Foundations of Software Engineering

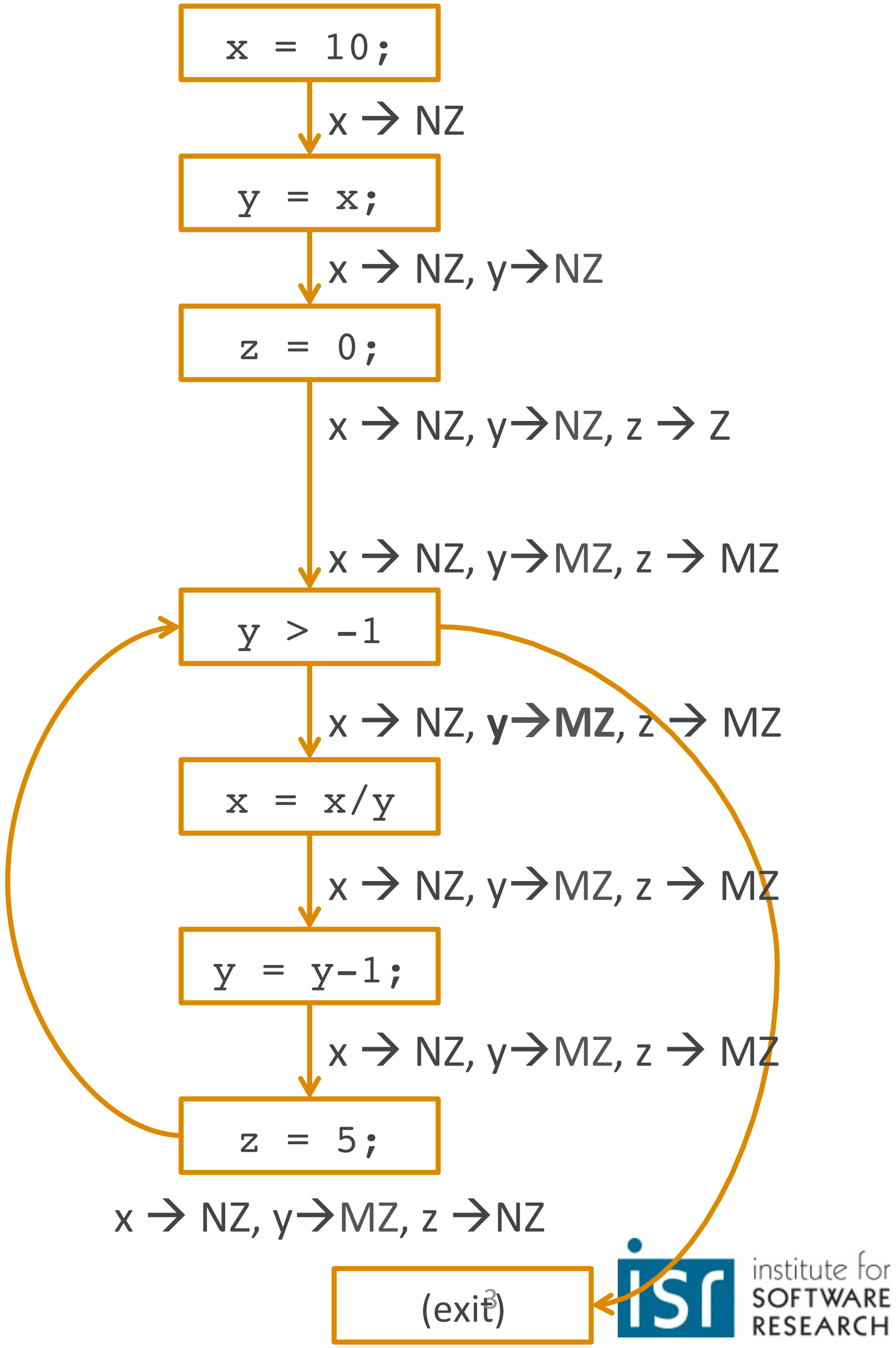
Lecture 14 – Static Analysis 2, or: the Halting Problem Strikes Back

institute for SOFTWARE RESEARCH

Learning goals

- Explain at a high level why static analyses cannot be sound, complete, and terminating.
- Give an example of a technique to increase precision, and assess tradeoffs in analysis design.
- Understand symbolic execution and its applicability, especially when combined with dynamic techniques for test case generation.
- Characterize and choose between tools that perform static analyses.

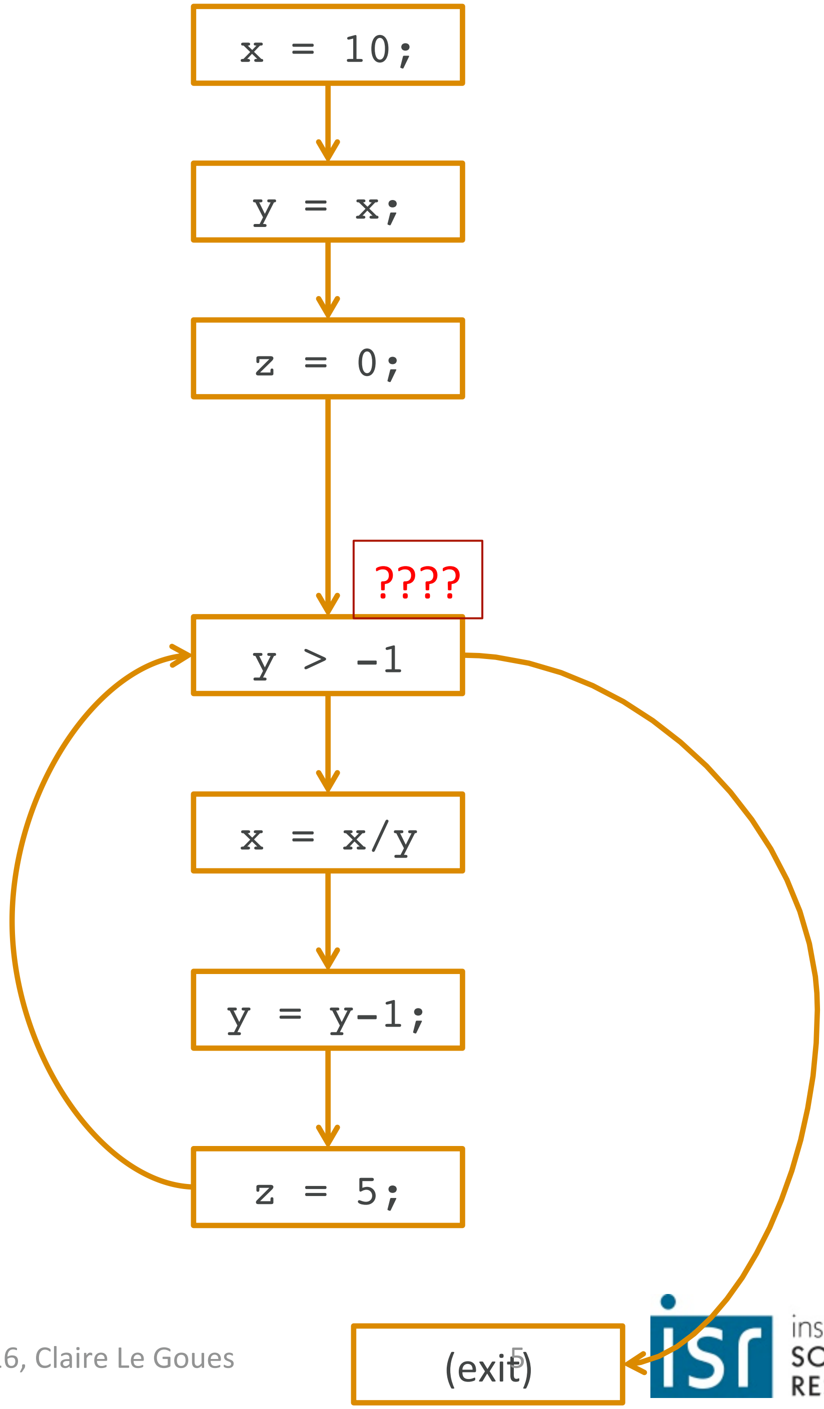
Warning! Possible division by zero error!



