

Advanced Software Security

8. Introduction to Program Verification

Kihong Heo



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



#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>

EDSGER W. DIJKSTRA

Fundamental contributions to
programming as a high,
intellectual challenge.



A.M.

TURING

A W A R D

1972



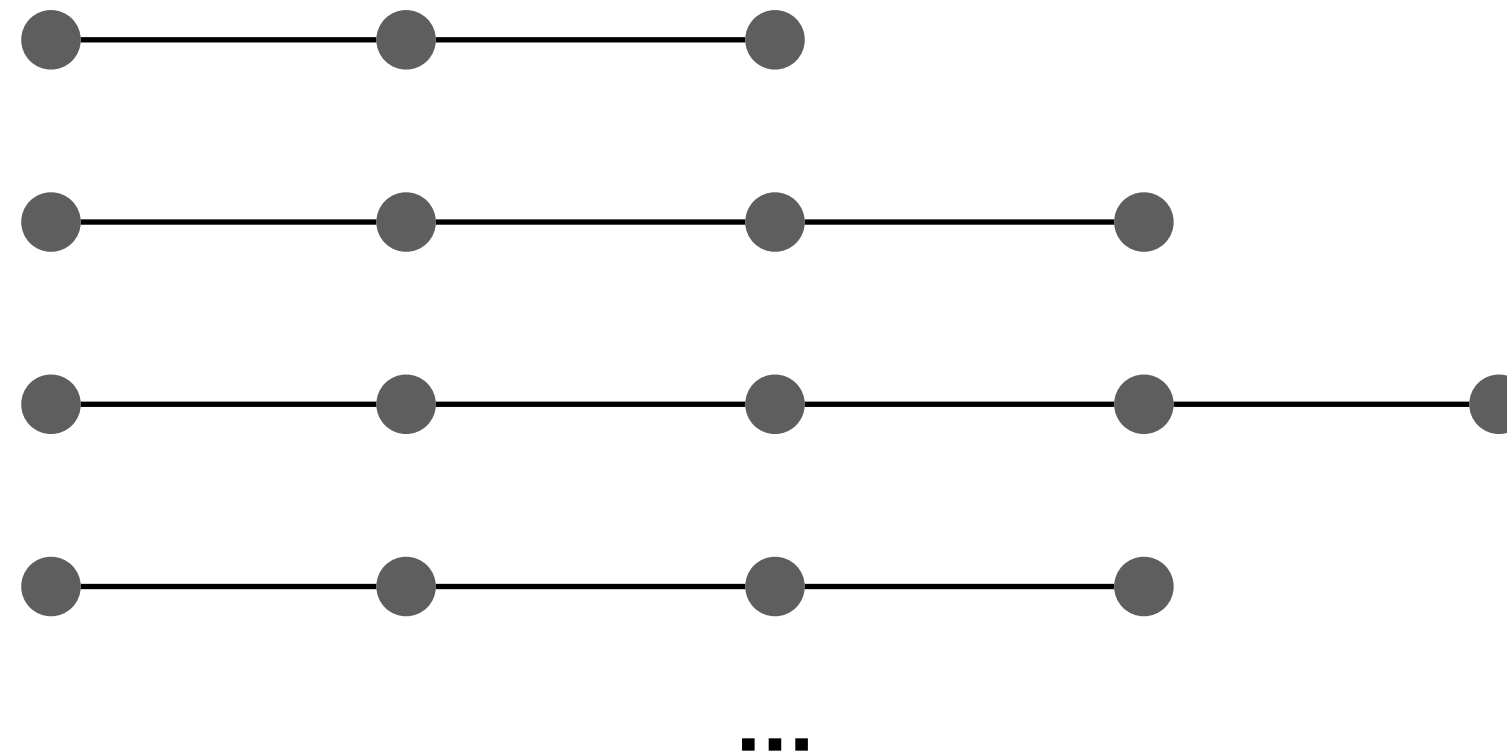
Program Verification

- 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 a set of (infinite) set of traces
- $\llbracket P \rrbracket$: a set of all possible execution traces

