

# Program Analysis

## 4. Concepts in Program Analysis

Kihong Heo



# Impact of Poor Software Quality



The Patriot Missile (1991)  
Floating-point roundoff  
28 soldiers died



The Ariane-5 Rocket (1996)  
Integer Overflow  
\$100M



NASA's Mars Climate Orbiter (1999)  
Meters-Inches Miscalculation  
\$125M

**CNN** U.S. | World | Politics | Money | Opinion | Health | Entertainment | Tech | Style | Travel | Sports | Video | Live TV

The 'Heartbleed' security flaw that affects most of the Internet

By Heather Kelly, CNN  
Updated 5:11 PM ET, Wed April 9, 2014

**Top stories**  
Trump: 'I th...  
Cory Book... against coll...

This dangerous Android security bug could let anyone hack your phone camera

By Anthony Spadafora November 23, 2019

Camera app vulnerabilities allow attackers to remotely take photos, record video and spy on users

**What Boeing's 737 MAX Has to Do With Cars: Software**

Investigators believe faulty software contributed to two fatal crashes. A newly discovered fault will likely will keep the 737 MAX grounded until the fall.

(Image credit: Shutterstock.com)



Homeland Security warns that certain heart devices can be hacked



New in Life & Style

Hannahith 4th-graders bond through poetry, art and Steph Curry  
10:31 PM

6 ways to celebrate Valentine's Day in Lake Geneva  
8:53 AM

Six ways to keep your kids healthy during winter  
8:36 AM

See More

# Cost of Software Quality Assurance



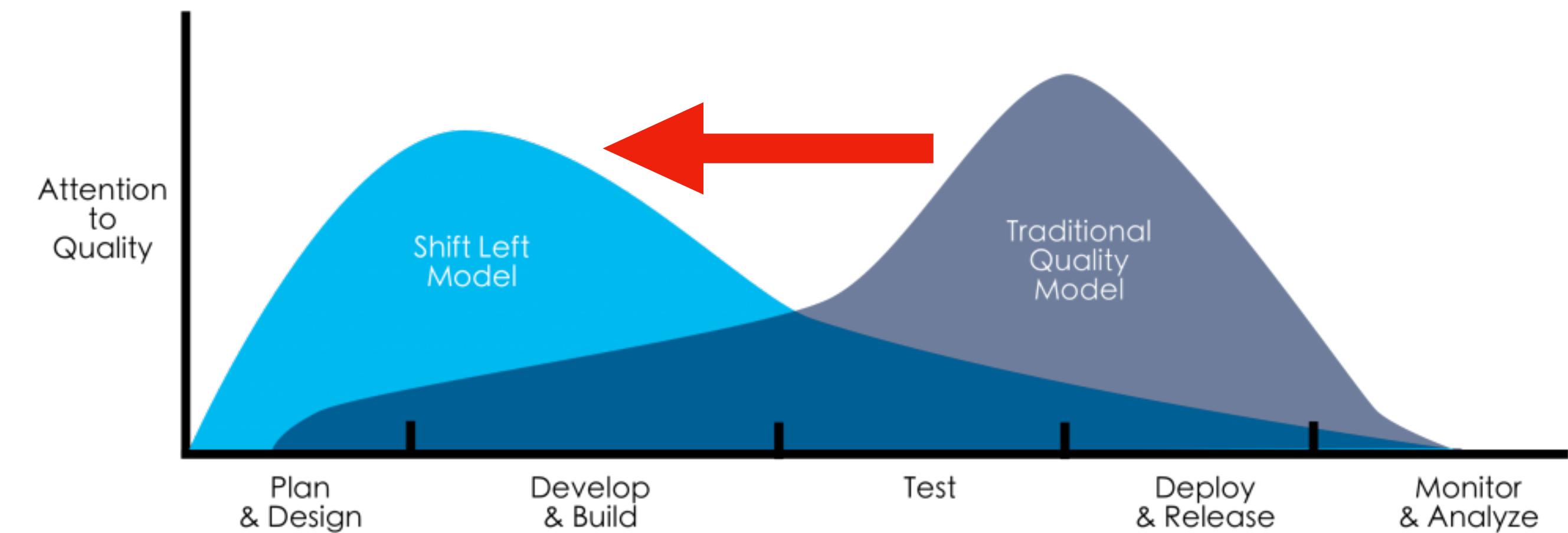
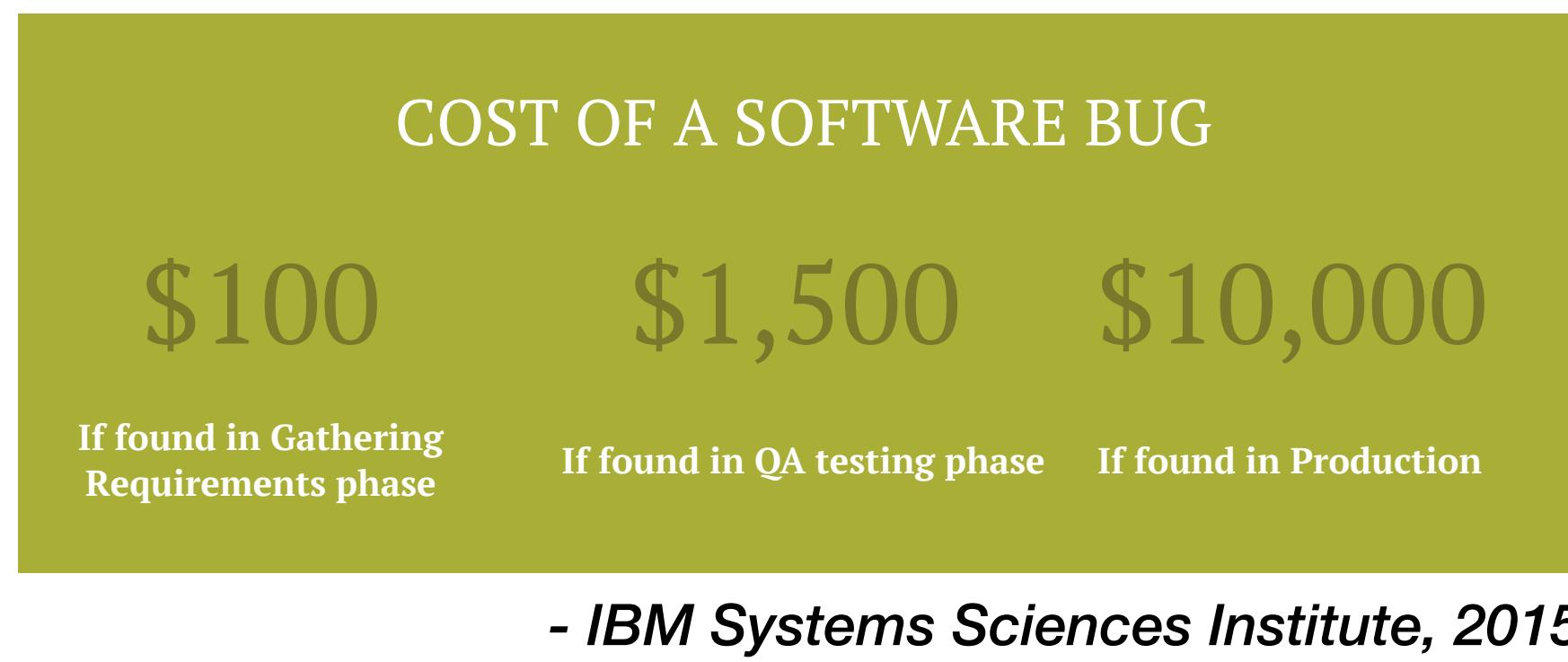
*“We have as **many testers** as we have developers.  
And testers spend **all their time testing**, and developers spend  
**half their time testing**. We’re more of a testing, a quality software  
organization than we’re a software organization”*  
- Bill Gates

**Q:** What is the solution to improve software quality at low cost?

**A:** Program analysis

# Discovering Software Errors

- The first step of SW reliability
- Key issue: how to detect SW errors as early as possible?



