Program Analysis

4. Concepts in Program Analysis

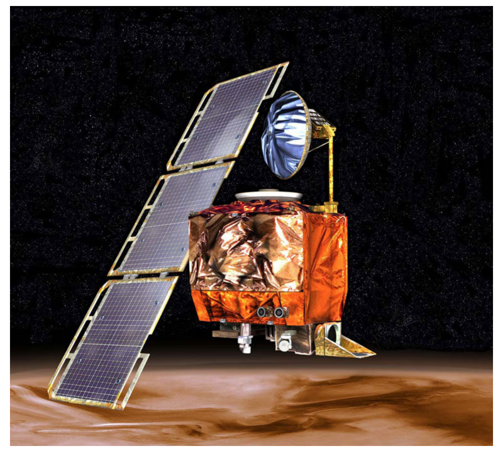
Kihong Heo

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

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

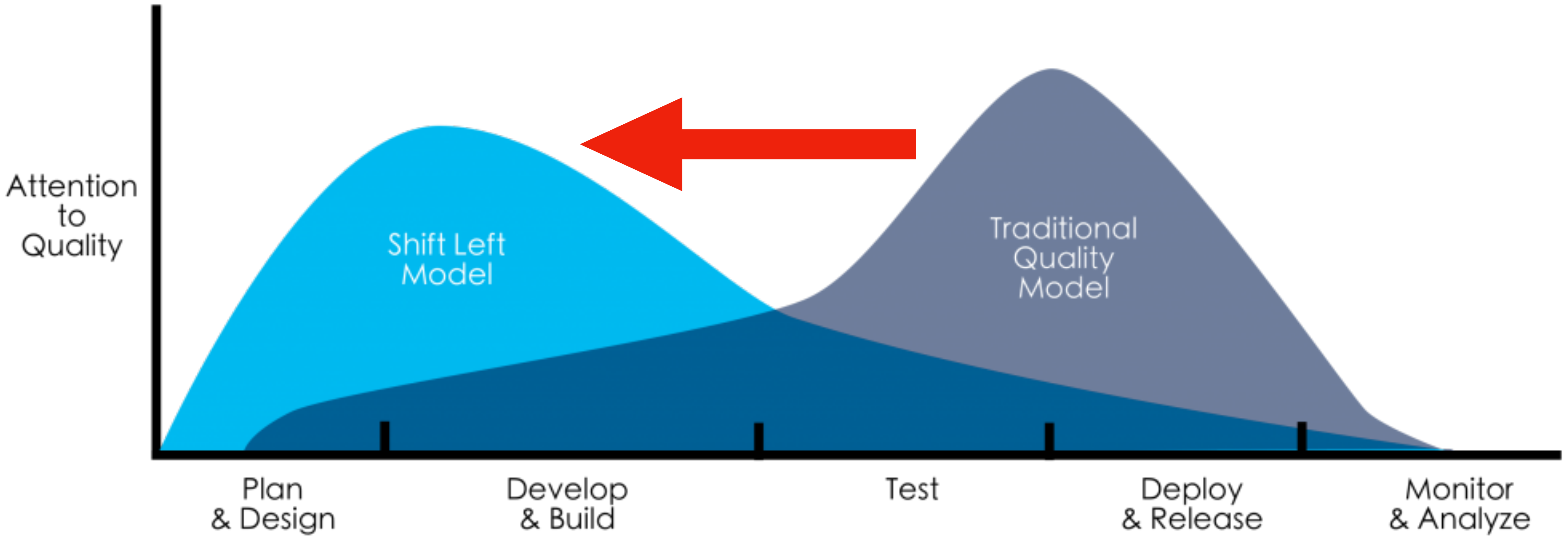
Q: What is the solution to improve software quality at low cost?

