Program Analysis

Adapted from slides by Hilton, Aldrich, and Le Goues

Learning Goals Today

- Be able to explain how soundness and completeness trade off in the design of tools that aim to find bugs automatically.

Spot the Bug

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa,
                                 SSLBuffer signedParams,
                                 uint8_t *signature,
                                 UInt16 signatureLen) {
    OSStatus err;
    …
    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
    …
 fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
}
```

Spot the Bug

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa,
                                 SSLBuffer signedParams,
                                 uint8_t *signature,
                                 UInt16 signatureLen) {
    OSStatus err;
    …
    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
    …
 fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
}
```

KEVIN POULSEN SECURITY FEB 22, 2014 11:27 AM

Behind iPhone's Critical Security Bug, a Single Bad 'Goto'

Like everything else on the iPhone, the critical crypto flaw announced in iOS 7 yesterday turns out to be a study in simplicity and elegant design: a single spurious "goto" in one part of Apple's authentication code that accidentally bypasses the rest of it.

ZD NET

tomorrow
belongs to those who embrace it
today

trending tech innovation business security advice buyin

/ tech

Home / Tech / Security

Apple's 'goto fail' tells us nothing good about Cupertino's software delivery process

The fact that Apple's infamous SSL validation bug actually got out into the real world is pretty terrifying.

Written by **Matt Baxter-Reynolds,** Contributor
March 19, 2014 at 3:00 a.m. PT

How Should Apple Have Found the Bug?

- Better code review?
- Better testing?
- Formal verification?
- Today's approach: *analyze the program*'s *source code*

Spot the Bug

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa,
                                 SSLBuffer signedParams,
                                 uint8_t *signature,
                                 UInt16 signatureLen) {
    OSStatus err;
    …
    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
    …
 fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
}
```

This code is unreachable. Isn't that a warning sign?

Hard-To-Find Bugs

- Often on a hard-to-execute codepath (need specific test cases)
- Can't actually test code exhaustively (too many paths, way too many states)
- Instead:
  - Identify relevant properties (e.g. code never dereferences NULL)
  - Try to prove program has those properties

Program Analysis

- Goal: answer questions about a program
- Examples:
  - Might this code ever dereference NULL?
  - Can I find any cases in which this code definitely divides by zero?

Now, for Some Theory

- The *halting problem* is a classic problem in CS
- Halting problem: Determine whether a program will halt on a specific input
  - Want: H such that $H(P(x))$ returns true if and only if $P(x)$ terminates
- Note: H has to work for *all* programs P and inputs x
- Halting problem was proven undecidable by Church in 1935 and Turing in 1936 (take CSE 105 to see the proof)

Rice's Theorem (Henry Rice, 1953)

- "Any nontrivial property about the language recognized by a Turing machine is undecidable."
- Implication: interesting static analyses will be imperfect (some false positives, false negatives, or sometimes not terminate)

Proof (by Contradiction)

- Suppose you have a function, **`divides_by_zero`**, that determines whether an input program divides by zero.

```
int oops(program p, input i) {
  p(i);
  return 5/0;
}
```

- if oops(p, i) divides by 0, p(i) must have halted
- if oops(p, i) does NOT divide by 0, p(i) loops forever
- If you want to know if p(i) halts, use this trick!

```
bool halts(program p, input i) {
  return divides_by_zero(oops(p,i));
}
```

halts() solves the halting problem…but we know that is impossible!

slide adapted from Aldrich and Le Goues

Soundness and Completeness

- A **sound** analysis finds all bugs (in a category of bugs).
  - No false negatives (doesn't fail to find a bug)
- A **complete** analysis only reports bugs (in a category of bugs).
  - No false positives (doesn't report bogus bugs)
- Generally, analyses are either **unsound** or **incomplete** (or both!)

Trust

- If a sound analysis says a program is safe, it is
  - (won't miss bugs)
- If a complete analysis reports a bug, the program is buggy
  - (won't report bogus bugs)

Static Analysis vs. Dynamic Analysis

- **Static analysis** is the analysis of programs *without* executing them
  - Usually want to find bugs or prove safety properties (the absence of bugs)
  - Often, static analyses can be made sound
- **Dynamic analysis** allows running programs
  - Dynamic analyses are more likely to be complete (only report bugs)

Static Analysis

- Key properties:
  - Liveness: "this good thing eventually happens" (e.g. server generates a response)
  - Safety: "this bad thing never happens" (e.g. dividing by zero)

Example

