

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 | U.S.

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

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

A large red heart outline with liquid dripping down from it, symbolizing the 'Heartbleed' bug.

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

A smartphone lying on a keyboard, with a green digital skull icon on its screen, symbolizing a security breach.

AIRLINE MARSHALL TRANSPORTATION 08:30:2019 07:00 AM

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 keep the 737 MAX grounded until the fall.

A Boeing 737 MAX airplane captured from a low angle, flying through a cloudy sky.

Homeland Security warns that certain heart devices can be hacked

New in Life & Style

Homeroom fifth-graders bond through poetry, art and Steph Curry

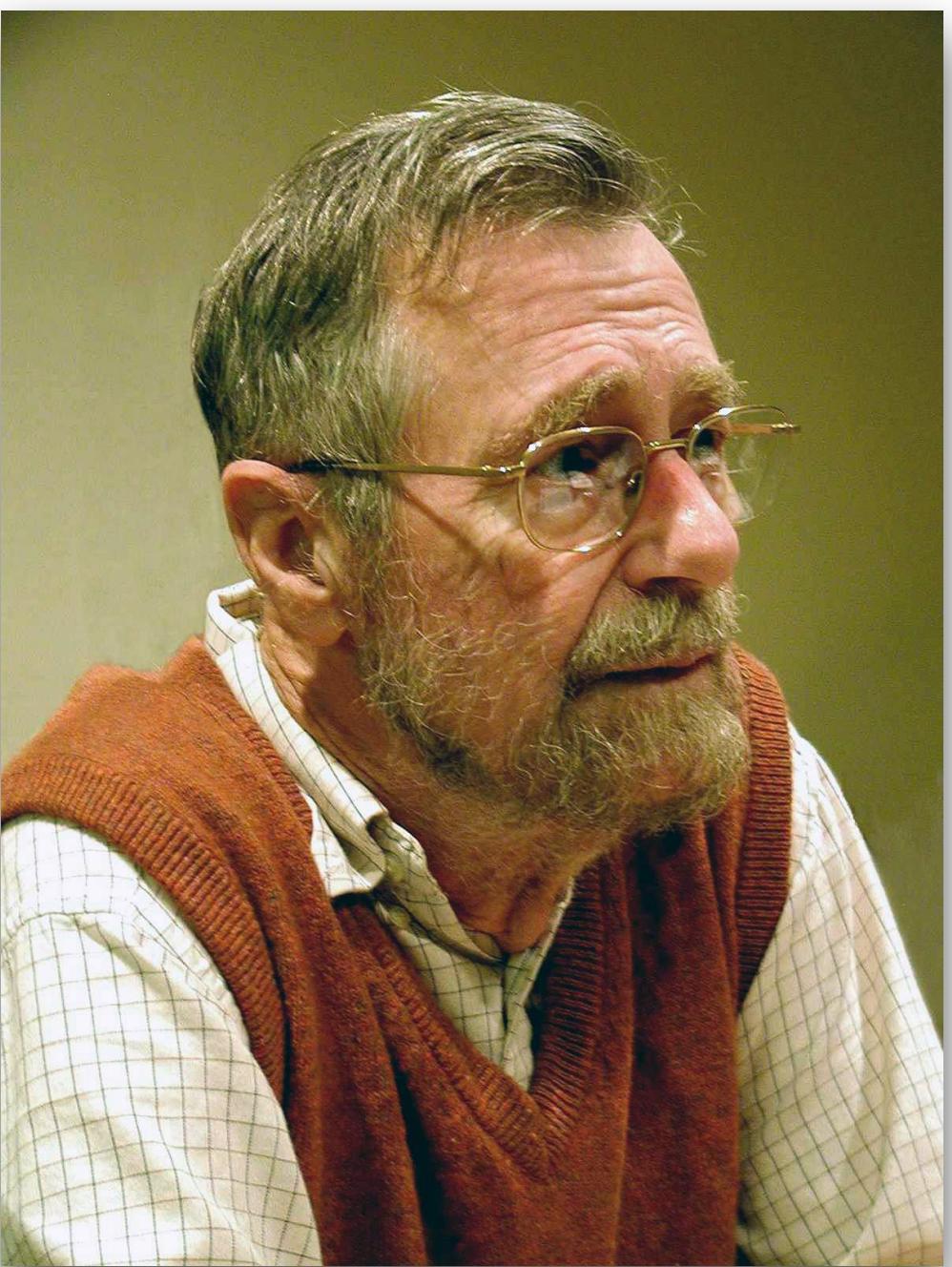
6 ways to celebrate Valentine's Day in Lake Geneva

Six ways to keep your kids healthy during winter

See More

Exterior view of the St. Jude Medical building, featuring a large sign with the company name and a main entrance sign.