A: Program analysis

Discovering Software Errors

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

- IBM Systems Sciences Institute, 2015



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

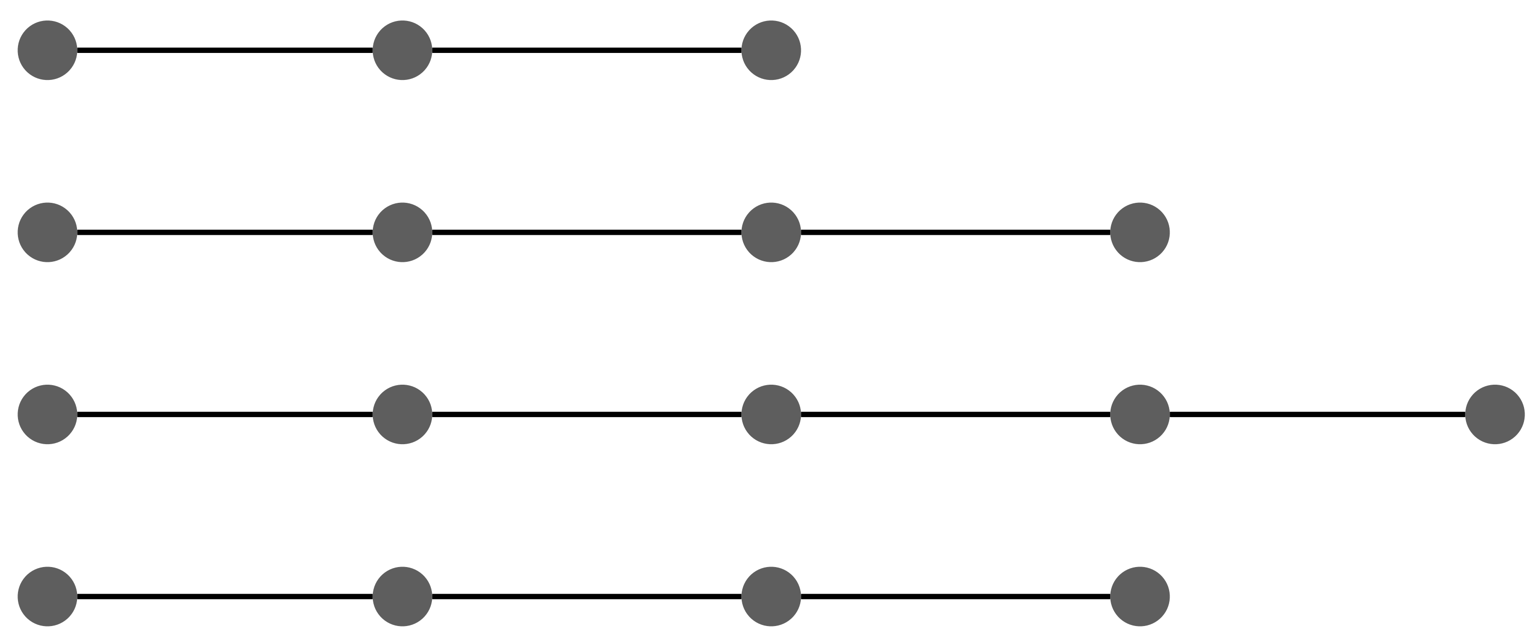
Properties성질

- 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

Trace실행경로

- 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

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

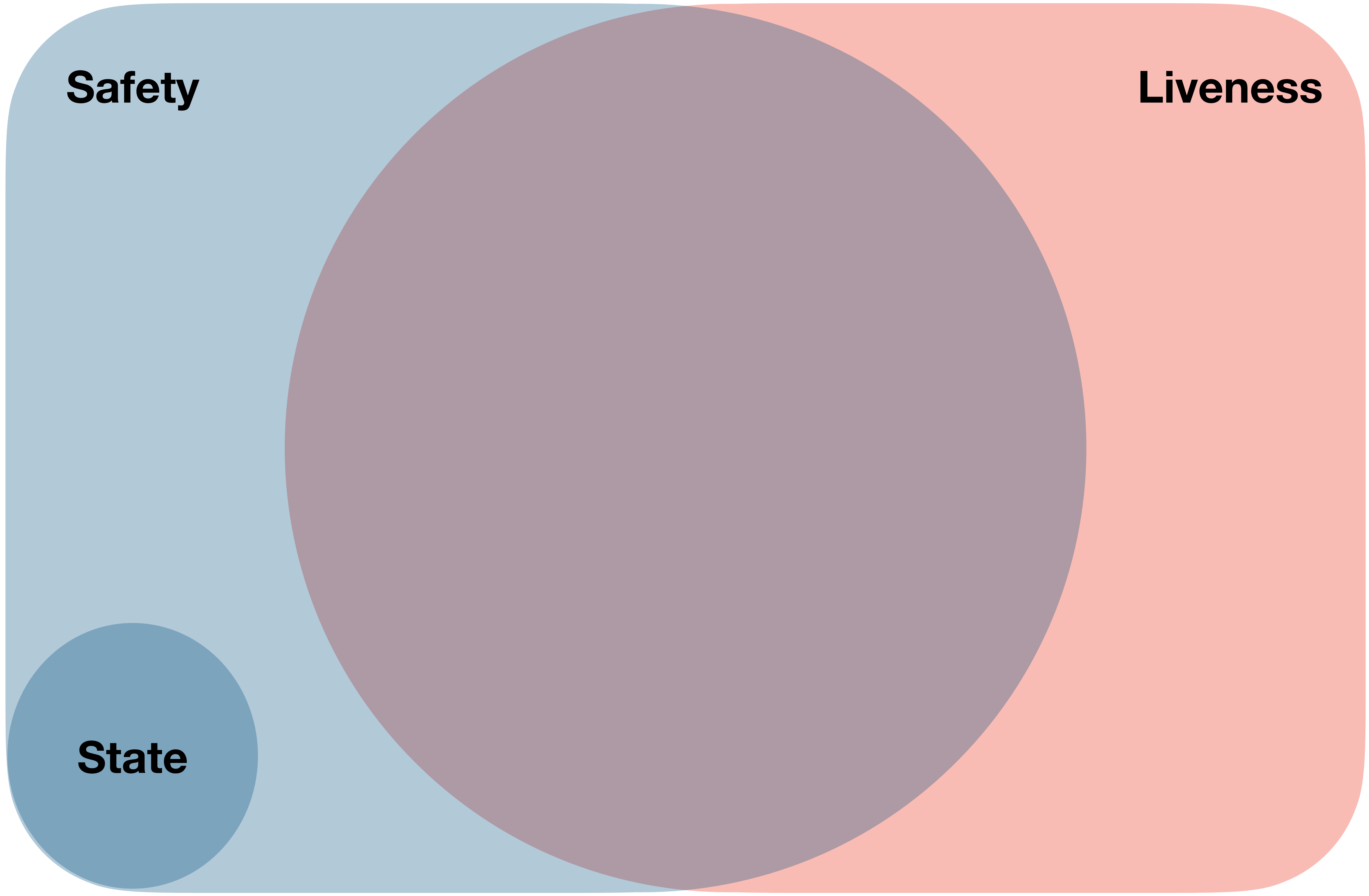


...

