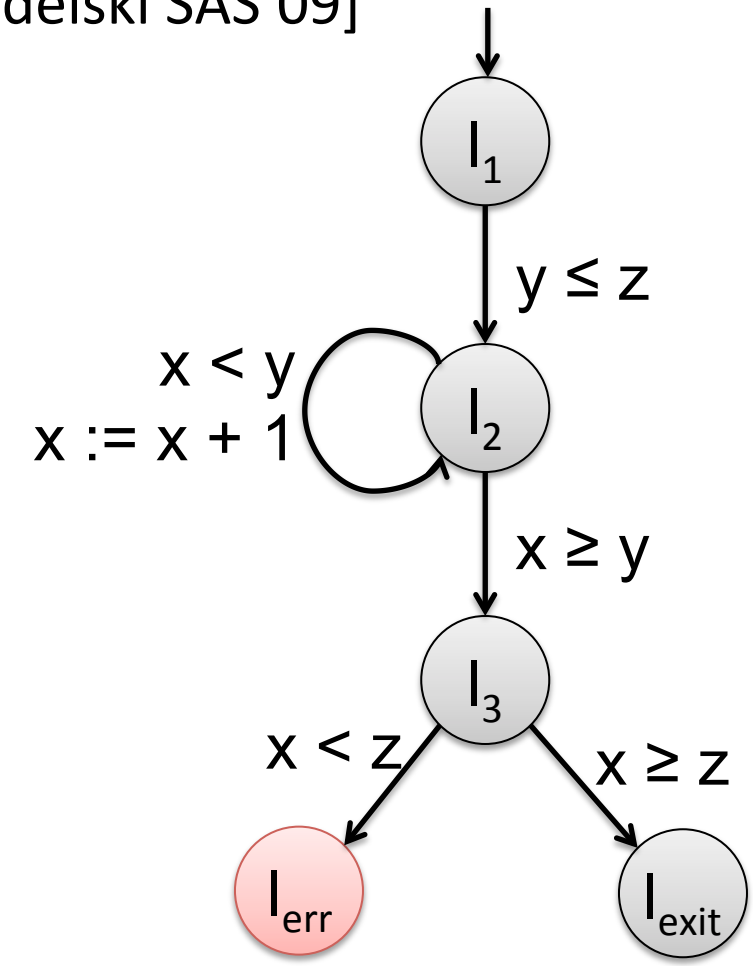
- A word that is accepted by the automaton:

