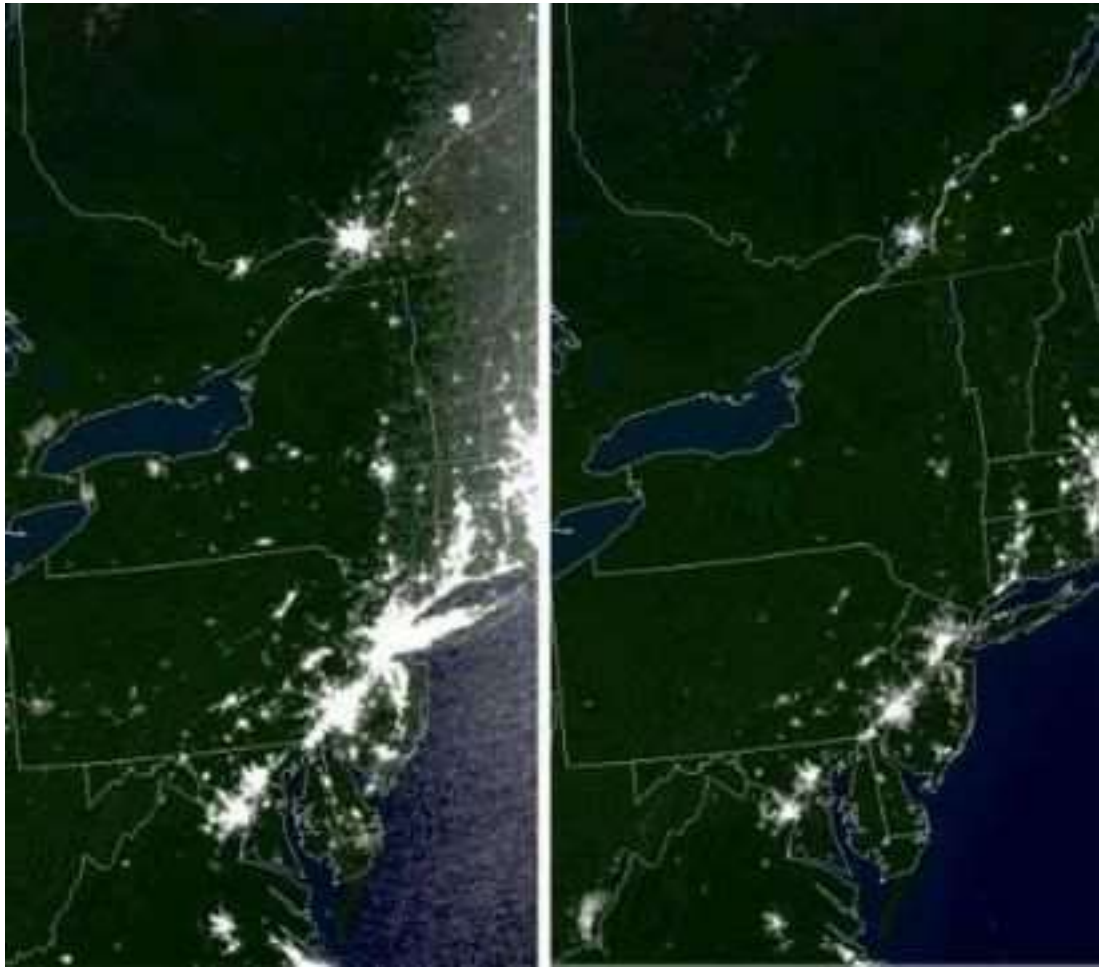
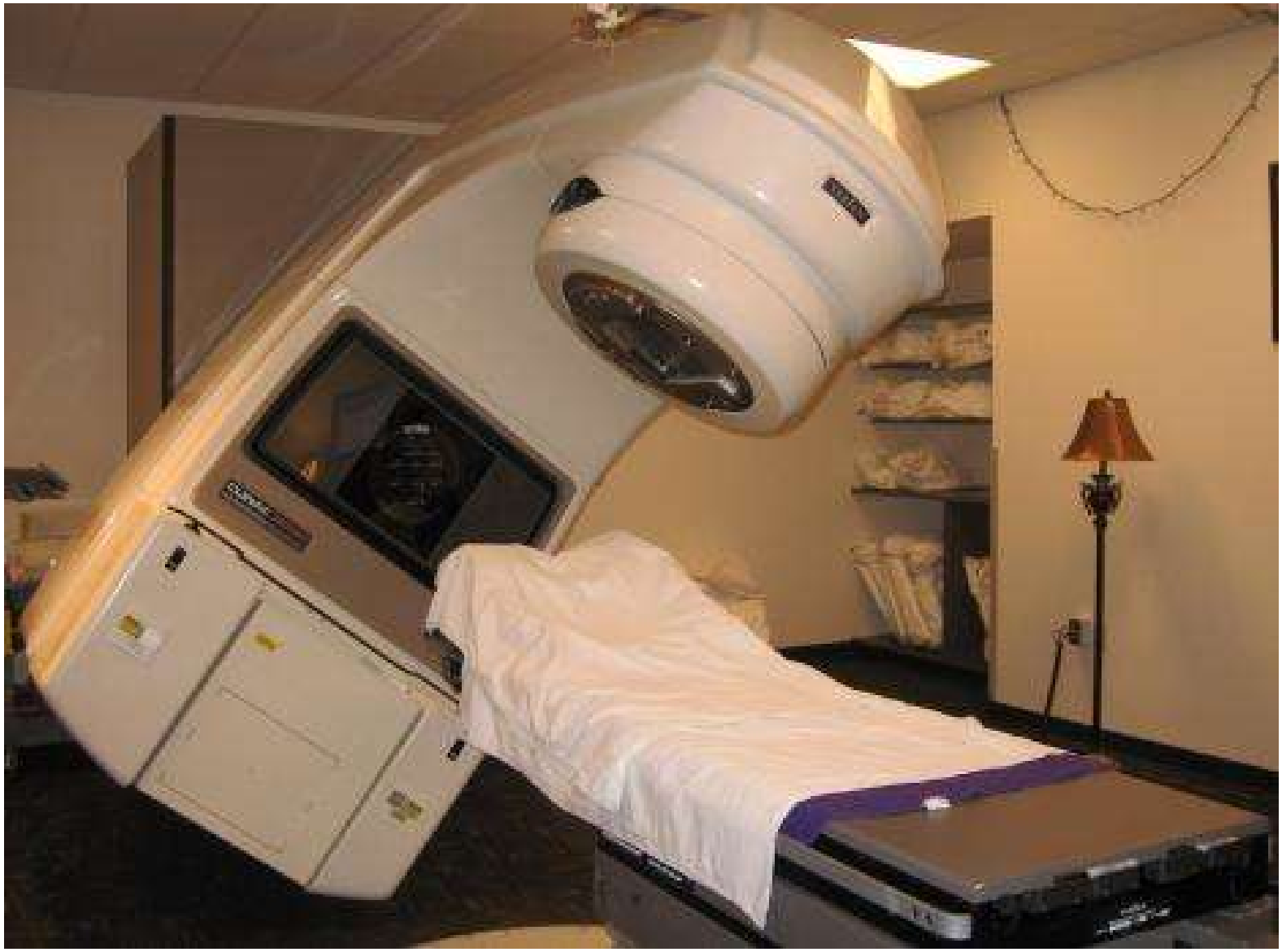
Introduction to Program Analysis

