# Program Reasoning

## 3. Concepts in Program Verification

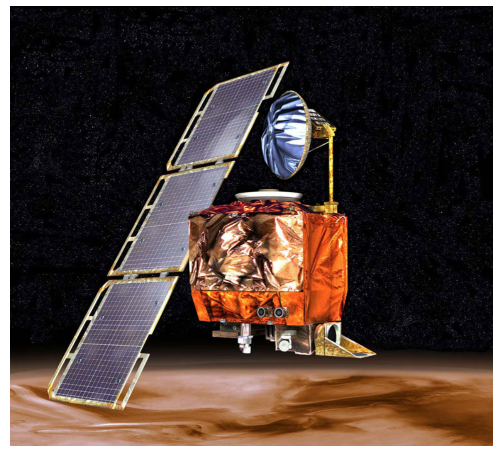
Kihong Heo

KAIST

# 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

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

(Image credit: Shutterstock.com)

AARIAN MARSHALL TRANSPORTATION 06.30.2019 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 will keep the 737 MAX grounded until the fall.

Homeland Security warns that certain heart devices can be hacked



Glen Stubbe / AP

By Tribune news services . Contact Reporter
Associated Press

JANUARY 11, 2017, 8:56 AM | WASHINGTON

New in Life & Style

Interfaith 4th-graders bond through poetry, art and Steph Curry
2:03 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