Order doesn’t actually matter

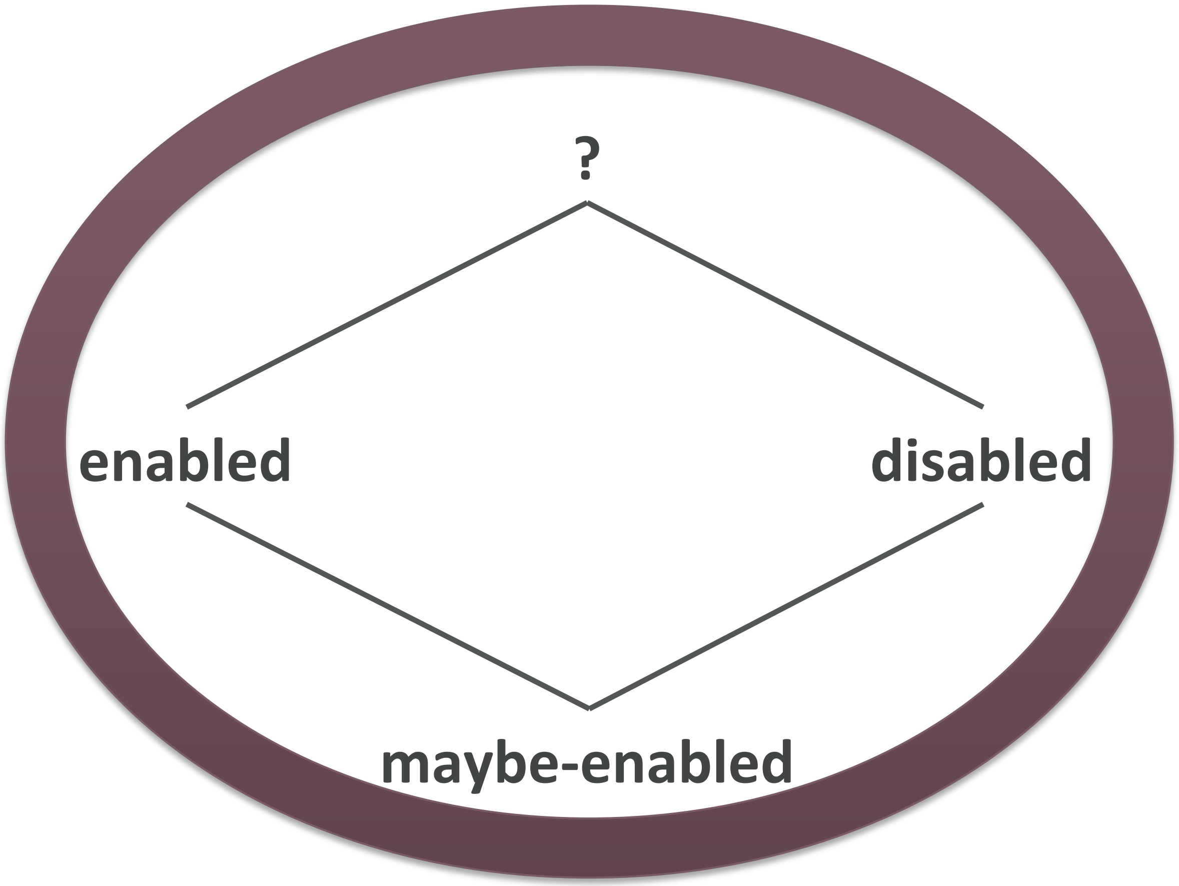
- Can process instructions in whatever order we want, until the information doesn’t change over the whole program.
- But, there’s a problem…

```
x = 10;
y = x;
z = 0;
while (y > -1) {
   x = x/y;
   y = y-1;
   z = 5;
}
```



WHAT IS THE INPUT STATE TO AN INSTRUCTION THAT HAS A PREDECESSOR WE HAVEN'T PROCESSED YET??

Example: interrupt checker



Complete lattices

- The $\bot$ value: bottom, the opposite of top. $\forall\, l,\ \bot \sqcup l = l$
- Join function: always moves up.

WHEN DO WE STOP?

Termination intuition

- A **fixed point** of a function is a data value *v* that a function maps to itself:
  - $f(v) = v$
- The flow function is the mathematical function.
- The dataflow analysis state at each fix point is the data values.

Simple algorithm

```
1. for all node indexes i do
2.     input[i] = $\bot$
3. input[ firstInstruction ] = $initial_A$
4. while not at fixed point
5.     pick an instruction i
6.     output = flow(i, input[i])
7.     for j in succs ( i )
8.         input[j] = input[j] $\sqcup$ output
```

Example of Worklist

1. [a := 0]
2. [b := 0]
3. while [a < 2] do
4.   [b := a];
5.   [a := a + 1];
6. [a := 0]

```
1.  for all node indexes i do
2.    input[i] = ⊥
3.  input[ firstInstruction ] =
    initial_A
4.  while not at fixed point
5.    pick an instruction i
6.    output = flow(i, input[i])
7.    for j in succs ( i )
8.      input[j] = input[j] ⊔ output
```

Kildall's Worklist Algorithm

```
1.  worklist = new Set();
2.  for all node indexes i do
3.     input[i] = $\bot_A$;
4.  input[entry] = $\text{initial}_A$;
5.  worklist.add(all nodes);
6.  while (!worklist.isEmpty()) do
7.     i = worklist.pop();
8.     output = flow(input[i], i);
9.     for j $\in$ succ(i) do
10.       if ! (output $\sqsubseteq$ input[j])
11.          input = input[j] $\sqcup$ output
12.          worklist.add(j)
```

Note on line 5: it's OK to just add entry to worklist if the flow functions cannot return bottom, which is true for our example but not generally.