Trace Properties개별경로 성질

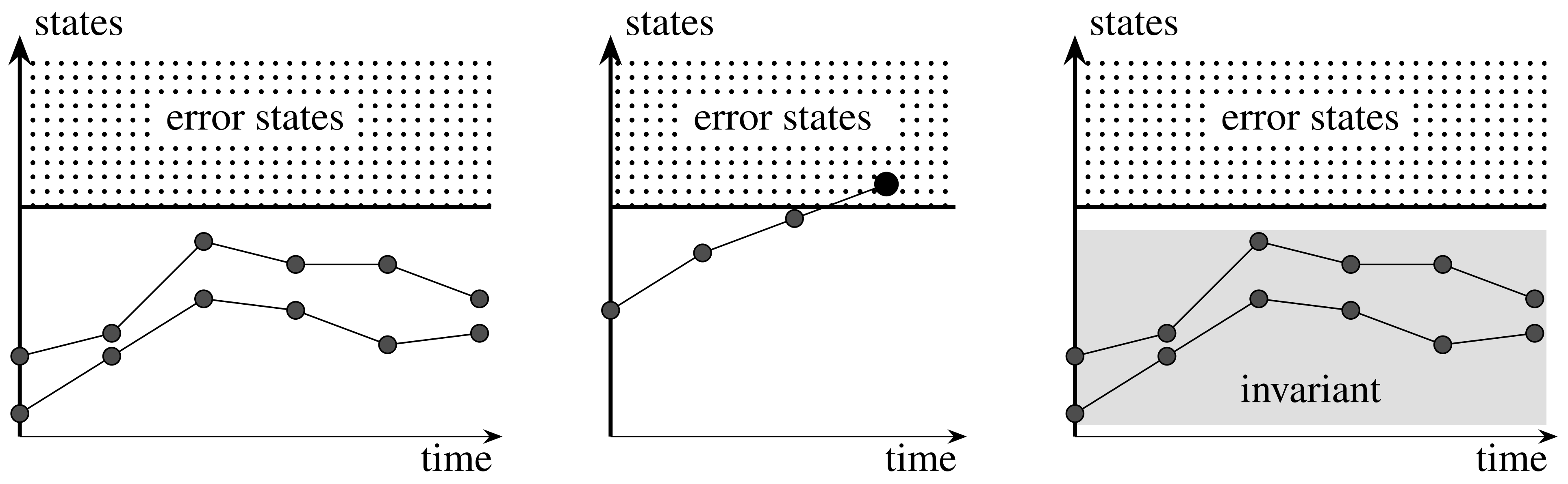
- A semantic property $\mathscr{P}$ that can be defined by a **set of execution traces** that satisfies $\mathscr{P}$
  - Ex1: “all traces that satisfies x != 0”
  - Ex2: “all traces where the value of y at termination is the same as the one in the entry point”
- Program P satisfies property $\mathscr{P}$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

Trace Properties



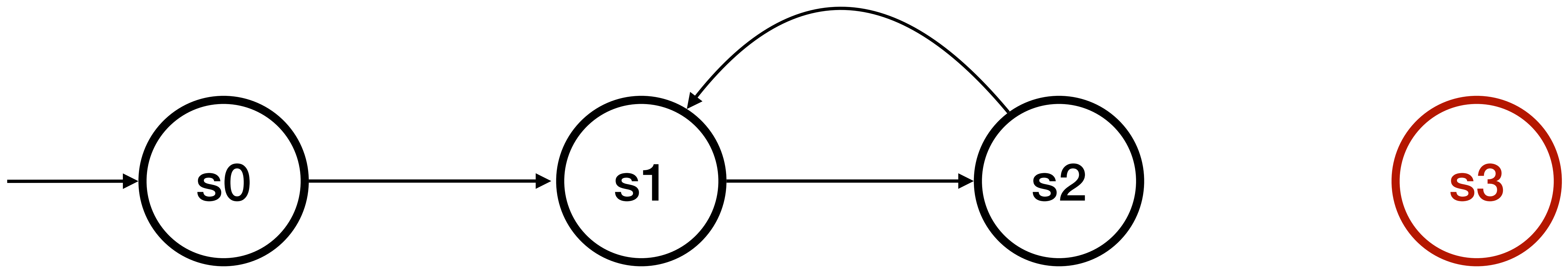
Safety Property항상성질

- 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

Example



Reachable states <= 0 step : {s0}
Reachable states <= 1 step : {s0, s1}
Reachable states <= 2 steps : {s0, s1, s2}
Reachable states <= 3 steps : {s0, s1, s2}
Reachable states <= 4 steps : {s0, s1, s2}
…
Reachable states <= 100 steps : {s0, s1, s2}
…
Reachable states < ∞ steps : {s0, s1, s2}

Invariant불변식

- Assertions supposed to be **always true** and **remain unchanged** after any operations
  - 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”