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

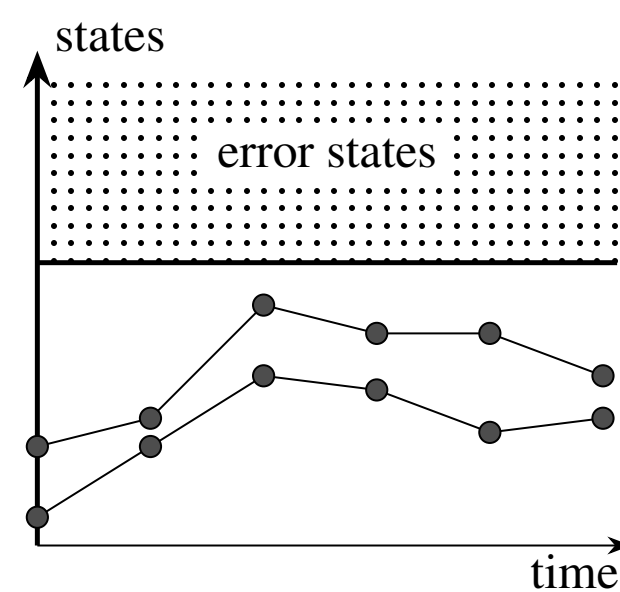


Trace Properties

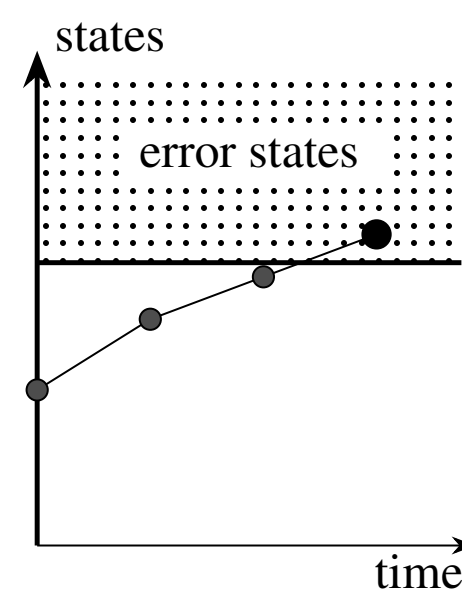
- A semantic property \mathcal{P} that can be defined by a **set of execution traces** that satisfies \mathcal{P}
 - Ex1: “all traces that satisfies $x \neq 0$ at line 10”
 - Ex2: “all traces where the value of y at line 97 is the same as the one in the entry point”
- Program P satisfies property \mathcal{P} 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

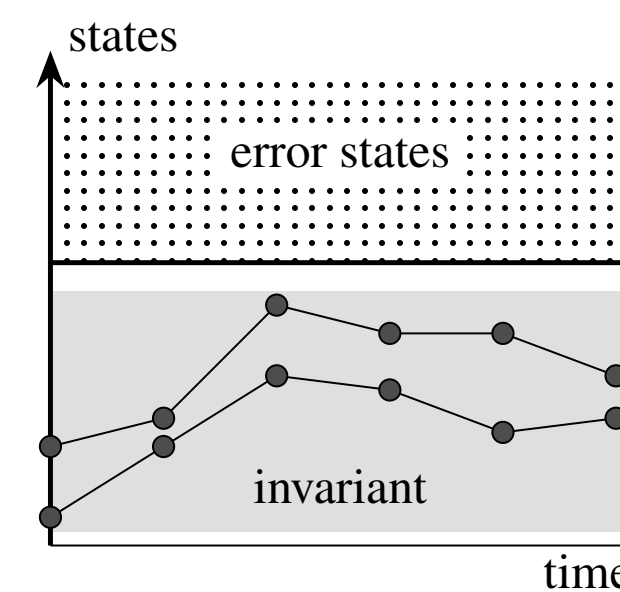
- A program **never** exhibit a behavior observable within **finite time**
 - “Bad things will never occur”
 - Bad things: integer overflow, buffer overrun, deadlock, etc
- If false, then there exists a **finite counterexample**
- To prove: all executions never reach error states



(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

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

```
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 ≤ x ≤ 10”

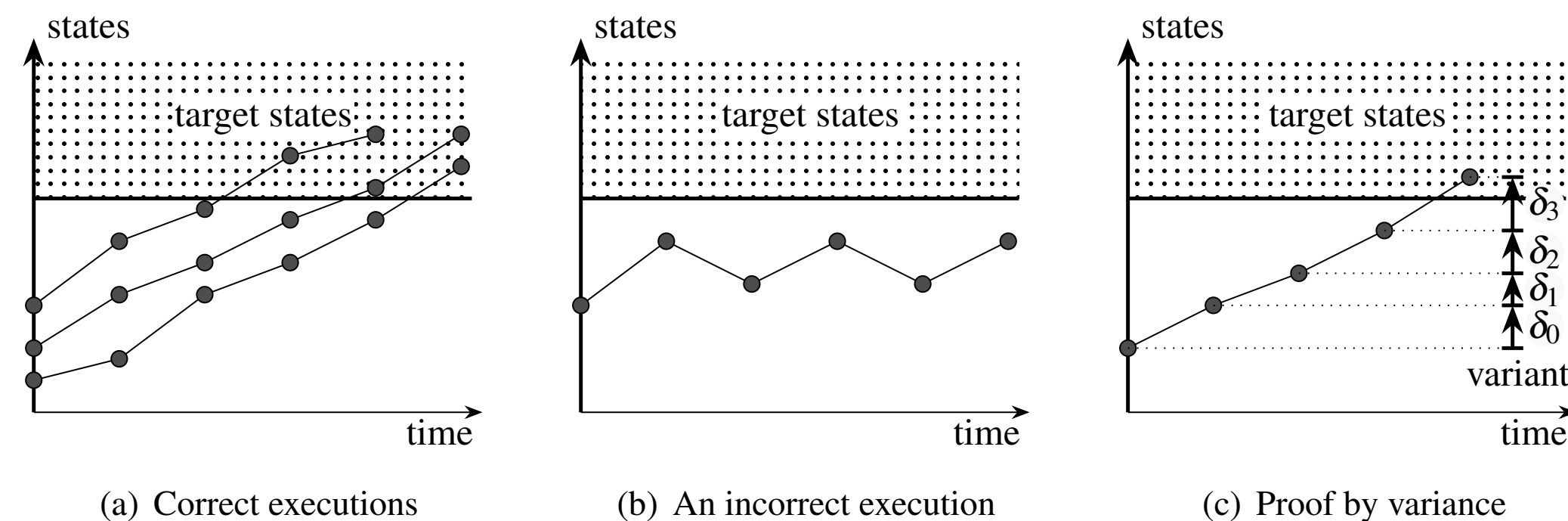
Example: Division-by-Zero