The Bad News: Rice's Theorem

"Any nontrivial property about the language recognized by a Turing machine is undecidable.“

Henry Gordon Rice, 1953

Every static analysis is necessarily incomplete or unsound or undecidable (or multiple of these)

WHAT DOES THAT MEAN, AND ALSO, WHY?

Let’s translate…

Anything interesting

"Any nontrivial property about the language recognized by a Turing machine is un cidabl

Program

Henry G 953

Computer

Why?Infinite loops.

- I have a program, and it takes input.
- That program is written in a reasonable programming language, so it has loops.
- One way a program with loops can go horrifically awry is that it can loop infinitely.
- It's often hard to tell the difference between a program that just takes a long time to execute, and a program that's stuck in an infinite loop.

Computability theory says…

- **Halting problem:** the problem of determining whether a given program will halt/terminate on a given input.
- A *general* algorithm that solves this problem is impossible.
  - More specifically: it's undecidable (it's possible to get a *yes* answer, but not a *no* answer).
  - (sometimes you can use heuristics, but solving it generally for all programs is still out.)
- The proof here is very elegant. But trust me: this problem is extremely impossible.

OK, so?

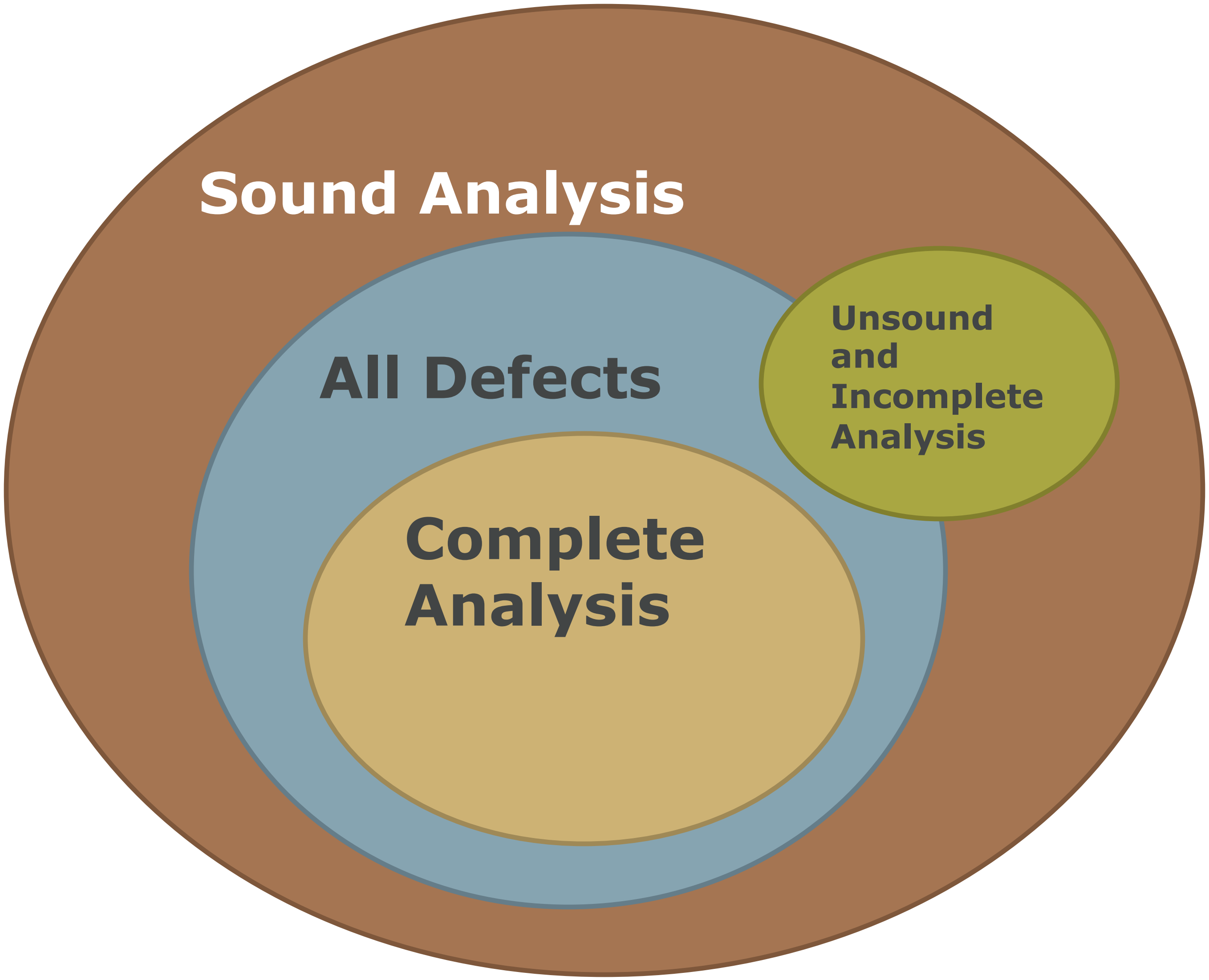
- If you could always statically tell if any program had a non-trivial property (never dereferences null, always releases all file handles, etc, etc), you could also generally solve the halting problem.
- …but the halting problem is *definitely* impossible.
- So: no static analysis is perfect. They will always have false positives or false negatives (or both).
- *All tools make tradeoffs.*

institute for SOFTWARE RESEARCH

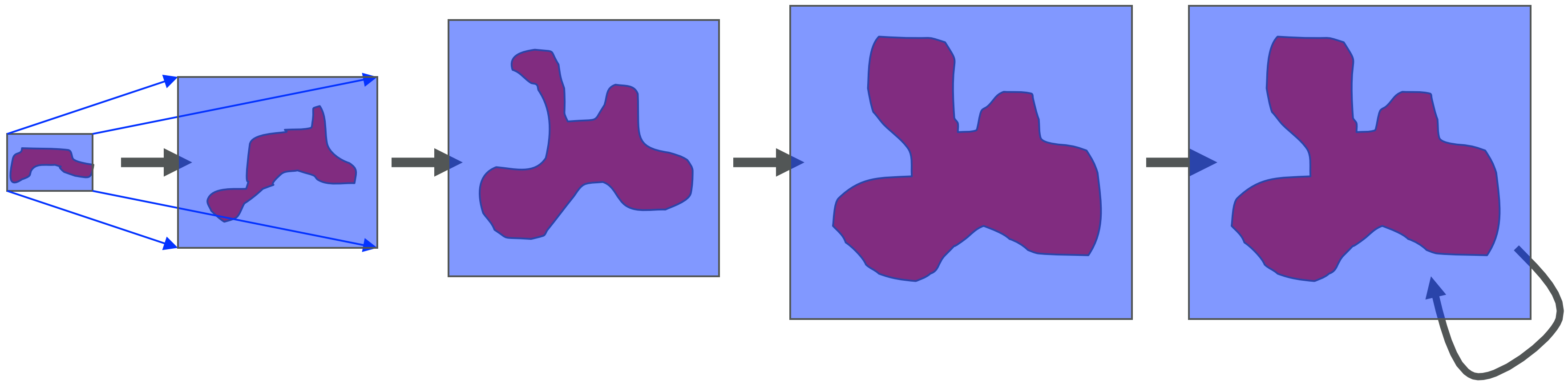
	Error exists	No error exists
Error Reported	True positive (correct analysis result)	False positive
No Error Reported	False negative	True negative (correct analysis result)