# What to Analyze?

CWE Definitions		
Sort Results By : CWE Number Vulnerability Count		
Total number of cwe definitions : 668 Page : 1 (This Page) 2 3 4 5 6 7 8 9 10 11 12 13 14		
<a href="#">Select</a> <a href="#">Select&amp;Copy</a>		
CWE Number	Name	Number Of Related Vulnerabilities
<a href="#">119</a>	Failure to Constrain Operations within the Bounds of a Memory Buffer	<a href="#">12328</a>
<a href="#">79</a>	Failure to Preserve Web Page Structure ('Cross-site Scripting')	<a href="#">11807</a>
<a href="#">20</a>	Improper Input Validation	<a href="#">7669</a>
<a href="#">200</a>	Information Exposure	<a href="#">6316</a>
<a href="#">89</a>	Improper Sanitization of Special Elements used in an SQL Command ('SQL Injection')	<a href="#">5643</a>
<a href="#">22</a>	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	<a href="#">2968</a>
<a href="#">94</a>	Failure to Control Generation of Code ('Code Injection')	<a href="#">2400</a>
<a href="#">125</a>	Out-of-bounds Read	<a href="#">2122</a>
<a href="#">287</a>	Improper Authentication	<a href="#">1746</a>
<a href="#">284</a>	Access Control (Authorization) Issues	<a href="#">1627</a>
<a href="#">416</a>	Use After Free	<a href="#">1256</a>
<a href="#">190</a>	Integer Overflow or Wraparound	<a href="#">1113</a>
<a href="#">476</a>	NULL Pointer Dereference	<a href="#">900</a>
<a href="#">78</a>	Improper Sanitization of Special Elements used in an OS Command ('OS Command Injection')	<a href="#">788</a>
<a href="#">787</a>	Out-of-bounds Write	<a href="#">737</a>
<a href="#">362</a>	Race Condition	<a href="#">615</a>
<a href="#">59</a>	Improper Link Resolution Before File Access ('Link Following')	<a href="#">518</a>
<a href="#">77</a>	Improper Sanitization of Special Elements used in a Command ('Command Injection')	<a href="#">489</a>
<a href="#">400</a>	Uncontrolled Resource Consumption ('Resource Exhaustion')	<a href="#">463</a>
<a href="#">611</a>	Information Leak Through XML External Entity File Disclosure	<a href="#">393</a>
<a href="#">434</a>	Unrestricted Upload of File with Dangerous Type	<a href="#">385</a>
<a href="#">732</a>	Incorrect Permission Assignment for Critical Resource	<a href="#">350</a>
<a href="#">74</a>	Failure to Sanitize Data into a Different Plane ('Injection')	<a href="#">327</a>
<a href="#">798</a>	Use of Hard-coded Credentials	<a href="#">319</a>
<a href="#">772</a>	Missing Release of Resource after Effective Lifetime	<a href="#">306</a>
<a href="#">269</a>	Improper Privilege Management	<a href="#">305</a>
<a href="#">601</a>	URL Redirection to Untrusted Site ('Open Redirect')	<a href="#">265</a>
<a href="#">502</a>	Deserialization of Untrusted Data	<a href="#">257</a>
<a href="#">134</a>	Uncontrolled Format String	<a href="#">216</a>
<a href="#">704</a>	Incorrect Type Conversion or Cast	<a href="#">180</a>
<a href="#">415</a>	Double Free	<a href="#">173</a>



