

# Static Program Analysis

Mooly Sagiv

# Hoare Proof Rules for Partial Correctness

$$\{A\} \text{ skip } \{A\}$$

$$\{B[a/X]\} X:=a \{B\}$$

$$\frac{\{P\} S_0 \{C\} \quad \{C\} S_1 \{Q\}}{\{P\} S_0; S_1 \{Q\}}$$

$$\{P\} S_0; S_1 \{Q\}$$

$$\frac{\{P \wedge b\} S_0 \{Q\} \quad \{P \wedge \neg b\} S_1 \{Q\}}{\{P\} \text{ if } b \text{ then } S_0 \text{ else } S_1 \{Q\}}$$

$$\{P\} \text{ if } b \text{ then } S_0 \text{ else } S_1 \{Q\}$$

$$\frac{\{I \wedge b\} S \{I\}}{\{I\} \text{ while } b \text{ do } S \{I \wedge \neg b\}}$$

$$\{I\} \text{ while } b \text{ do } S \{I \wedge \neg b\}$$

$$\frac{\models P \Rightarrow P' \quad \{P'\} S \{Q'\} \quad \models Q' \Rightarrow Q}{\{P\} S \{Q\}}$$

$$\{P\} S \{Q\}$$

# Challenges in Proving Partial Correctness

- Specifying what the program is supposed to do
- Writing loop invariants
- Decision procedures for proving implications

# Static Analysis

- Automatically infer sound invariants from the code
- Prove the absence of certain program errors
- Prove user-defined assertions
- Report bugs before the program is executed

# Simple Correct C code

```
main() {  
    int i = 0, *p =NULL, a[100];  
    for (i=0 ; i <100, i++) {  
        a[i] = i;  
        p = malloc(1, sizeof(int));  
        *p = i;  
        free(p);  
        p = NULL;  
    }  
}
```

# Simple Correct C code

```
main() {  
    int i = 0, *p=NULL, a[100];  
    for (i=0 ; i <100, i++) {  
        { 0 <= i < 100 }  
        a[i] = i;  
        { p == NULL:}  
        p = malloc(1, sizeof(int));  
        { alloc(p) }  
        *p = i;  
        { alloc(p) }  
        free(p);  
        { !alloc(p) }  
        p = NULL;  
        { p==NULL }  
    }  
}
```

# Simple Incorrect C code

```
main() {  
    int i = 0, *p=NULL, a[100], j;  
    for (i=0 ; i <j , i++) {  
        { 0 <= i < j }  
        a[i] = i;  
        p = malloc(1, sizeof(int));  
        { alloc(p) }  
        p = malloc(1, sizeof(int));  
        { alloc(p) }  
        free(p);  
        free(p);  
    }  
}
```

# Sound (Incomplete) Static Analysis

- It is undecidable to prove interesting program properties
- Focus on **sound** program analysis
  - When the compiler reports that the program is correct it is indeed correct for every run
  - The compiler may report spurious (false alarms)



# A Simple False Alarm

```
int i, *p=NULL;
```

```
...
```

```
if (i >=5) {  
    p = malloc(1, sizeof(int));  
}
```

```
...
```

```
if (i >=5) {  
    *p = 8;  
}
```

```
...
```

```
if (i >=5) {  
    free(p);  
}
```

# A Complicated False Alarm

```
int i, *p=NULL;
```

```
...
```

```
if (foo(i)) {  
    p = malloc(1, sizeof(int));  
}
```

```
...
```

```
if (bar(i )) {  
    *p = 8;  
}
```

```
...
```

```
if (zoo(i)) {  
    free(p);  
}
```

# Foundation of Static Analysis

- Static analysis can be viewed as interpreting the program over an “abstract domain”
- Execute the program over larger set of execution paths
- Guarantee sound results
  - Whenever the analysis reports that an invariant holds it indeed hold

