Advanced Software Security

8. Introduction to Program Verification

Kihong Heo

KAIST

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

ACM - Association for Computing Machinery
9분 ·

#ACMTuringAward recipient Edsger Wybe Dijkstra was born on this day in 1930. Dijkstra received the 1972 ACM A.M. Turing Award for fundamental contributions to programming as a high, intellectual challenge; for eloquent insistence and practical demonstration that programs should be composed correctly, not just debugged into correctness; for illuminating perception of problems at the foundations of program design.

https://bit.ly/3nVC3ux

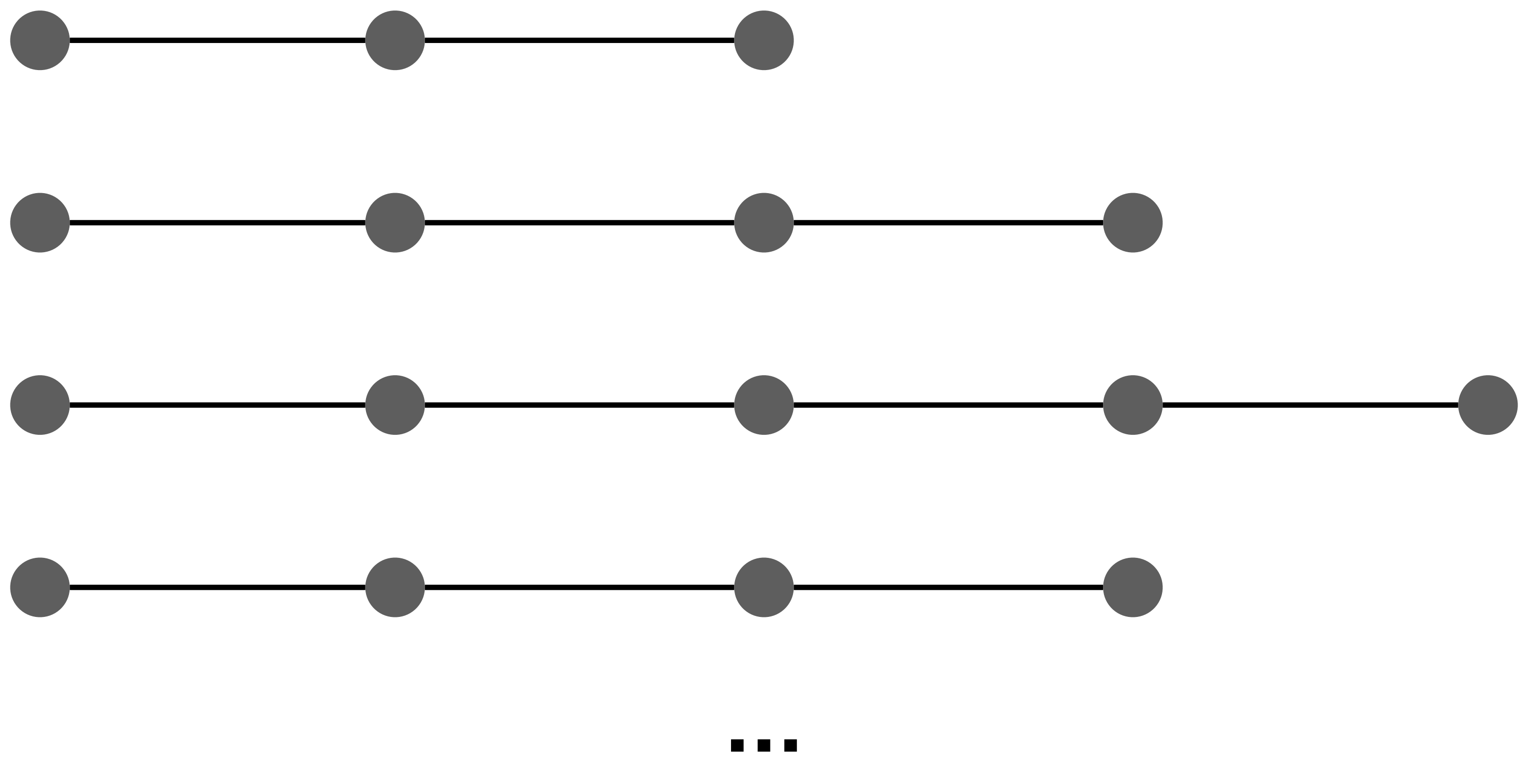
Program Verification

- Prove a given program satisfies the target properties
- Property: points of interest in programs
  - E.g., “p == NULL?”, “idx < size?”, “arr sorted?”, 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 an (infinite) set of traces
- $[\![P]\!]$: a set of all possible execution traces