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;
```

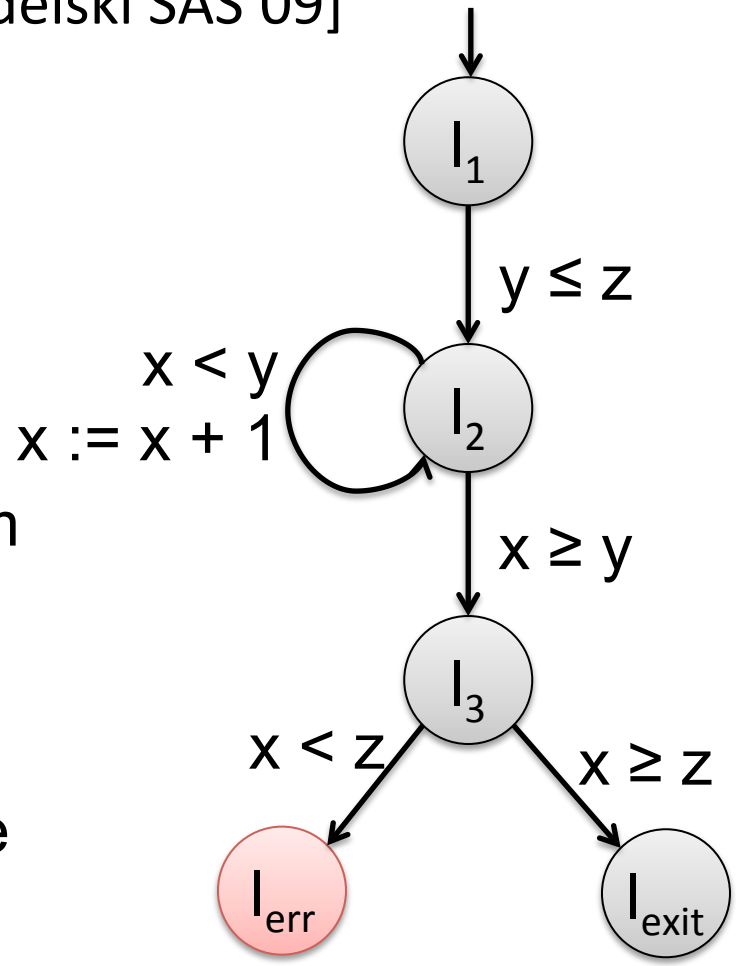


Nodes are program states
Edges are program transitions

Programs as Automata

[Heizmann, Hoenicke, Podelski SAS'09]

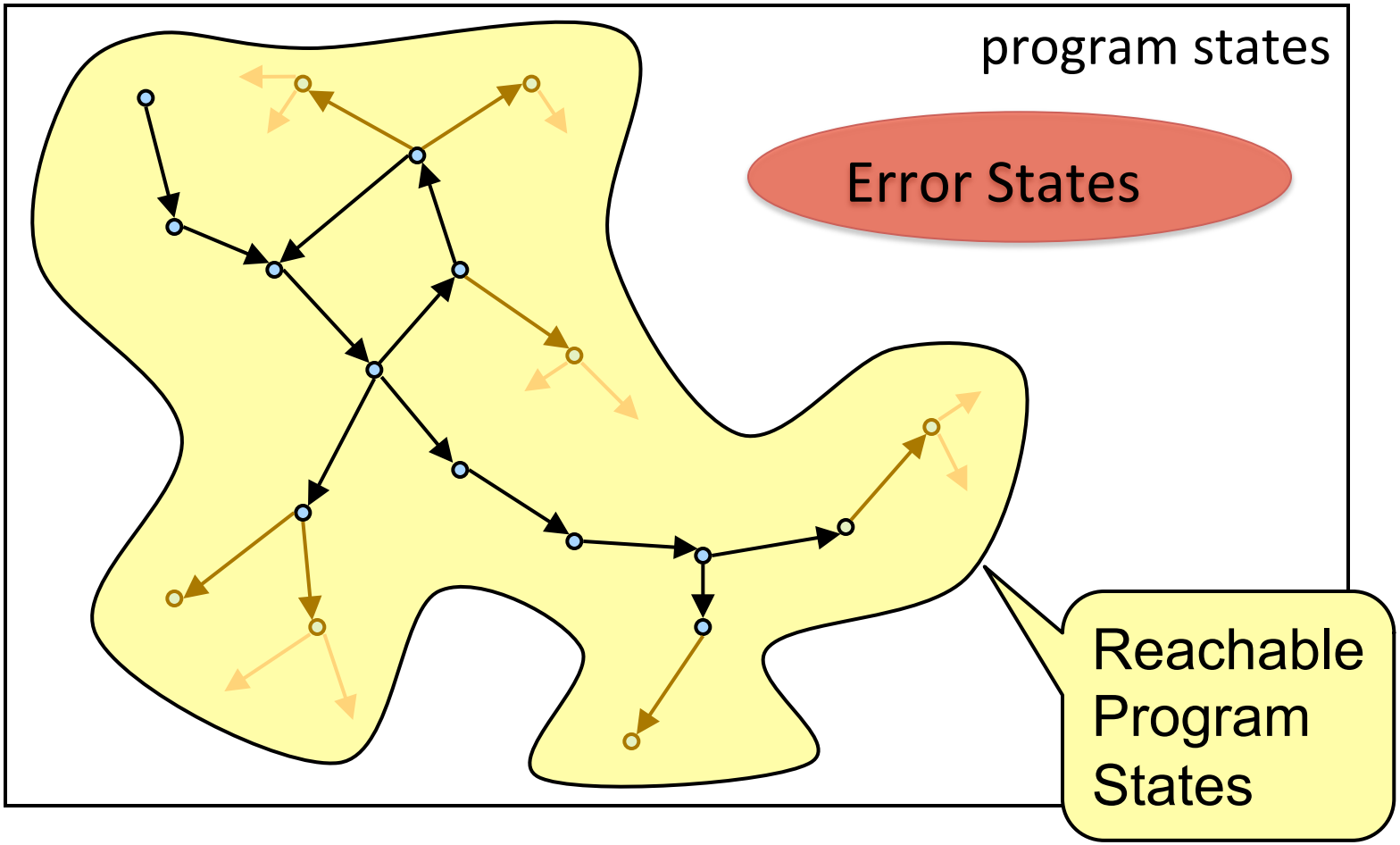
```
1: assume y ≤ z;
2: while x < y do x := x + 1;
3: assert x ≤ z;
```



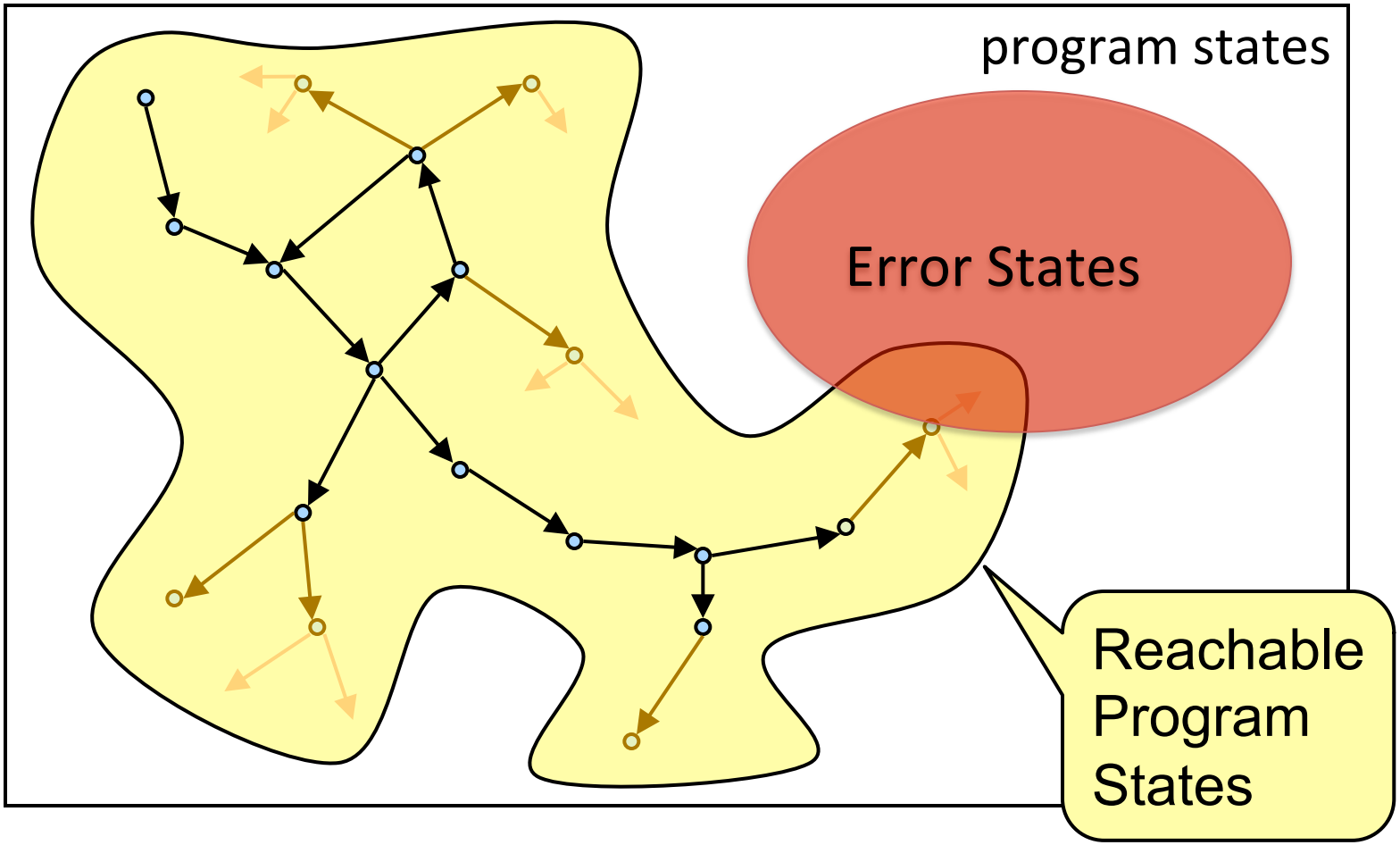
- error path = finite word of program statements