# 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**

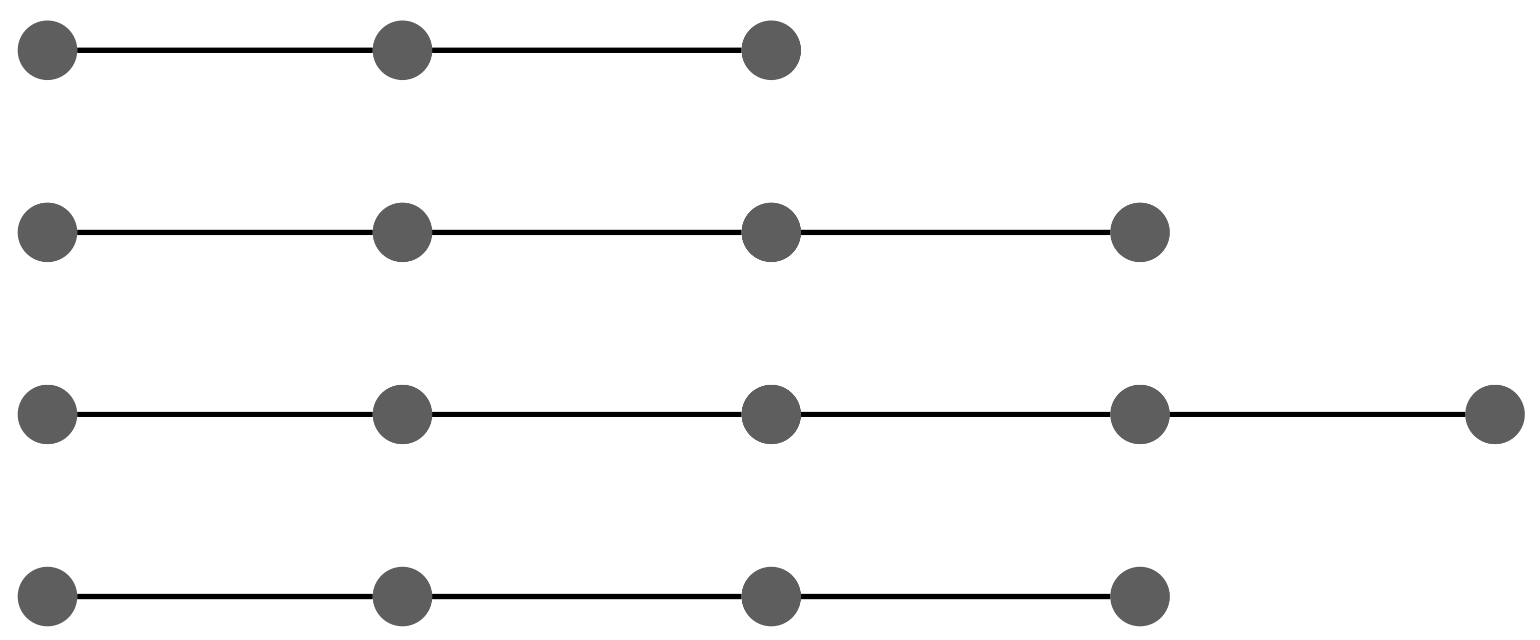
# Properties

- Points of interest in programs
  - for verification, bug detection, optimization, understanding, etc
  - E.g.,“p == NULL?”, “idx < 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
- $[\![P]\!]$ : a set of all possible execution traces

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



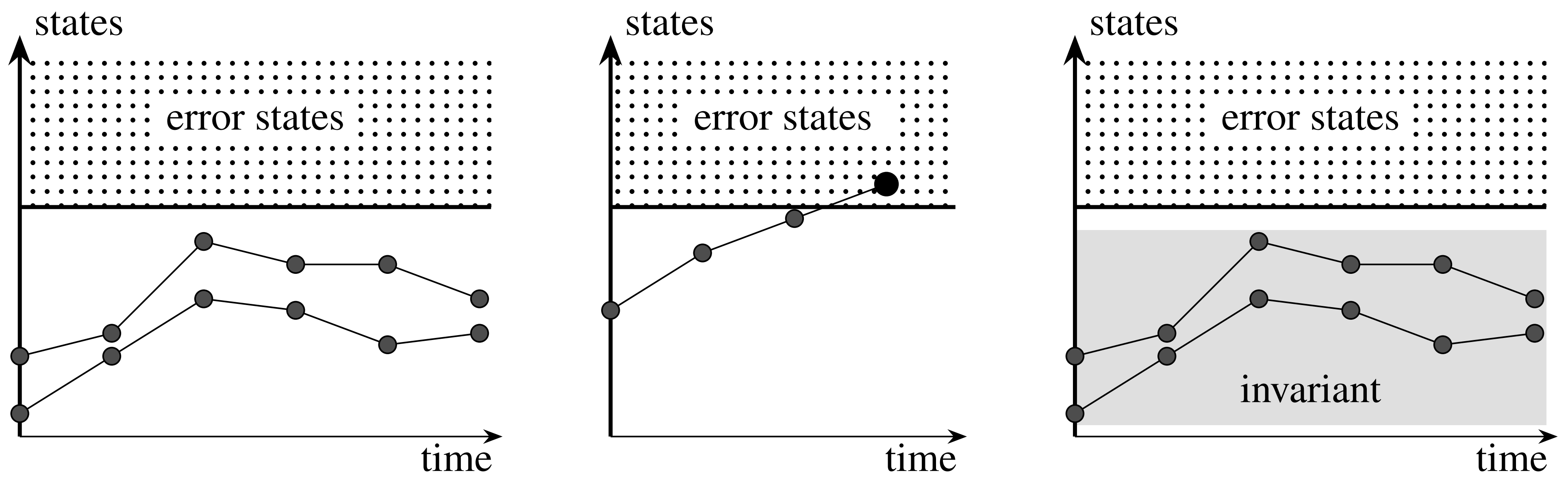
...

# Trace Properties

- A semantic property $\mathscr{P}$ that can be defined by a **set of execution traces** that satisfies $\mathscr{P}$
  - Ex1: “all traces that satisfies x != 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 $\mathscr{P}$ iff $[\![P]\!] \subseteq T_{\mathscr{P}}$
- 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




(a) Correct executions (b) An incorrect execution (c) Proof by invariance

# 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 >= 0”

Loop invariant 3: “0 <= x <= 10”