**Heartbleed, 2014  
OpenSSL  
CVE-2014-0160**



**Shellshock, 2014  
Bash  
CVE-2014-6271**



**goto fail, 2014  
MacOS / iOS  
CVE-2014-1266**

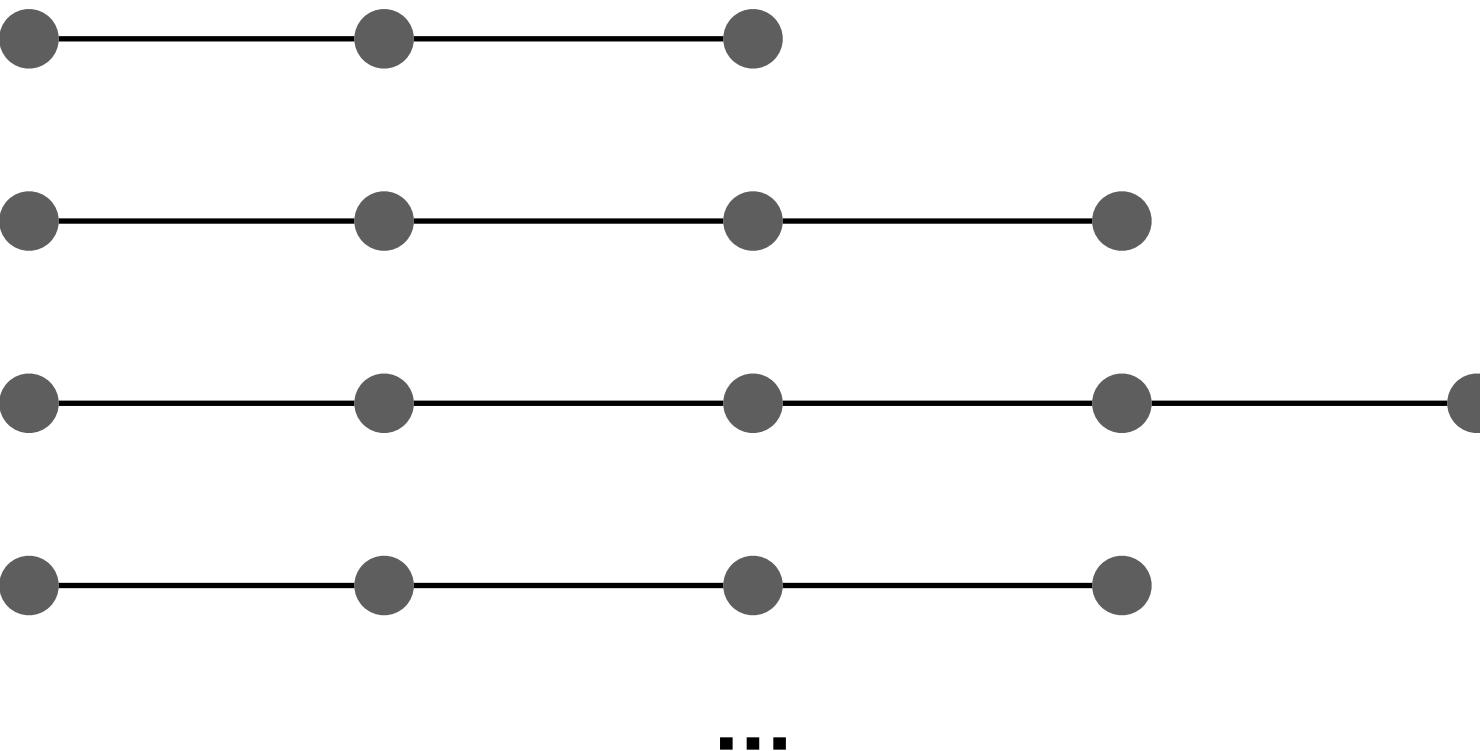
# Properties

- Points of interest in programs
  - for verification, bug detection, optimization, understanding, etc
  - E.g., “ $p == \text{NULL?}$ ”, “ $\text{idx} < \text{size?}$ ”, “ $\text{fp}$  can be only f, g, or h?”, “value of x”, etc
- Two categories:
  - Trace properties = properties of individual execution traces
    - safety properties + liveness properties
  - Information-flow properties = properties of multiple execution traces

# Trace

- Trace = a list of states
- Recall small-step operational semantics
- A program can have a set of (infinite) set of traces
- $\llbracket P \rrbracket$  : a set of all possible execution traces

$$\begin{aligned} & (2 \times 2 \times 2) \times (2 + 1) \\ \rightarrow & (4 \times 2) \times (2 + 1) \\ \rightarrow & 8 \times (2 + 1) \\ \rightarrow & 8 \times 3 \\ \rightarrow & 24 \end{aligned}$$

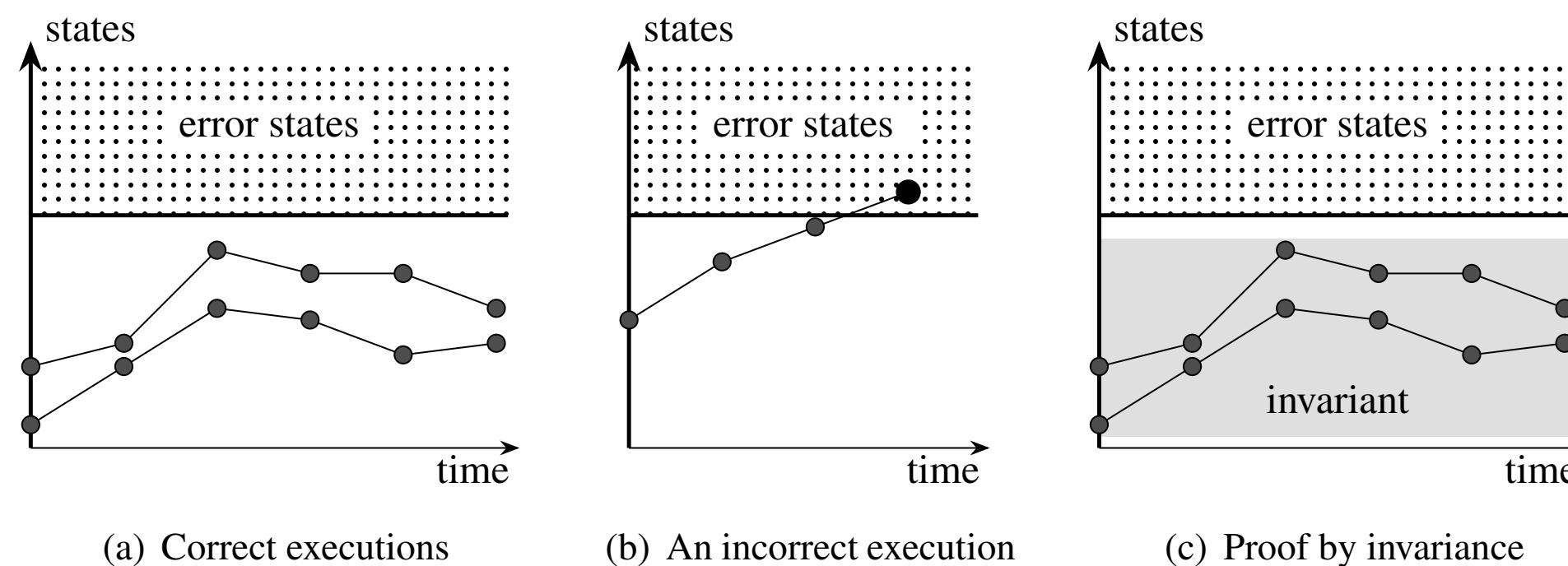


# Trace Properties

- A semantic property  $\mathcal{P}$  that can be defined by a **set of execution traces** that satisfies  $\mathcal{P}$ 
  - Ex1: “all traces that satisfies  $x \neq 0$  at line 10”
  - Ex2: “all traces where the value of  $y$  at line 97 is the same as the one in the entry point”
- Program  $P$  satisfies property  $\mathcal{P}$  iff  $\llbracket P \rrbracket \subseteq T_{\mathcal{P}}$
- State properties: defined by a set of states (so, obviously trace properties)
  - E.g., division-by-zero, integer overflow
- Any trace property: the conjunction of a safety and a liveness property

# Safety Property

- A program **never** exhibit a behavior observable within **finite time**
  - “Bad things will never occur”
  - Bad things: integer overflow, buffer overrun, deadlock, etc
- If false, then there exists a **finite counterexample**
- To prove: all executions never reach error states



# Invariant

- Assertions supposed to be **always true**
  - Starting from a state in the invariant, any computation step also leads to another state in the invariant (i.e., fixed point!)
  - E.g., “x has an int value during the execution”, “y is larger than 1 at line 5”
- Loop invariant: assertion to be true at the beginning of every loop iteration

$$\frac{\{B \wedge I\} \ C \ \{I\}}{\{I\} \text{ while } B \ C \{\neg B \wedge I\}}$$

```
x = 0;  
while (x < 10) {  
    x = x + 1;  
}  
assert(x > 0);  
assert(x == 10);
```

Loop invariant 1: “x is an integer”

Loop invariant 2: “x is a positive integer”

Loop invariant 3: “ $0 \leq x \leq 10$ ”