# Even/Odd Abstract Interpretation

- Determine if an integer variable is even or odd at a given program point

# Example Program

*/\* x=? \*/*

while (x !=1) do { */\* x=? \*/*

    if (x %2) == 0

*/\* x=E \*/*                    { x := x / 2; }           */\* x=? \*/*

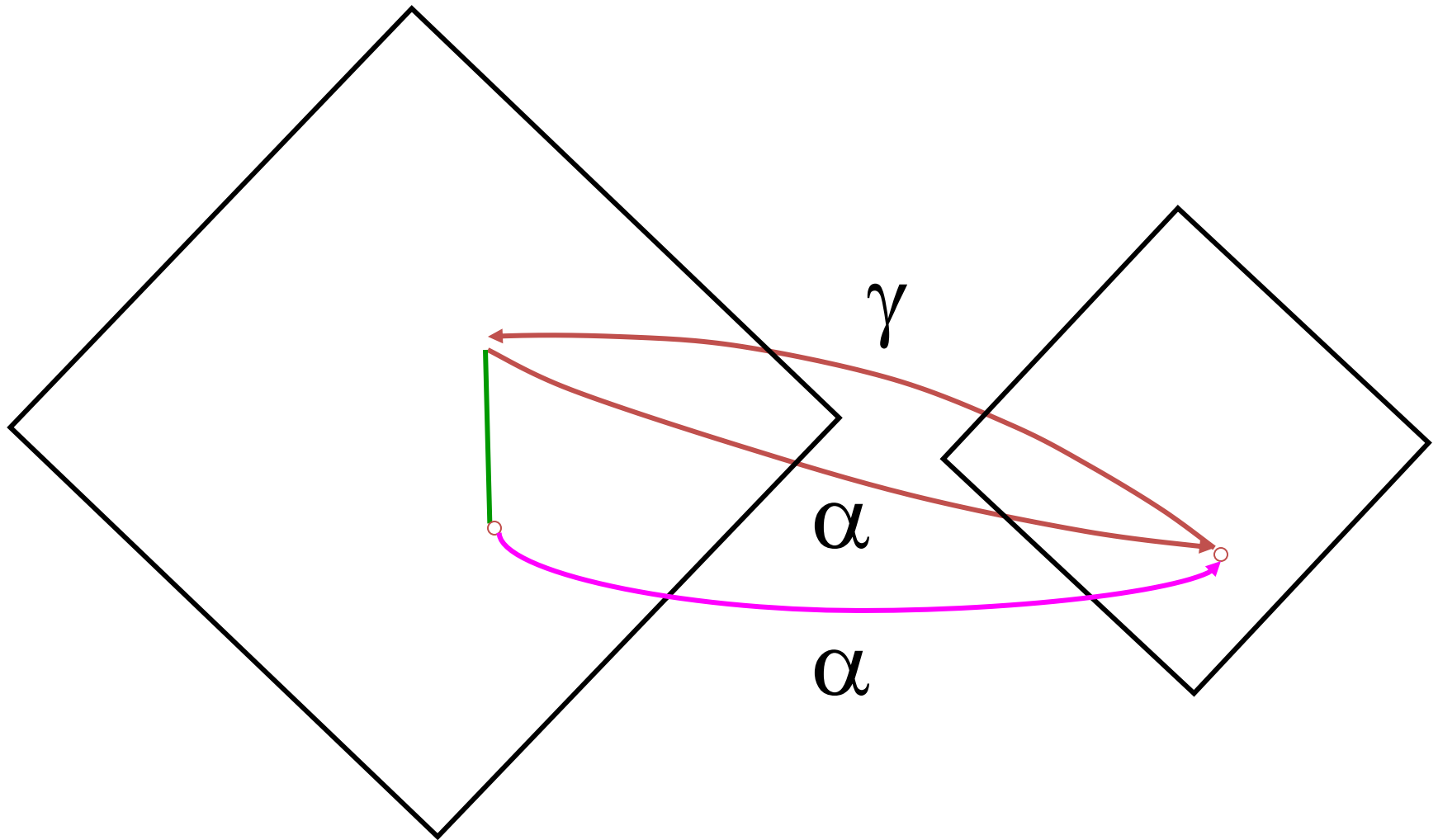
        else

*/\* x=O \*/*                    { x := x \* 3 + 1;  
                                  assert (x %2 ==0); } */\* x=E \*/*