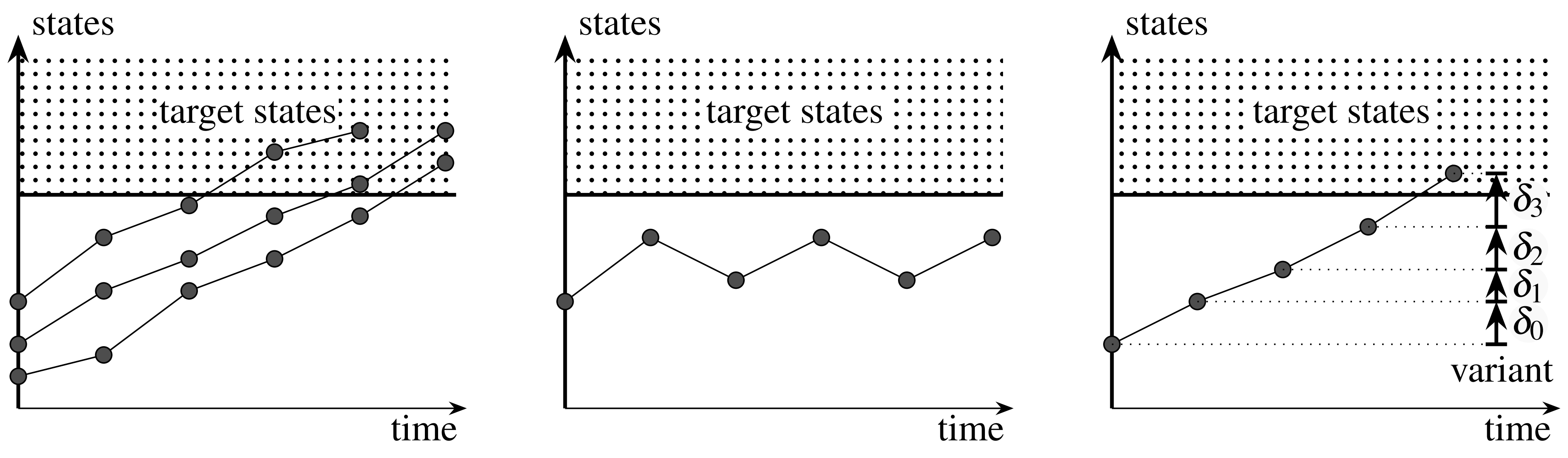
# Example: Division-by-Zero

```c
int main(){
  int x = input();
  x = 2 * x - 1;
  while (x > 0) {
    x = x - 2;
  }
  assert(x != 0);
  return 10 / x;
}
```

```c
int main(){
  int x = input();
  x = 2 * x;
  while (x > 0) {
    x = x - 2;
  }
  assert(x != 0);
  return 10 / x;
}
```

# 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




(a) Correct executions (b) An incorrect execution (c) Proof by variance

# 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 c that stores the value of "step counter"
  - Initially, c is equal to zero
  - Each program execution step increments c by one

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

```
// 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?
```