Invariant in Art

by hyerinshelly, 2022 🥇

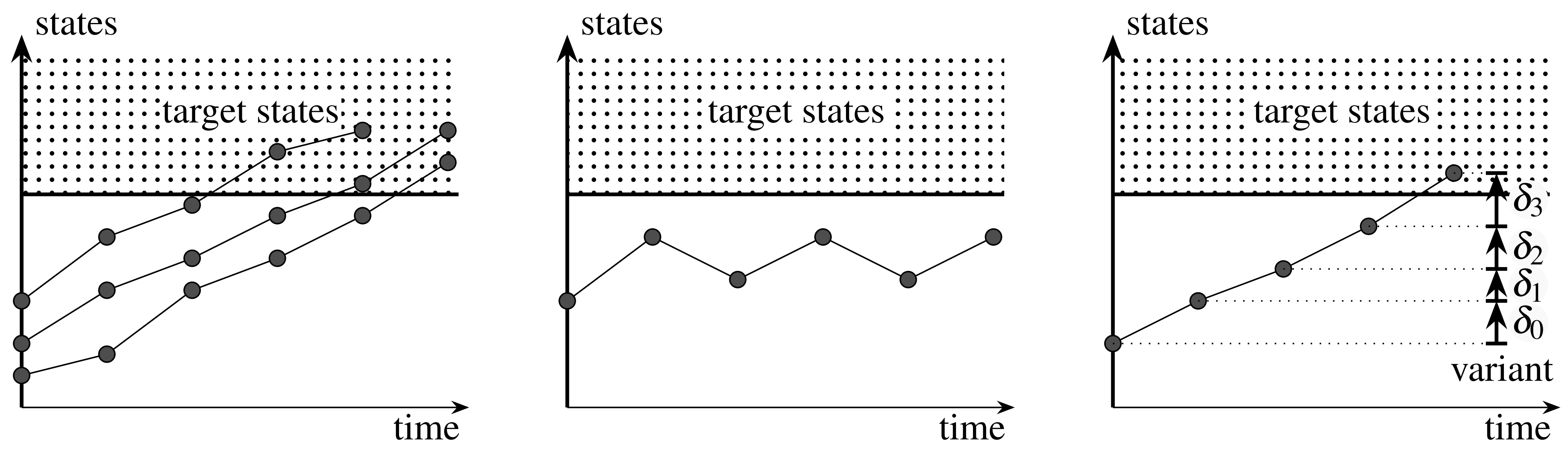
Example: Division-by-Zero

```
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;
}
```

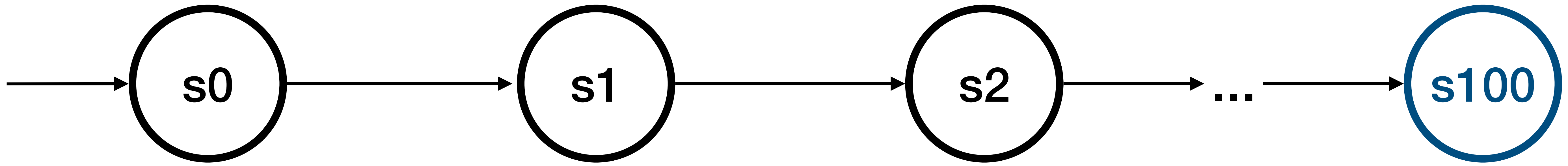
Liveness Property결국성질

- A program will **eventually** exhibit a behavior observable within **finite time**
  (A program will **never** exhibit a behavior observable only after **infinite 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

Example



Shortest distance after 0 step : 100
Shortest distance after 1 step : 99
Shortest distance after 2 steps : 98
Shortest distance after 3 steps : 97

...

(if we are sure that the distance will keep decreasing)

...

Reachable states after 100 steps : 0

Variant변동식

- A quantity that **evolves towards** the set of target states
(so guarantee any execution eventually reaches the set)
- Usually, a value that is strictly decreasing for some well-founded order relation
  - Well-founded order: there is no infinite decreasing chain
  - E.g., an integer value that is always positive and strictly decreasing

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

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

Example: Termination

```
// A factorial program
fun fact(n) =
  r = 1;
  i = 1;
  while (i <= n) {
    r = r * i;
    i = i + 1;
  }
}
```

$i <= n + 1$

$0 <= n - i + 1 \quad \wedge \quad n - i + 1$ is strictly decreasing $\quad \Rightarrow \quad$ termination