- program = automaton that accepts error paths
- fault local. = eliminate loops in the error path

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

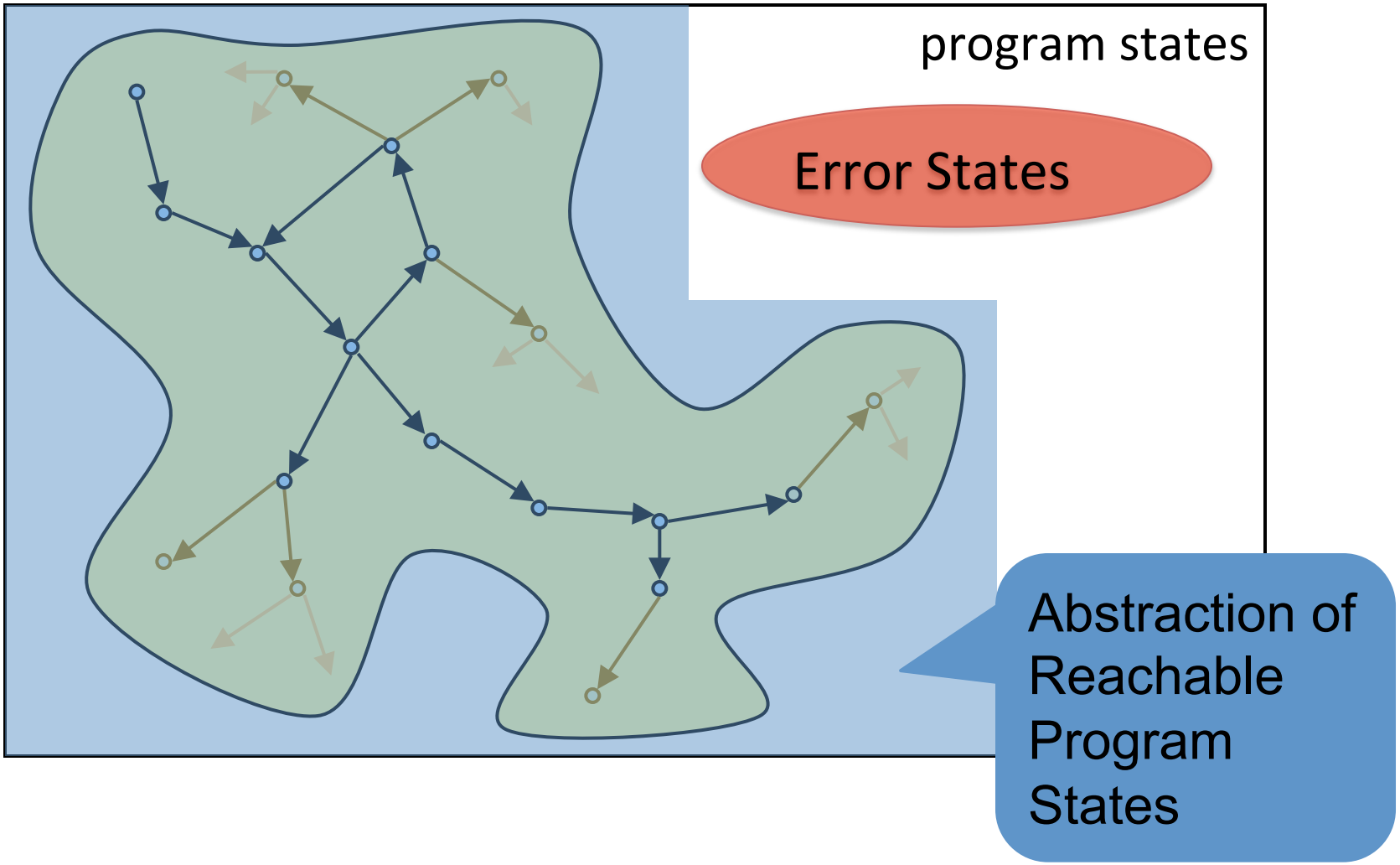
Executions are runs of the automaton



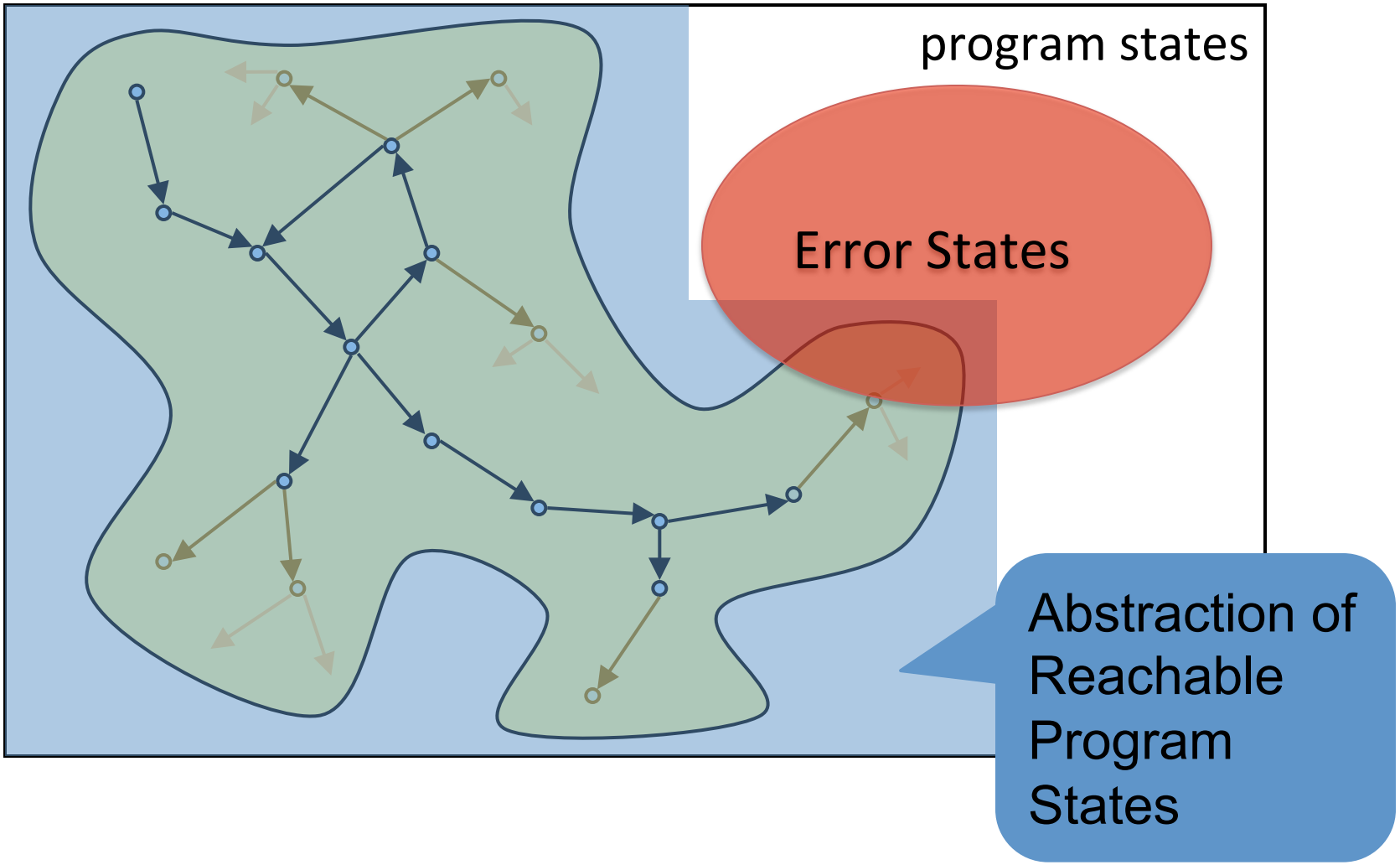
Executions are runs of the automaton



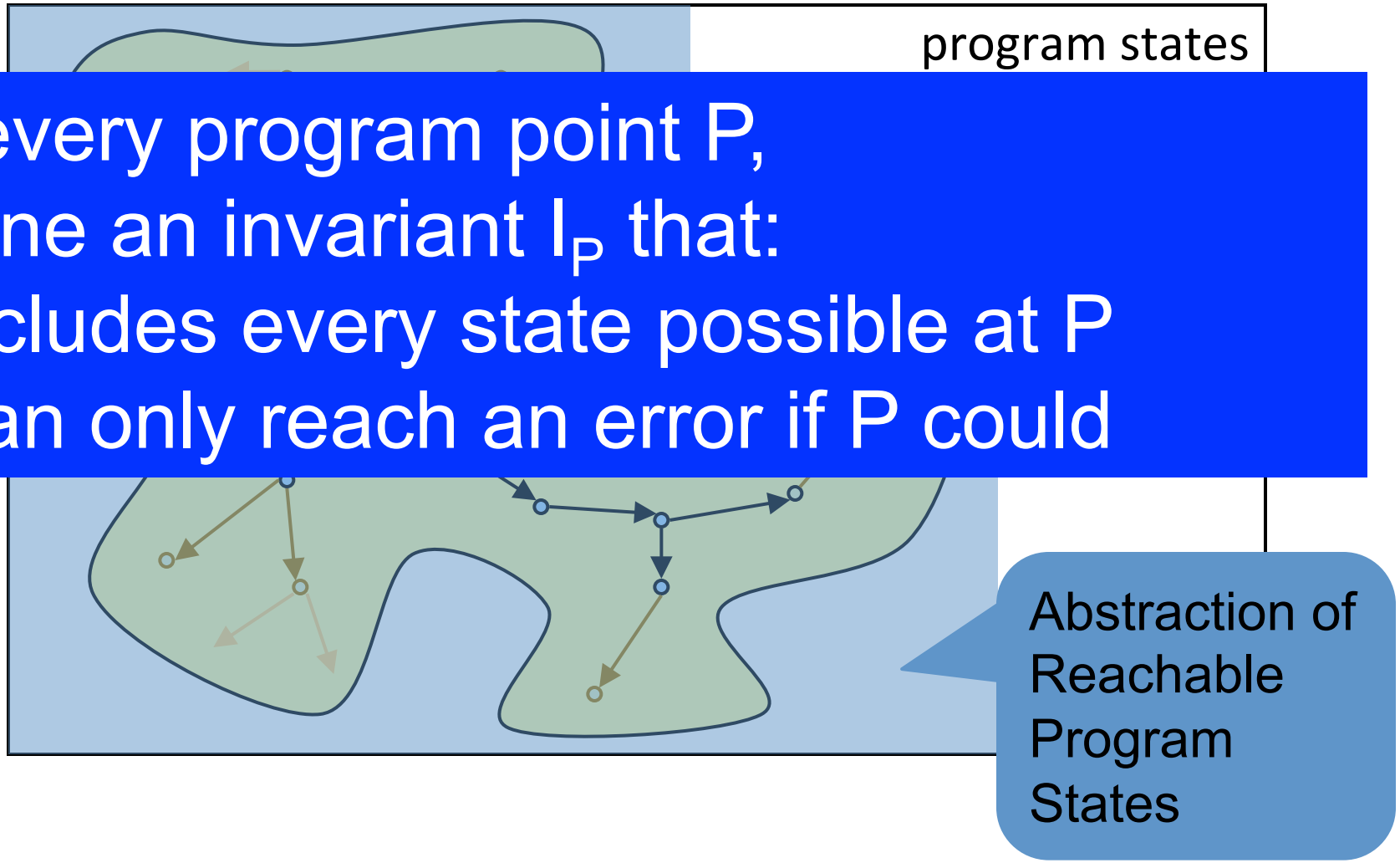