c <= 3n + 2

**0 <= 3n + 2 - c ∧ 3n + 2 - c is strictly decreasing ⇒ termination**

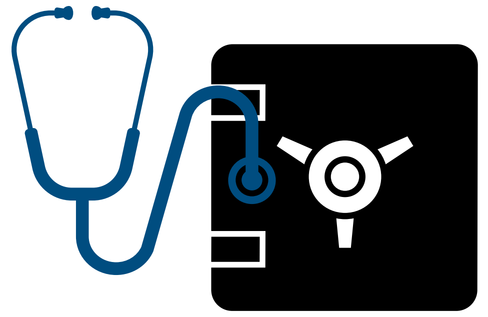
# Example

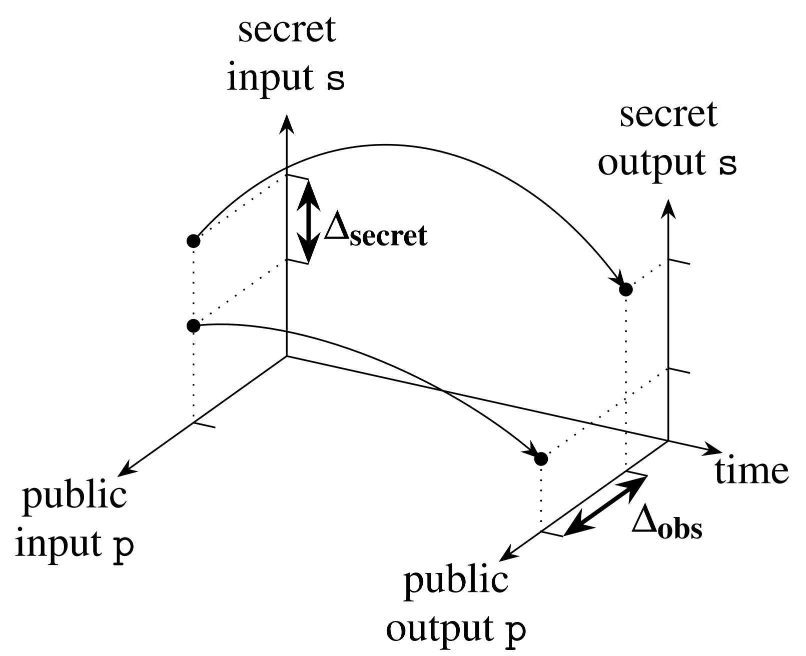
- Correctness of a sorting algorithm as trace property