17-355/17-665/17-819: Program Analysis

Jonathan Aldrich and Claire Le Goues





Is there a bug in this code?

```
1./* from Linux 2.3.99 drivers/block/raid5.c */
2.static struct buffer_head *
3.get_free_buffer(struct stripe_head * sh,
4.                int b_size) {
5.  struct buffer_head *bh;
6.  unsigned long flags;
7.  save_flags(flags);
8.  cli(); // disables interrupts
9.  if ((bh = sh->buffer_pool) == NULL)
10.    return NULL;
11.  sh->buffer_pool = bh -> b_next;
12.  bh->b_size = b_size;
13.  restore_flags(flags); // re-enables interrupts
14.  return bh;
15.}
```

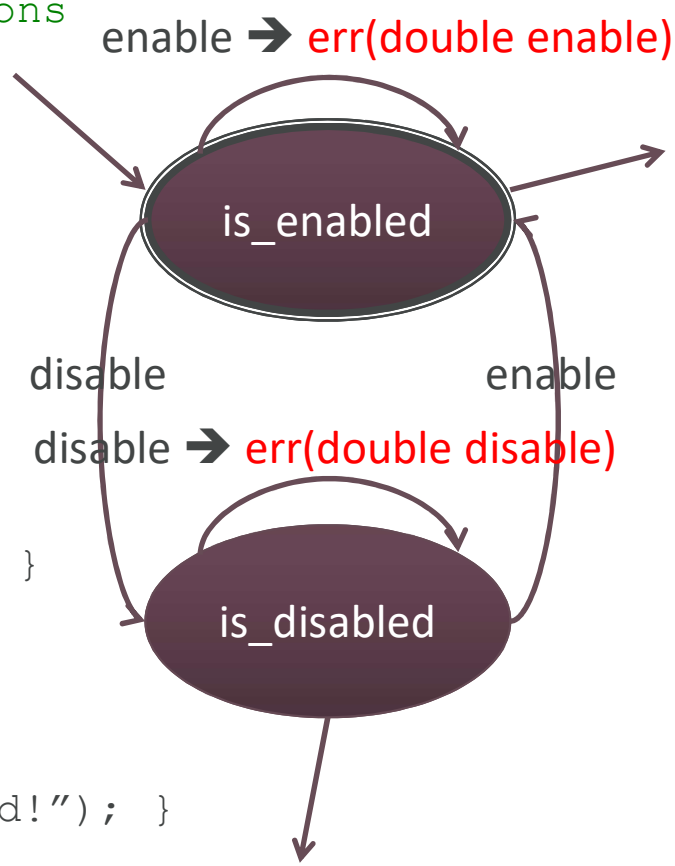
ERROR: function returns with interrupts disabled!