```
def n2s(n: int, b: int):
  if n <= 0: return '0'
  r = ''
  while n > 0:
    u = n % b
    if u >= 10:
      u = chr(ord('A') + u-10)
    n = n // b
    r = str(u) + r
  return r
```

- What types can 'u' have at each line?
- Can 'u' be negative?
- Will `n2s` always return a value?
- Can there be division by zero?
- Will the returned value ever include a '-'?

Example credit: Hilton et al.

Static Analysis Techniques

- Linters
  - Shallow syntax analysis (unsound, incomplete, unclear properties)
- Type checking (lots of research here)
  - Ensures program has well-defined semantics
- Data flow analysis, abstract interpretation (lots of research here too)
  - Is a[i] always within bounds?
  - Typical answers: "yes", "no", "maybe"

Sound Analyses…

A. Only report problems that occur in practice, but may miss some bugs

B. Only find some bugs in a given class and may report problems that will not occur in practice

C. Find all bugs in a given class, but may report problems that will not occur in practice

D. Find all bugs in a given class and only report problems that really occur

E. Find all bugs, so can't exist in real life

Sound Analyses…

A. Only report problems that occur in practice, but may miss some bugs

B. Only find some bugs in a given class and may report problems that will not occur in practice

C. Find all bugs in a given class, but may report problems that will not occur in practice

D. Find all bugs in a given class and only report problems that really occur

E. Find all bugs, so can't exist in real life

Complete Analyses…

A. Only report problems that occur in practice, but may miss some bugs

B. Only find some bugs in a given class and may report problems that will not occur in practice

C. Find all bugs in a given class, but may report problems that will not occur in practice

D. Find all bugs in a given class and only report problems that really occur

E. Find all bugs, so can't exist in real life

Complete Analyses…

A. **Only report problems that occur in practice, but may miss some bugs**

B. Only find some bugs in a given class and may report problems that will not occur in practice

C. Find all bugs in a given class, but may report problems that will not occur in practice

D. Find all bugs in a given class and only report problems that really occur

E. Find all bugs, so can't exist in real life

Pattern-Based Bug Detection

- e.g. SpotBugs
- Example: if a method acquires a lock, it should release it on all paths

```
Lock l = ...;
l.lock();
try {
    // do something
    l.unlock();
}
```

Oops! `l` remains locked if an exception is thrown