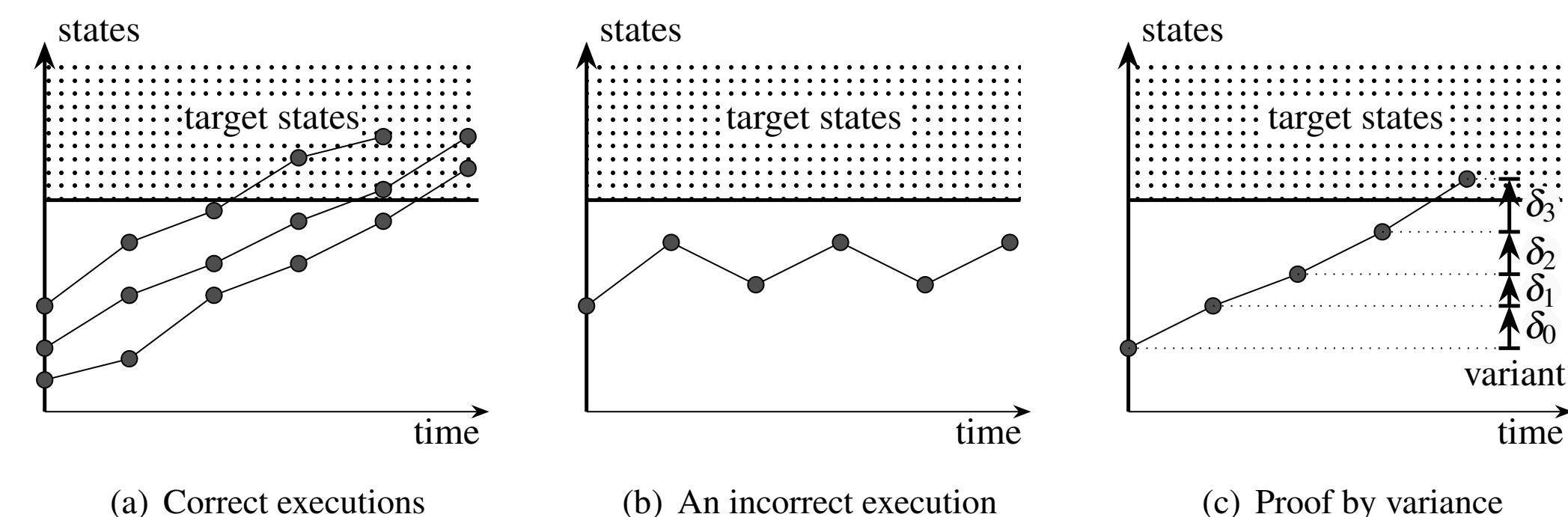
# Example: Division-by-Zero

```
1: int main(){
2:     int x = input();
3:     x = 2 * x - 1;
4:     while (x > 0) {
5:         x = x - 2;
6:     }
7:     assert(x != 0);
8:     return 10 / x;
9: }
```

```
1: int main(){
2:     int x = input();
3:     x = 2 * x;
4:     while (x > 0) {
5:         x = x - 2;
6:     }
7:     assert(x != 0);
8:     return 10 / x;
9: }
```

# Liveness Property

- A program will **never** exhibit a behavior observable only after **infinite time**  
(A program will **eventually** exhibit a behavior observable within **finite time**)
  - “Good things will eventually occur”
  - Good things: termination, fairness, etc
- If false then there exists an **infinite counterexample**
- To prove: all executions eventually reach target states



# Variant

- A quantity that **evolves towards** the set of target states (so guarantee any execution eventually reach the set)
- Usually, a value that is strictly decreasing for some well-founded order relation
  - Well-founded order: there exists a minimal element
  - E.g., an integer value is always positive and strictly decreasing

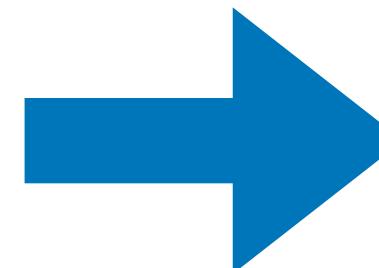
```
x = pos_int();  
while (x > 0) {  
    x = x - 1;  
}
```

**x is always a positive integer**  $\wedge$  **x is strictly decreasing**  $\Rightarrow$  **The program terminates**

# Example: Termination

- Introduce variable  $\underline{c}$  that stores the value of “step counter”
  - Initially,  $\underline{c}$  is equal to zero
  - Each program execution step increments  $\underline{c}$  by one

```
// A factorial program  
i = n;  
r = 1;  
while (i > 0) {  
    r = r * i;  
    i = i - 1;  
}
```



$\underline{c} \leq 3n + 2$

```
// An instrumented program  
i = n;  
r = 1;  
c = 2;  
while (i > 0) {  
    r = r * i;  
    i = i - 1;  
    c = c + 3;  
}  
// what is the value of c in the loop?
```

$0 \leq 3n + 2 - \underline{c} \wedge 3n + 2 - \underline{c}$  is strictly decreasing  $\Rightarrow$  termination

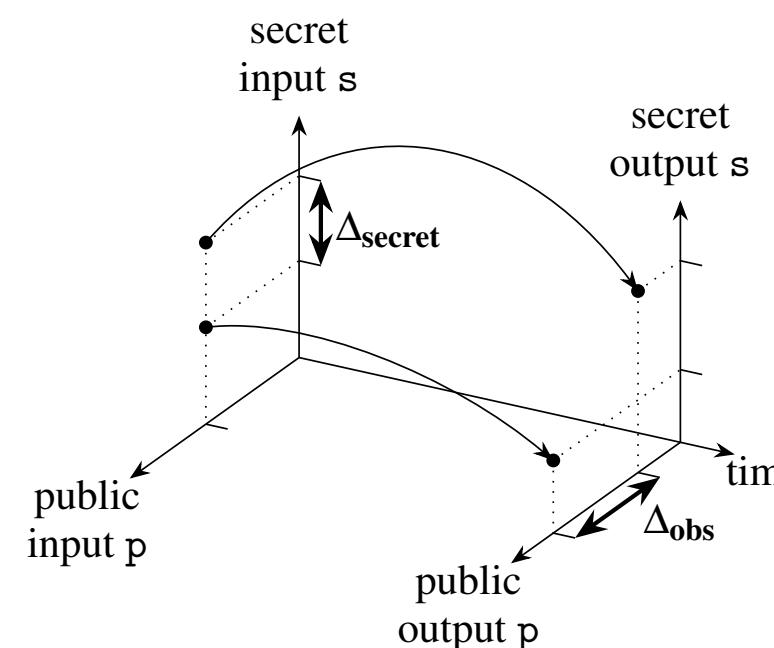
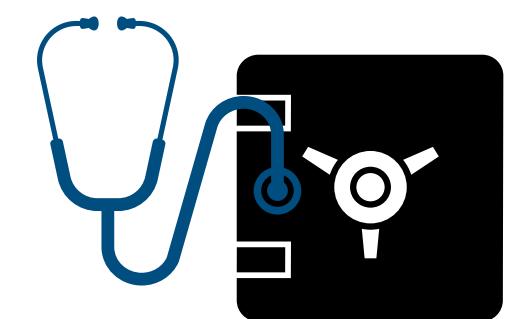
# Example