```
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

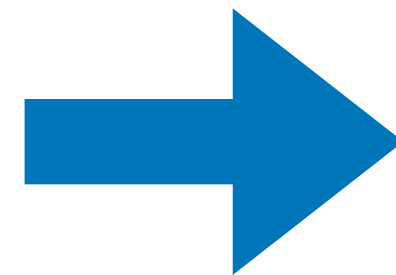
```
x = pos_int();  
while (x > 0) {  
    x = x - 1;  
}
```

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;
}
```



```
// 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?
```

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

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

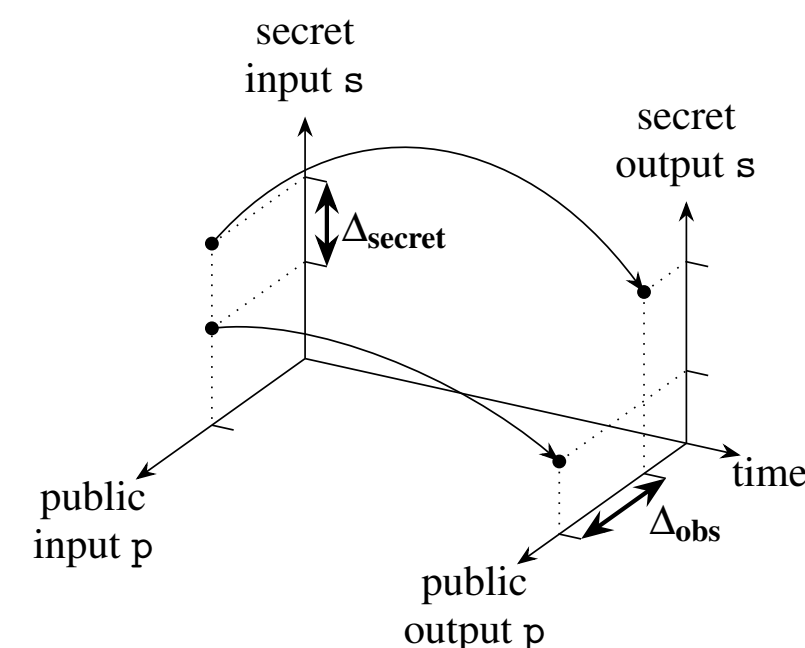
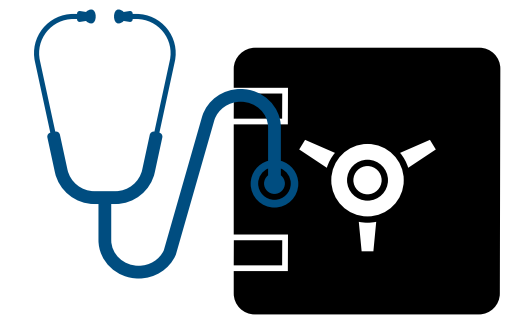
Example

- Correctness of a sorting algorithm as trace property

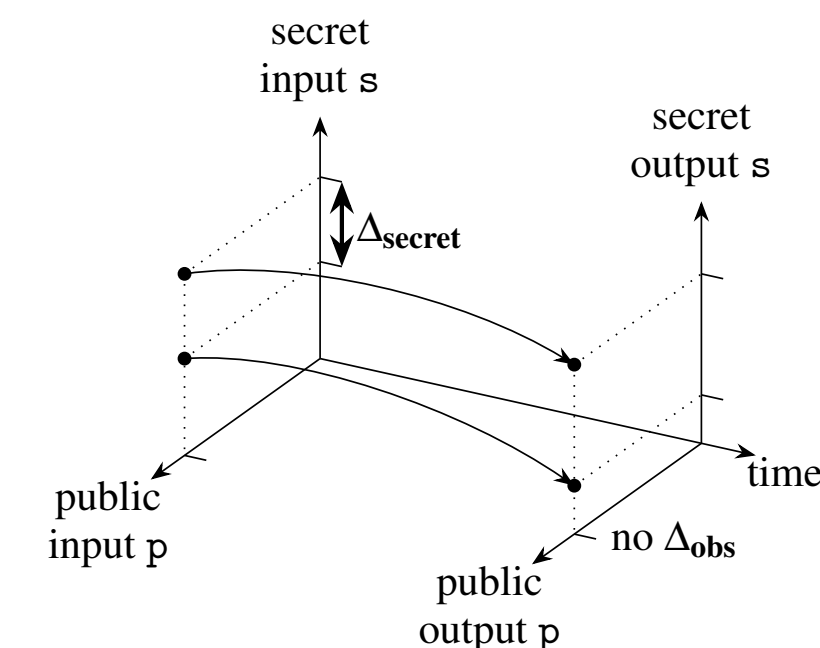
Property	Safety or Liveness?	State?
Should not fail with a run-time error		
Should terminate		
Should return a sorted array (if terminated)		
Should return an array with the same elements and multiplicity (if terminated)		

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]
```