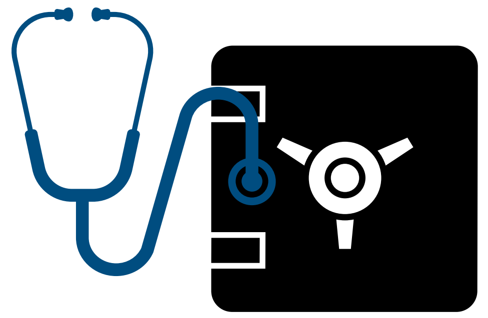
Example

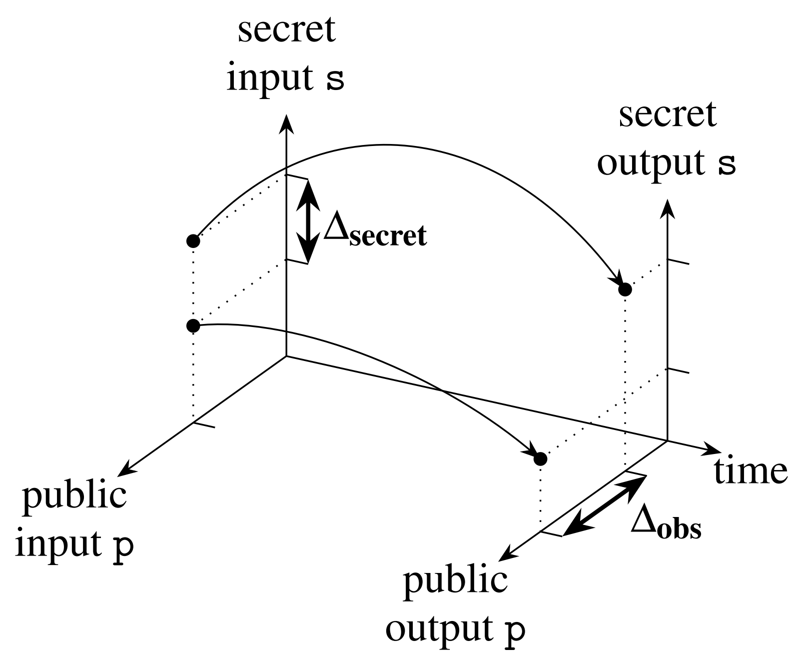
- Correctness of a sorting algorithm as trace property

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)		

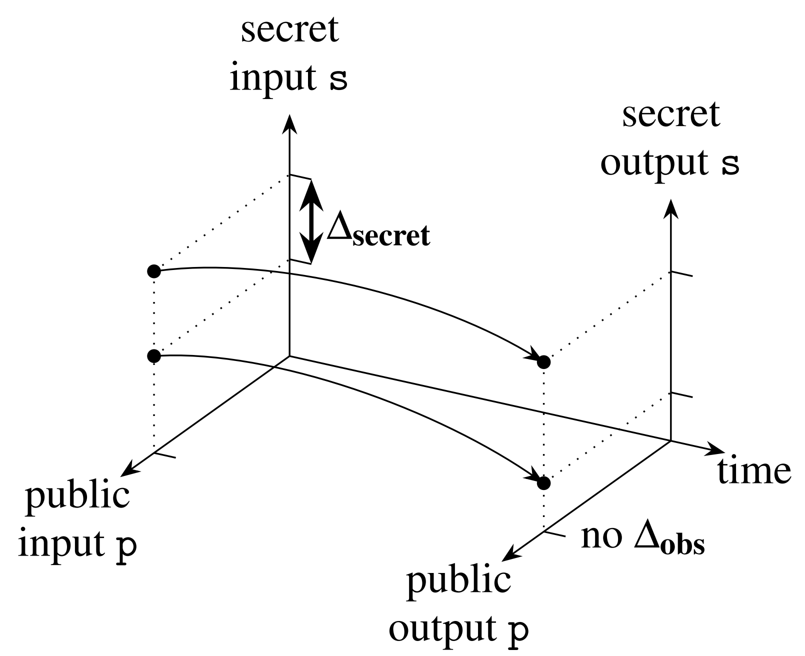
Information Flow Properties실행 간섭 성질

- 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

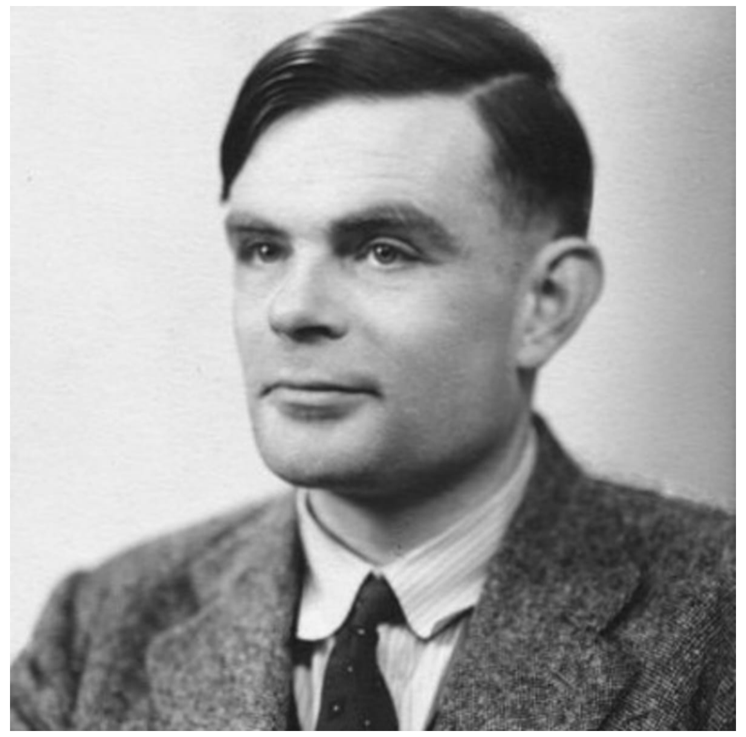
A Hard Limit: Undecidability계산불가능성

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

- Fact: 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!

If Decidable?