```
Lock l = ...;
l.lock();
try {
    // do something
} finally {
    l.unlock();
}
```

Tradeoffs

- Analysis must be super fast
- In general, these pattern-based detectors are **unsound** and **incomplete**
- Google recommends static analyzers have < 10% false positives [Sadowski]
  - Otherwise developers will turn them off!

https://abseil.io/resources/swe-book/html/ch20.html https://checkerframework.org/tutorial/webpages/get-started-cmd.html

Type-Based Approaches

- Idea: Extend the type system to enable reasoning about important properties

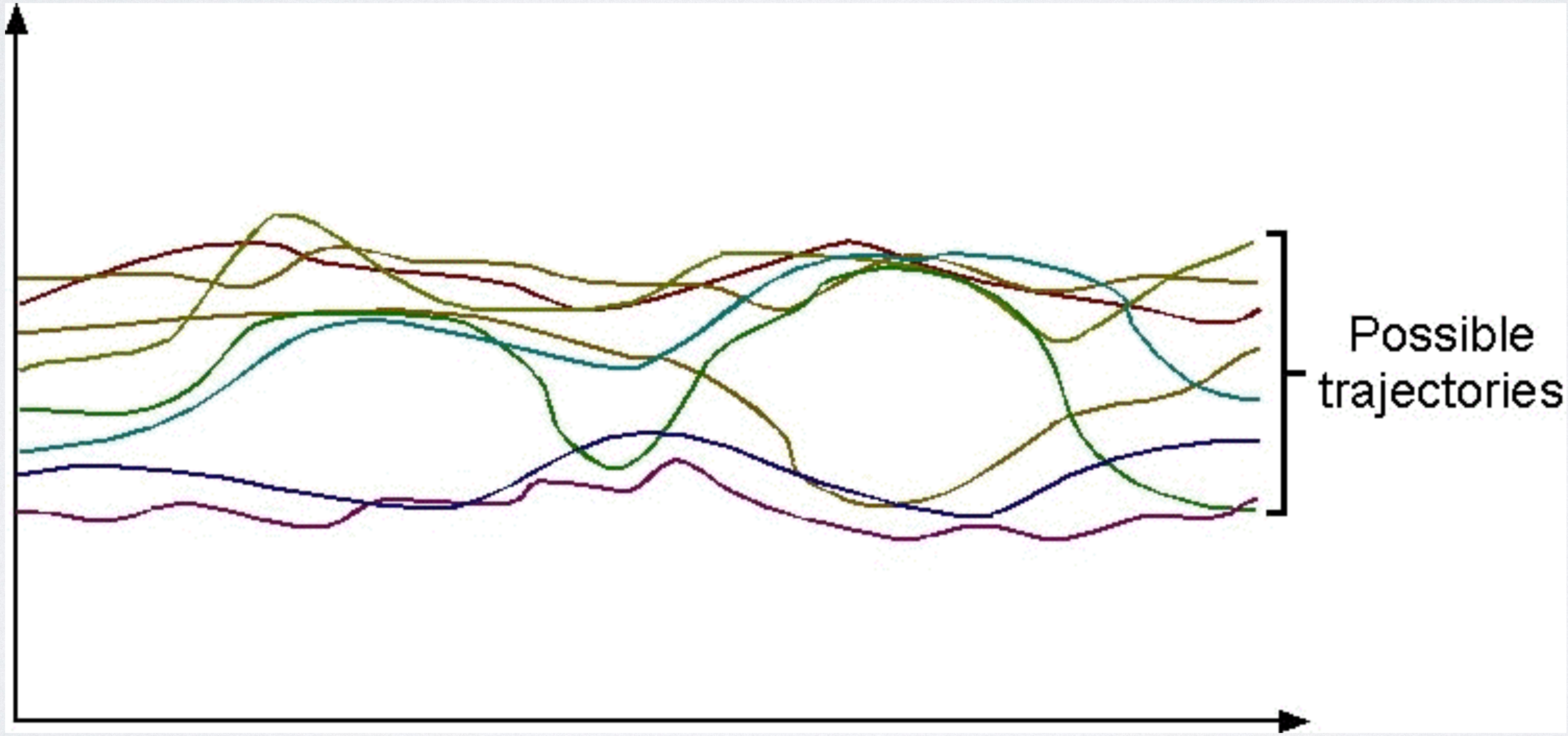
```
public class NullnessExample {
    public static void main(String[] args) {
        Object myObject = null;
        System.out.println(myObject.toString());
    }
}
```

```
$ javacheck -processor org.checkerframework.checker.nullness.NullnessChecker NullnessExample.java
```