Abstractions are OK as long as they do not change whether errors occur



Abstractions are OK as long as they do not change whether errors occur



Our idea: when possible, replace program states with invariants



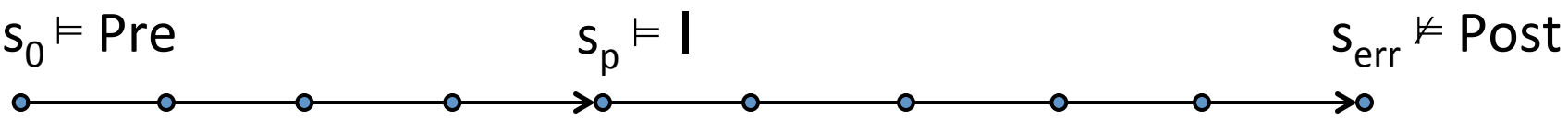
At every program point P,
define an invariant I_P that:

- Includes every state possible at P
- Can only reach an error if P could

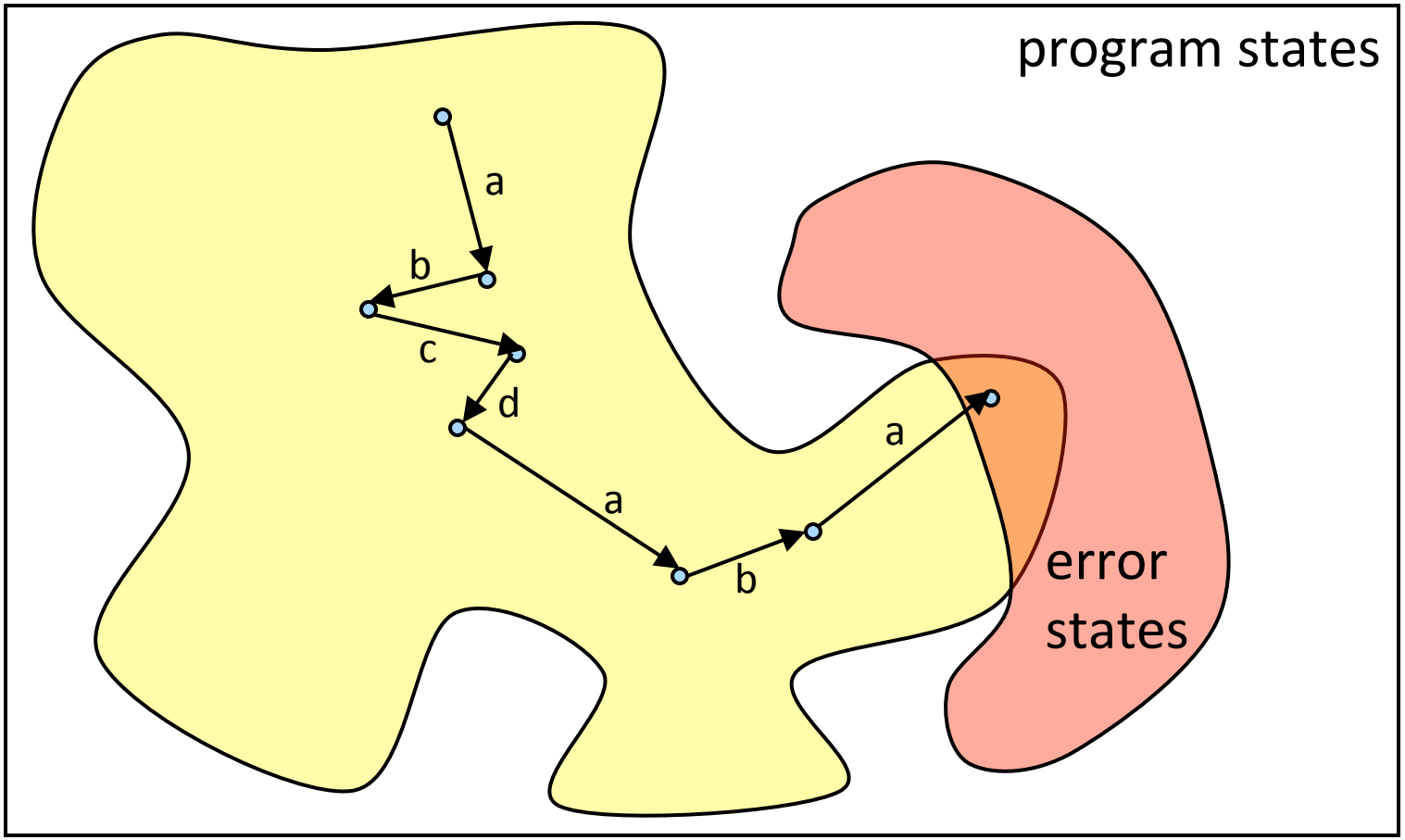