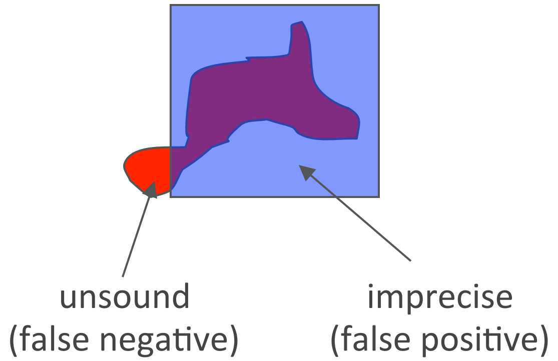
Sound Analysis:
- reports all defects
- -> no false negatives
- typically overapproximated

Complete Analysis:
- every reported defect is an actual defect
- -> no false positives
- typically underapproximated


Sound Analysis
All Defects
Unsound and Incomplete Analysis
Complete Analysis

Soundness and precision





🟥 Program state covered in actual execution

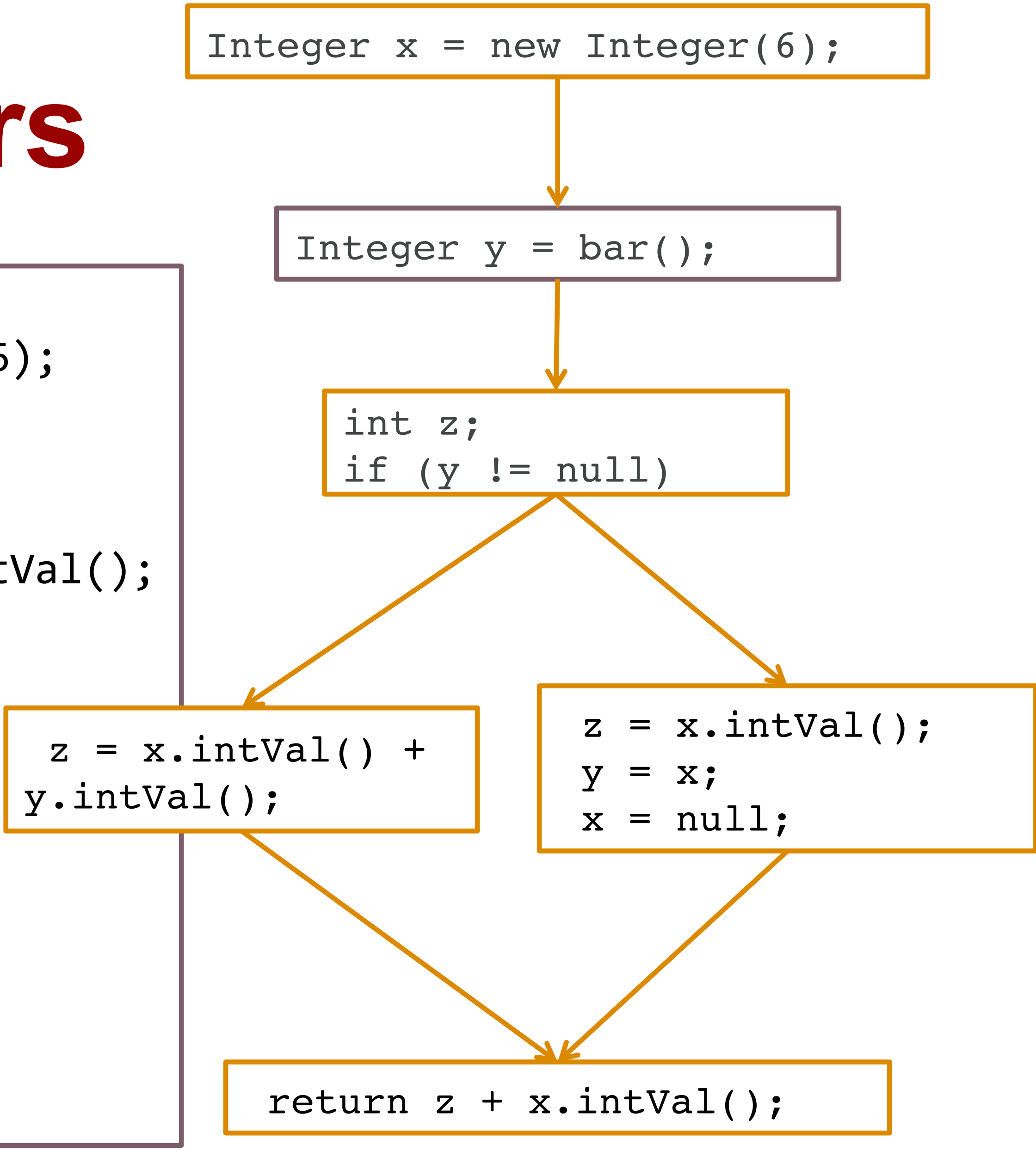
🟦 Program state covered by abstract execution with analysis

Sound vs. Heuristic Analysis

- Heuristic Analysis
  - FindBugs, checkstyle, …
  - Follow rules, approximate, avoid some checks to reduce false positives
  - May report false positives and false negatives
- Sound Static Analysis
  - Type checking, Not-Null, … (specific fault classes)
  - Sound abstraction, precise analysis to reduce false positives

Null pointers

```
int foo() {
    Integer x = new Integer(6);
    Integer y = bar();
    int z;
    if (y != null)
        z = x.intVal() + y.intVal();
    } else {
        z = x.intVal();
        y = x;
        x = null;
    }
    return z + x.intVal();
}
```



```
Integer x = new Integer(6);

Integer y = bar();

int z;
if (y != null)

z = x.intVal() +
y.intVal();

z = x.intVal();
y = x;
x = null;

return z + x.intVal();
```

What about that function call?

1. If you're worried about totally wacky control flow (exceptions, longjumps), they can be modeled in wackier/more complicated control flow graphs.
2. Ignore it by assuming that all functions return and tempering your claim: "assuming the program terminates, the analysis soundly computes..."
   – Most people don't bother; this is basically assumed.