```
NullnessExample.java:9: error: [dereference.of.nullable] dereference of possibly-null reference myObject
        System.out.println(myObject.toString());
                           ^
1 error
```

Reasoning About Behavior

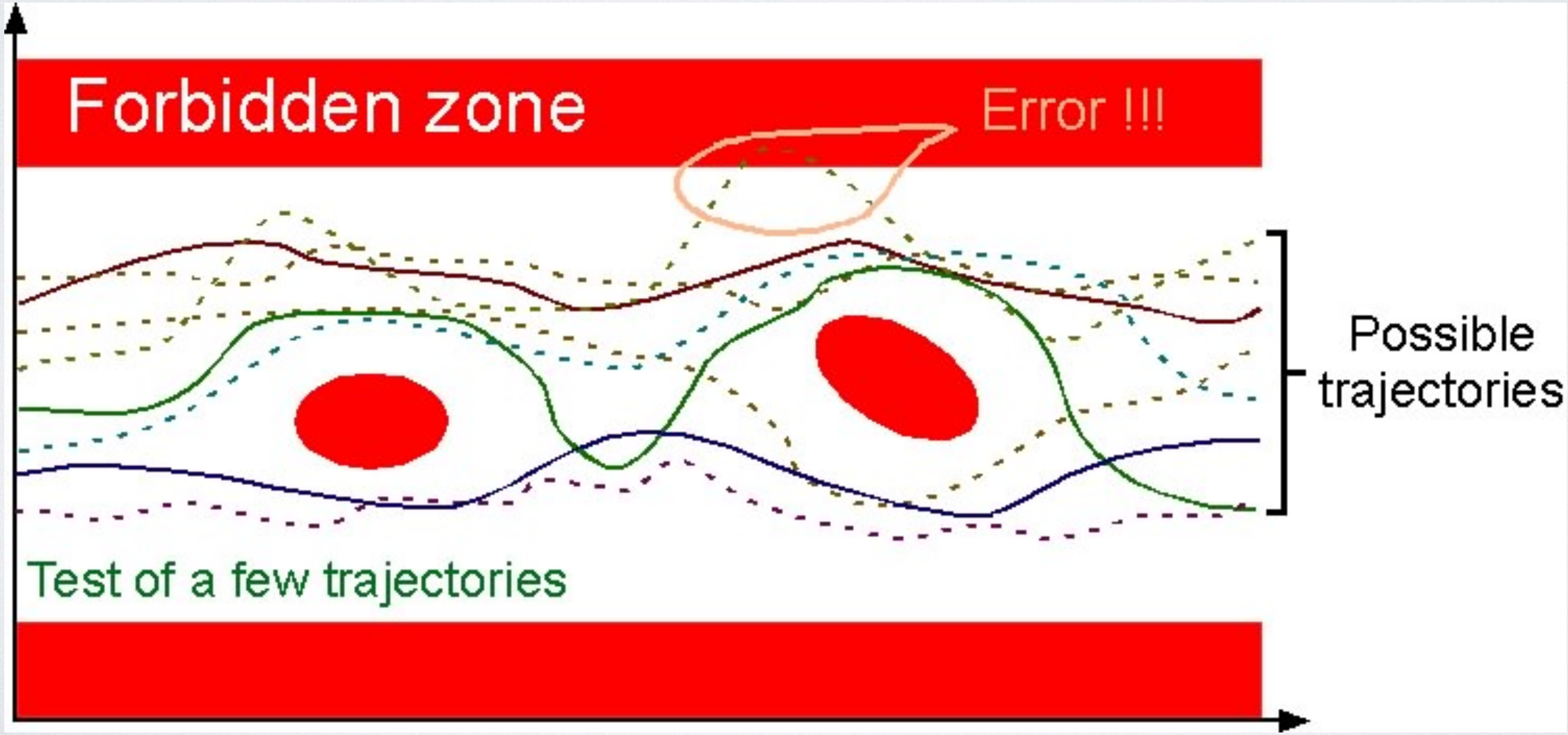
- Concrete semantics: all possible executions of a program



https://www.di.ens.fr/~cousot/AI/IntroAbsInt.html

Testing

- Can only test some of the possible trajectories



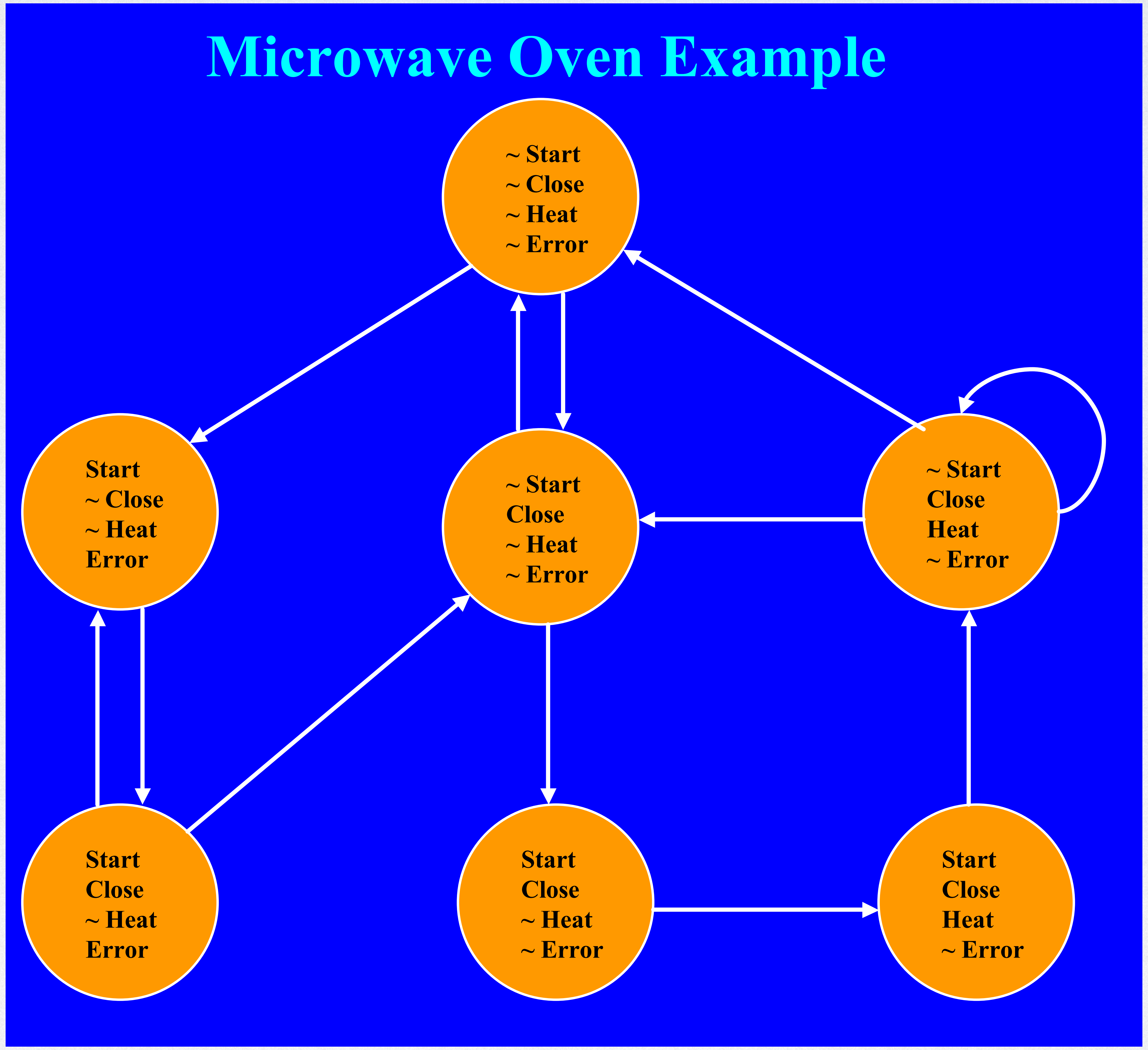
Possible Approaches

- Model Checking