- Correctness of a sorting algorithm as trace property

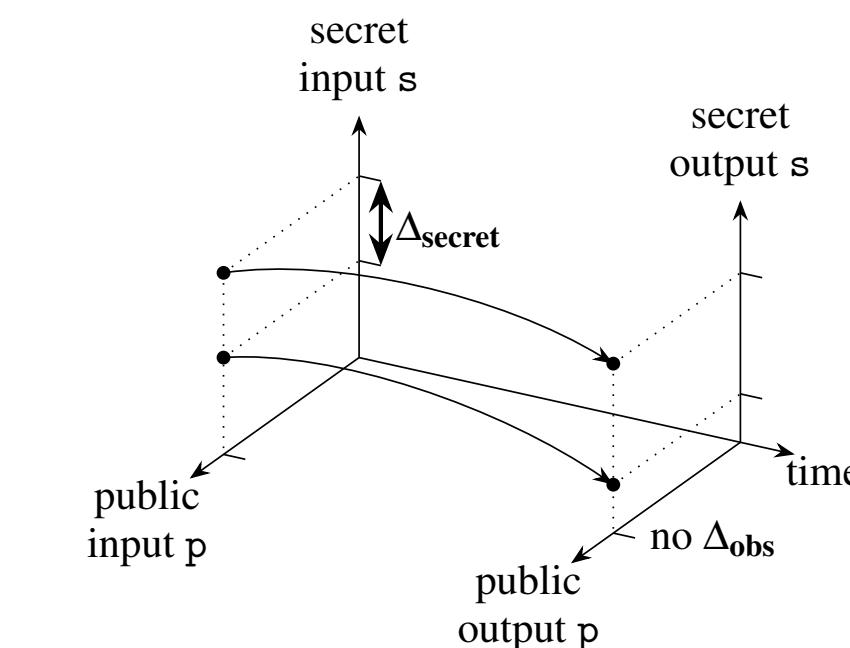
Property	Safety or Liveness?	State?
Should not fail with a run-time error		
Should terminate		
Should return a sorted array (if terminated)		
Should return an array with the same elements and multiplicity (if terminated)		

# Information Flow Properties

- Properties stating the absence of dependence between **pairs of executions**
  - Beyond trace properties: so called **hyper-properties**
- Mostly for security: multiple executions with public data should not derive private data
- E.g., a door lock beeps louder if a right digit is pressed at the right position



A pair of executions with insecure information flow



A pair of executions without insecure information flow

# Example

- Assume that variables s (secret) and p (public) take only 0 and 1

```
// Program 0  
p_out := p_in * [0, 1]
```

```
// Program 1  
p_out := p_in * s * [0, 1]
```

```
// Program 2  
p_out := p_in + [0, 1] - s
```

Input		Output
p	s	p
0	0	{0, 1}
0	1	{0, 1}
1	0	{0, 1}
1	1	{0, 1}

Input		Output
p	s	p
0	0	{0}
0	1	{0}
1	0	{0}
1	1	{0, 1}

Input		Output
p	s	p
0	0	{0, 1}
0	1	{0, 1}
1	0	{0, 1}
1	1	{0, 1}

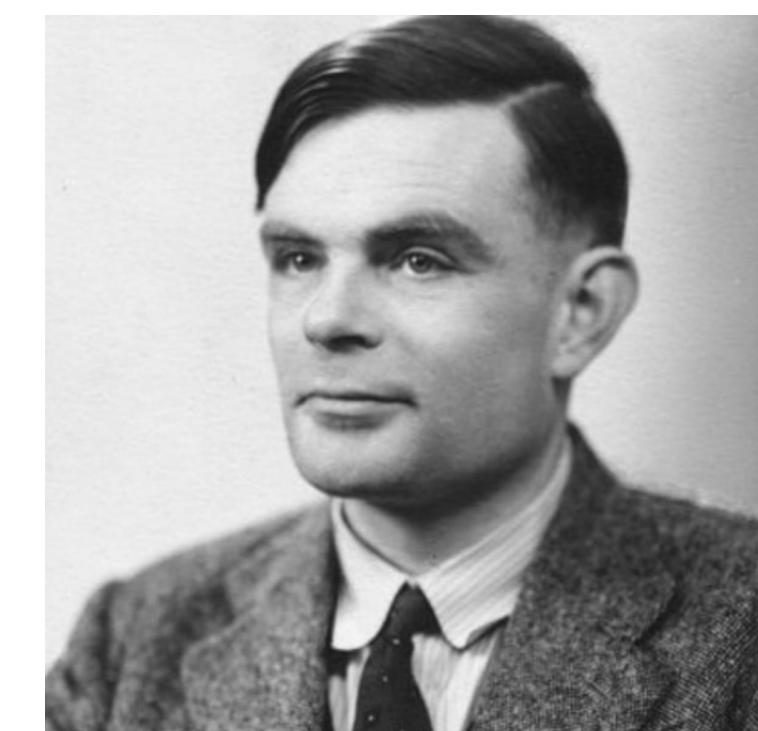
# A Hard Limit: Undecidability

**Theorem (Rice's theorem).** Any **non-trivial** semantic properties are **undecidable**.

- Non-trivial property: worth the effort of designing a program analyzer for
  - trivial: true or false for all programs
- Undecidable? If decidable, it can solves the Halting problem!

HP: Given a Turing machine  $T$  and an input  $i$ , does  $T$  eventually halt on  $i$ ?

Undecidable: There is no Turing machine that can solve HP!



# Informal Proof of Undecidability of HP

HP: Given a Turing machine  $T$  and an input  $i$ , does  $T$  eventually halt on  $i$ ?

- Assume  $H(T, i)$  returns true or false
- Let  $F(x) = \text{if } H(x, x) \text{ then loop() else halt()}$
- Does  $F(F)$  terminate?

# Informal Proof of Rice's Theorem

- Assumption: HP is undecidable
- An analyzer **A** for a property: “*This program always prints 1 and finishes*”
- Given a program **P**, generate **P'** = “**P**; print 1;”
- Analyze **P'** using **A**: **A(P')**
  - **A(P')** says “Yes”: **P** halts,
  - **A(P')** says “No”: **P** does not halt
- HP is decidable if we use **A** : contradiction!

# Toward Computability

## Undecidable

⇒ Automatic, terminating, and exact reasoning is impossible  
⇒ If we give up one of them, it is computable!

- Manual rather than automatic: assisted proving
  - require expertise and manual effort
- Possibly nonterminating rather than terminating: model checking, testing
  - require stopping mechanisms such as timeout
- Approximate rather than exact: static analysis
  - report spurious results

# Soundness and Completeness

- Given a semantic property  $\mathcal{P}$ , and an analysis tool  $A$
- If  $A$  were perfectly accurate,