}

*/\* x=O\*/*

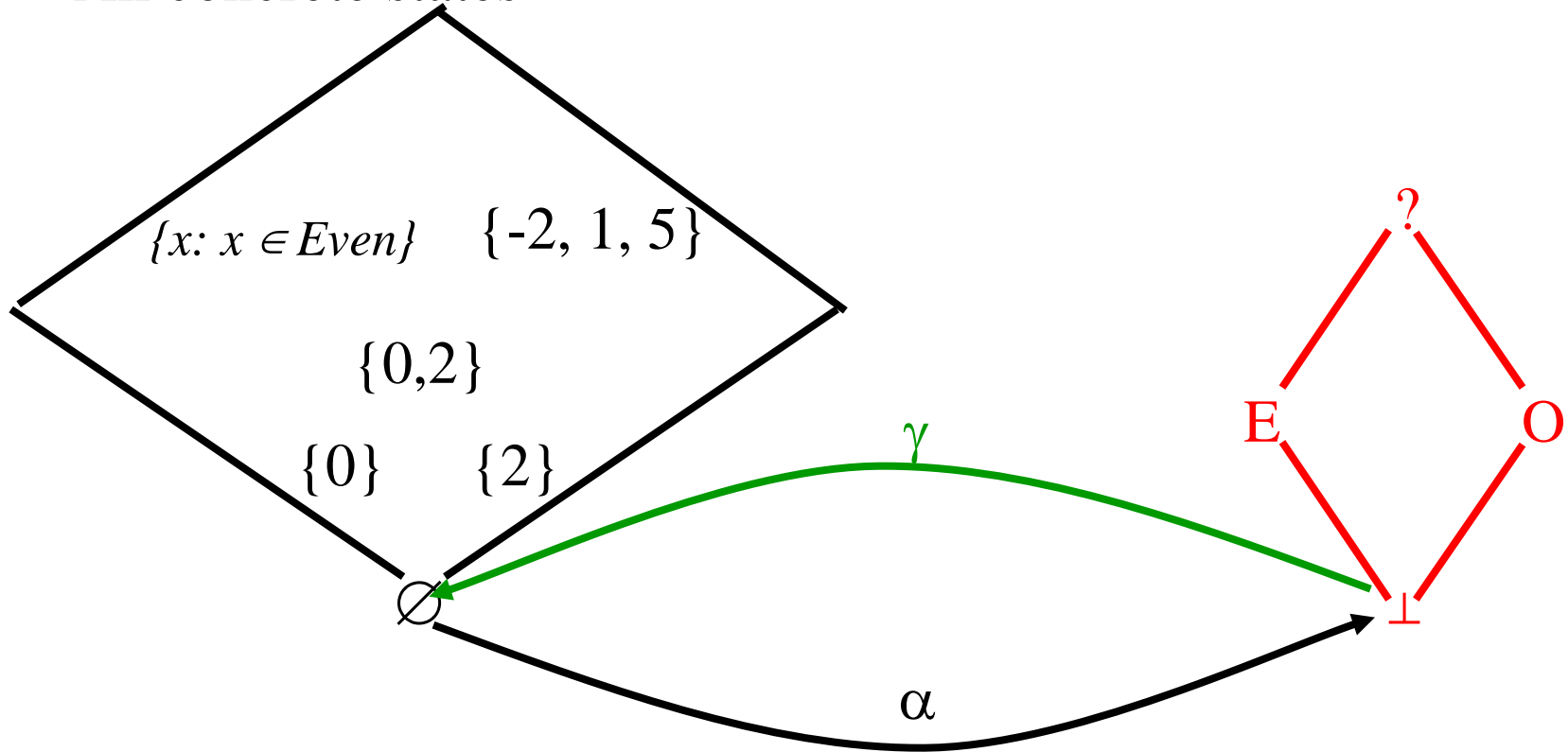
# Abstract Interpretation



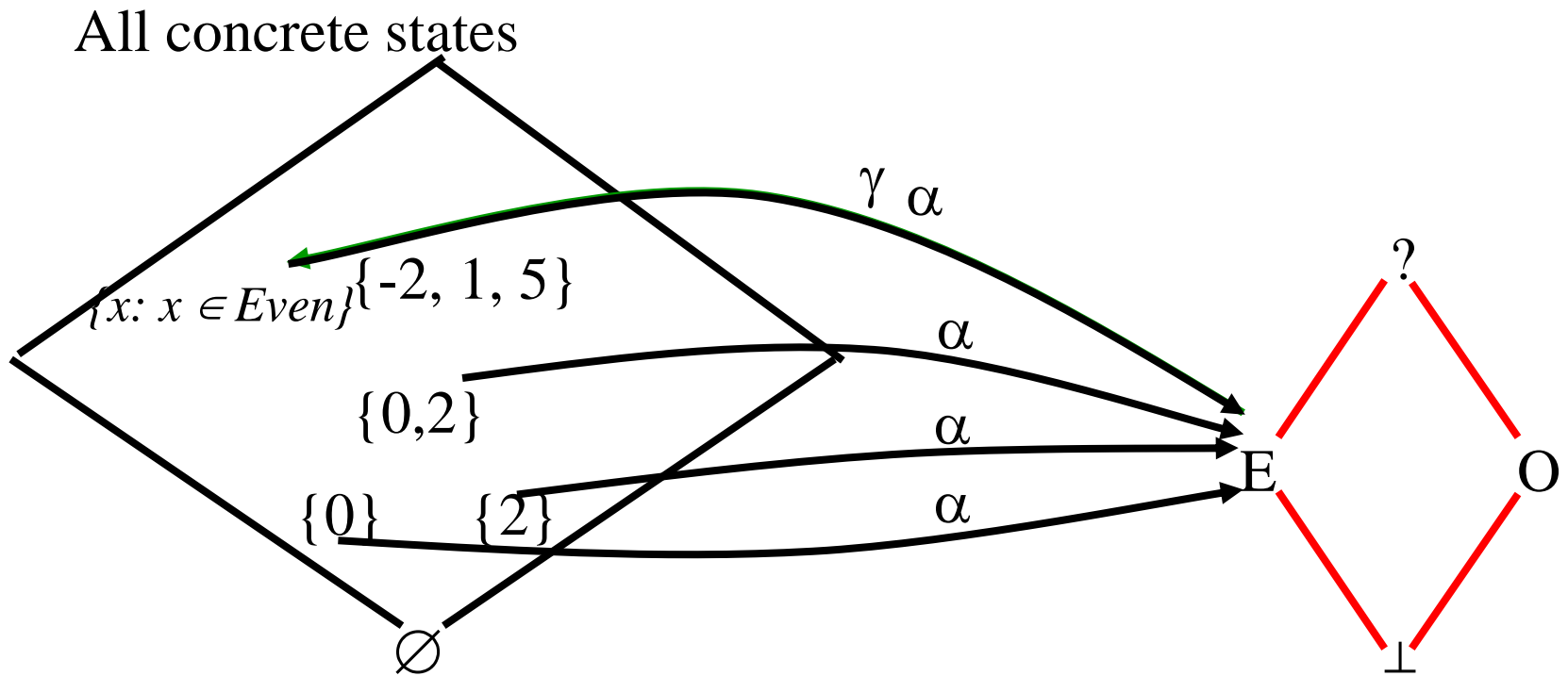
*Sets of stores*  $\xrightarrow{\alpha}$  *Descriptors of sets of stores*