  - Consider program to be a state machine
  - See if any possible state is "bad" (violates some safety property)
  - Draw transition diagram; make sure transitioning to bad state is impossible
- Abstract interpretation
  - Analyze program, annotating variables with information about possible values

Model Checking

- Goal: Explore all possible execution paths (via logic)
- Problem: too many execution paths (loops, recursion)
- Approach: *bounded* model checking (execute loops at most N times)

Model Checking Example (Ed Clarke)



Microwave Specification (Clarke)

- The oven doesn’t heat up until the door is closed.
- Not heat_up holds until door_closed
  - (~ heat_up) U door_closed

Model Checking Formalization (Clarke)

- Let M be a state-transition graph.
- Let f be the specification in temporal logic.
- Find all states s of M such that $M, s \models f$.

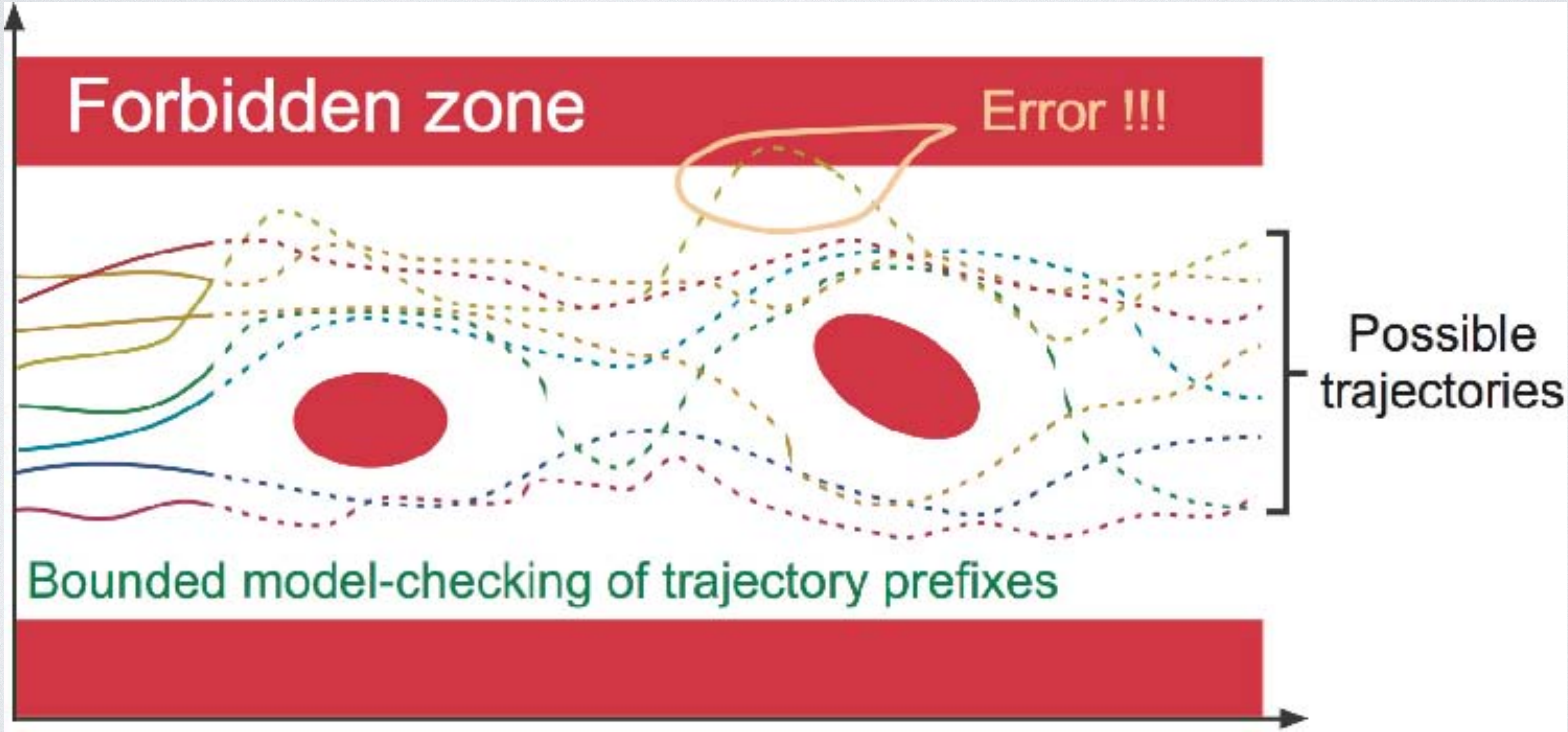
Tradeoffs

- Advantages: don't have to write proofs
- Disadvantages: state explosion; have to formally specify desired properties

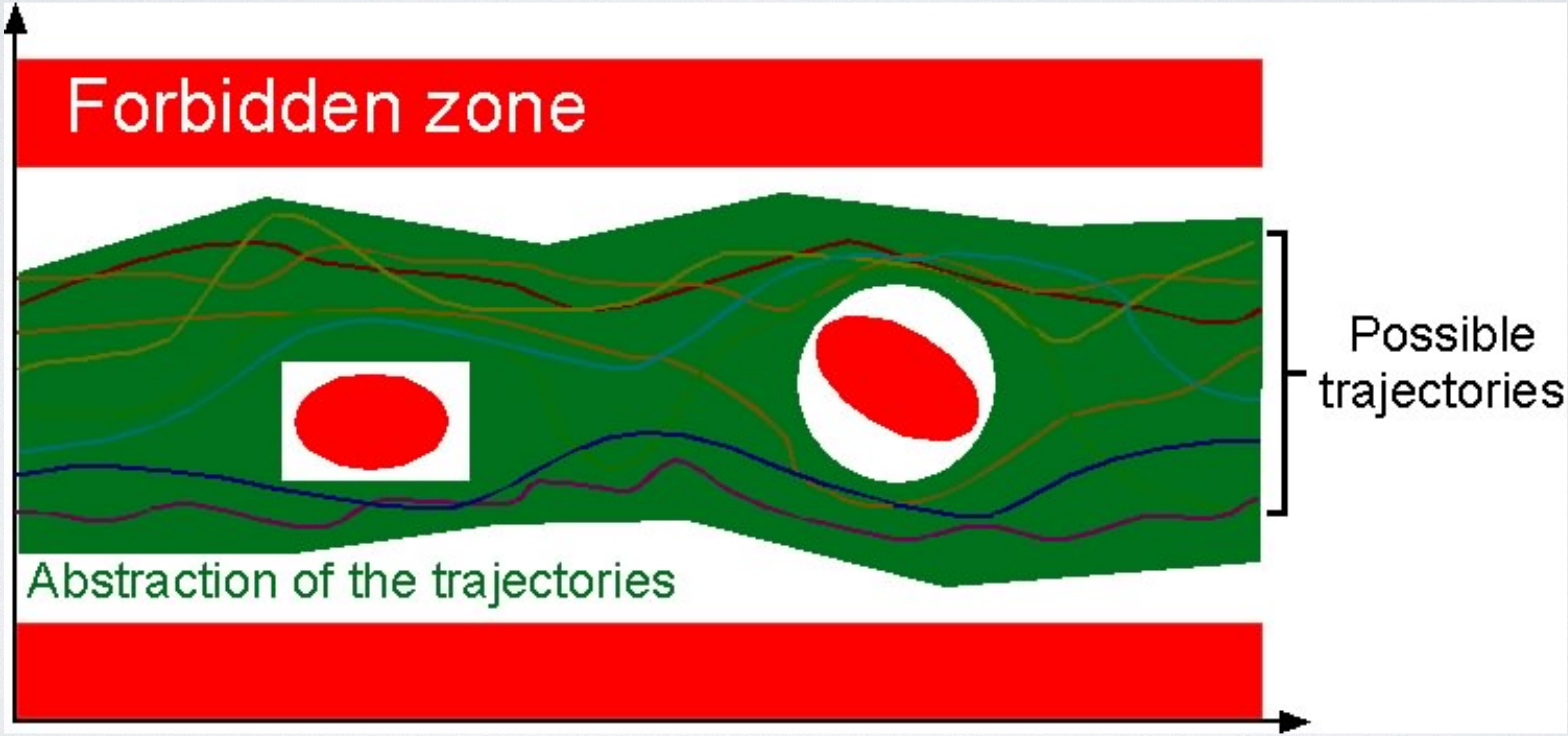