Towards Error-free SW



***“Program testing can be used to show the presence of bugs,
but never to show their absence!”***

- Edsger W. Dijkstra, 1970

Cost of Software Quality Assurance



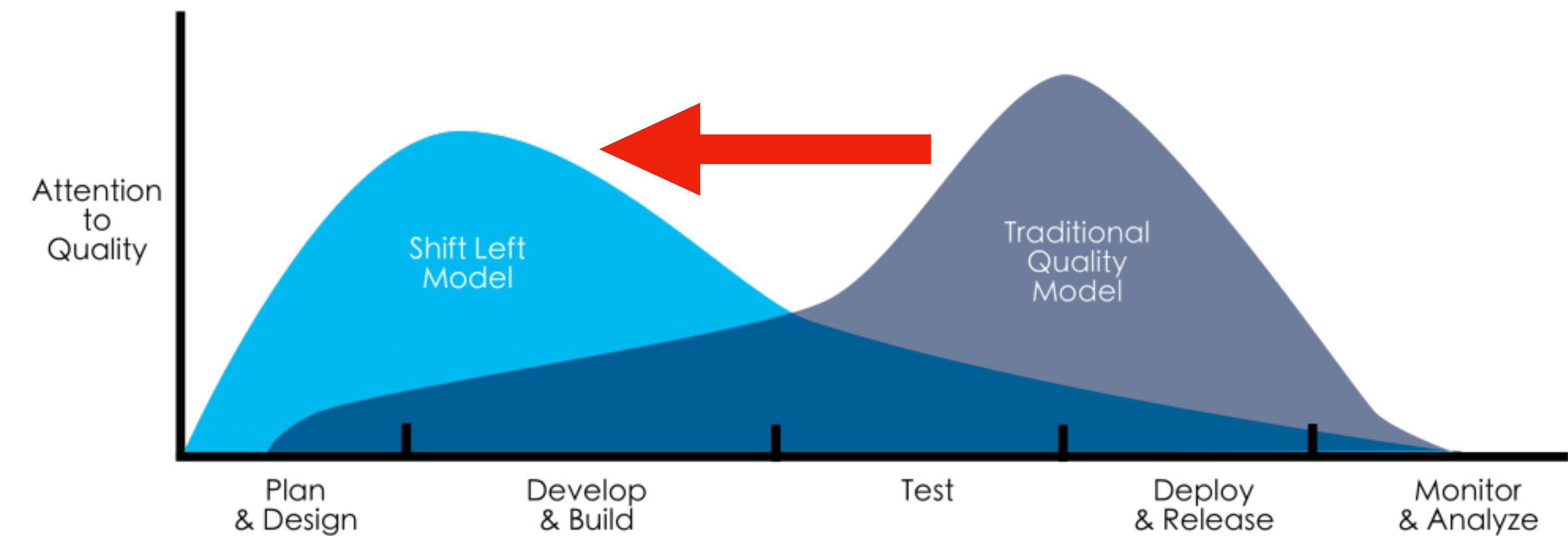
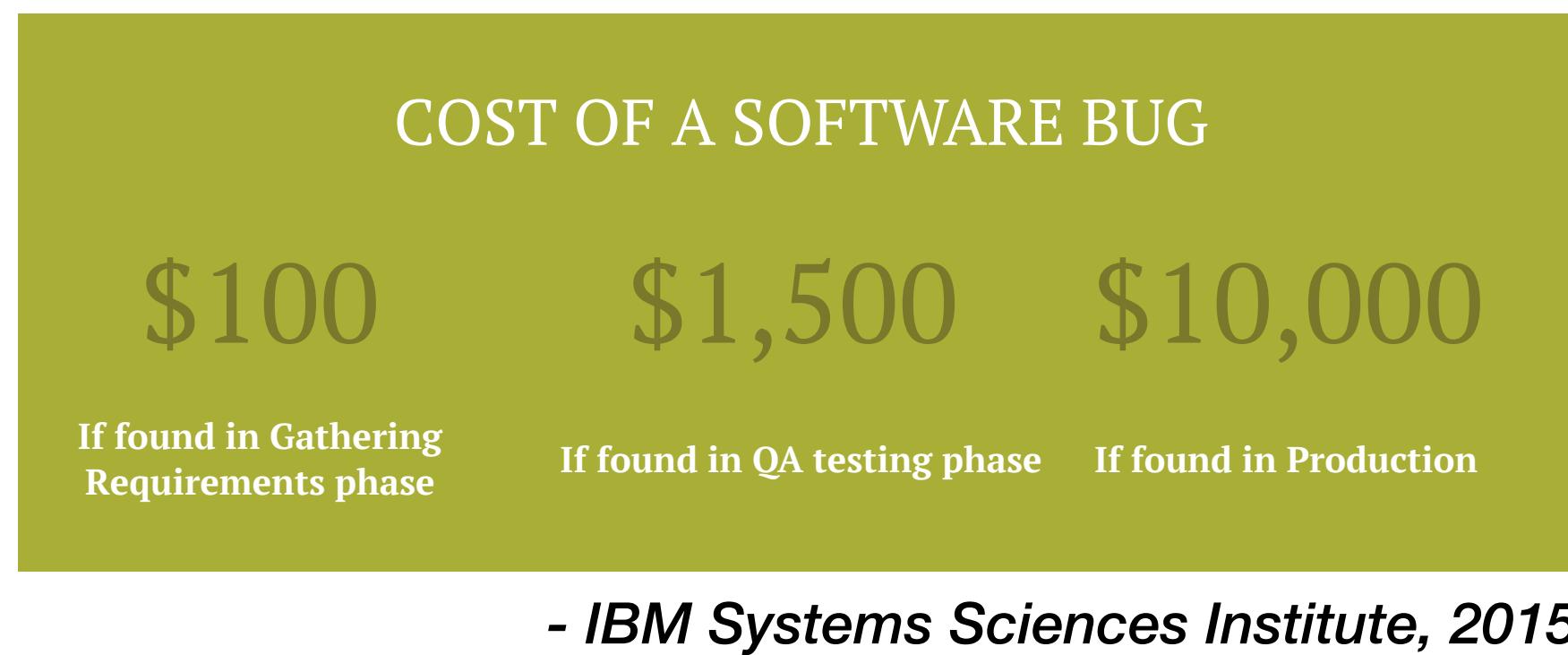
*“We have as **many testers** as we have developers.
And testers spend **all their time testing**, and developers spend
half their time testing. We’re more of a testing, a quality software
organization than we’re a software organization”*
- Bill Gates, 2002