| 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) | | |

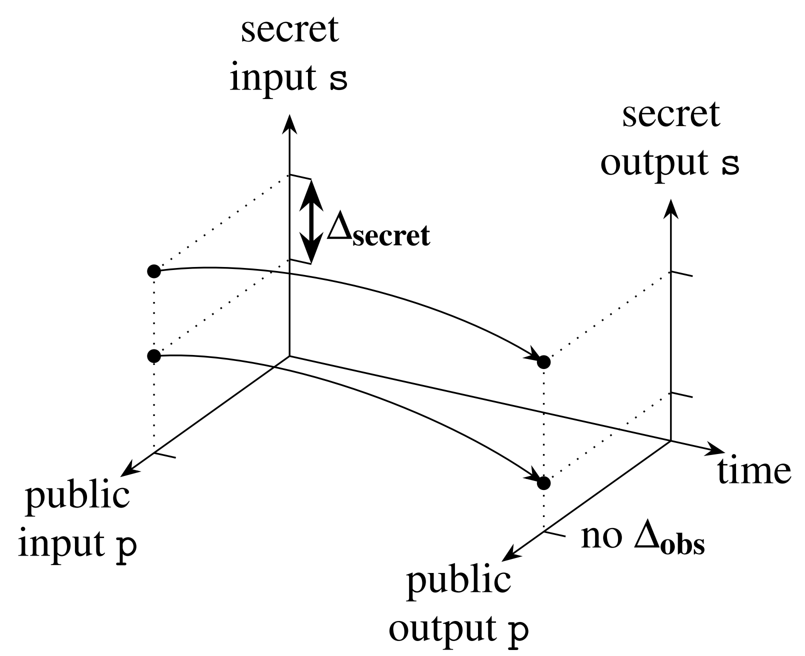
# Information Flow Properties

- 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} |

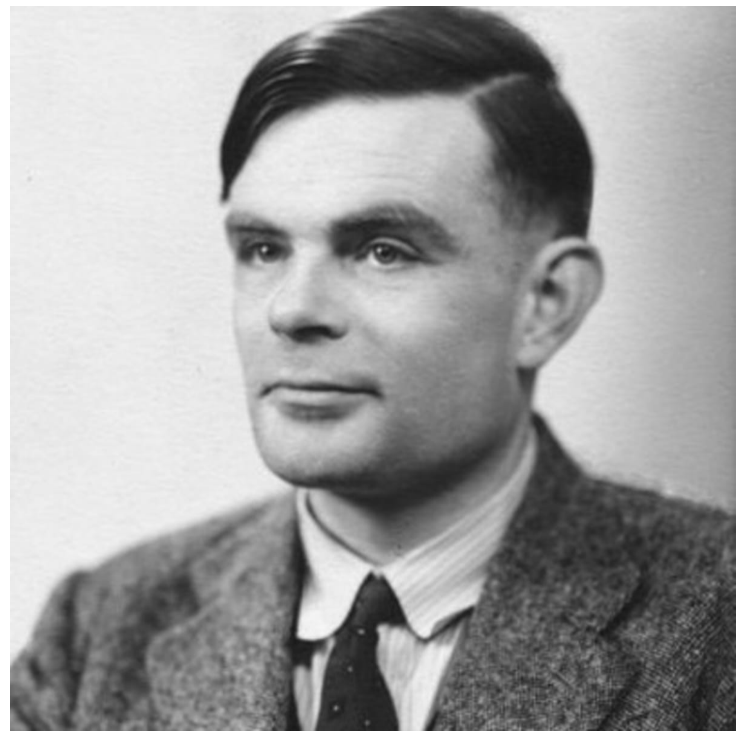
# A Hard Limit: Undecidability

**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 $F(x) = \text{if } H(x, x) \text{ then } \texttt{loop}() \text{ else } \texttt{halt}()$
- 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 $\mathscr{P}$, and an analysis tool $\mathbf{A}$

- If $\mathbf{A}$ were perfectly accurate,

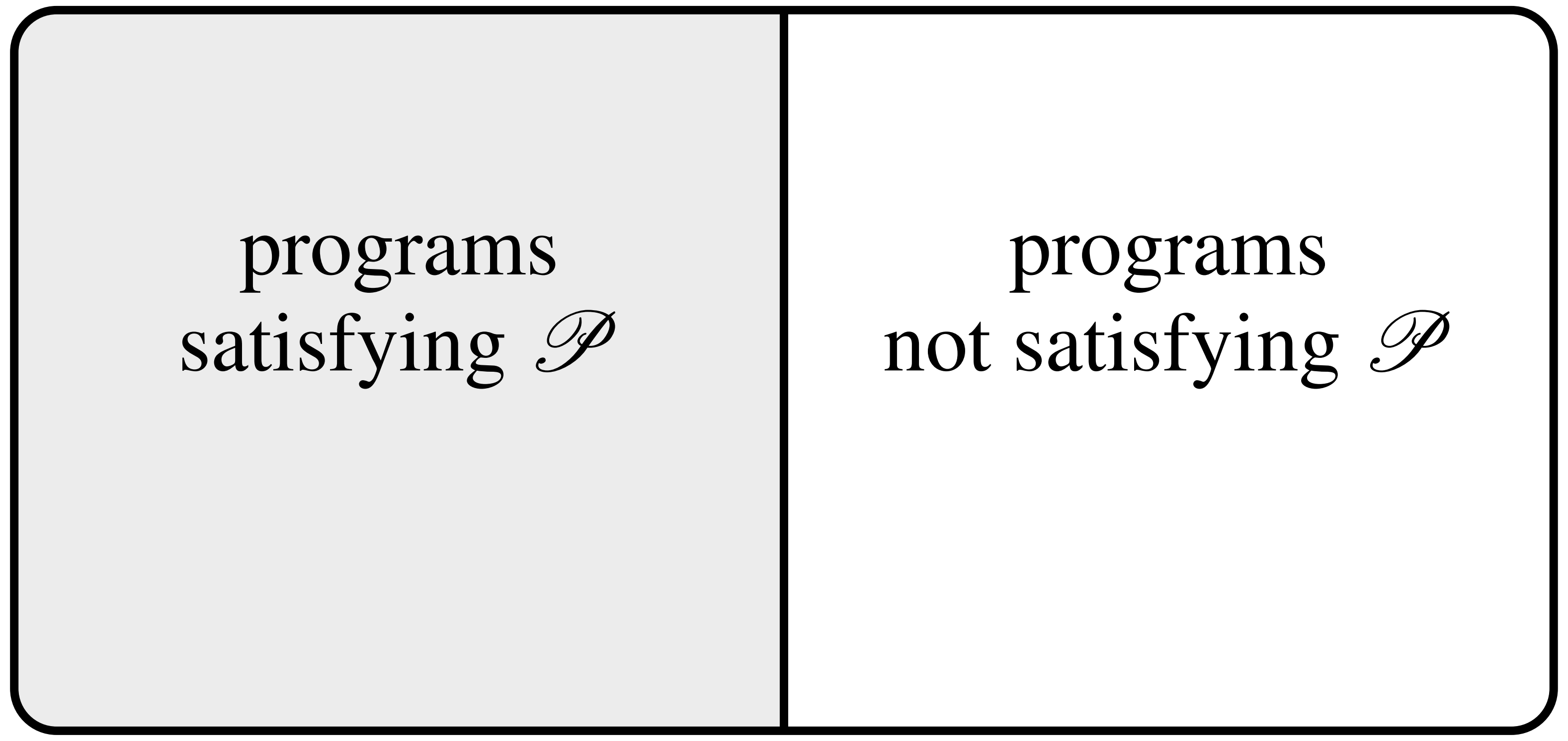
$$\text{For all program p, } \mathbf{A}(\text{p}) = \mathbf{true} \iff \text{p satisfies } \mathscr{P}$$