Error Invariants

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

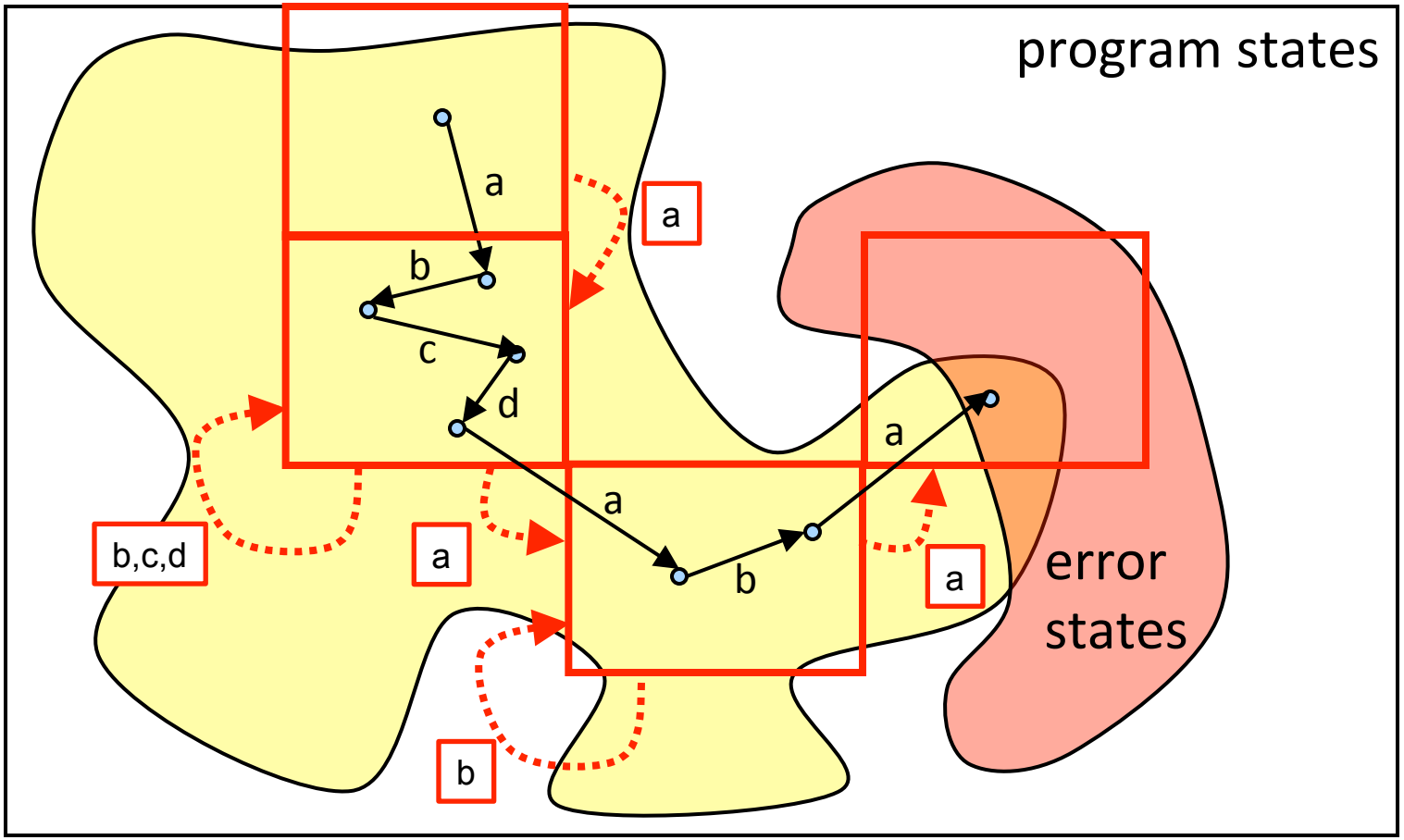
1. all states reachable by executing the prefix of τ up to position p satisfy I
2. all executions of the suffix of τ that start from p in a state that satisfies I, still lead to the same error.



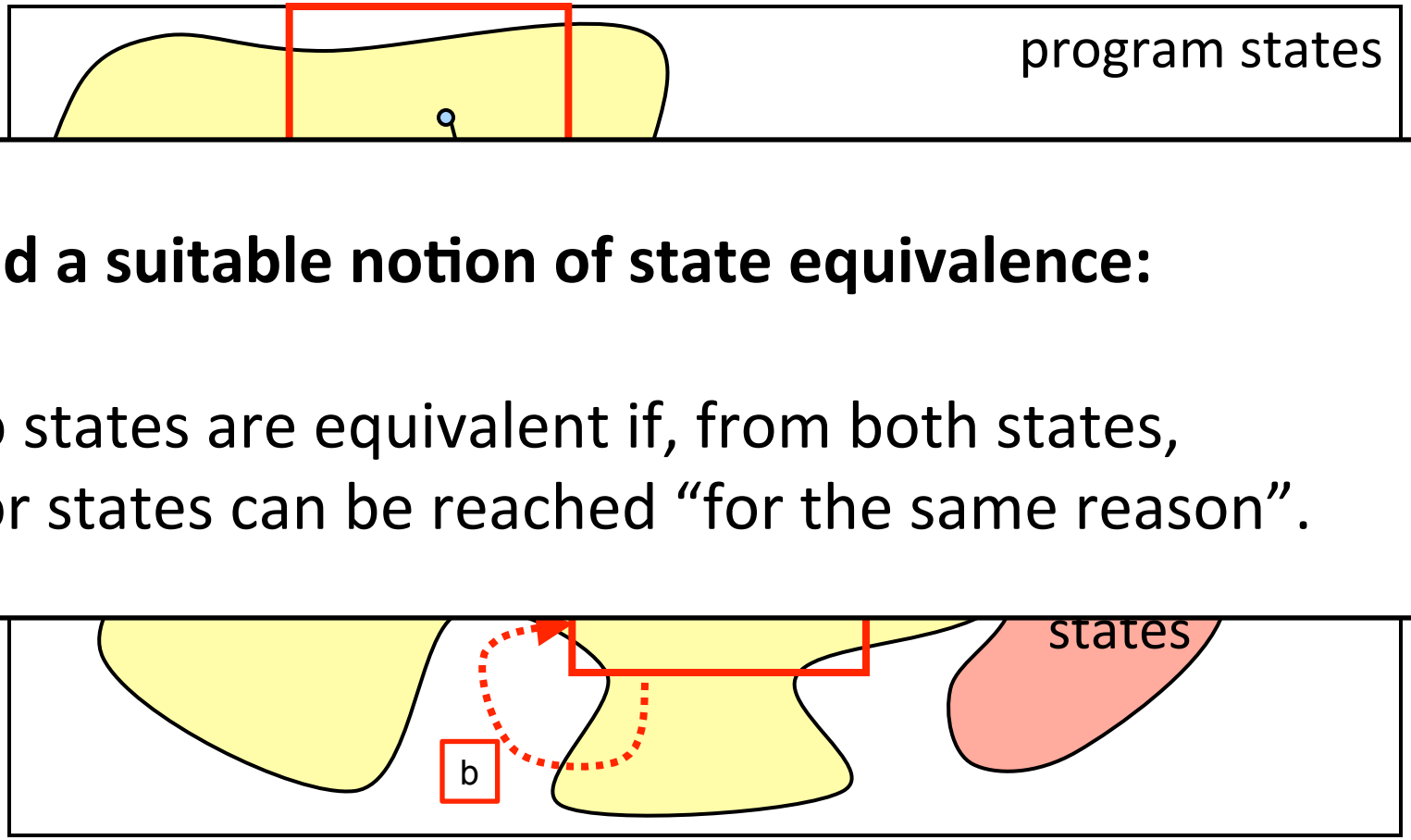
Fault Abstraction



Fault Abstraction