# Odd/Even Abstract Interpretation

All concrete states

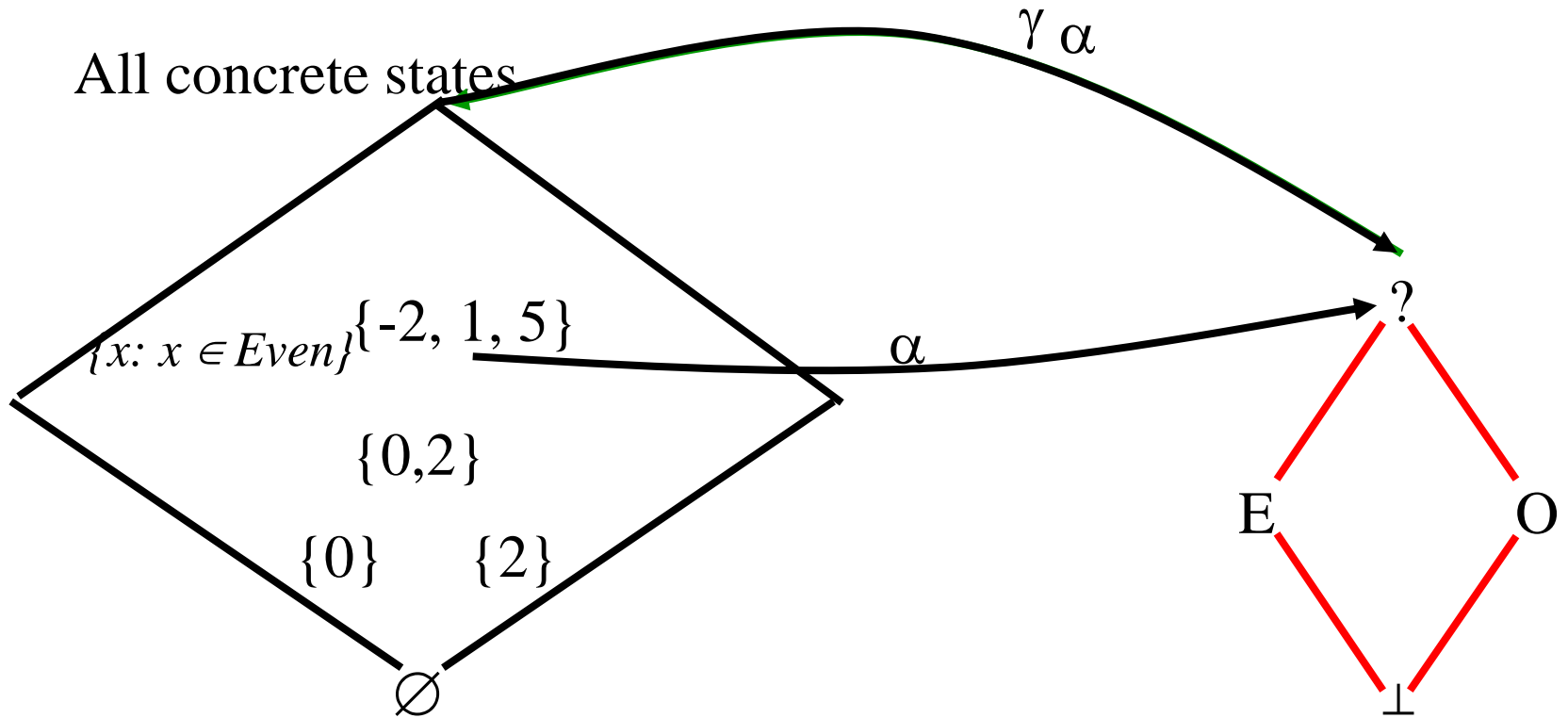


# Odd/Even Abstract Interpretation





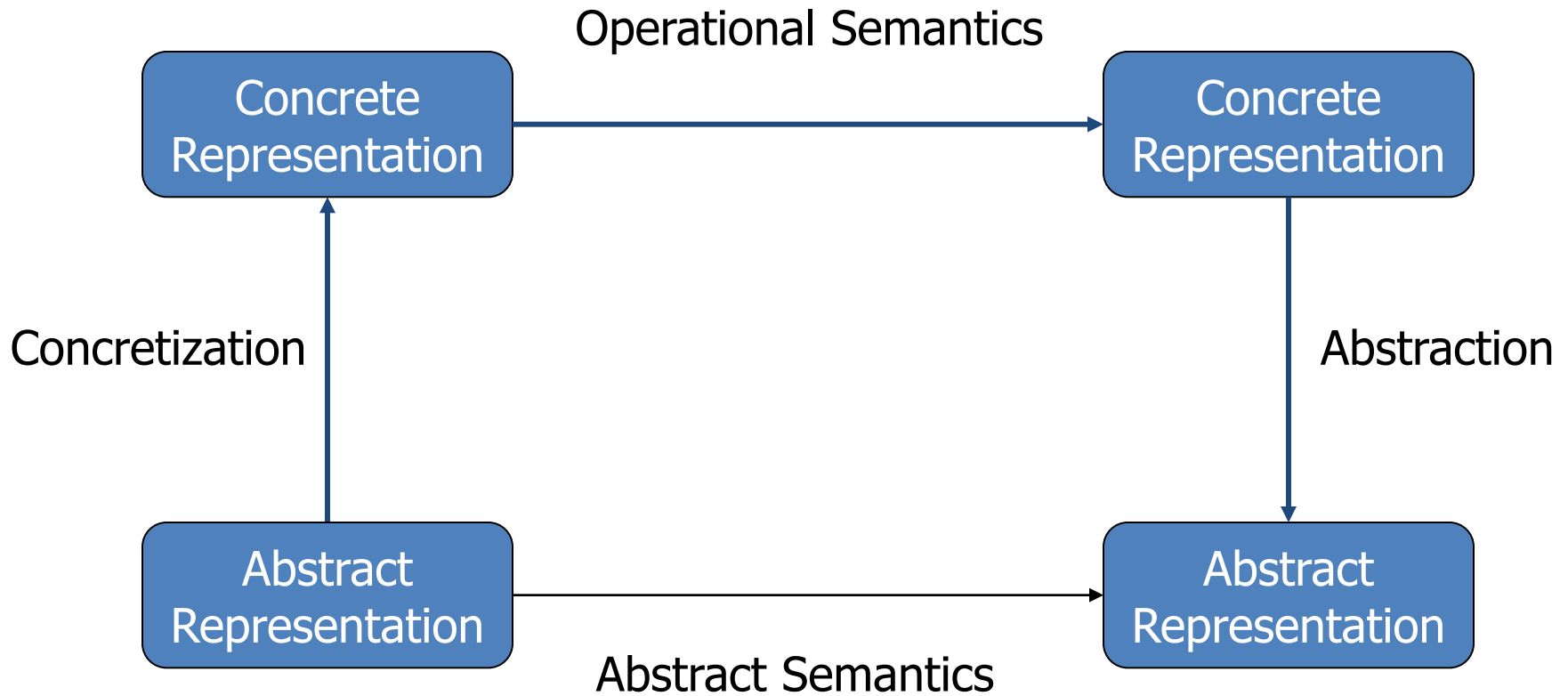
# Odd/Even Abstract Interpretation



# Example Program

```
while (x !=1) do {  
    if (x %2) == 0  
        { x := x / 2; }  
    else  
        /* x=O */ { x := x * 3 + 1;    /* x=E */  
                    assert (x %2 ==0); }  
}
```