- Many mathematical problems become trivial!

```
// Fermat's last theorem

for (a, b, c, n) in ℕ⁴ do
  if n > 2 && aⁿ + bⁿ = cⁿ then
    exit()
```

```
// Goldbach's conjecture

for k in Even do
  for (p, q) in Prime² do
    if k != p + q then
      exit()
```

…

…

…

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

Soundness안전성 and Completeness완전성

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

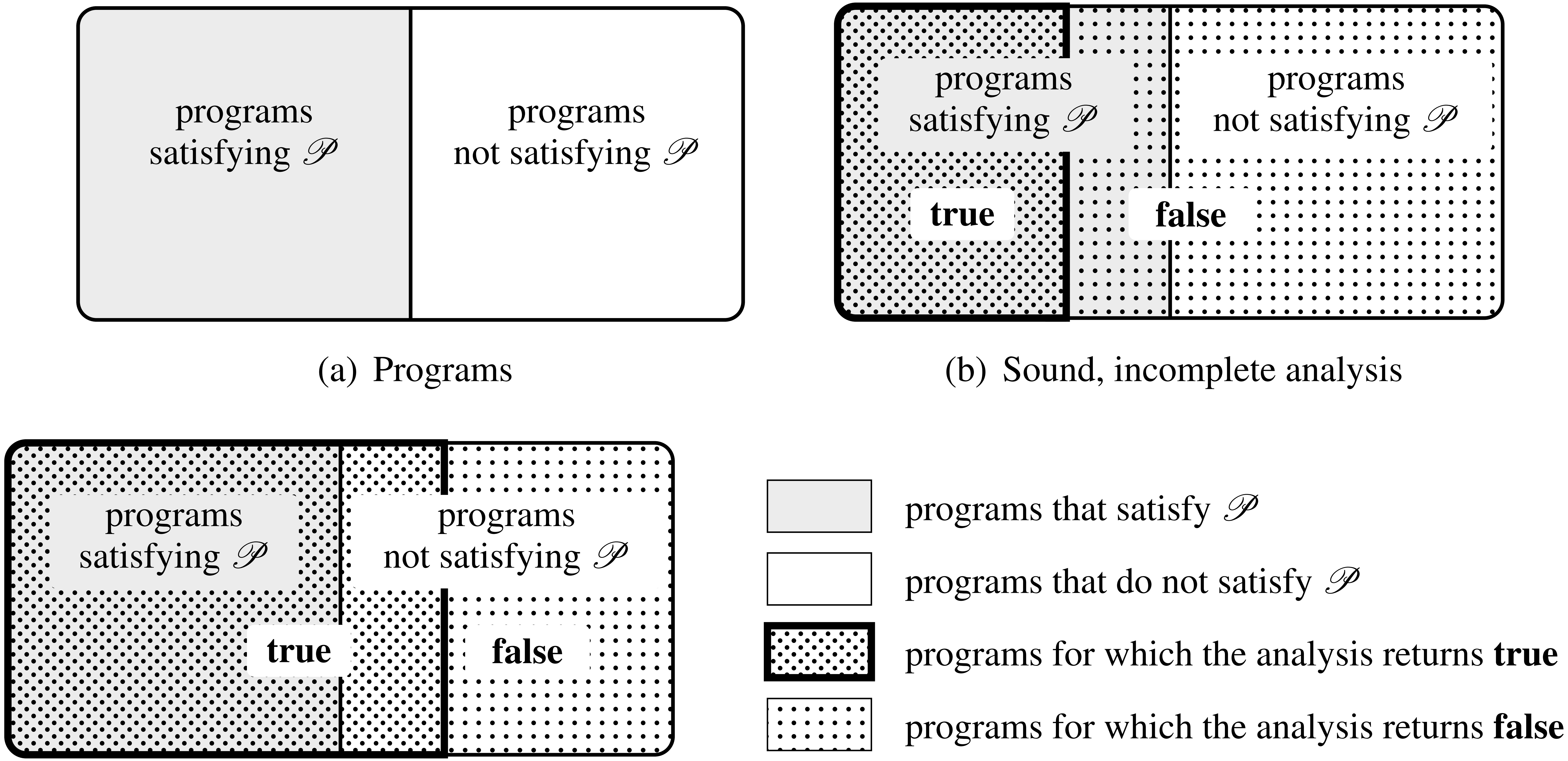
$$\text{For all program p, } \mathbf{A}(\text{p}) = \mathbf{true} \iff \text{p satisfies } \mathscr{P}$$