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

A: Program analysis

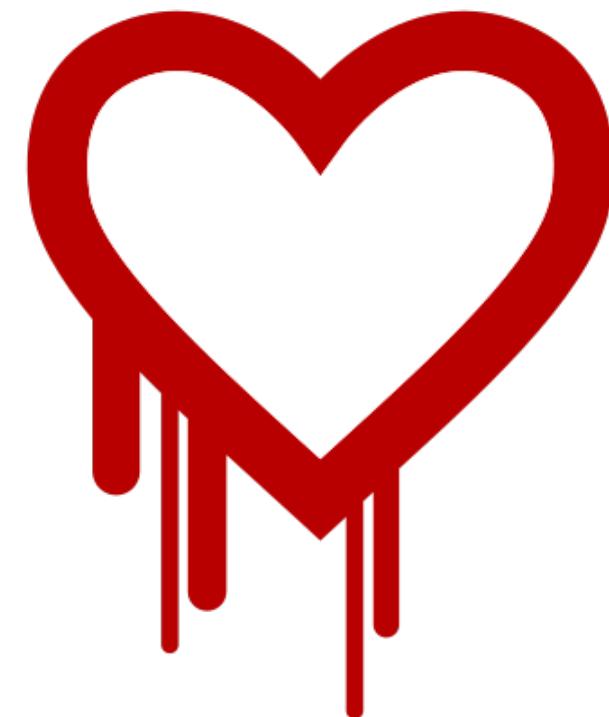
Discovering Software Errors

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



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



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



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

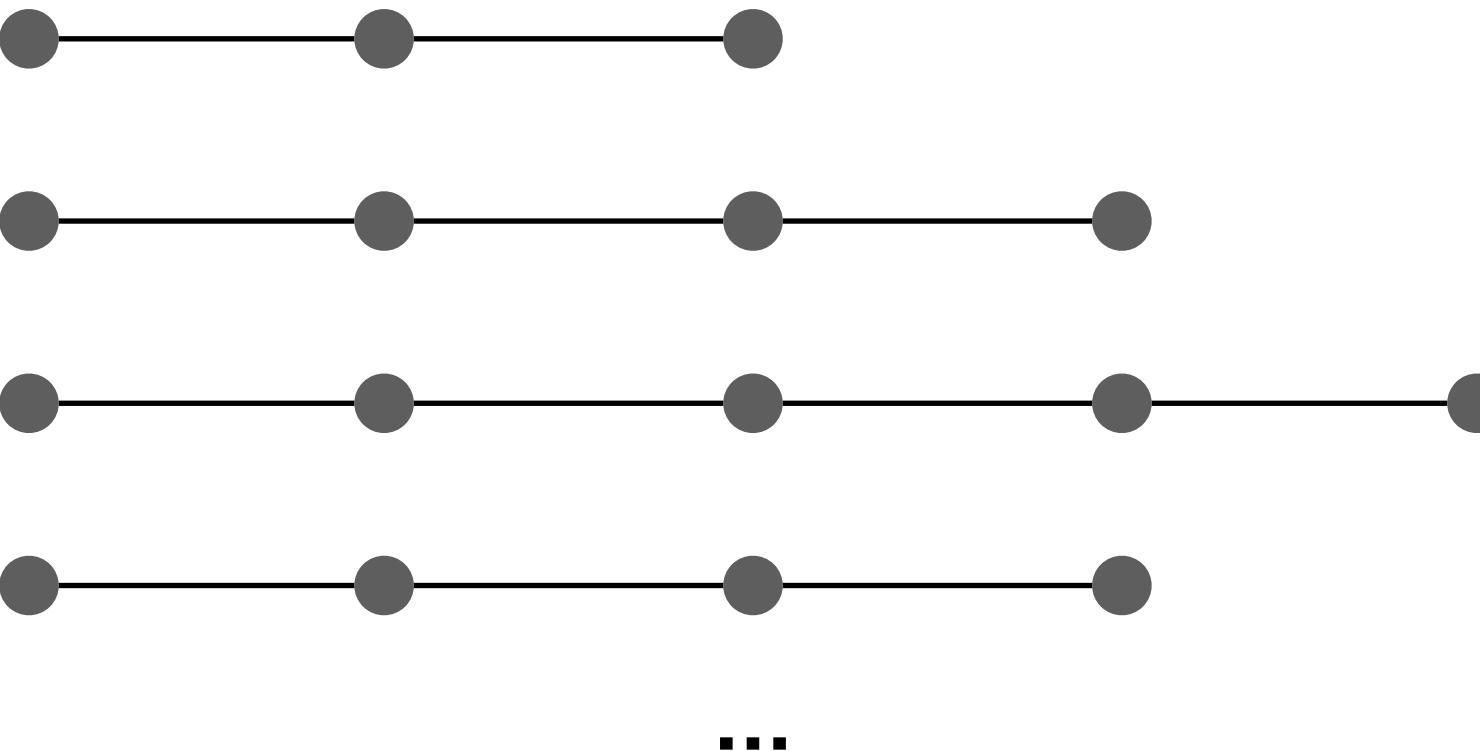
Properties

- Points of interest in programs
 - for verification, bug detection, optimization, understanding, etc
 - E.g., “ $p == \text{NULL?}$ ”, “ $\text{idx} < \text{size?}$ ”, “ 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