# (Best) Abstract Transformer



# Runtime vs. Static Testing

	Runtime	Static Analysis
Effectiveness	Missed Errors	False alarms
		Locate rare errors
Cost	Proportional to program's execution	Proportional to program's size
	No need to efficiently handle rare cases	Can handle limited classes of programs and still be useful

# Static Analysis Algorithms

- Generate a system of equations over the abstract values
- Iteratively compute the least solution to the system
- The solution is guaranteed to be sound
- The correctness of the invariants can be conservatively checked

# Example Interval Analysis

- Find a lower and an upper bound of the value of a single variable
- Can be generalized to multiple variables

# Simple Correct C code

```
main() {  
    int i = 0, a[100];  
    { [-minint, maxint] }  
    for (i=0 ; i <100, i++) {  
        {[0, 99]}  
        a[i] = i;  
        {[0, 99]}  
    }  
    {[100, 100]}
```

# The Power of Interval Analysis

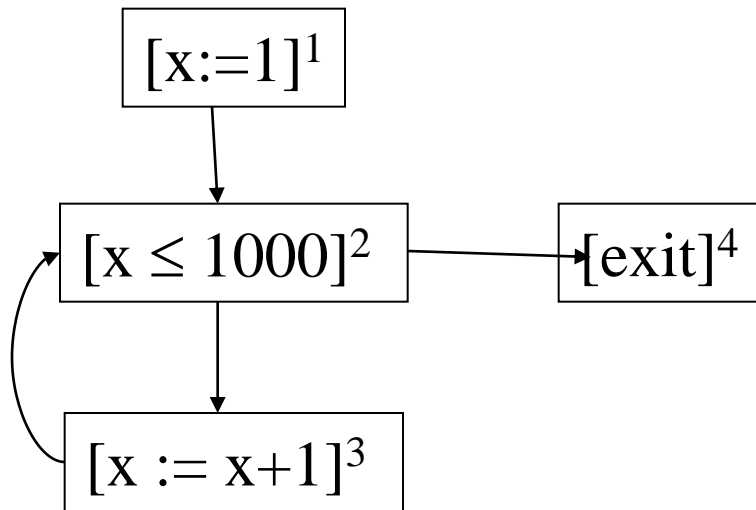
```
int f(x) {  
  { [minint , maxint] }  
  if (x > 100) {  
    { [101, maxint] }  
    return x -10 ;  
    { [91, maxint-10]; }  
  }  
  else {  
    { [minint, 100] }  
    return f(f(x+11))  
    { [91, 91] }  
  }  
}
```



# Example Program

## Interval Analysis

```
[x := 1]1 ;  
while [x ≤ 1000]2  
do  
  [x := x + 1;]3
```



# Abstract Interpretation of Atomic Statements

$$\llbracket \text{skip} \rrbracket^\# [l, u] = [l, u]$$

$$\llbracket x := 1 \rrbracket^\# [l, u] = [1, 1]$$

$$\llbracket x := x + 1 \rrbracket^\# [l, u] = [l, u] + [1, 1] = [l + 1, u + 1]$$

# Equations Interval Analysis

$[x := 1]^1$  ;