File open/close

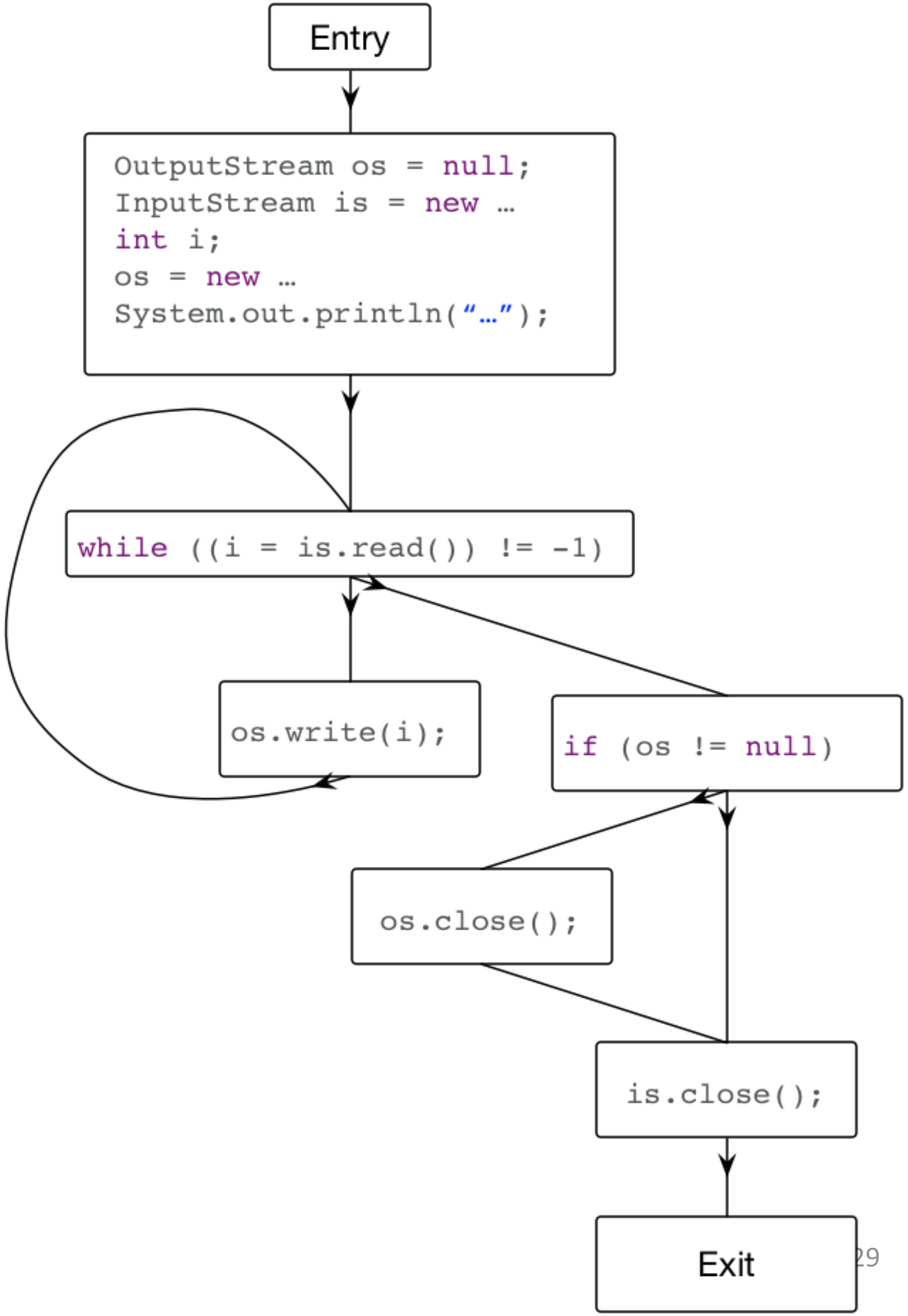
- Abstract domain: file open, file closed, file maybe-open.
- Transfer and joins left as exercise to the reader…

```java
public class StreamDemo {
  public static void main(String[] args) throws Exception {
     OutputStream os = null;
     InputStream is = new FileInputStream("in.txt");
     int i;
     try {
      os = new FileOutputStream("out.txt");
      System.out.println("Copying in progress...");
      while ((i = is.read()) != -1) {
         os.write(i);
      }
      if (os != null) {
        os.close();
       }
     } catch (IOException e) {
        e.printStackTrace();
     }
      is.close();
    }
}
```

Design choices: representation and abstract domain

- What if we don't model the try/catch?

institute for SOFTWARE RESEARCH



```
OutputStream os = null;
InputStream is = new …
int i;
os = new …
System.out.println("…");
```

```
while ((i = is.read()) != -1)
```

```
os.write(i);
```