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

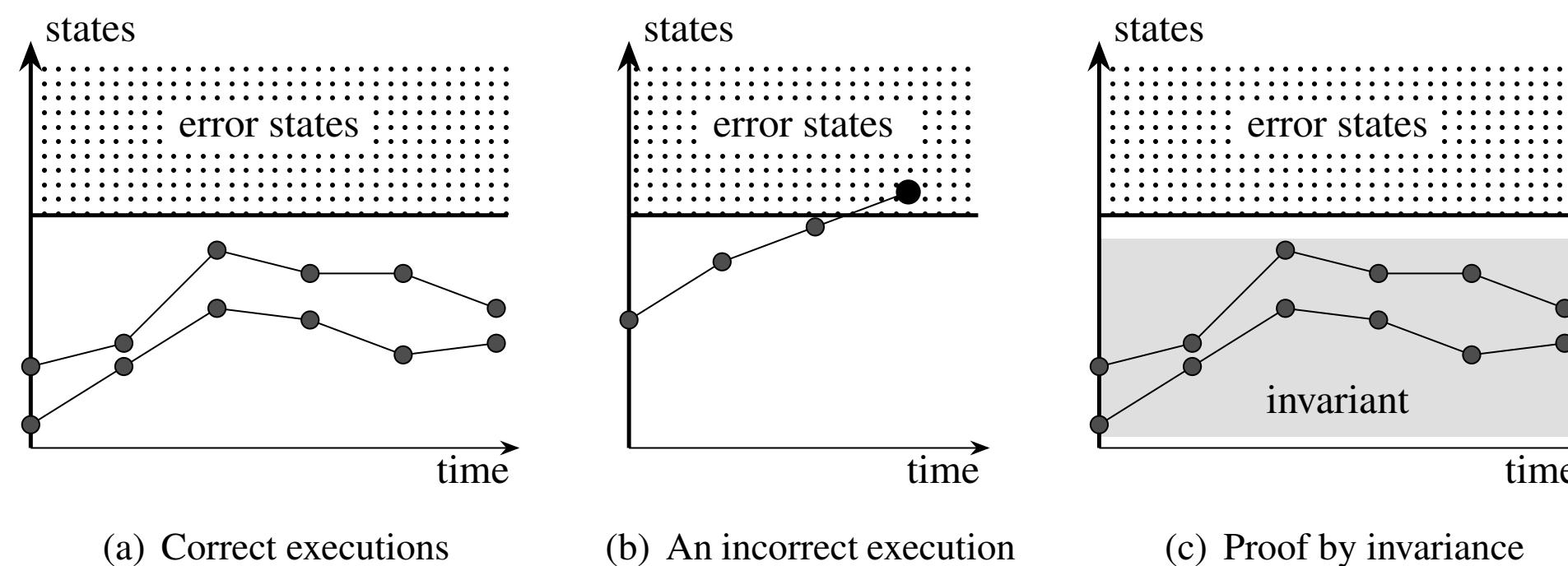


Trace Properties

- 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

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

Loop invariant 1: “x is an integer”

Loop invariant 2: “ $x \geq 0$ ”