while  $[x \leq 1000]^2$

do

$[x := x + 1;]^3$

$En(1) = [\text{minint}, \text{maxint}]$

$Ex(1) = [1, 1]$

$In(2) =$

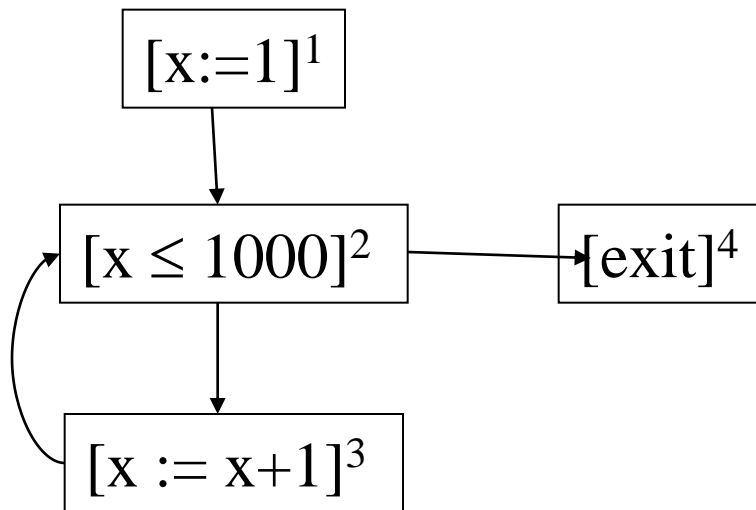
$Ex(2) = In(2)$

$En(3) =$

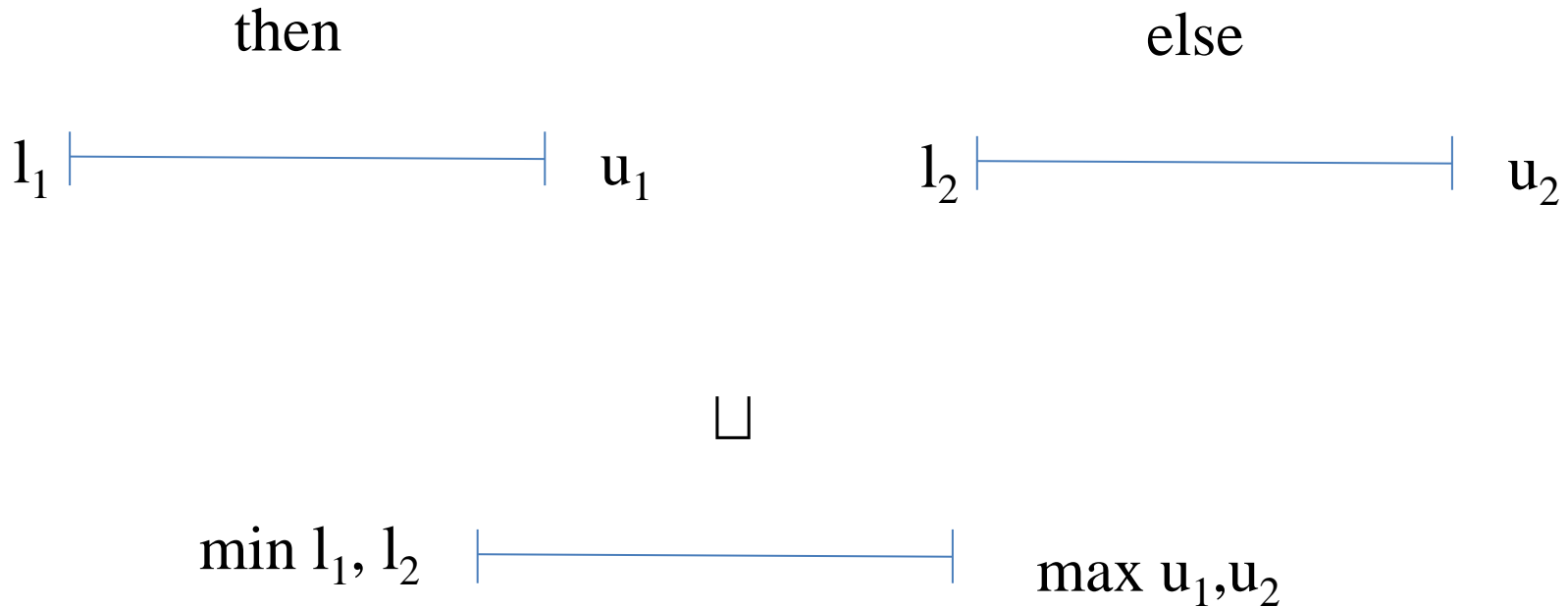
$Ex(3) = In(3) + [1, 1]$

$En(4) =$

$Ex(4) = In(4)$



# Abstract Interpretation of Joins



$$[l_1, u_1] \sqcup [l_2, u_2] = [\min(l_1, l_2), \max(u_1, u_2)]$$

# Equations Interval Analysis

$[x := 1]^1$  ;