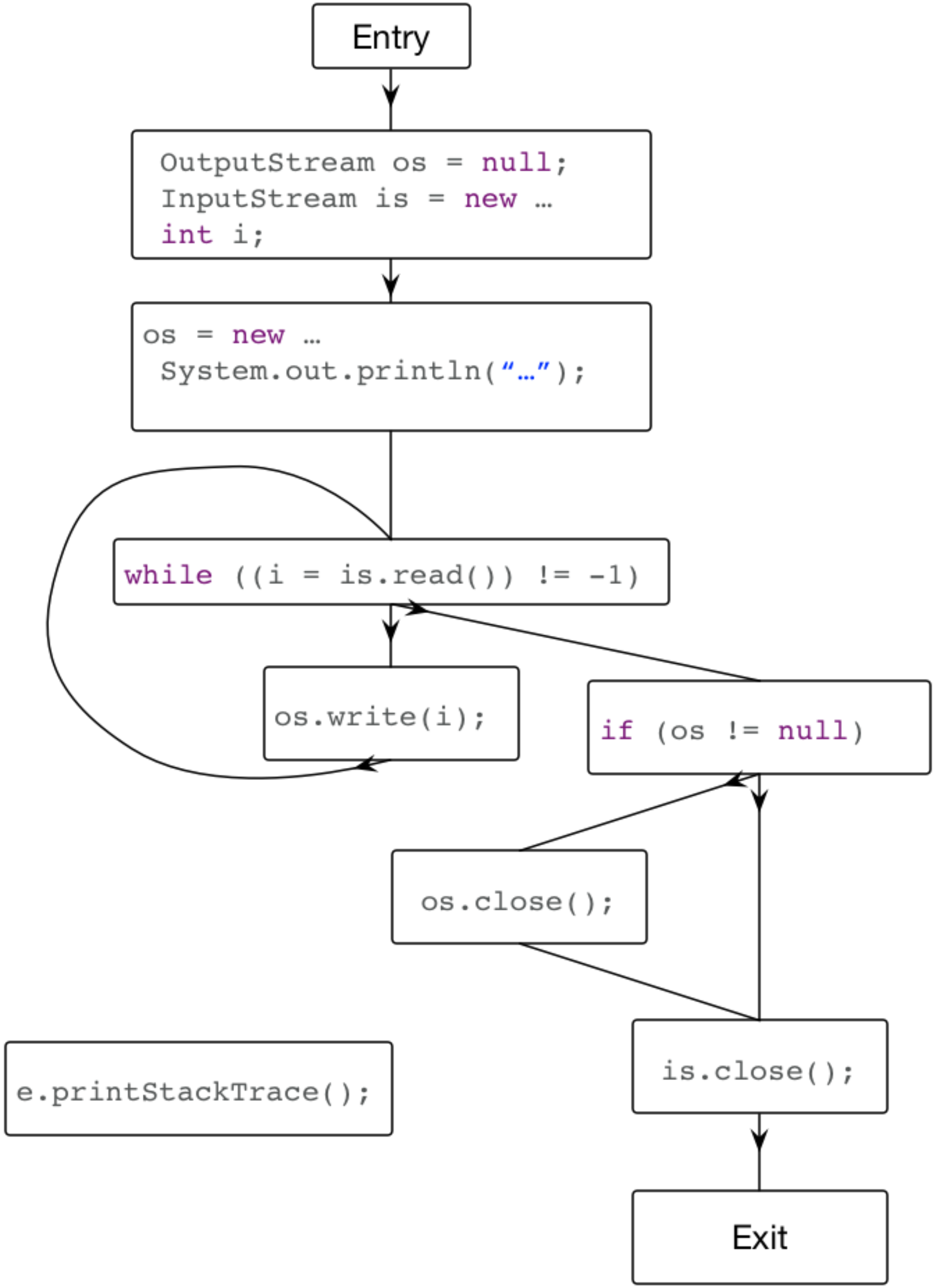
```
if (os != null)
```

```
os.close();
```

```
is.close();
```

Design choices: representation and abstract domain

- What if we don't model the try/catch?
- If we do…how should we include it?


Entry
OutputStream os = null;
InputStream is = new …
int i;
os = new …
System.out.println("…");
while ((i = is.read()) != -1)
os.write(i);
if (os != null)
os.close();
is.close();
e.printStackTrace();
Exit

Design choices: representation and abstract domain

- What if we don't model the try/catch?
- If we do…how should we include it?
- …what about non-IOExceptions?
- Broader question: How precisely should we model semantics?
  - E.g., Of instructions, of conditional checks, etc.

Upshot: analysis as approximation

- Analysis must approximate in practice
  - False positives: may report errors where there are really none
  - False negatives: may not report errors that really exist
  - All analysis tools have either false negatives or false positives
- Approximation strategy
  - Find a pattern P for correct code
    - which is feasible to check (analysis terminates quickly),
    - covers most correct code in practice (low false positives),
    - which implies no errors (no false negatives)
- Analysis can be pretty good in practice
  - Many tools have low false positive/negative rates
  - A sound tool has no false negatives
    - Never misses an error in a category that it checks

Symbolic Execution

- Execute program with symbolic inputs.

```
y = read()
y = 2 * y
if (y == 12)
   fail()
print("OK")
```

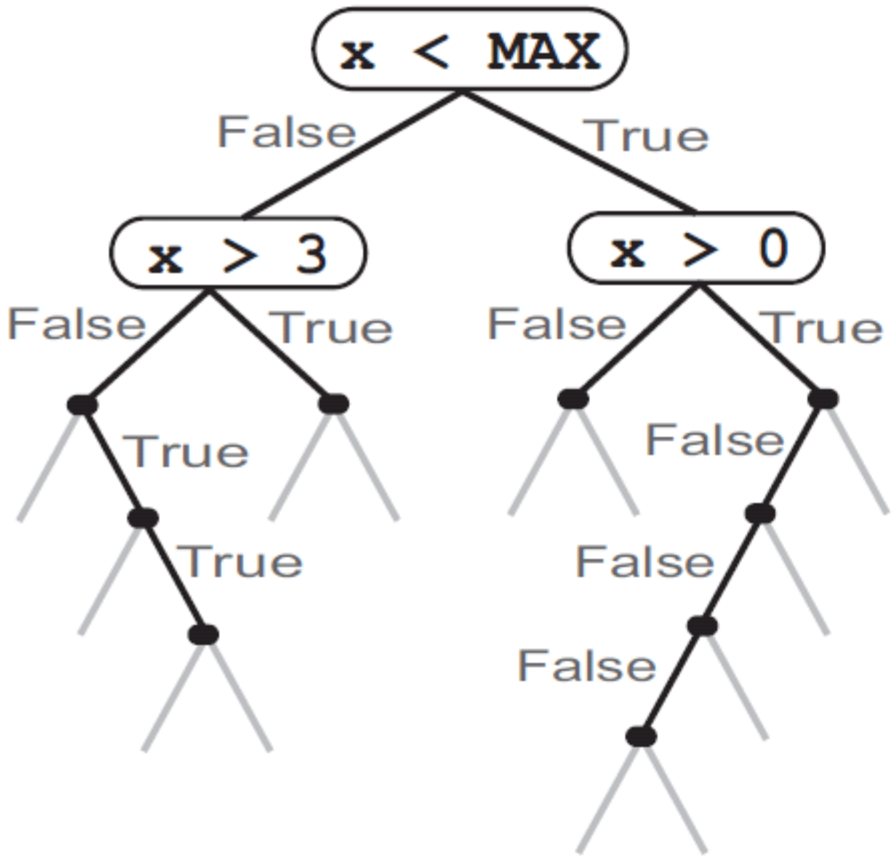
```
y = α
y = 2*α
Successful path
condition:
  y = 2*α
```

- Used for verification, test generation.