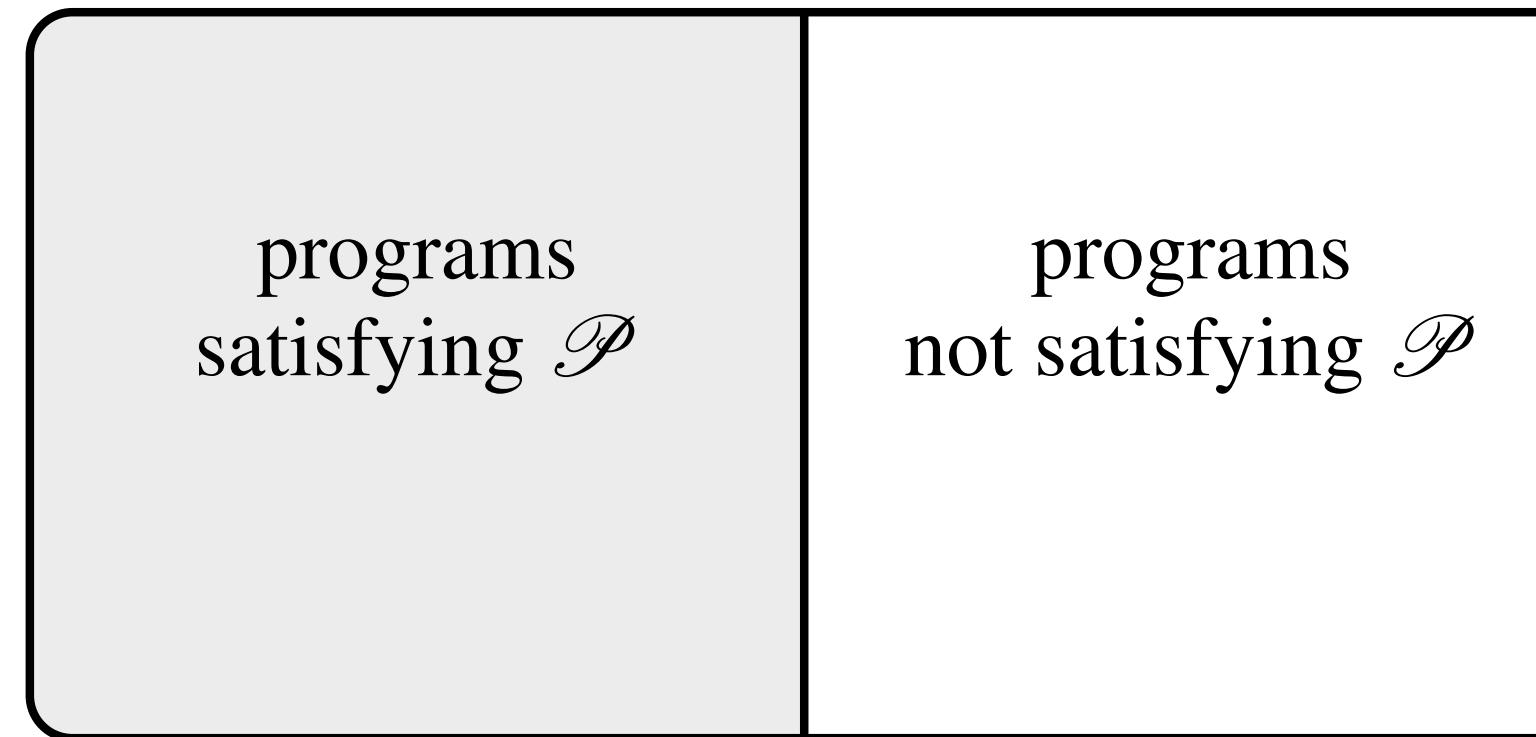
For all program  $p$ ,  $A(p) = \text{true} \iff p \text{ satisfies } \mathcal{P}$

which consists of

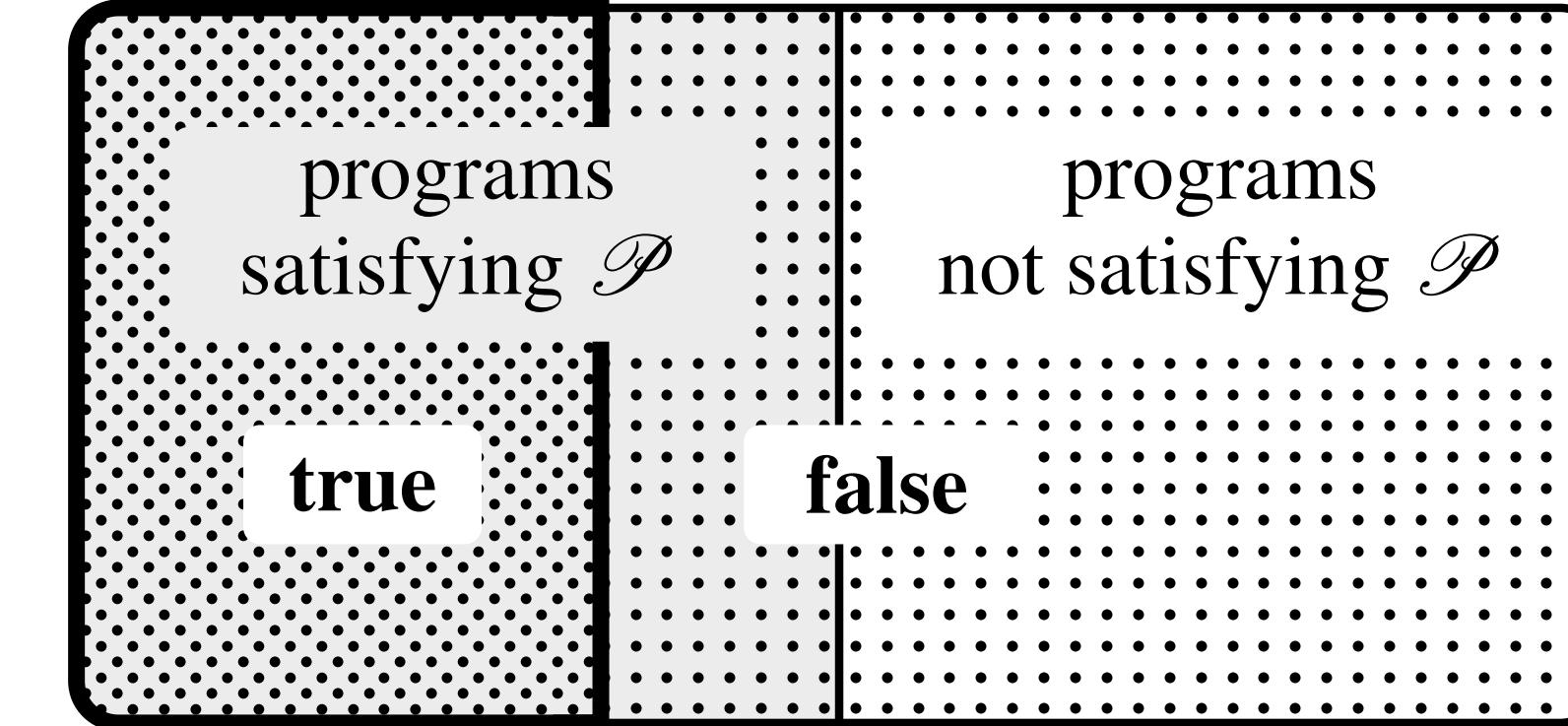
For all program  $p$ ,  $A(p) = \text{true} \Rightarrow p \text{ satisfies } \mathcal{P}$  **(soundness)**

For all program  $p$ ,  $A(p) = \text{true} \Leftarrow p \text{ satisfies } \mathcal{P}$  **(completeness)**