while  $[x \leq 1000]^2$

do

$[x := x + 1;]^3$

$En(1) = [\text{minint}, \text{maxint}]$

$Ex(1) = [1, 1]$

$En(2) = En(1) \sqcup En(3)$

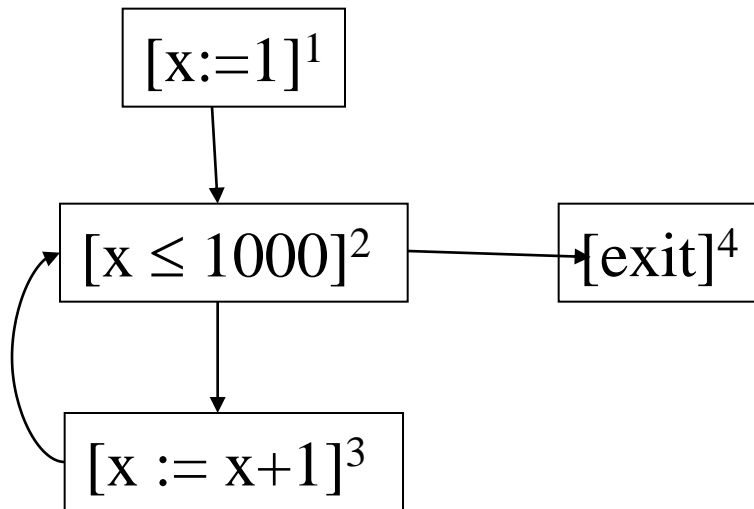
$Ex(2) = En(2)$

$En(3) =$

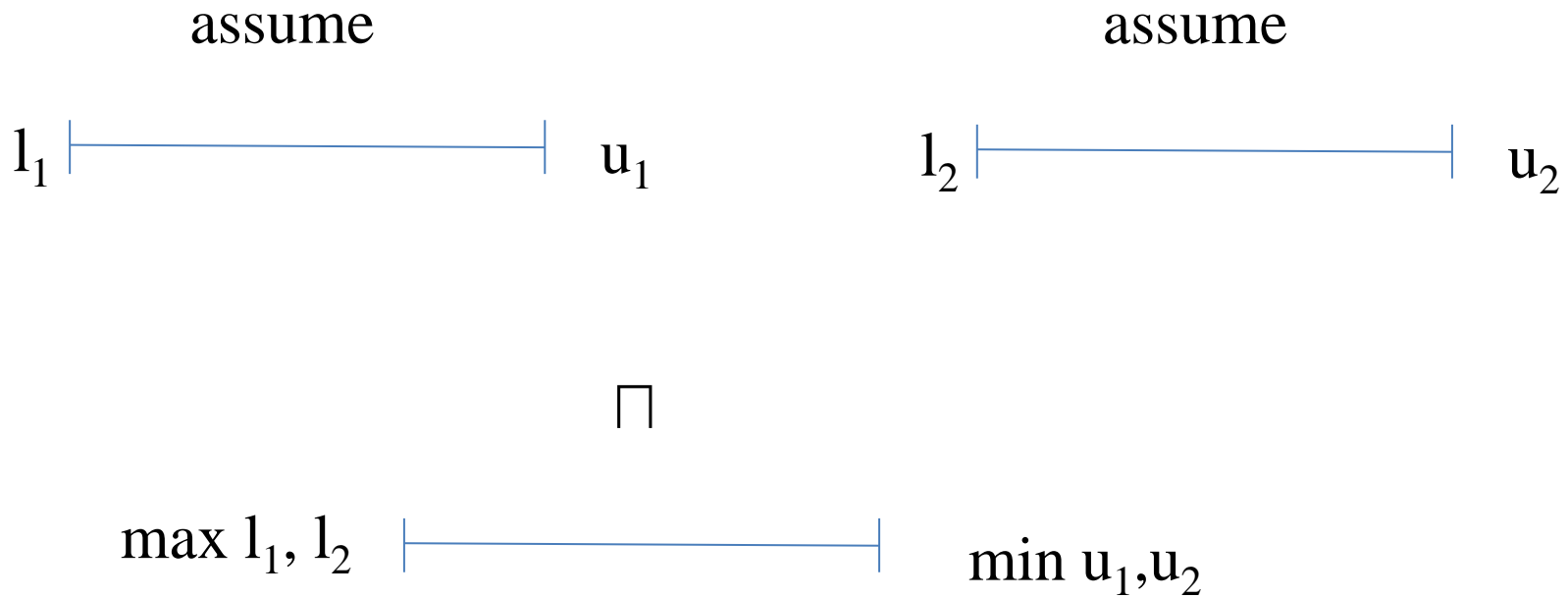
$Ex(3) = En(3) + [1, 1]$

$En(4) =$

$Ex(4) = En(4)$



# Abstract Interpretation of Meets



$$[l_1, u_1] \sqcap [l_2, u_2] = [\max(l_1, l_2), \min(u_1, u_2)]$$

# Equations Interval Analysis

$[x := 1]^1 ;$

while  $[x \leq 1000]^2$

do

$[x := x + 1;]^3$

$$En(1) = [\text{minint}, \text{maxint}]$$

$$Ex(1) = [1, 1]$$

$$En(2) = Ex(1) \sqcup Ex(3)$$

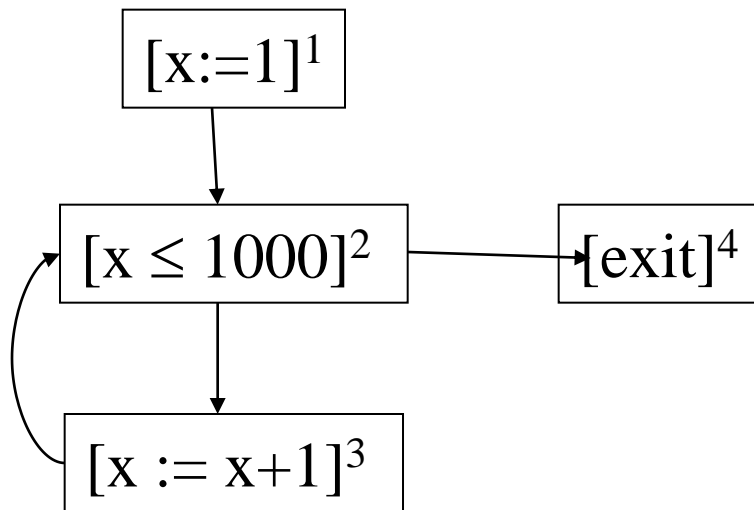
$$Ex(2) = En(2)$$

$$En(3) = Ex(2) \sqcap [\text{minint}, 1000]$$

$$Ex(3) = En(3) + [1, 1]$$

$$En(4) = Ex(2) \sqcap [1001, \text{maxint}]$$

$$Ex(4) = En(4)$$

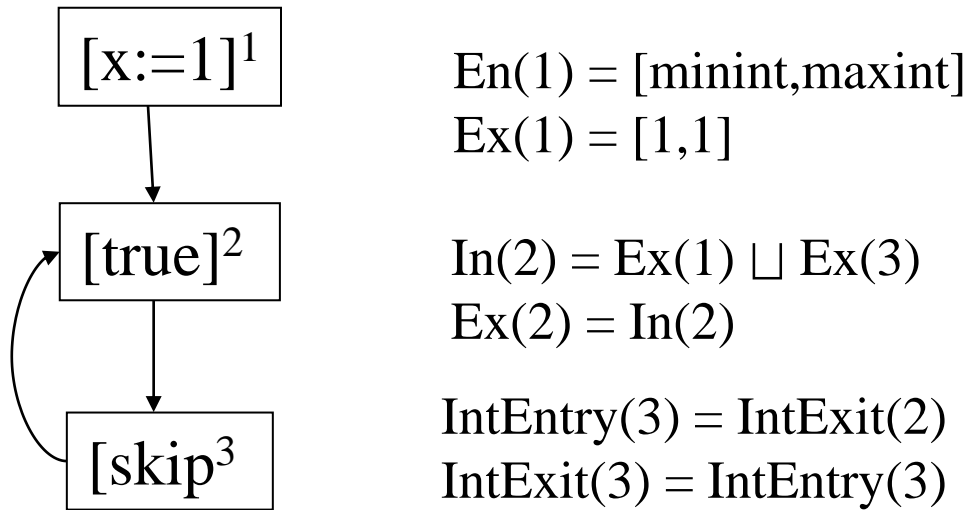


# Solving the Equations

- For programs with loops the equations have many solutions
- Every solution is sound
- Compute a minimal solution