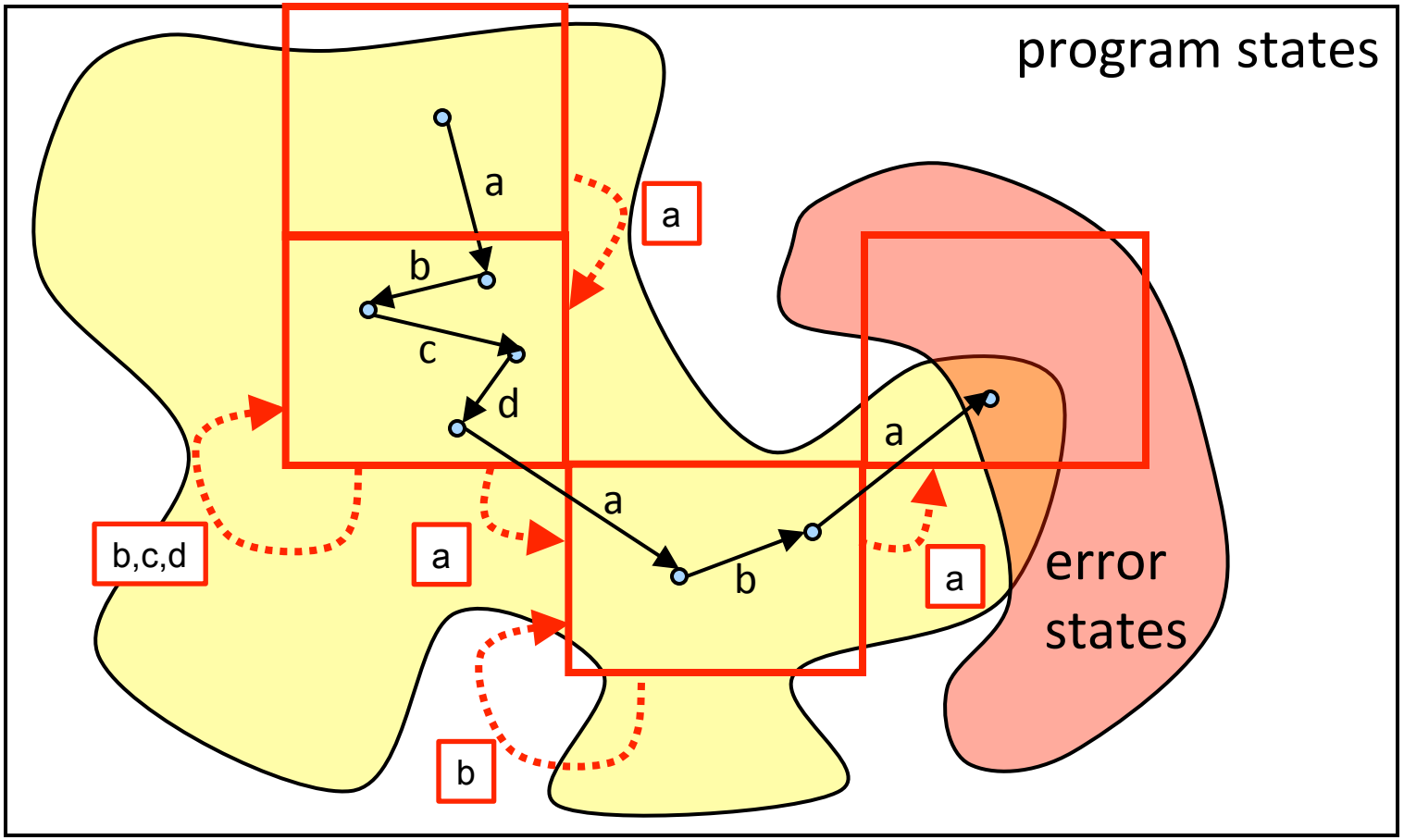
Fault Abstraction



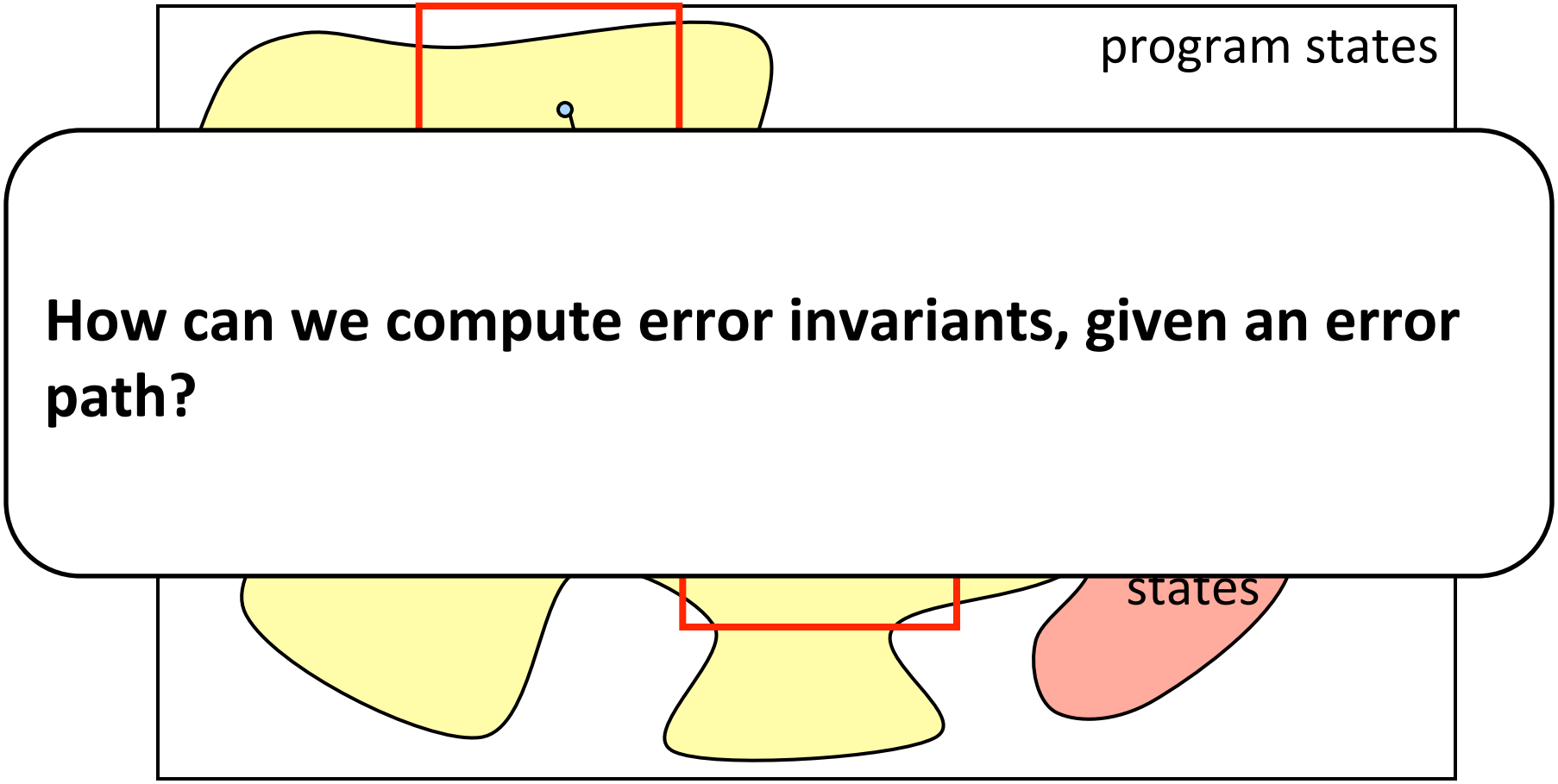
Need a suitable notion of state equivalence:

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

Restoring Invariants is sufficient



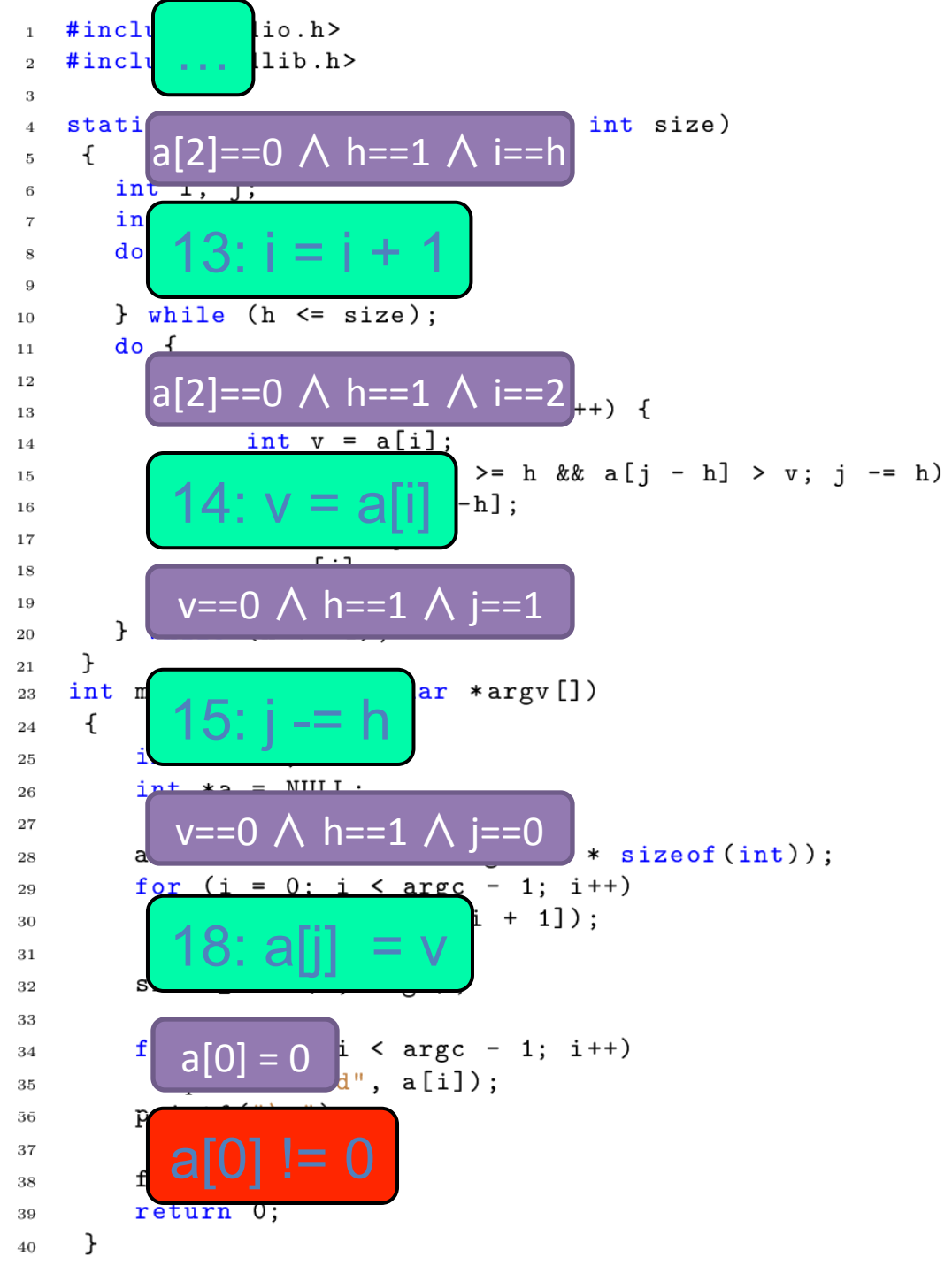
Restoring Invariants is sufficient



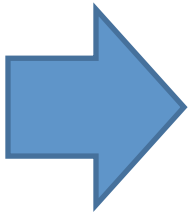
How can we compute error invariants, given an error path?