Symbolic Execution

- Exploring all paths

```
if (x<MAX) {
   if (x>0)
      …
   else
      …
} else {
   if (x>3)
      …
```



institute for SOFTWARE RESEARCH

Symbolic Execution: Limitations

- Path explosion
- Undecidable Path Constraints ($\alpha*\beta<10$)
- Nontermination with unlimited loop bounds (while (x<y))

Practical scalability today: ~10,000 lines of code

Dynamic Symbolic Execution

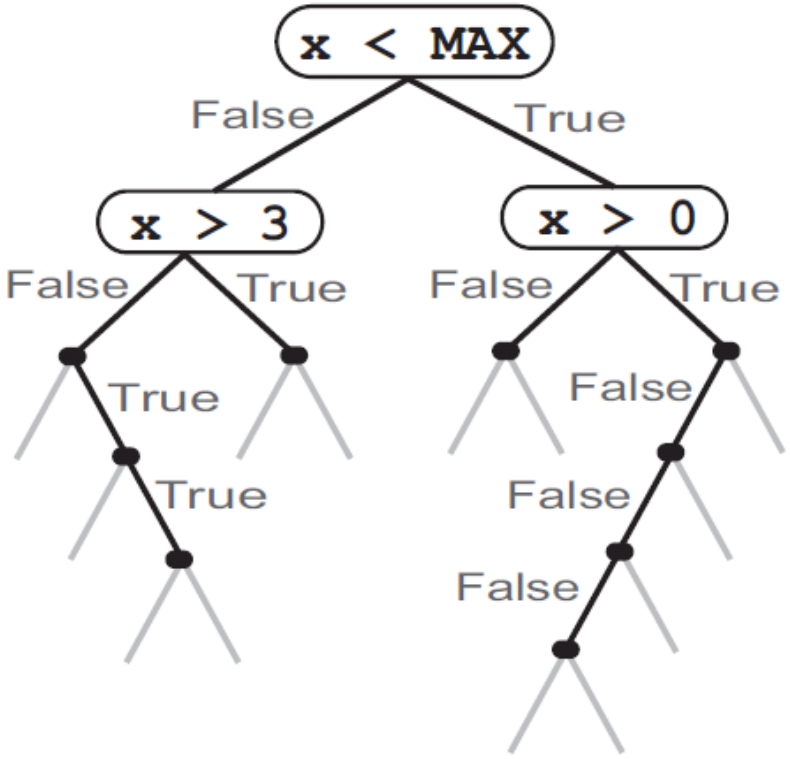
- Mixing Concrete and Symbolic Values
- Unsound -> Test Case Generation
- Given Unsolvable Constraint or Loop Bound: just guess one variable and continue

$\alpha*\beta<10$
$\alpha*2<10$

institute for SOFTWARE RESEARCH

Automatic white-box test generation

- Dynamic Symbolic Execution to guide Fuzz Testing
- Microsoft SAGE
  - In production on Office, Windows
  - 200+ machines
  - 3 B+ constraints



The general procedure

- Start with random inputs.
- Execute the program.
  - Identify the paths/decisions/statements covered by the test case.
  - Collect **path constraints** corresponding to the execution.
- Flip one of the constraints, ask a **constraint solver** to give new inputs to force the execution down a different path.

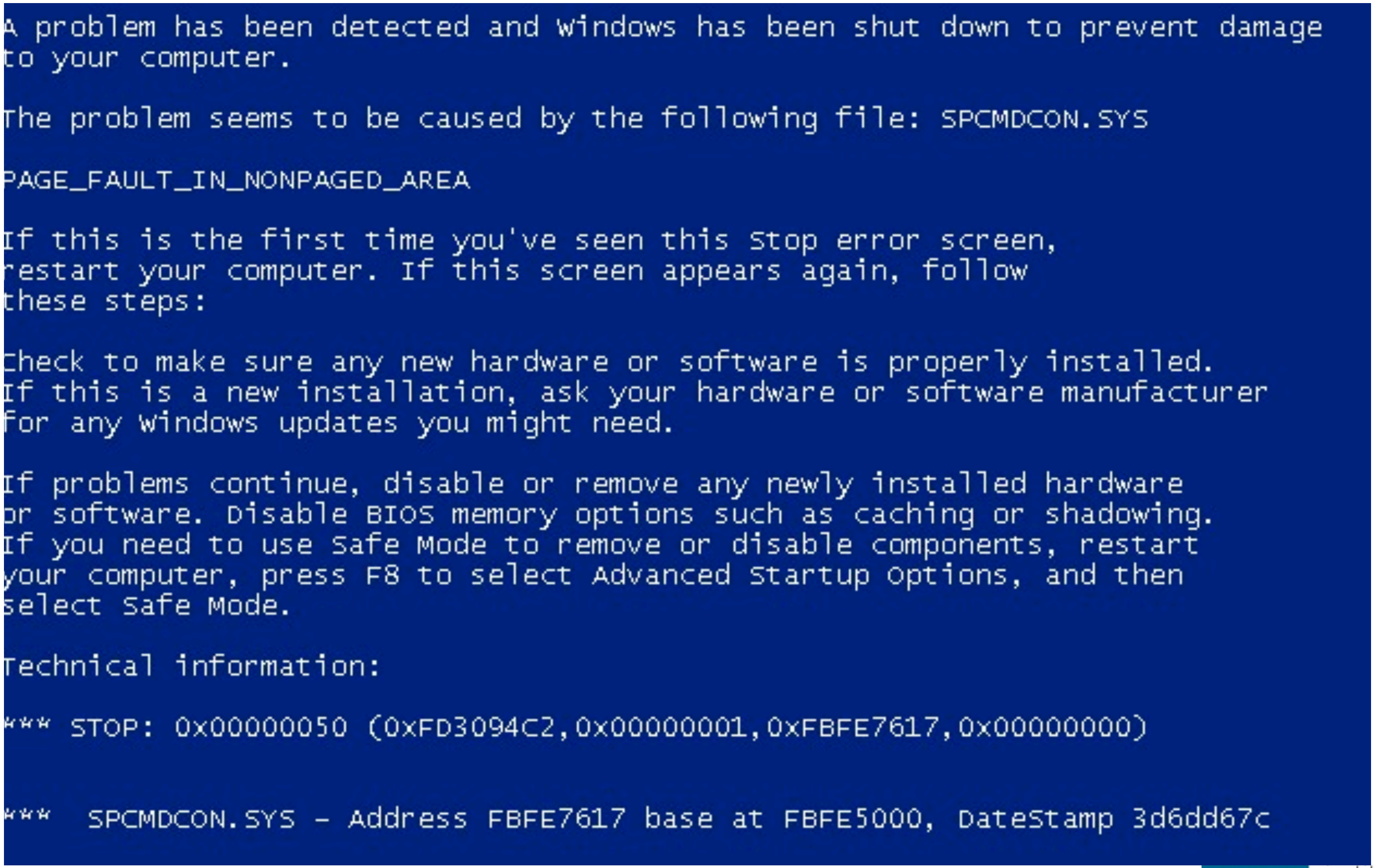
Making things better: termination

- Secret weapon: define your abstraction such that it is finite.
- If you come to a statement and you've already explored a given state for that statement, stop.
  - The analysis depends on the code and the current state; continuing the analysis from this program point and state would yield the same results.
- If the number of possible states isn't finite, you're stuck.
  - Your analysis may not terminate.
- Common solution: cap the number of paths/ loop iterations to 0, 1, or 2.

Check out…

- PEX: Automated White Box Testing for .NET
  - Technique out of Microsoft Research
  - Extension to Visual Studio
- Pex4Fun: educational programming web game based on PEX.