which consists of

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

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

Soundness안전성 and Completeness완전성



(a) Programs

(b) Sound, incomplete analysis

(c) Unsound, complete analysis

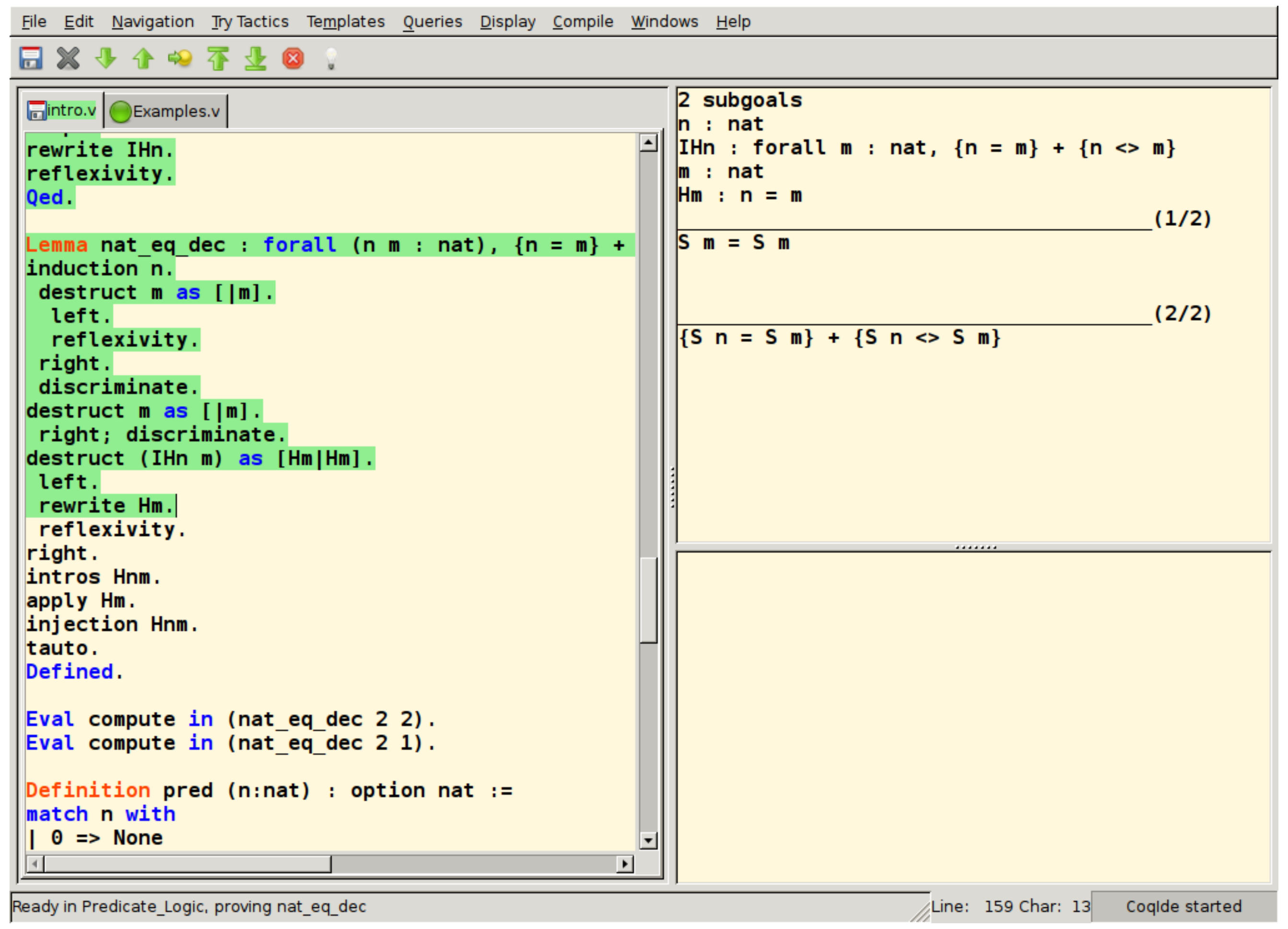
(d) Legend

Testing

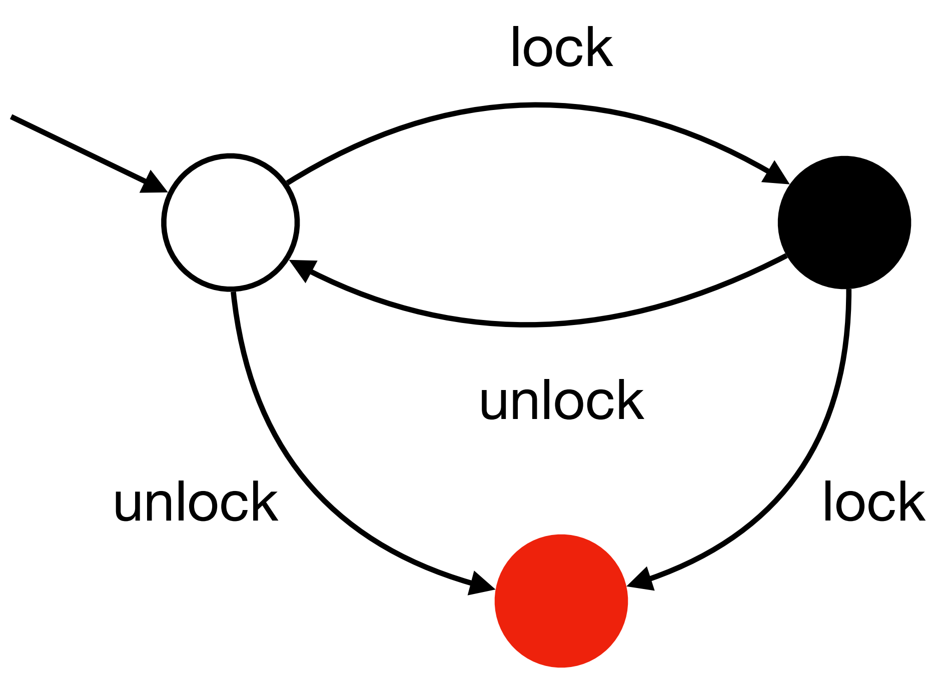
- Check a set of **finite executions**
  - e.g., random testing, concolic (**conc**rete + symb**olic**) testing
- In general, **unsound yet complete**
  - Unsound: cannot prove the absence of errors
  - Complete: produce counterexamples (i.e., erroneous inputs)
- Example: Google’s oss-fuzz (https://github.com/google/oss-fuzz)