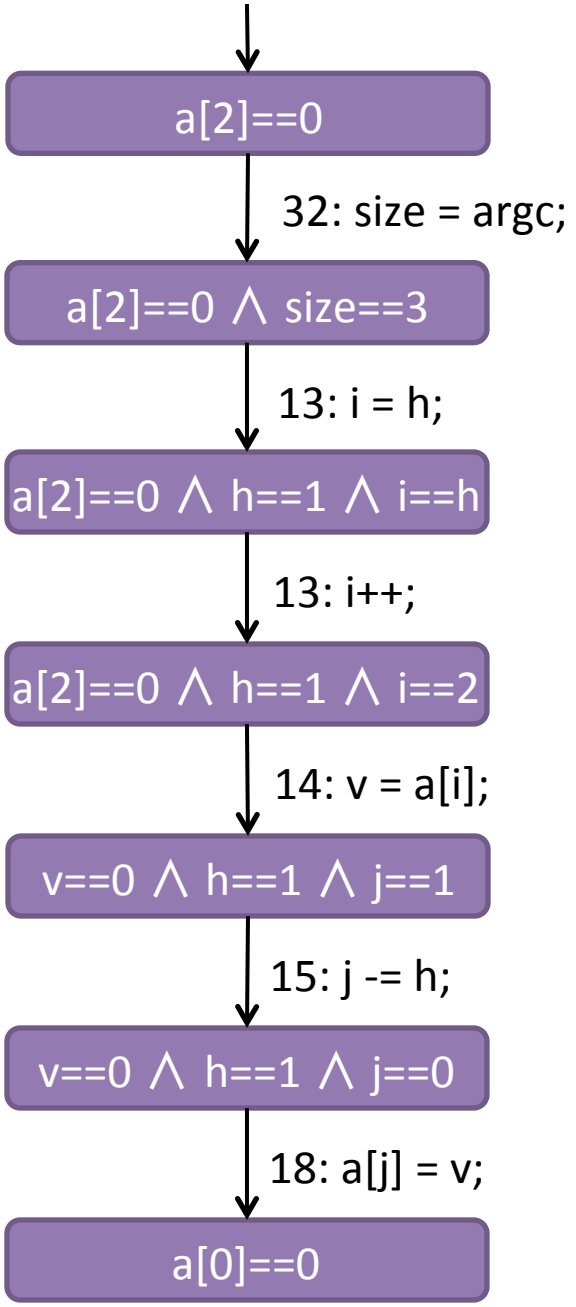
Step 1: Encode in SMT

- Each statement becomes an SMT assertion
- Add the violated assertion
- We now have an UNSAT formula
- Calculate an invariant following each statement
  - Technically a Craig interpolant
  - Note that they are not always minimal.
- This forms our automaton
  - states are the invariants
  - edges are the statements