Why do Static Analysis



```
A problem has been detected and windows has been shut down to prevent damage
to your computer.

The problem seems to be caused by the following file: SPCMDCON.SYS

PAGE_FAULT_IN_NONPAGED_AREA

If this is the first time you've seen this Stop error screen,
restart your computer. If this screen appears again, follow
these steps:

Check to make sure any new hardware or software is properly installed.
If this is a new installation, ask your hardware or software manufacturer
for any Windows updates you might need.

If problems continue, disable or remove any newly installed hardware
or software. Disable BIOS memory options such as caching or shadowing.
If you need to use Safe Mode to remove or disable components, restart
your computer, press F8 to select Advanced Startup Options, and then
select Safe Mode.

Technical information:

*** STOP: 0x00000050 (0xFD3094C2,0x00000001,0xFBFE7617,0x00000000)


***  SPCMDCON.SYS - Address FBFE7617 base at FBFE5000, DateStamp 3d6dd67c
```

Quality assurance at Microsoft

- Original process: manual code inspection
  - Effective when system and team are small
  - Too many paths to consider as system grew
- Early 1990s: add massive system and unit testing
  - Tests took weeks to run
    - Diversity of platforms and configurations
    - Sheer volume of tests
  - Inefficient detection of common patterns, security holes
- Non-local, intermittent, uncommon path bugs
  Was treading water in Windows Vista development
- Early 2000s: add static analysis

Impact at Microsoft

- Thousands of bugs caught monthly
- Significant observed quality improvements
  - e.g. buffer overruns latent in codebaes
- Widespread developer acceptance
  - Check-in gates
  - Writing specifications

Tools: Compilers

- Type checking, proper initialization API, correct API usage

Program	**Compiler output**

`int add(int x,int y) {`
`  return x+y;`
`}`
 	
`void main() {`
`  add(2);`
`}`
	`$> error: too few arguments to function 'int add(int, int)'`

- Compile at a high warning level
  - **`$>gcc -Wall`**

Tools: lint and splint

- Lint was originally a static checker for C code
  - Flagged suspicious and non-portable constructs in C
  - Stronger checking than typical compiler
  - Also uses embedded annotations
  - Creates internal structures to analyze program state and detect problematic arrangements
    - Parse program and analyze state of variables, functions, etc.
- Splint (Secure Programming Lint) is modern version of lint
- "Lint-like" or "Lint" tools now refers to any tool that flags suspicious code usage.
- Some companies run such checkers at checkin.
  - (Ed note: except Apple, apparently??)

http://www.splint.org/

Splint Example

Code (ex.c)

```c
int main() {
  char c;
  while (c != 'x');
  {
    c = getchar();
    if (c = 'x')
      return 1;
  }
  return 0;
}
```

Splint output

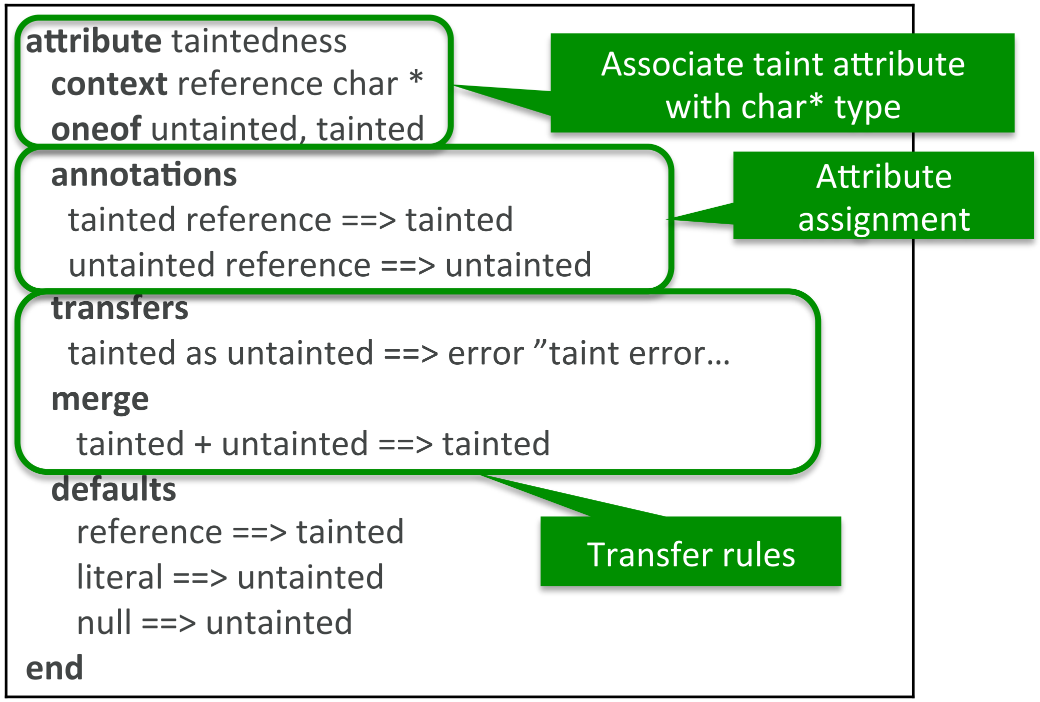
```
$> splint ex.c

Splint 3.1.1 --- 19 Jul 2006

ex.c:3:10: Variable c used before definition. An rvalue is used that may not be initialized to a value on some execution path. (Use -usedef to inhibit warning)

ex.c:3:10: Suspected infinite loop.  No value used in loop test (c) is modified by test or loop body. This appears to be an infinite loop. Nothing in the body of the loop or the loop test modifies the value of the loop test. Perhaps the specification of a function called in the loop body is missing a modification. (Use -infloops to inhibit warning)

ex.c:5:5: Assignment of int to char: c = getchar()
  To make char and int types equivalent, use +charint.
…
```

Splint is extendable

Extending Splint to Analyze Taintedness

- Tainting marks data as untrusted
  - Tainted data originates from the user/external environment
  - Mark (taint) data as untrusted and analyze program to determine how/where it is used
- We can extend splint to analyze taintedness at compile time

Tainted character pointers



```
attribute taintedness
  context reference char *
  oneof untainted, tainted
  annotations
    tainted reference ==> tainted
    untainted reference ==> untainted
  transfers
    tainted as untainted ==> error "taint error…
  merge
    tainted + untainted ==> tainted
  defaults
    reference ==> tainted
    literal ==> untainted
    null ==> untainted
end
```

Using the new definition in annotations

```
int printf  (/*@untainted@*/ char *fmt, ...);
```

http://www.splint.org/manual/html/sec10.html

Learning goals

- Explain at a high level why static analyses cannot be sound, complete, and terminating.
- Give an example of a technique to increase precision, and assess tradeoffs in analysis design.
- Understand symbolic execution and its applicability, especially when combined with dynamic techniques for test case generation.
- Characterize and choose between tools that perform static analyses.