which consists of

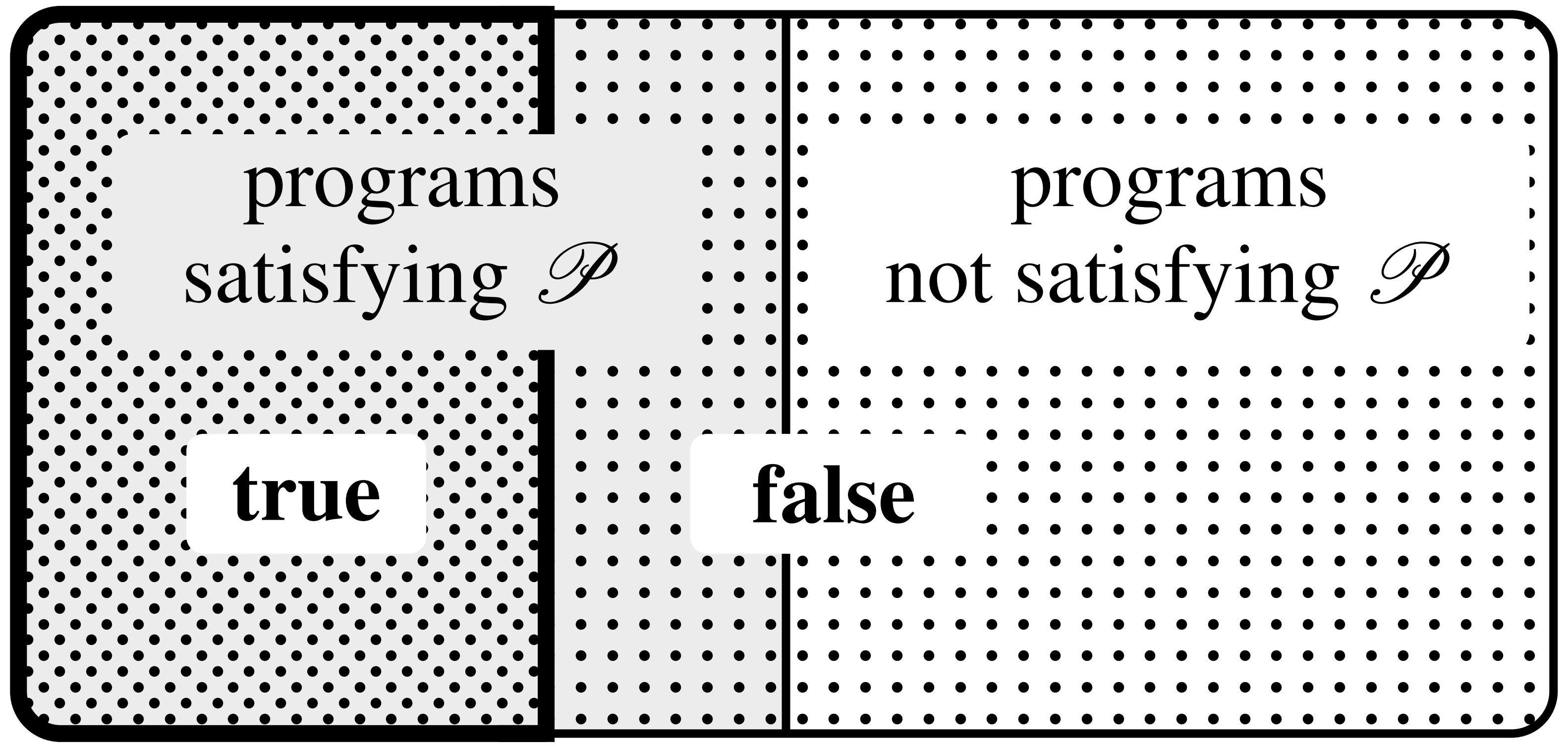
$$\text{For all program p, } \mathbf{A}(\text{p}) = \mathbf{true} \Rightarrow \text{p satisfies } \mathscr{P} \qquad \textbf{(soundness)}$$