```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;
         }
   } 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;
 }
```





Real world example: SED

```
1  struct vector *the_program = 0;
2  ...
3  struct re_pattern_buffer *last_regex;
4  ...
5  int main (int argc, char **argv)
6  {
7    ...
8    compile_file (...);
9    ...
10   read_file (...);
11 }
12 ...
13 void compile_file (...)
```

```
45 void read_file (...)
46 {
47   ...
48   execute_program(the_program);
49   ...
50 }
51 ...
52 void execute_program (struct vector *vec)
53 {
54   struct sed_cmd *cur_cmd;
55   ...
56   cur_cmd = vec->v;
57   ...
```

Full Program: 11990 LOC
Error Trace: 985 LOC

```
28   ...
29   cur_cmd = vector->v + vector->v_length;

30   ...
31   compile_address (&(cur_cmd->a1));
32   ...
33   return vector;
34 }
35 ...
36 int compile_address(struct addr *addr)
37 {
38   ...
39   #ifndef FAULTY_F_AG_19
40    compile_regex ();
41   #endif
42   addr->addr_regex = last_regex;
43   ...
44 }
```

```
73   unsigned num_regs = rxb->re_nsub + 1;
74   ...
75 }
```

+ 11915 more LOC

Reduced SED trace

```
//I00: true
3: last_regex = 0;
//I01: last_regex == 0 && addr = 0x123
42: addr->addr_regex = last_regex;
//I02: mem[0x123]==0 && addr=0x123
65: rxb = addr->addr_regex;
//I03: rxb==0;
num_regs = rxb->re_nsub + 1
//I04 : false
```

Results

	sed	gzip
LOC Full Program	11990	7328
LOC Error Trace	985	232
LOC address-sensitive slice	11	51
LOC address-insensitive slice	4	49

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

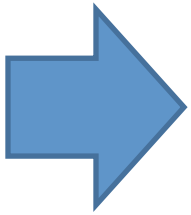
e must not be **null**

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

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

In conclusion:

```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;
         }
   } 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;
 }
```



...

a[2]==0 ∧ h==1 ∧ i==h

13: i = i + 1

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

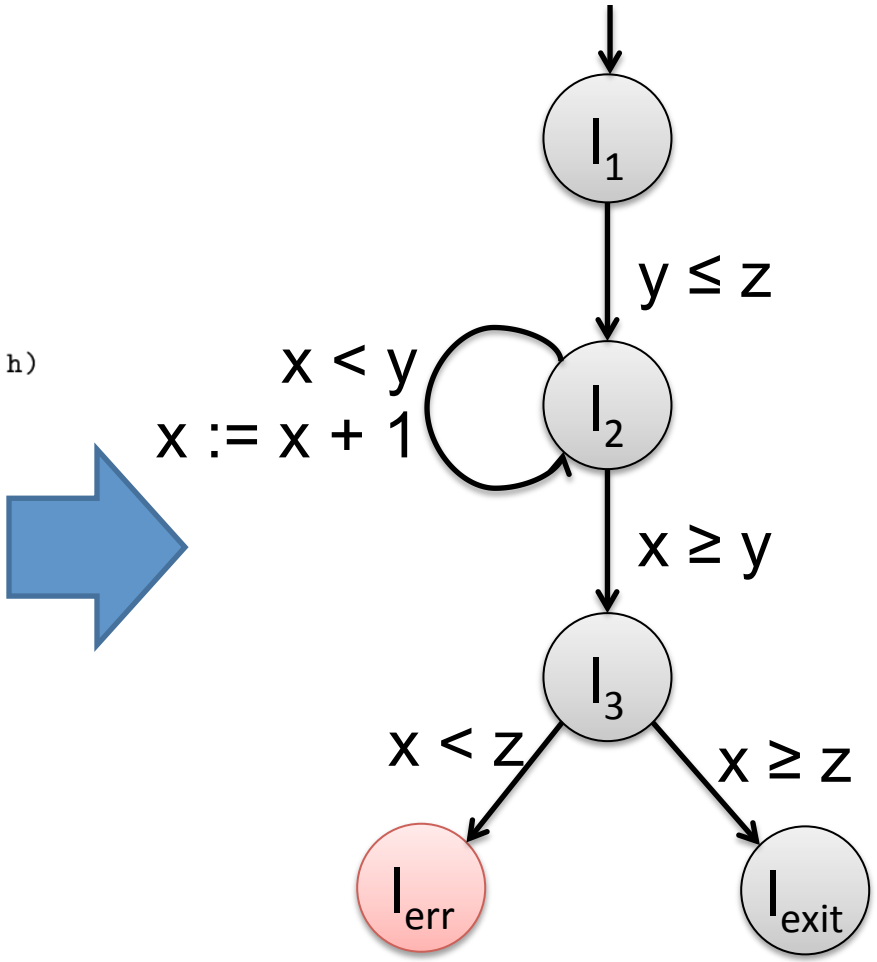
18: a[j] = v

a[0] = 0

a[0] != 0

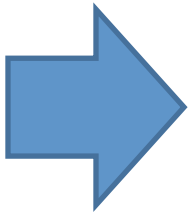
In conclusion:

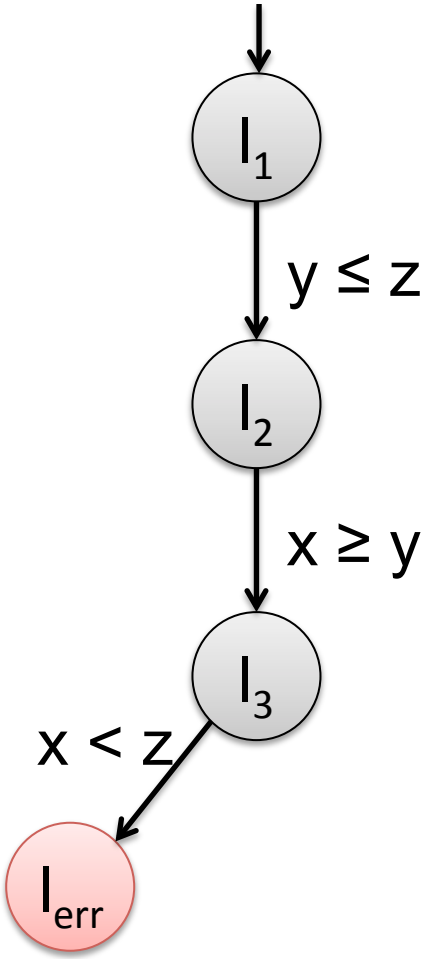
```c
#include<stdio.h>
#include<stdlib.h>

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;
         }
   } 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;
 }
```



In conclusion:

```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;
         }
   } 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;
 }
```





In conclusion:

```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
           int
           for

           if

         }
   } while (h
 }
int main(int a
 {
    int i = 0;
    int *a = NU

    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;
 }
```

Thanks for your attention

☺