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

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

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

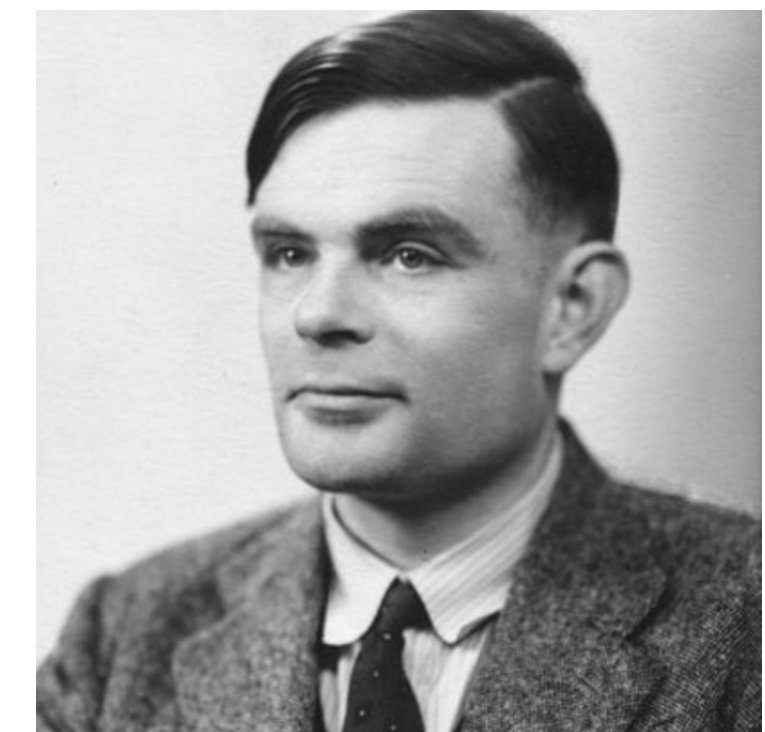
A Hard Limit: Undecidability

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 solve 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