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

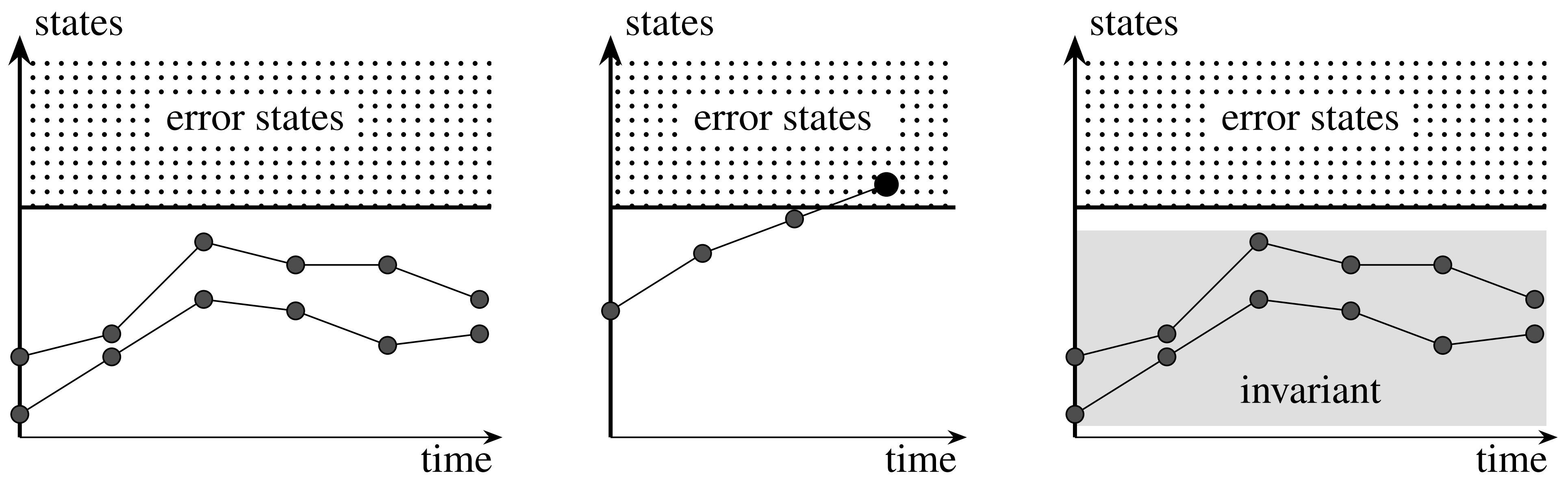


Trace Properties

- 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

$$\frac{\{B \land I\}\ C\ \{I\}}{\{I\}\ \texttt{while}\ B\ C\{\neg B \land I\}}$$

```
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 <= x <= 10”

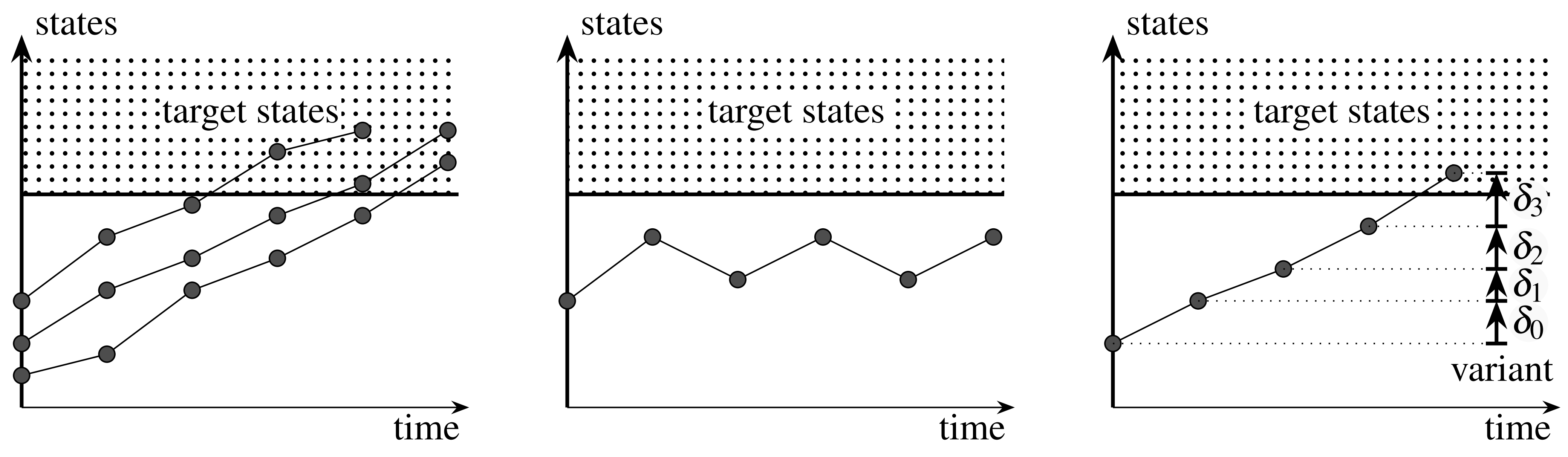
Example: Division-by-Zero

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