Model Checking Success Story

- In early 2000s: Windows users were plagued by blue screens of death
- Most common cause: driver bugs (not Microsoft's fault)
- Solution: model check drivers

Bounded Model Checking



Abstract Interpretation



Example: Numerical Intervals

- Ideally: figure out what values variables can have
  - But that requires running the program with all inputs ☹️
- Instead, track bounds [L, H] for each variable

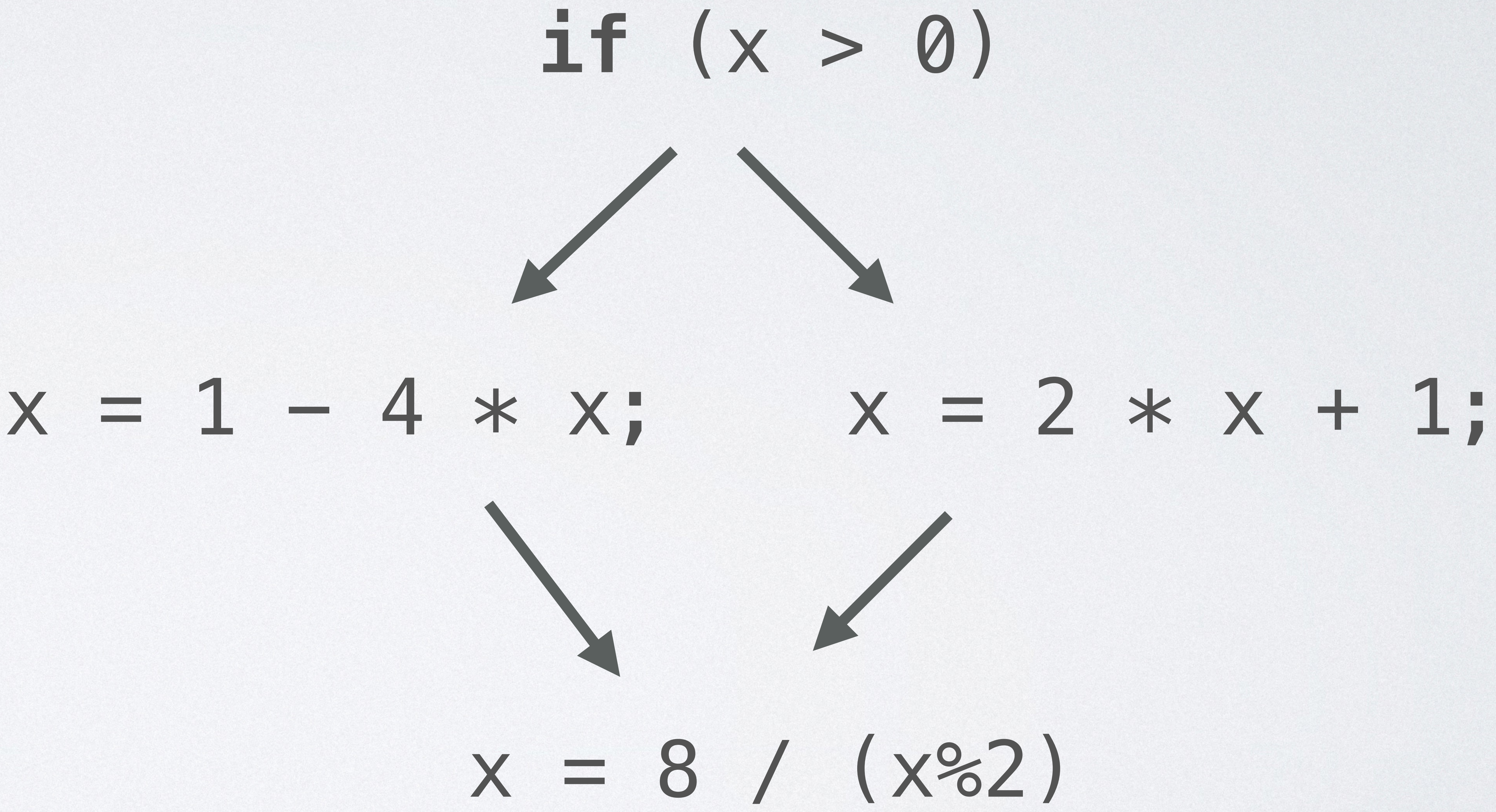
Will This Code Divide by Zero?

Source code