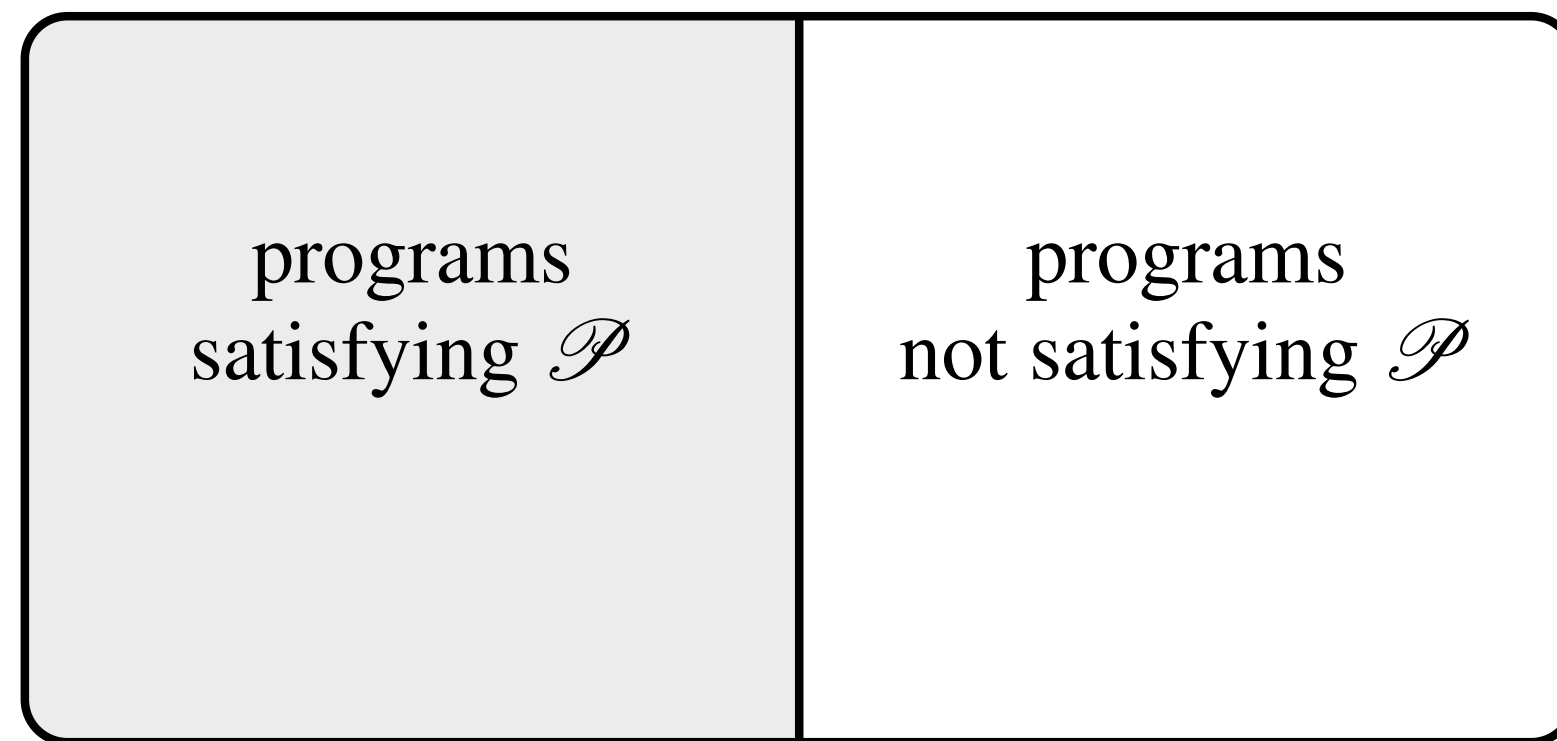
- Given a semantic property \mathcal{P} , and an analysis tool **A**
- If **A** were perfectly accurate,

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

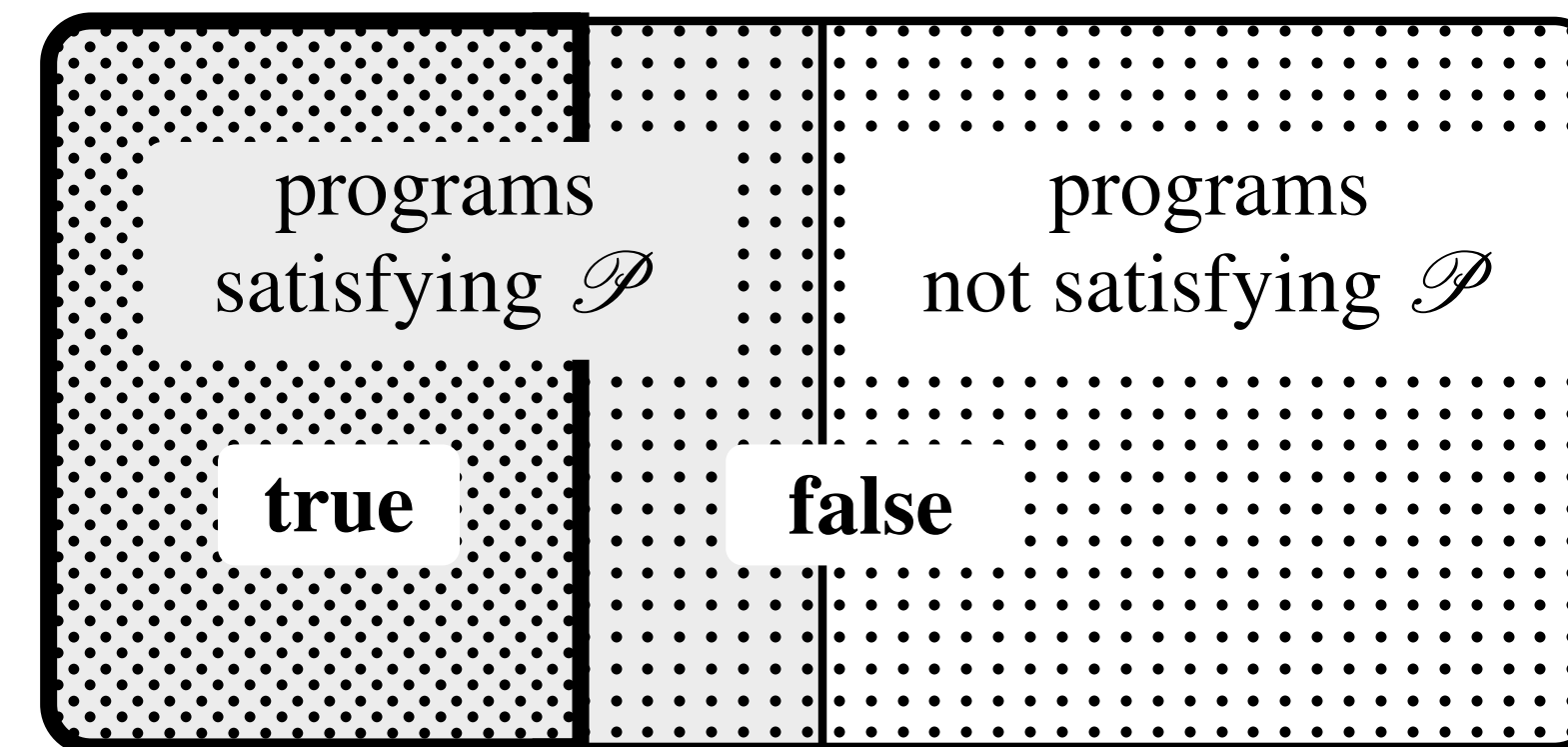
For all program p , $\mathbf{A}(p) = \mathbf{true} \Rightarrow p \text{ satisfies } \mathcal{P}$ **(soundness)**