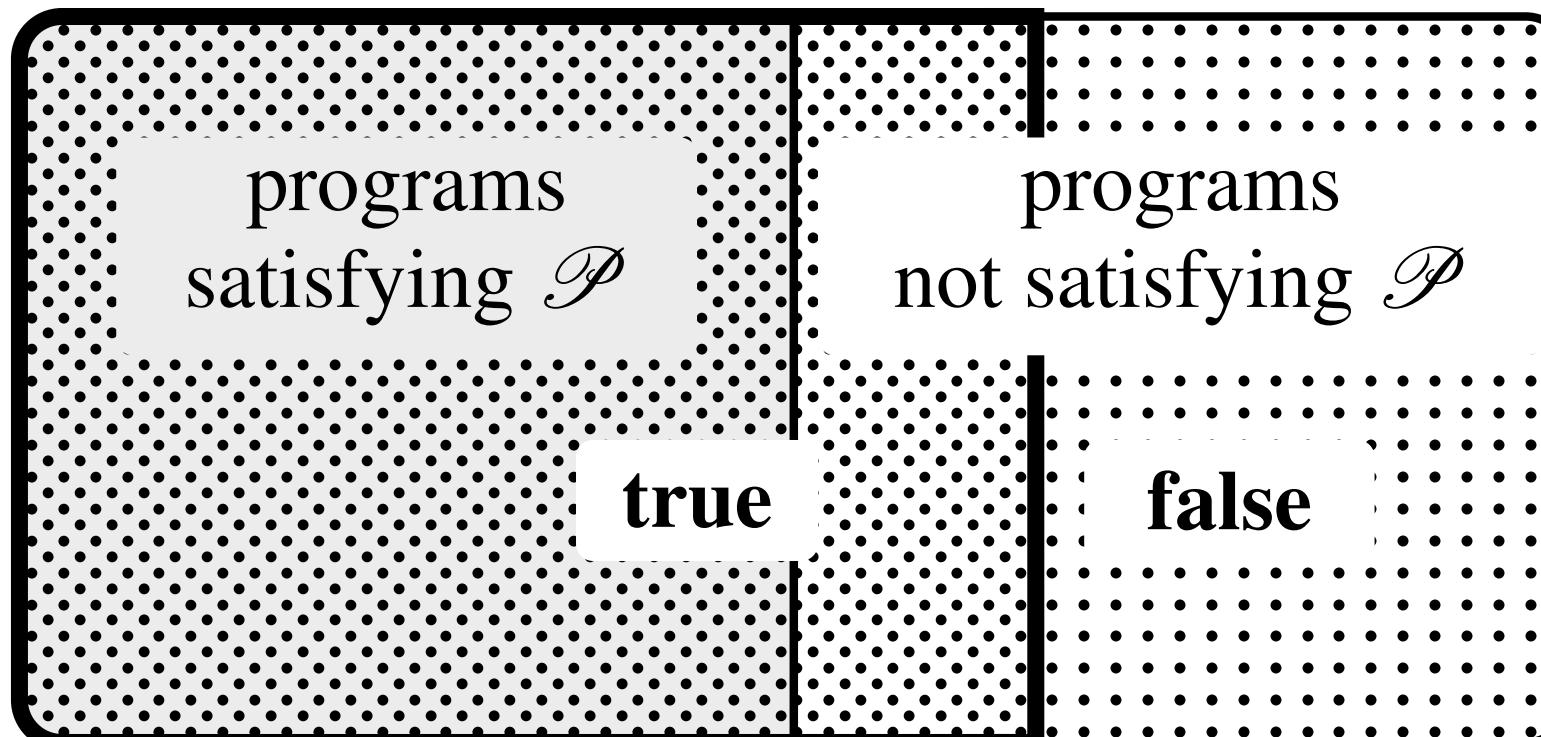
# Soundness and Completeness



(a) Programs



(b) Sound, incomplete analysis



(c) Unsound, complete analysis

- programs that satisfy  $\mathcal{P}$
- programs that do not satisfy  $\mathcal{P}$
- programs for which the analysis returns **true**
- programs for which the analysis returns **false**

(d) Legend

# Testing

- Check a set of **finite executions**
  - e.g., random testing, concolic (**concrete + symbolic**) testing
- In general, **unsound yet complete**
  - Unsound: cannot prove the absence of errors
  - Complete: produce counterexamples (i.e., erroneous inputs)
- Example: Google's oss-fuzz (<https://github.com/google/oss-fuzz>)

# Assisted Proving

- Machine-assisted proof techniques
  - Relying on user-provided proofs or invariants
  - Using proof assistants (e.g., Coq, Isabelle/HOL)
- **Sound and complete** (up to the ability of the proof assistant)
  - require manual effort / expertise
  - Example: CompCert (verified C compiler), seL4 (verified microkernel)

The screenshot shows the Coq proof assistant interface. The top menu bar includes File, Edit, Navigation, Try Tactics, Templates, Queries, Display, Compile, Windows, and Help. The main window has two tabs: Intro.v and Examples.v. The Examples.v tab is active, displaying a proof script for a lemma named nat\_eq\_dec. The script uses tactics like rewrite, reflexivity, induction, and discriminate. It also includes eval compute and definition statements. The right side of the interface shows the state of the proof, with 2 subgoals and 2 hypotheses: n : nat, IHn : forall m : nat, {n = m} + {n > m}, m : nat, Hm : n = m. The bottom status bar indicates Line: 159 Char: 13 Coqide started.

```
File Edit Navigation Try Tactics Templates Queries Display Compile Windows Help
Intro.v Examples.v
rewrite IHn.
reflexivity.
Qed.

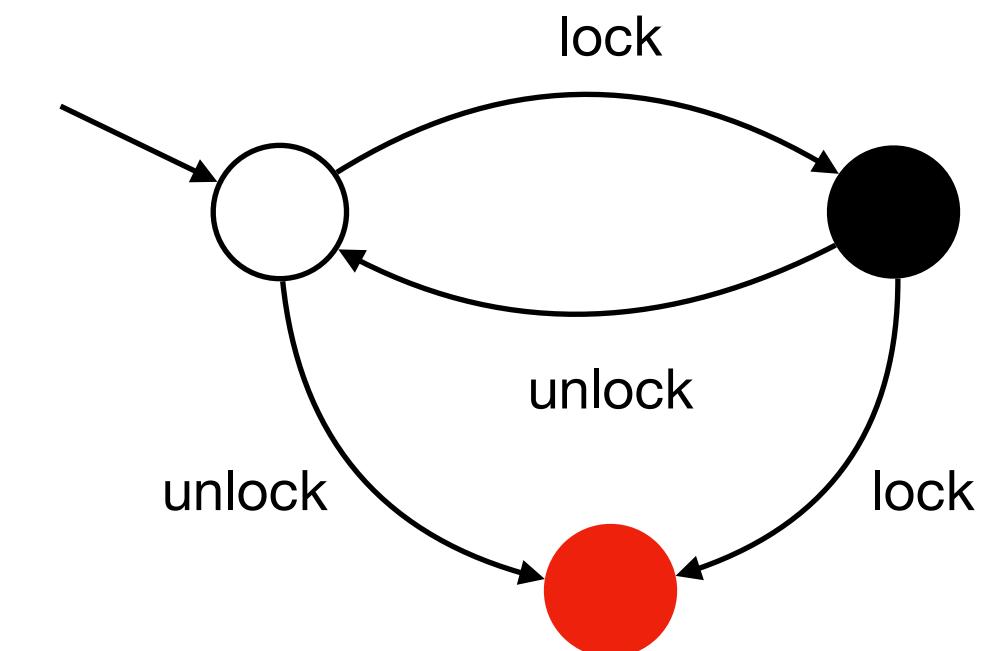
Lemma nat_eq_dec : forall (n m : nat), {n = m} + {n > m}.
induction n.
destruct m as [|m].
left.
reflexivity.
right.
discriminate.
destruct m as [|m].
right; discriminate.
destruct (IHn m) as [Hm|Hm].
left.
rewrite Hm.
reflexivity.
right.
intros Hnm.
apply Hm.
injection Hnm.
tauto.
Defined.

Eval compute in (nat_eq_dec 2 2).
Eval compute in (nat_eq_dec 2 1).

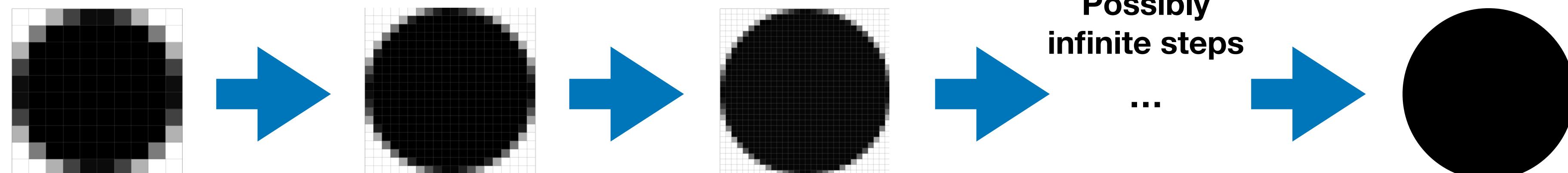
Definition pred (n:nat) : option nat :=
match n with
| 0 => None
| _ => Some n
End.
Ready in Predicate_Logic, proving nat_eq_dec
Line: 159 Char: 13 Coqide started
```