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

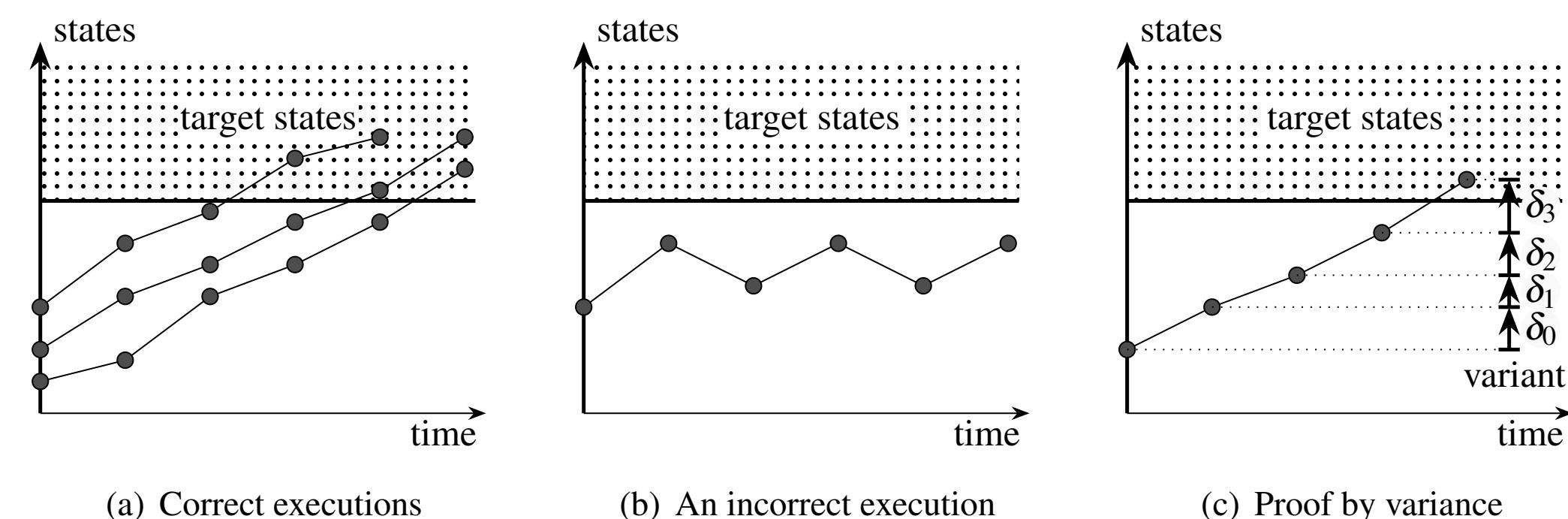
Example: Division-by-Zero

```
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

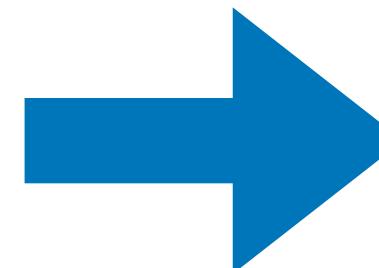
```
x = pos_int();  
while (x > 0) {  
    x = x - 1;  
}
```

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

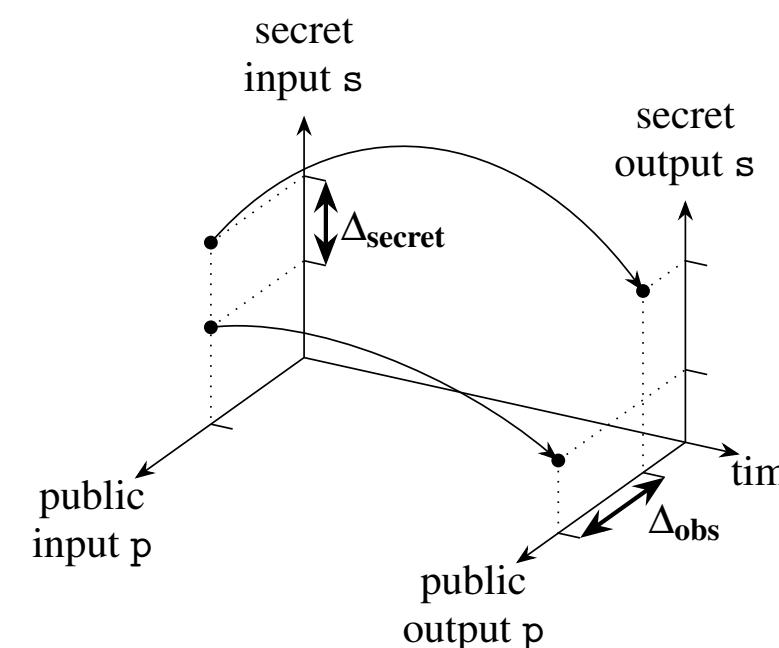
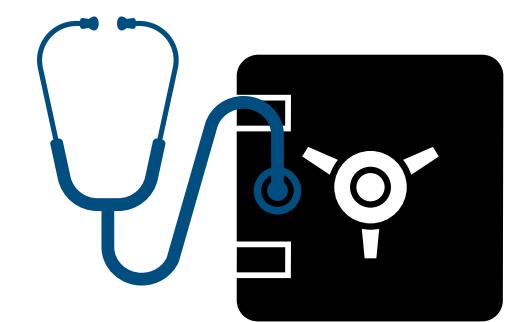
Example

- Correctness of a sorting algorithm as trace property

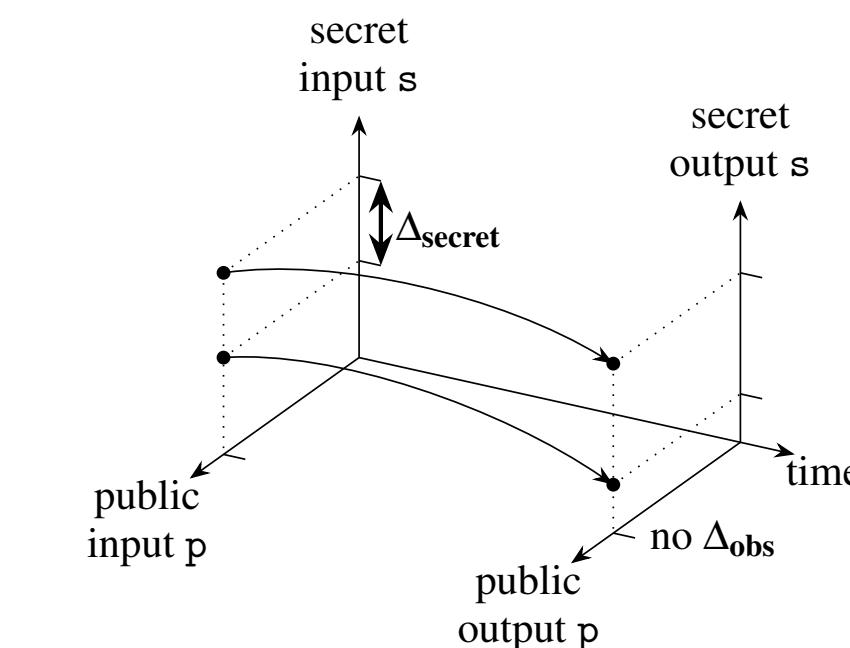
Property	Safety or Liveness?	State?
Should not fail with a run-time error		
Should terminate		
Should return a sorted array (if terminated)		
Should return an array with the same elements and multiplicity (if terminated)		

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}

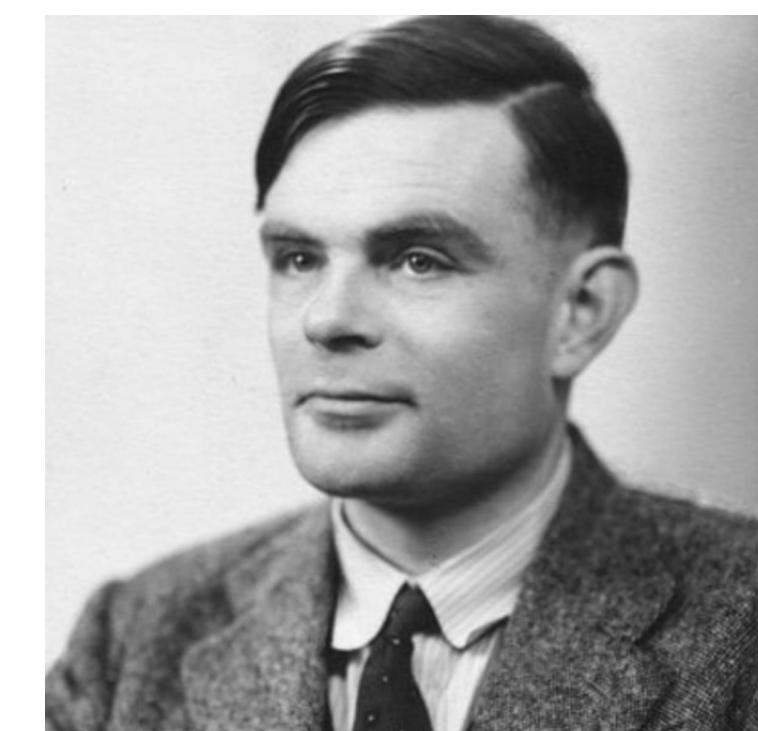