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

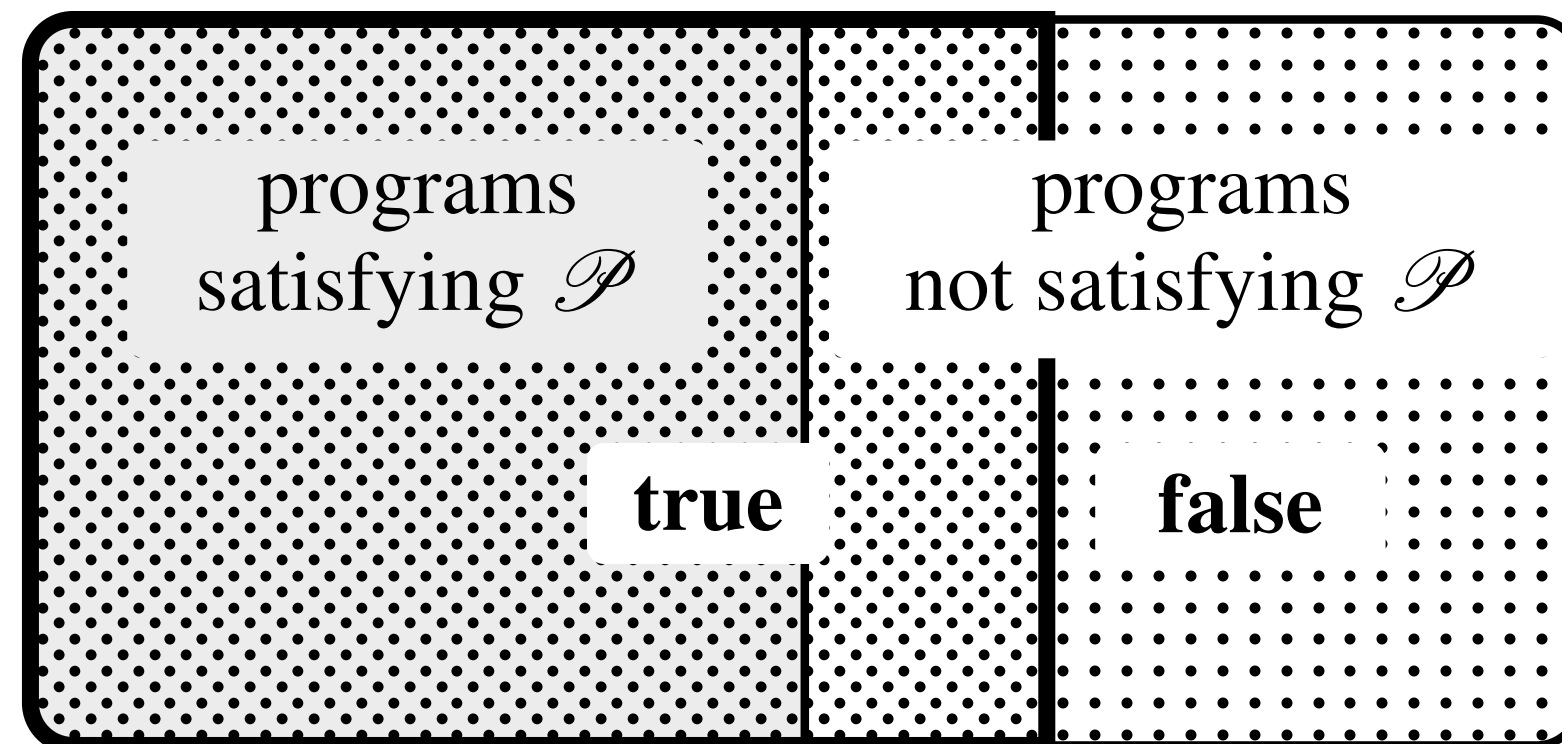
Soundness and Completeness



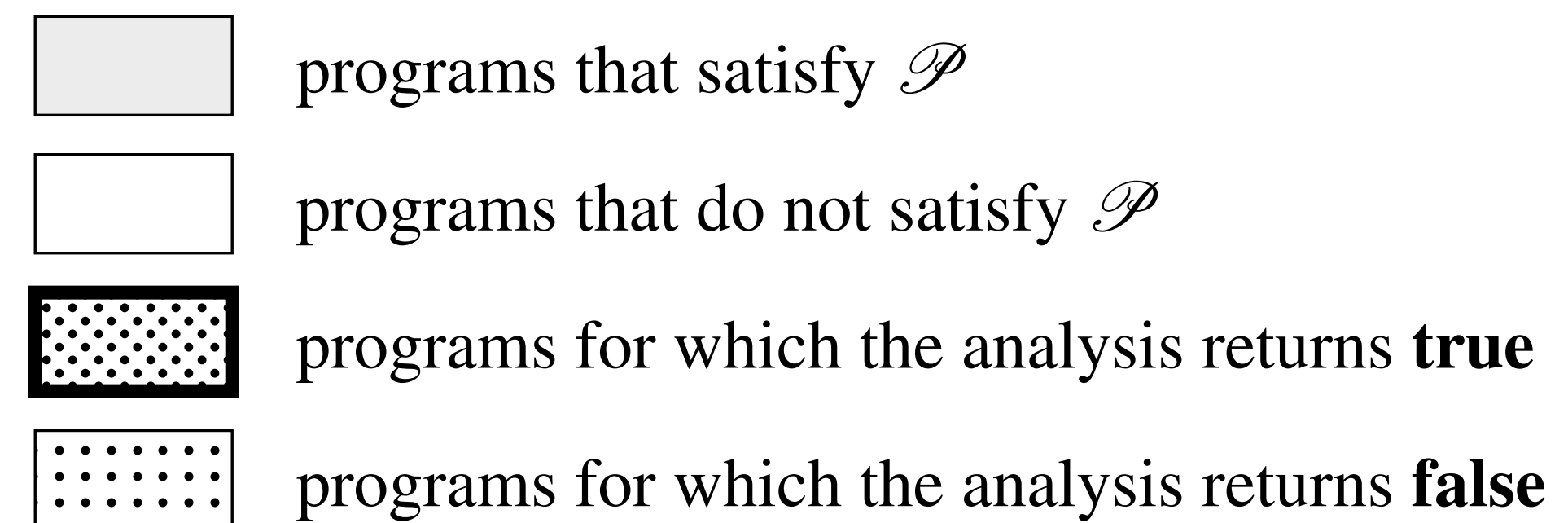
(a) Programs



(b) Sound, incomplete analysis



(c) Unsound, complete analysis



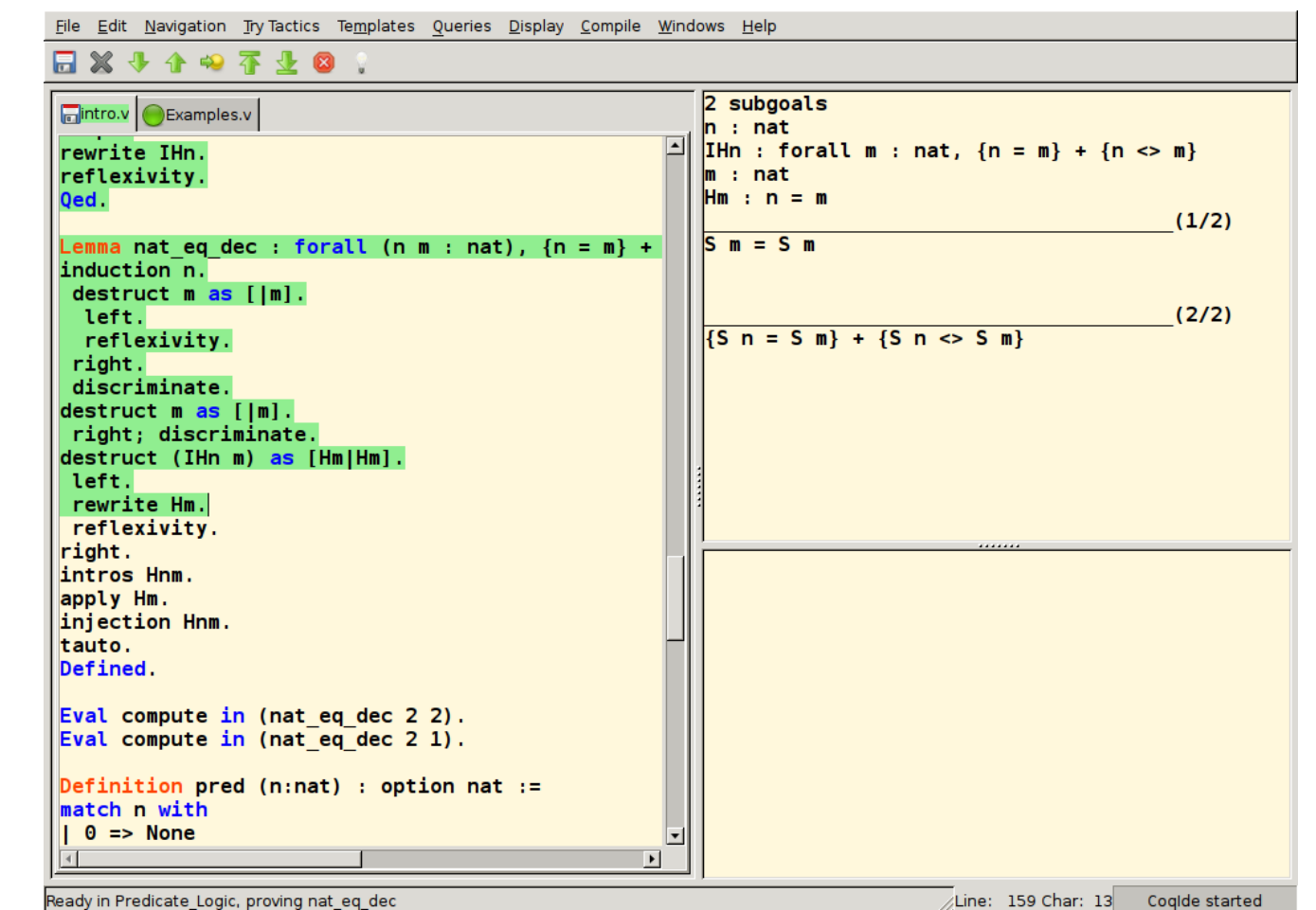
(d) Legend

Testing

- Check a set of **finite executions**
 - e.g., random testing, concolic (**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 main window displays a proof script for a