# An Example with Multiple Solutions



En[1]	Ex[1]	En[2]	Ex[2]	En[3]	Ex[3]	Comments
$[-\infty, \infty]$	$[1, 1]$	$[-\infty, \infty]$	$[-\infty, \infty]$	$[-\infty, \infty]$	$[-\infty, \infty]$	Maximal
$[-\infty, \infty]$	$[1, 1]$	$[1, 1]$	$[1, 1]$	$[1, 1]$	$[1, 1]$	Minimal
$[-\infty, \infty]$	$[1, 2]$	$[1, 2]$	$[1, 2]$	$[1, 2]$	$[1, 2]$	Solution
$[-\infty, \infty]$	$\perp$	$[1, 1]$	$[1, 1]$	$[1, 2]$	$[1, 2]$	Not a solution

# Computing Minimal Solution

- Initialize the interval at the entry according to program semantics
- Initialize the rest of the intervals to empty
- Iterate until no more changes

# Iterations Interval Analysis

$$\text{IntEntry}(1) = [\text{minint}, \text{maxint}]$$

$$\text{IntExit}(1) = [1, 1]$$

$$\text{IntEntry}(2) = \text{IntExit}(1) \sqcap \text{IntExit}(3)$$

$$\text{IntExit}(2) = \text{IntEntry}(2)$$

$$\text{IntEntry}(3) = \text{IntExit}(2) \sqcap [\text{minint}, 1000] \quad \text{IntEntry}(4) = \text{IntExit}(2) \sqcap [1001, \text{maxint}]$$

$$\text{IntExit}(3) = \text{IntEntry}(3) + [1, 1]$$

$$\text{IntExit}(4) = \text{IntEntry}(4)$$

En[1]	Ex[1]	En[2]	Ex[2]	En[3]	Ex[3]	In[4]	Ex[4]
$[-\infty, \infty]$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
	[1, 1]						
		[1, 1]					
			[1, 1]				
				[1, 1]			
					[2, 2]		
		[1, 2]					

# Widening

- Accelerate the convergence of the iterative procedure by jumping to a more conservative solution
- Heuristic in nature
- But simple to implement

# Widening for Interval Analysis

- $\perp \nabla [c, d] = [c, d]$
- $[a, b] \nabla [c, d] = [$   
    if  $a \leq c$   
        then  $a$   
        else  $-\infty$ ,  
    if  $b \geq d$   
        then  $b$   
        else  $\infty$   
    ]

# Iterations with widening

$$\text{IntEntry}(1) = [\text{minint}, \text{maxint}]$$

$$\text{IntEntry}(2) = \text{IntEntry}(2) \nabla (\text{IntExit}(1) \sqcup \text{IntExit}(3))$$

$$\text{IntExit}(1) = [1, 1]$$

$$\text{IntExit}(2) = \text{IntEntry}(2)$$

$$\text{IntEntry}(3) = \text{IntExit}(2) \sqcap [\text{minint}, 1000] \quad \text{IntEntry}(4) = \text{IntExit}(2) \sqcap [1001, \text{maxint}]$$

$$\text{IntExit}(3) = \text{IntEntry}(3) + [1, 1]$$

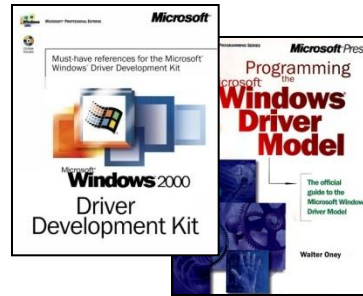
$$\text{IntExit}(4) = \text{IntEntry}(4)$$

En[1]	Ex[1]	En[2]	Ex[2]	En[3]	Ex[3]	In[4]	Ex[4]
$[-\infty, \infty]$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
	[1, 1]						
		[1, 1]					
			[1, 1]				
				[1, 1]			
					[2, 2]		
		[1, $\infty$ ]					
			[1, $\infty$ ]				
				[1, 1000]			

# Some Success Stories

- The SLAM Static Driver Verification (MSR)
- Polyspace (INRIA, Mathworks)
- aiT (Abslint)
- [The Astrée Static Analyzer](#)

# Static Driver Verifier



Rules

## Static Driver Verifier

Precise  
API Usage Rules  
(SLIC)

Environment  
model



Defects

100% path  
coverage



**SLAM**  
`if(node->l; i++ & node->end) node;{  
procs,`

Driver's Source Code in C



# Example SLAM Application

The screenshot displays the Static Driver Verifier (SDV) interface. The main window is titled "Static Driver Verifier Report Page - [Static Driver Verifier Defect Viewer]". It is divided into several panes:

- Trace Tree:** Shows the execution path starting from `sdv_main` and ending at `sdv_stub_custom_main_end`. The path includes `DriverRead`, `CustomAcquireLock`, `SLIC_CustomAcquireLock_exit`, `CustomMemMove`, and `DriverRead`. The final `DriverRead` call is highlighted in blue.
- Source Code:** Shows the source code for `fail_driver1.c`. The function `DriverRead` is visible, with line 45 highlighted in blue: `return CUSTOM_STATUS_UNSUCCESSFUL;`. The code includes declarations for `CUSTOM_LOCK`, `int reads`, `int writes`, `char buffer[512]`, and `PCUSTOM_IRP Irp`. It also shows the `CustomAcquireLock` and `CustomReleaseLock` functions being called.
- State:** Shows the current state of the program at Step 30: `status==1`, `SLAM guard==&(DriverData.Lock)`, `s==1`, and `s!=0`.
- Defect(s):** Shows a defect titled "Defect(s)(2)" under the component "customlock". The defect is marked with a red 'X' and a blue 'X' icon, indicating a failure.

The status bar at the bottom of the window displays the message: "The driver has returned from an entry point without releasing the lock."

# Summary

- Static analysis is powerful
- Can locate rear bugs
- Challenges
  - Specification
  - Scalability
  - False alarms