# Model Checking

- Automatic technique to verify if a model satisfies a specification
  - Model of the target program (finite automata)
  - Specification written in a logical formula
  - Verification via an exhaustive search of the state space (graph reachability)
- **Sound and complete with respect to the model**
  - May incur infinite model refinement steps
  - Example: SLAM (MS Windows device driver verifier)



Check: calls to lock and unlock must alternate



# Static Analysis

- **Over-approximate** (not exact) the set of all program behavior
- In general, **sound and automatic, but incomplete**
  - May have spurious results
- Based on a foundational theory : Abstract interpretation
- Variants:
  - under-approximating static analysis: automatic, complete, unsound
  - bug finder: automatic, unsound, incomplete, and heuristics
- Example: type systems, ASTREE, Facebook Infer, Sparrow, etc

# Example

```
1: static char *curfinal = "HDACB  FE";      curfinal: buffer of size 10
2:
3: keysym = read_from_input();                keysym : any integer
4:
5: if ((KeySym)(keysym) >= 0xFF9987)
6: {
7:     unparseputc((char)(keysym - 0xFF91 + 'P'), pty);
8:     key = 1;
9: }
10: else if (keysym >= 0)
11: {
12:     if (keysym < 16)                      keysym: [0, 15]
13:     {
14:         if (read_from_input())
15:         {
16:             if (keysym >= 10) return;       keysym: [0, 9]
17:             curfinal[keysym] = 1;        keysym: [0, 9]
18:         }
19:     else
20:     {
21:         Buffer-overflow          curfinal[keysym] = 2;    size of curfinal: [10, 10]
22:     }
23: }
24: if (keysym < 10)                          keysym: [0, 9]
25: unparseput(curnal[keysym], pty);
26: }
```

# Approximation

- Compute approximated (inaccurate) semantics instead of exact semantics

- Inaccurate  $\neq$  incorrect

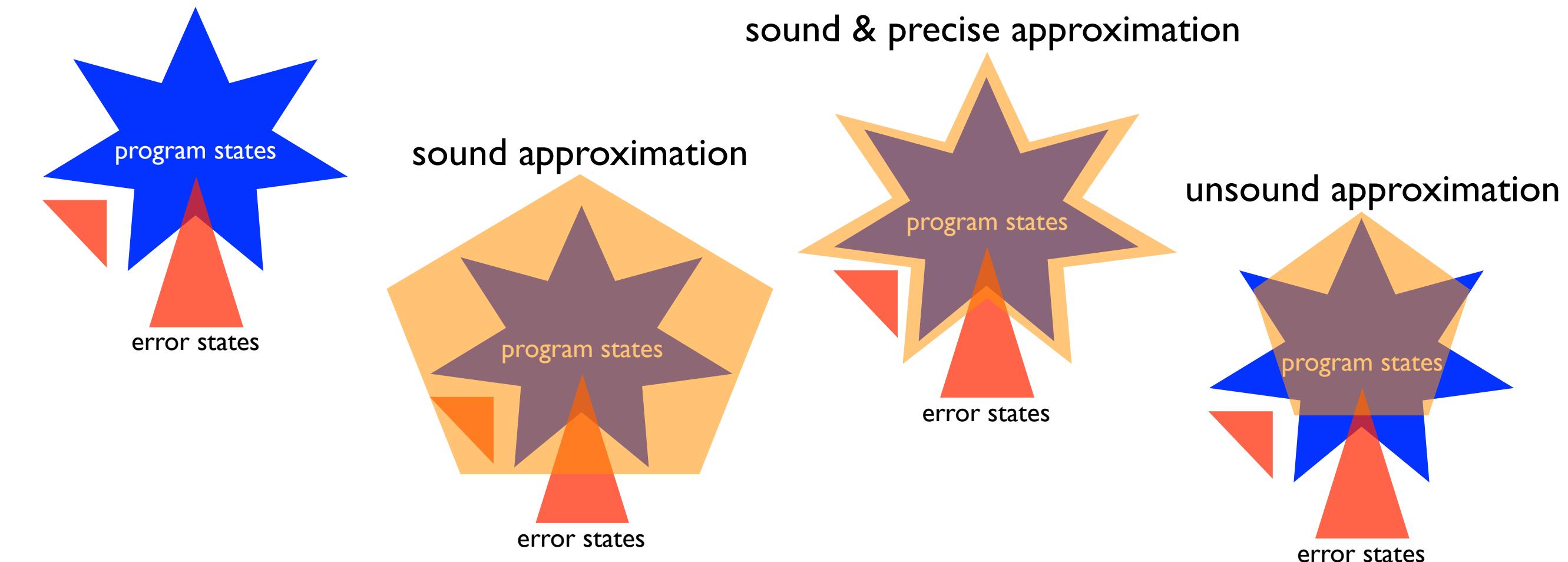
- E.g., reality:  $\{2, 4, 6, 8, \dots\}$

answer 1: “even” (exact)

answer 2: “positive” (conservative)

answer 3: “multiple of 4” (omissive)

answer 4: “odd” (wrong)



- Given a program and property, the analysis answers “Yes”, “No”, or “Don’t know”
- Key point: choosing a right approximation to prove a given target property

# Principle of Static Analysis

- How to design a sound approximation of real executions?
- How to guarantee the termination of static analysis?



## A: Abstract Interpretation

# Summary

- Property: point of interest in a program (safety, liveness, information flow, etc)
- Program analysis: check whether a property is satisfied or not
- Hard limit of program analysis: generally undecidable problem
- Practical solutions

	Automatic	Sound	Complete	Object	When
Testing	Yes	No	Yes	Program	Dynamic
Assisted Proving	No	Yes	Yes/No	Model	Static
Model Checking of finite-state model	Yes	Yes	Yes	Finite Model	Static
Conservative Static Analysis	Yes	Yes	No	Program	Static
Bug Finding	Yes	No	No	Program	Static