```
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?
```

c <= 3n + 2

0 <= 3n + 2 - c $\wedge$ 3n + 2 - c is strictly decreasing $\Rightarrow$ termination

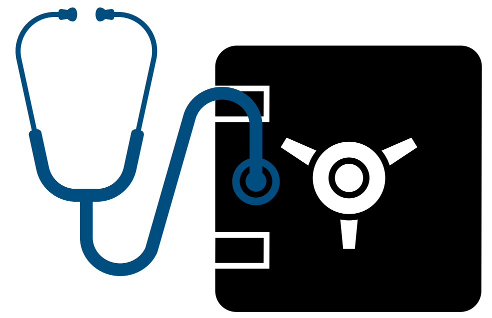
Example

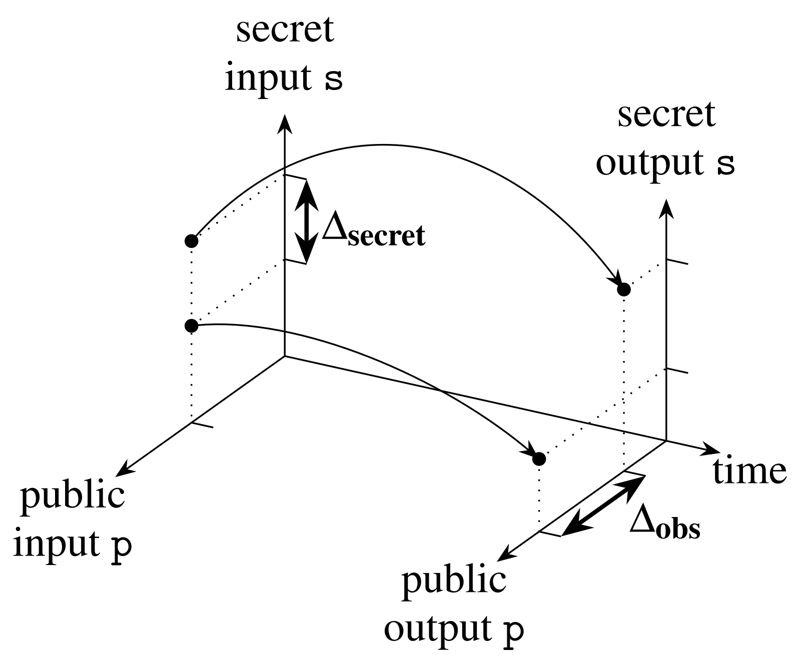
- Correctness of a sorting algorithm as trace property

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)		

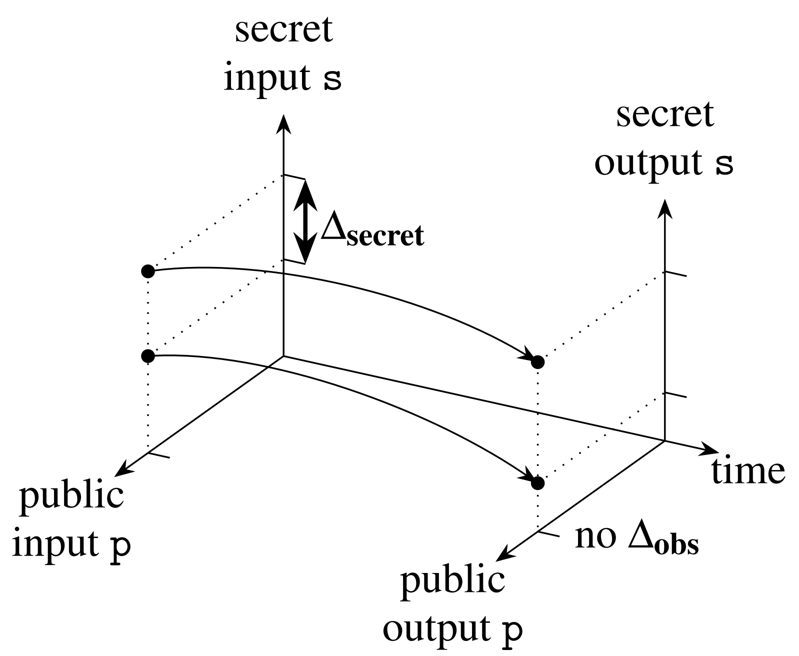
Information Flow Properties

- 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}

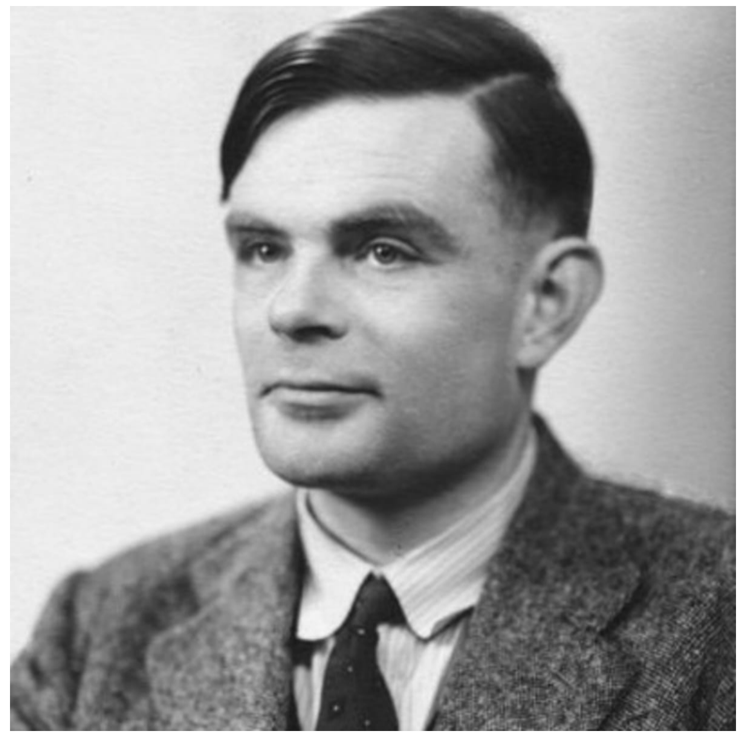