```
if (x > 0) {
    x = 2 * x + 1;
}
else {
   x = 1 - 4 * x;
}

x = 8 / (x%2)
```

Control Flow Graph



Defining an Abstract Domain

- We need to know if (x % 2) could be 0
- Let's track whether x could be even or odd.
  - Don't track all the values x could have.
- Abstract domain: {even, odd}

Analysis

{-∞, ∞}; {even, odd} **Precondition**

```
if (x > 0)
```

{-∞, ∞}; {even, odd} **Postcondition**

{-∞, 0}; {even, odd}

```
x = 1 - 4 * x;
```

{1, ∞}; {odd}

{1, ∞}; {even, odd}

```
x = 2 * x + 1;
```

{3, ∞}; {odd}

{1, ∞}; {odd}

```
x = 8 / (x%2)
```

Abstract Interpretation Uses Abstract Domains to…

A. Store concrete program states for exhaustive analysis.

B. Reduce the number of cases that must be reasoned about

C. Ensure a program executes faster by precomputing all possible outputs.

D. Simulate program execution for every possible input combination.

E. Identify the most efficient algorithm for solving a given problem.

Conclusion

- We can find lots of bugs by analyzing code
- But analyses are generally unsound, incomplete, or both
- Software engineers hate false positives, so choose analyses wisely