$$\text{For all program p, } \mathbf{A}(\text{p}) = \mathbf{true} \Leftarrow \text{p satisfies } \mathscr{P} \qquad \textbf{(completeness)}$$

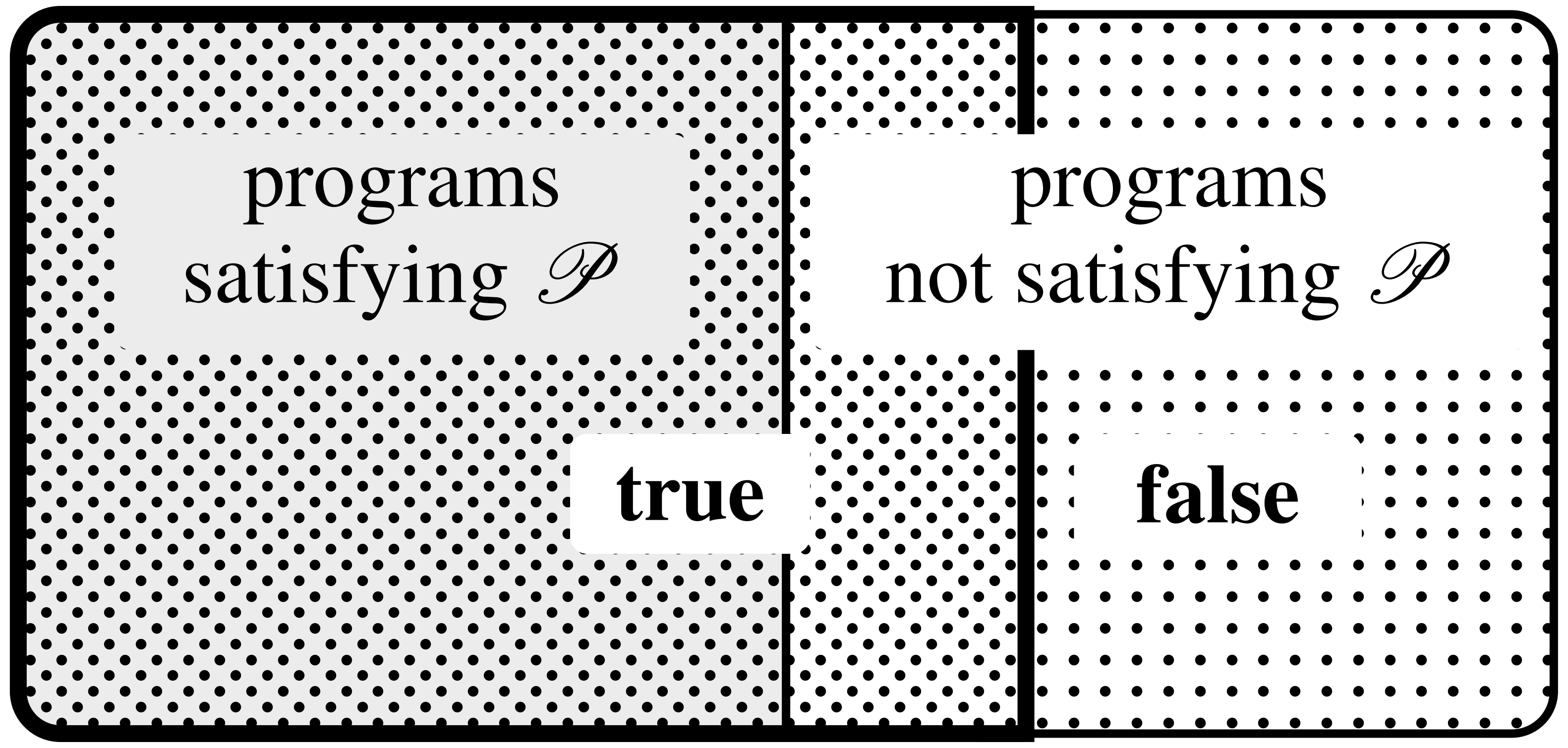
# Soundness and Completeness



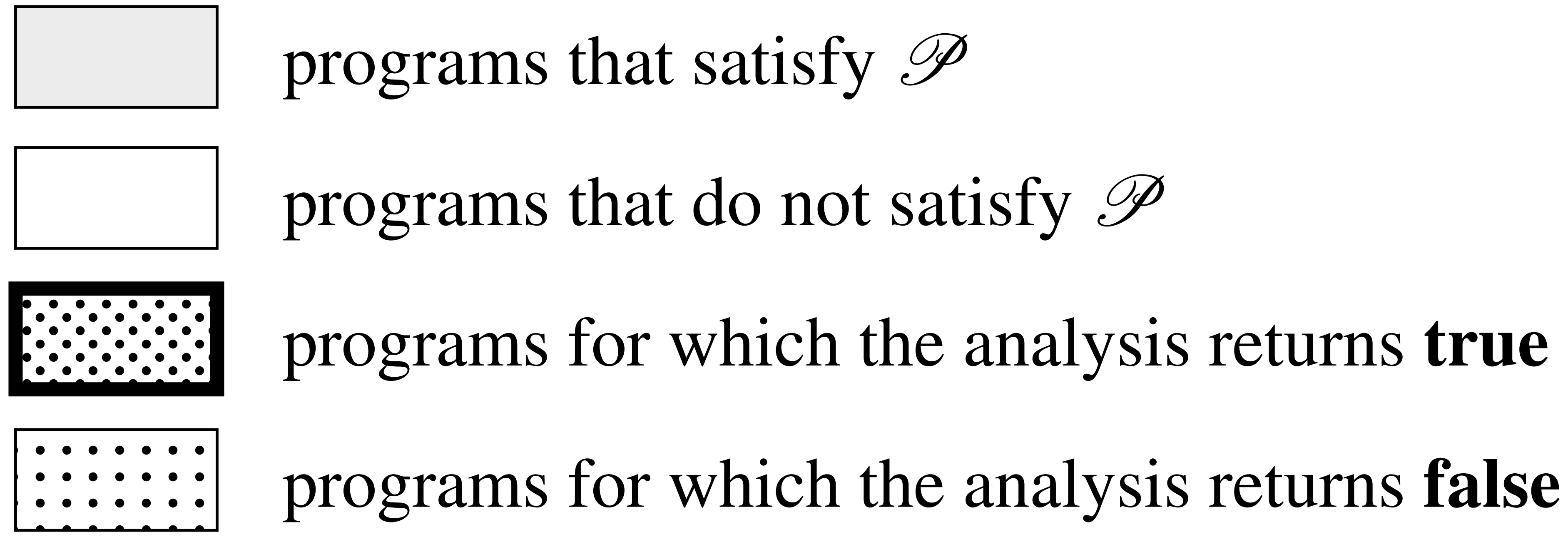

(a) Programs




(b) Sound, incomplete analysis




(c) Unsound, complete analysis




(d) Legend

# Program Verification

- Prove a given program satisfies the target properties
    - Loop invariants provided by the user or another program analyzer
- **Sound and complete** if a “good” invariant is provided
- **Sound and incomplete** if an imprecise invariant is provided
- **Sound, complete, yet non-terminating** if the invariant generation does not terminate
- How to describe the target property (specification)?
- How to prove the target property (specification)?

**A: Program Logic**

# Summary

- Property: point of interest in a program (safety, liveness, information flow, etc)
- Program verification: check whether a property is satisfied or not
- Hard limit of program analysis: generally undecidable problem
- Practical solutions:
    - **Manual** rather than **automatic**
    - **Possibly nonterminating** rather than **terminating**
    - **Approximate** rather than **exact**