A Hard Limit: Undecidability

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 $F(x) = \text{if } H(x, x) \text{ then } \texttt{loop}() \text{ else } \texttt{halt}()$
- 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

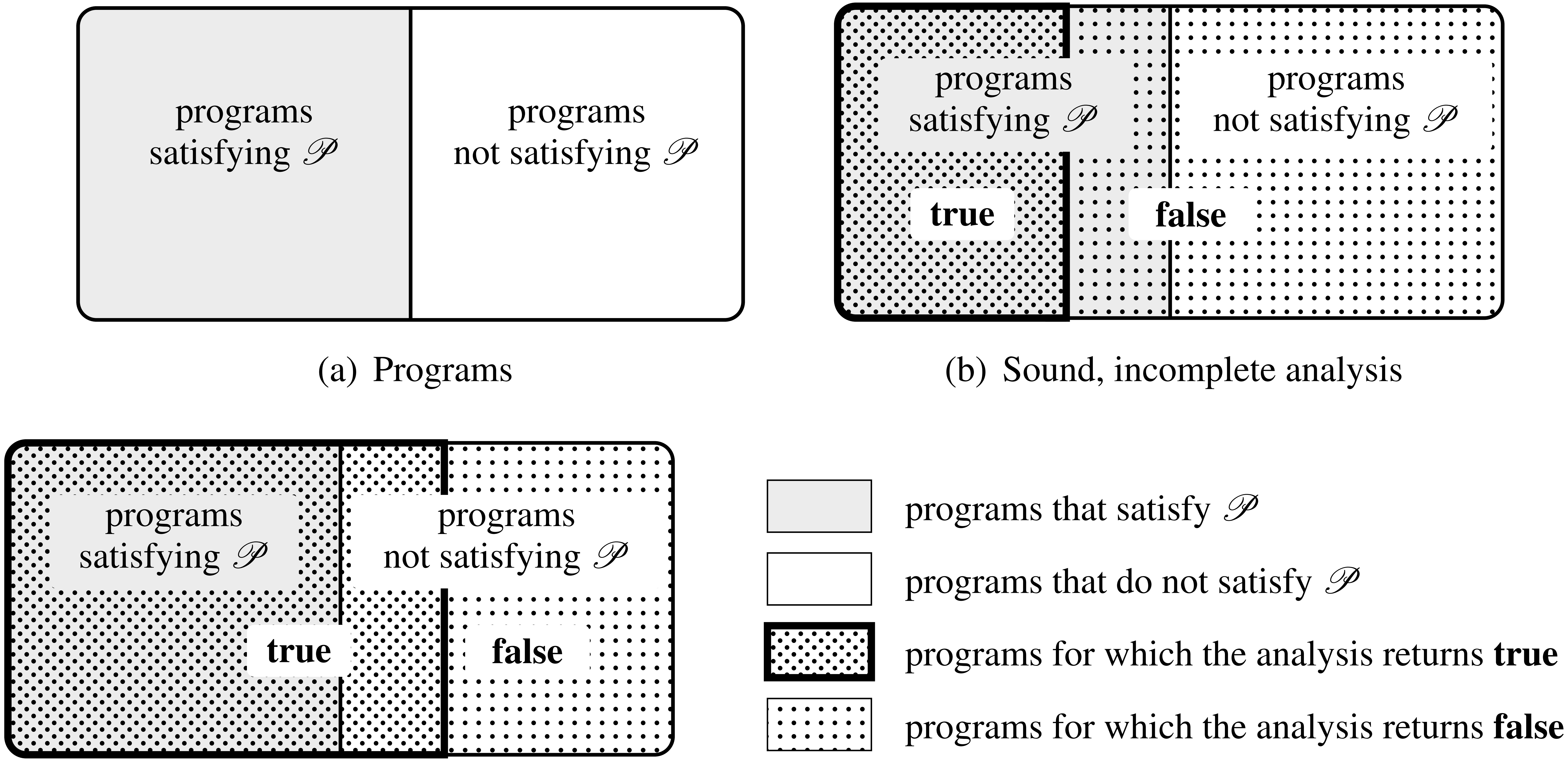
- Given a semantic property $\mathscr{P}$, and an analysis tool **A**
- If **A** were perfectly accurate,

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

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