A Hard Limit: Undecidability

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,

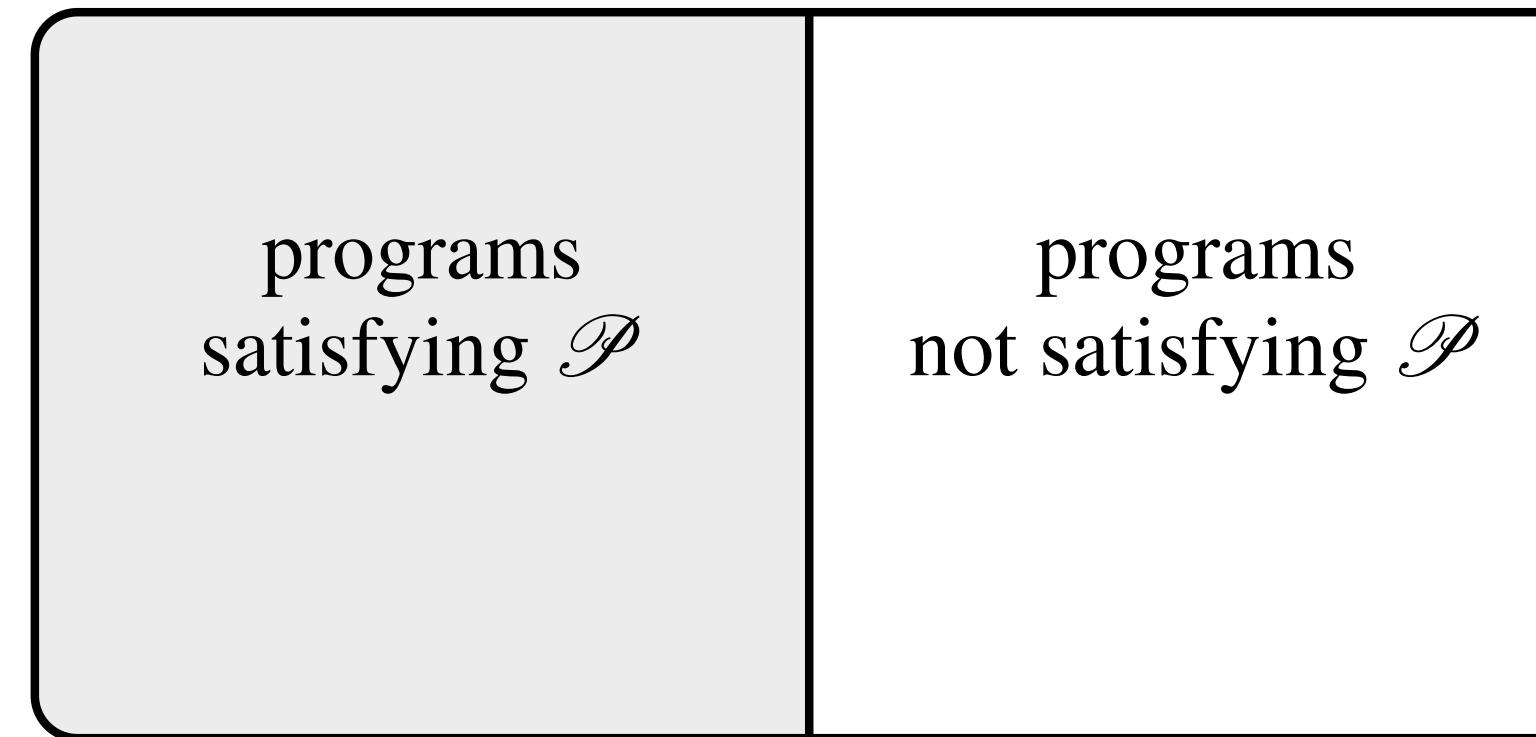
For all program p , $A(p) = \text{true} \iff p \text{ satisfies } \mathcal{P}$

which consists of

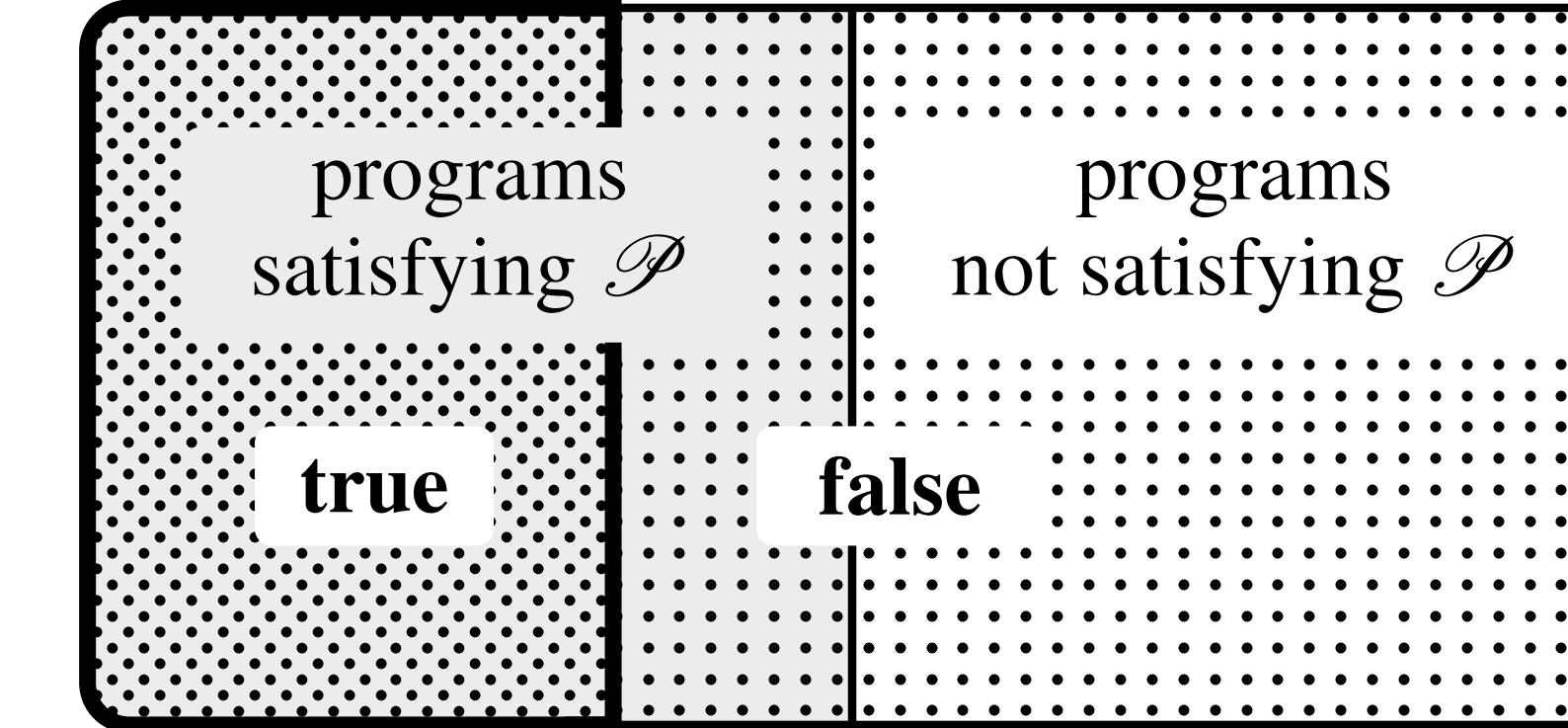
For all program p , $A(p) = \text{true} \Rightarrow p \text{ satisfies } \mathcal{P}$ **(soundness)**

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

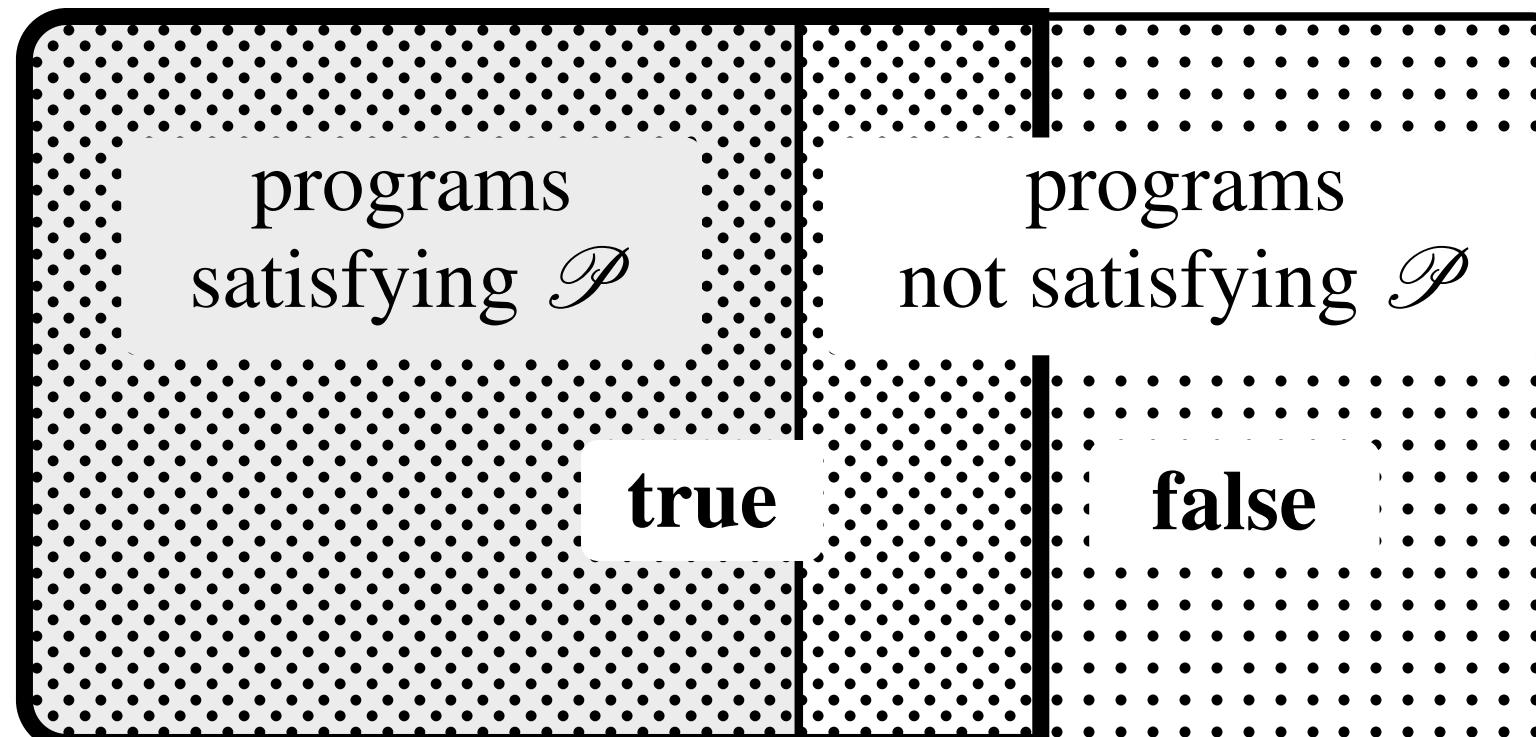
Soundness and Completeness



(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)

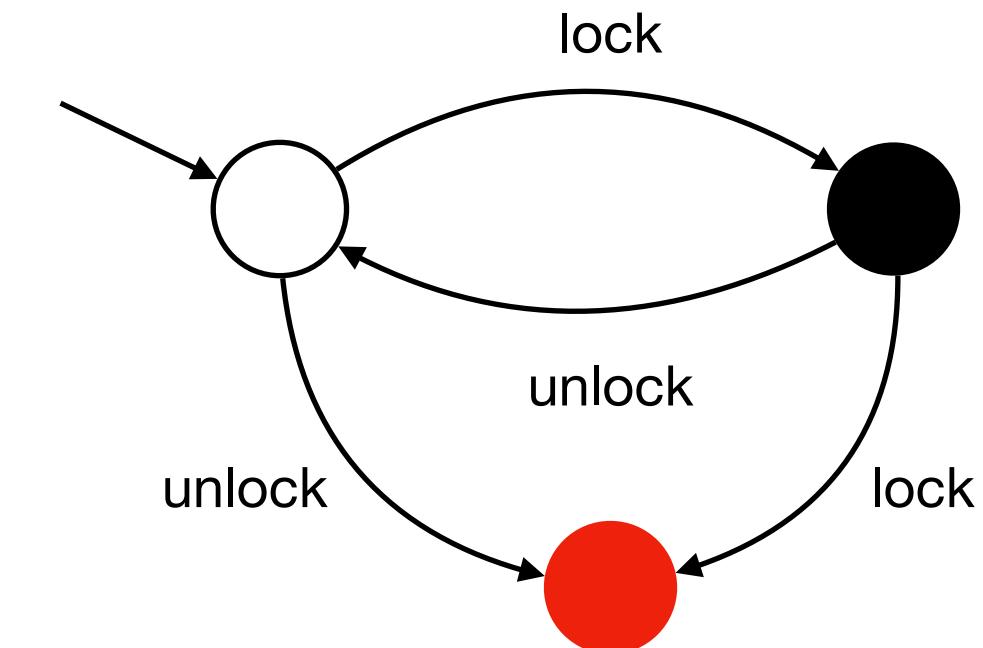
The screenshot shows the Coq proof assistant interface. The top menu bar includes File, Edit, Navigation, Try Tactics, Templates, Queries, Display, Compile, Windows, and Help. The main window has two tabs: Intro.v and