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)

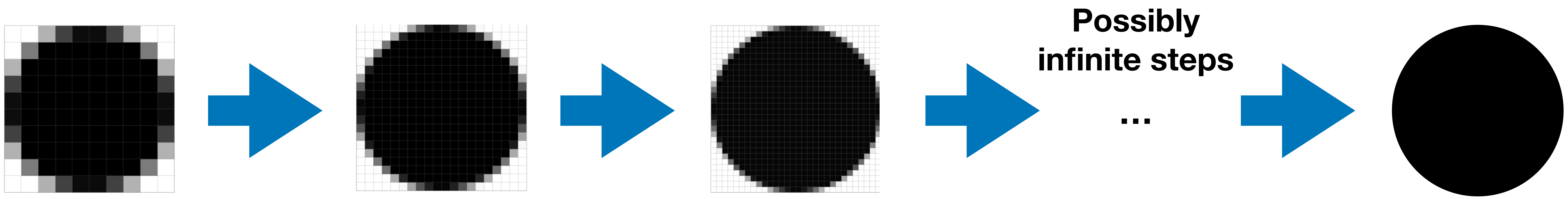


Model Checking



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 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)



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

```
static char *curfinal = “HDACB  FE”;

keysym = read_from_input();

if ((KeySym)(keysym) >= 0xFF9987)
{
    unparseputc((char)(keysym - 0xFF91 + ‘P’), pty);
    key = 1;
}
else if (keysym >= 0)
{
    if (keysym < 16)
    {
        if (read_from_input())
        {
            if (keysym >= 10) return;
            curfinal[keysym] = 1;
        }
        else
        {
            curfinal[keysym] = 2;
        }
    }
    if (keysym < 10)
        unparseput(curfinal[keysym], pty);
}
```

curfinal: buffer of size 10

keysym : any integer

keysym: [0, 15]

SAFE

keysym: [0, 9]

Buffer-overrun

size of curfinal: [10, 10]
keysym: [0, 15]

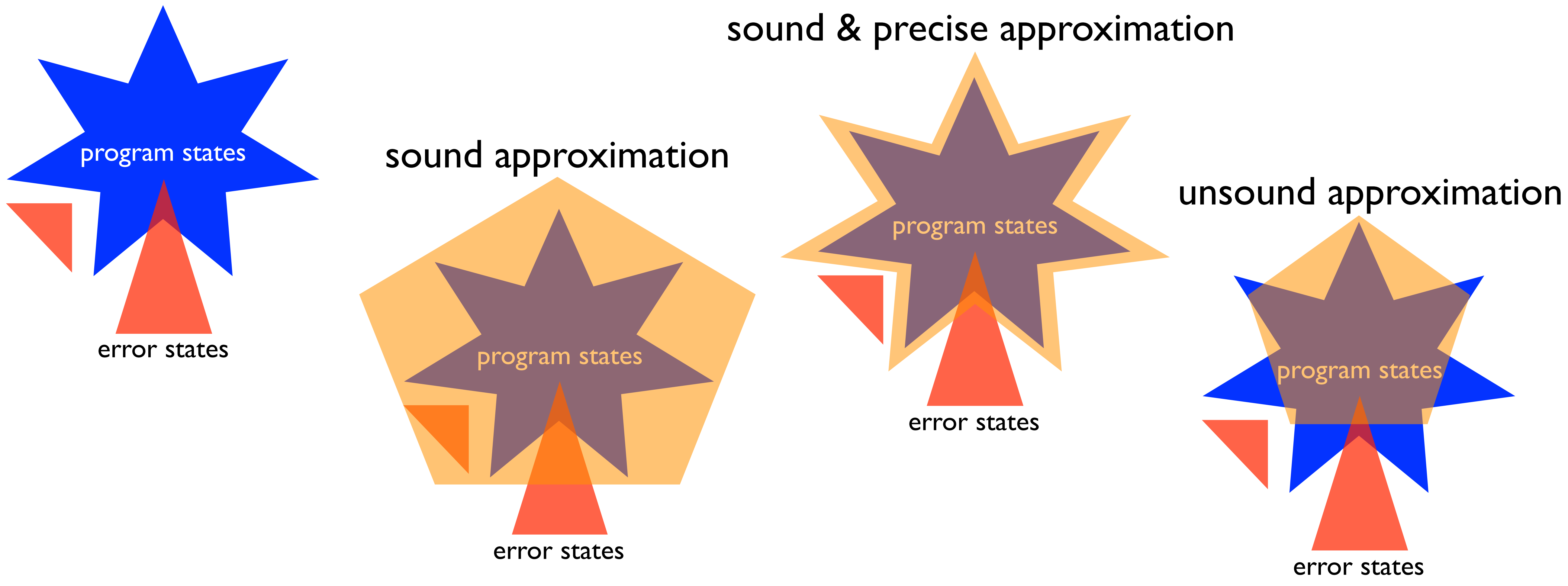
keysym: [0, 9]

SAFE

Approximation

- Compute approximated (inaccurate) semantics instead of exact semantics
  - Inaccurate ≠ incorrect
  - E.g., reality: {2, 4, 6, 8, …}

    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