lemma named `nat_eq_dec`. The script uses tactics like `forall`, `induction`, `destruct`, `left`, `right`, `reflexivity`, `discriminate`, `rewrite`, `intros`, `apply`, `injection`, and `tauto` to prove the decidability of equality for natural numbers. The right-hand pane shows the current goals, including the lemma statement and the induction hypothesis. The bottom status bar indicates the current line and character position.

```
File Edit Navigation Try Tactics Templates Queries Display Compile Windows Help
Coq Intro 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).

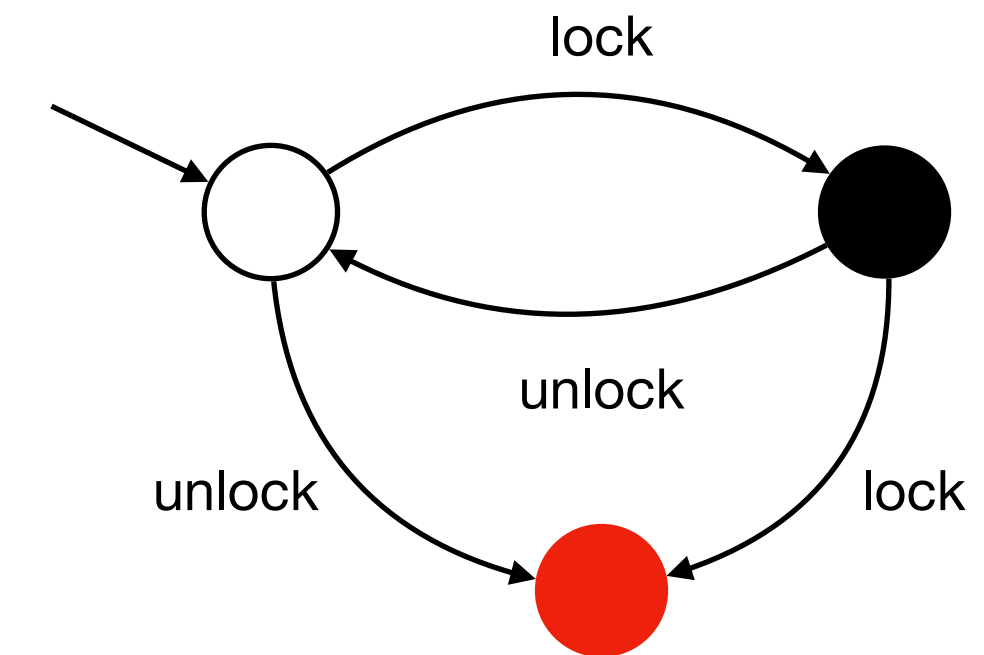
Definition pred (n:nat) : option nat :=
match n with
| 0 => None
```

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

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 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)



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

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