Examples.v. The Examples.v tab is active, displaying a proof script for a lemma named nat_eq_dec. The script uses tactics like rewrite, reflexivity, induction, destruct, discriminate, and eval compute. It also defines a predicate pred. The right pane shows the state of the proof, with two subgoals listed:

- (1/2) subgoals n : nat
IHn : forall m : nat, {n = m} + {n > m}
m : nat
Hm : n = m
- (2/2) S m = S m
{S n = S m} + {S n > S m}

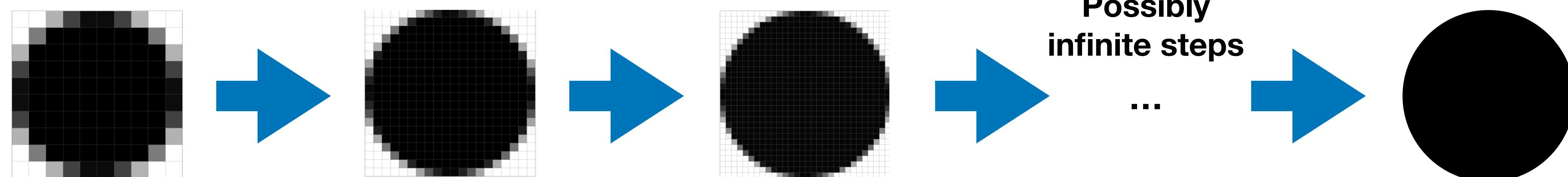
At the bottom of the interface, status information includes "Ready in Predicate_Logic, proving nat_eq_dec", "Line: 159 Char: 13", and "Coqide started".

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:         keysym: [0, 15]
23:     }
24:     if (keysym < 10)                   keysym: [0, 9]
25:     unparseput(curninal[keysym], pty);
26: }
```