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

Soundness and Completeness



(a) Programs

(b) Sound, incomplete analysis

(c) Unsound, complete analysis

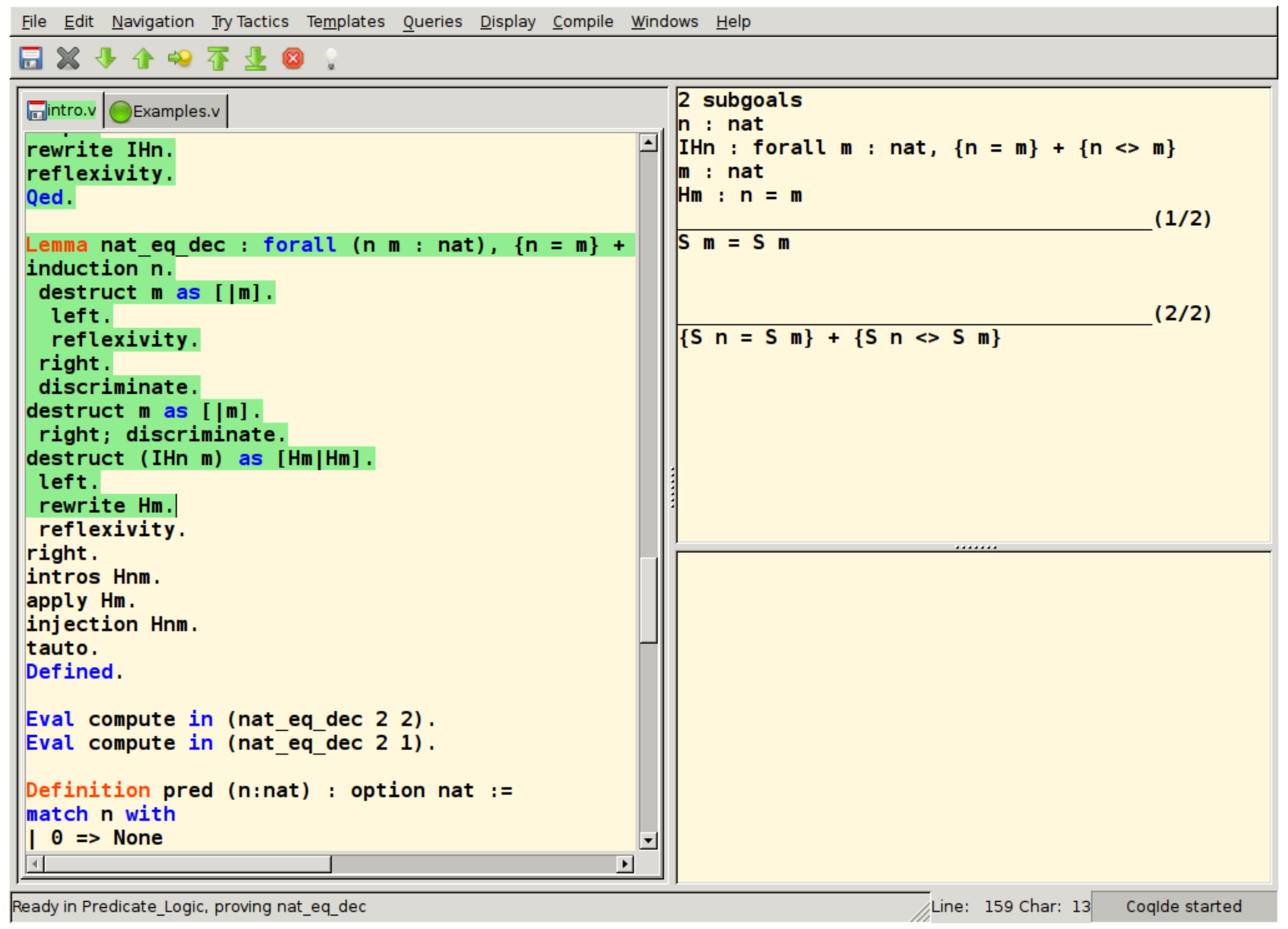
(d) Legend

Testing

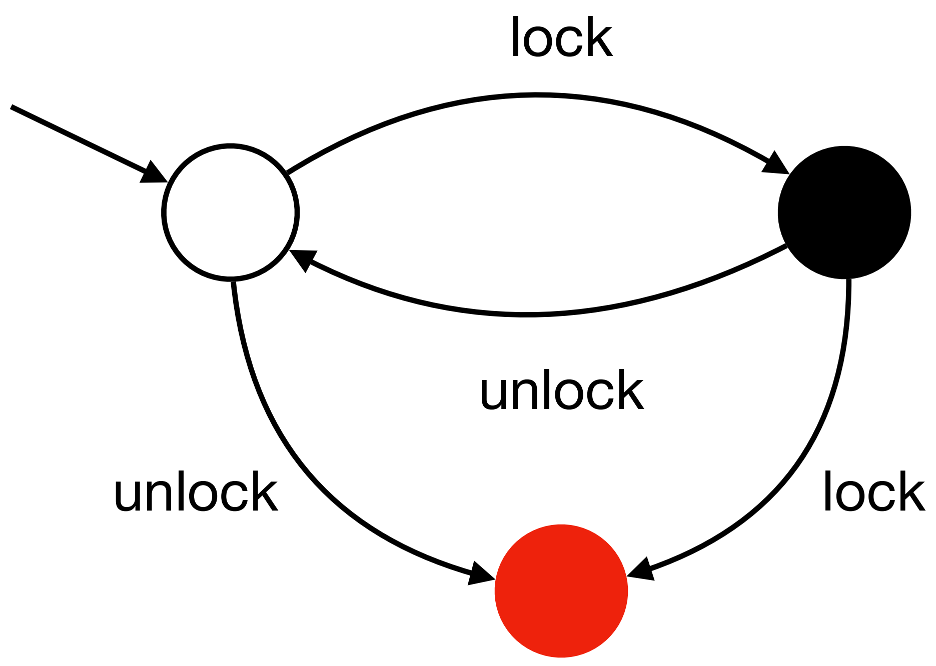
- Check a set of **finite executions**
  - e.g., random testing, concolic (**conc**rete + symb**olic**) testing
- In general, **unsound yet complete**
  - Unsound: cannot prove the absence of errors
  - Complete: produce counterexamples (i.e., erroneous inputs)
- Example: Google's oss-fuzz (https://github.com/google/oss-fuzz)

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)



Model Checking



Check: calls to lock and unlock must alternate

- Automatic technique to verify if a model satisfies a specification
  - Model of the target program (finite automata)
  - Specification written in logical formula
  - Verification via 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)

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

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