Approximation

- Compute approximated (inaccurate) semantics instead of exact semantics

- Inaccurate \neq incorrect

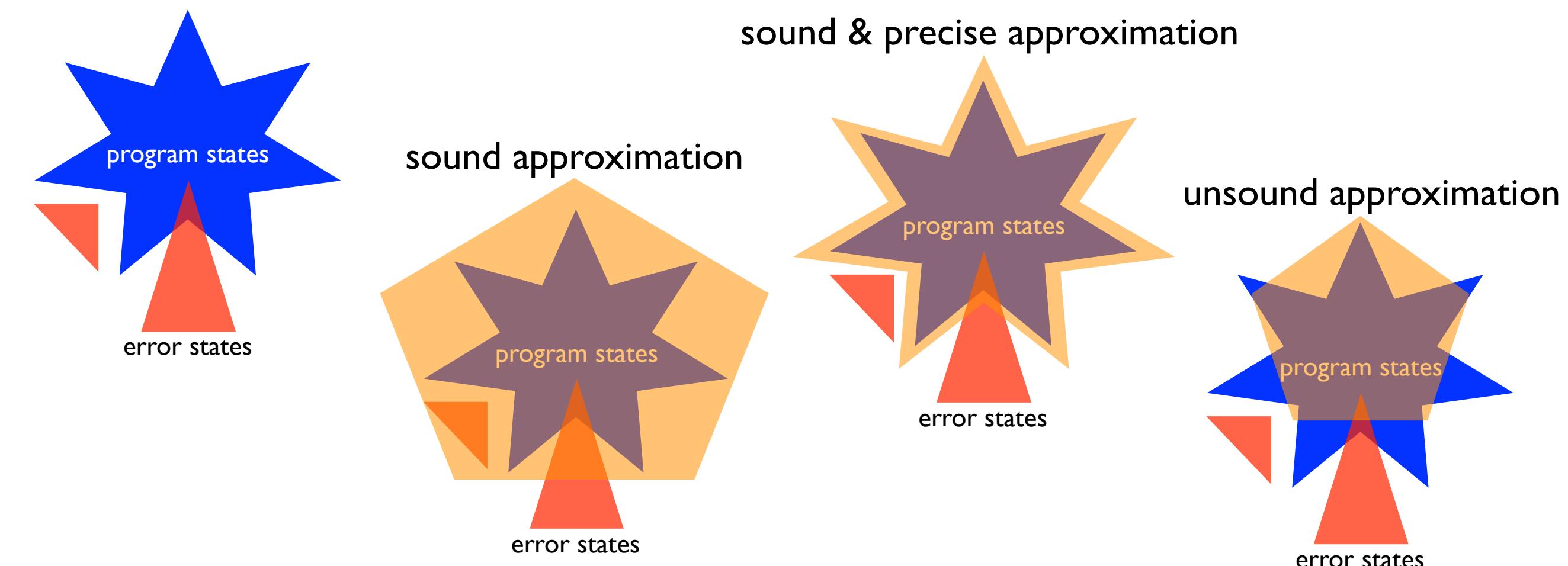
- E.g., reality: $\{2, 4, 6, 8, \dots\}$

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