Example from Engler et al., *Checking system rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '000

Could you have found it?

- How often would that bug trigger?
- What happens if you return from a driver with interrupts disabled?
- Consider: that's one function
  - …in a 2000 LOC file
  - …in a module with 60,000 LOC
  - …IN THE LINUX KERNEL
- **Moral:** *Some defects are very difficult to find via testing, inspection.*

```
1. sm check_interrupts {
2. // variables; used in patterns
3. decl { unsigned } flags;
4. // patterns specify enable/disable functions
5. pat enable = { sti() ; }
6.             | { restore_flags(flags); } ;
7. pat disable = { cli() ; }
8. //states; first state is initial
9. is_enabled : disable ➔ is_disabled
10.    | enable ➔ { err("double enable"); }
11.;
12. is_disabled : enable ➔ is_enabled
13.     | disable ➔ { err("double disable"); }
14.//special pattern that matches when
15.// end of path is reached in this state
16.     | $end_of_path$ ➔
17.         { err("exiting with inter disabled!"); }
18.;
19.}
```



Example from Engler et al., *Checking system rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '000

```
1./* from Linux 2.3.99 drivers/block/raid5.c */
2.static struct buffer_head *
3.get_free_buffer(struct stripe_head * sh,
4.                int b_size) {
5.  struct buffer_head *bh;
6.  unsigned long flags;
7.  save_flags(flags);
8.  cli(); // disables interrupts
9.  if ((bh = sh->buffer_pool) == NULL)
10.    return NULL;
11.  sh->buffer_pool = bh -> b_next;
12.  bh->b_size = b_size;
13.  restore_flags(flags); // re-enables interrupts
14.  return bh;
15.}
```

Initial state: is_enabled

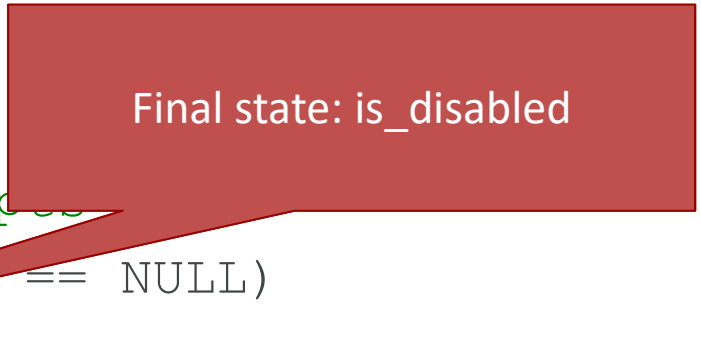
Example from Engler et al., *Checking system rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '000

```
1./* from Linux 2.3.99 drivers/block/raid5.c */
2.static struct buffer_head *
3.get_free_buffer(struct stripe_head * sh,
4.                int b_size) {
5.  struct buffer_head *bh;
6.  unsigned long flags;
7.  save_flags(flags);
8.  cli(); // disables interrupts
9.  if ((bh = sh->buffer_pool) == NULL)
10.    return NULL;
11.  sh->buffer_pool = bh -> b_next;
12.  bh->b_size = b_size;
13.  restore_flags(flags); // re-enables interrupts
14.  return bh;
15.}
```

Transition to: is_disabled

Example from Engler et al., *Checking system rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '000

```
1./* from Linux 2.3.99 drivers/block/raid5.c */
2.static struct buffer_head *
3.get_free_buffer(struct stripe_head * sh,
4.                int b_size) {
5.  struct buffer_head *bh;
6.  unsigned long flags;
7.  save_flags(flags);
8.  cli(); // disables interrupts
9.  if ((bh = sh->buffer_pool) == NULL)
10.    return NULL;
11.  sh->buffer_pool = bh -> b_next;
12.  bh->b_size = b_size;
13.  restore_flags(flags); // re-enables interrupts
14.  return bh;
15.}
```



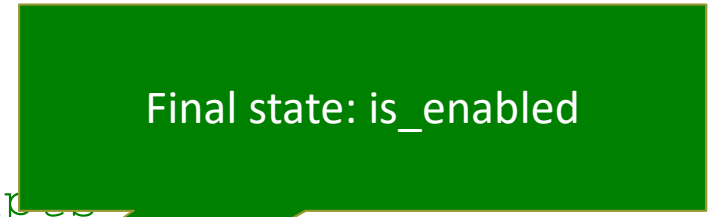
Example from Engler et al., *Checking system rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '000

```
1./* from Linux 2.3.99 drivers/block/raid5.c */
2.static struct buffer_head *
3.get_free_buffer(struct stripe_head * sh,
4.                int b_size) {
5.  struct buffer_head *bh;
6.  unsigned long flags;
7.  save_flags(flags);
8.  cli(); // disables interrupts
9.  if ((bh = sh->buffer_pool) == NULL)
10.    return NULL;
11.  sh->buffer_pool = bh -> b_next;
12.  bh->b_size = b_size;
13.  restore_flags(flags); // re-enables interrupts
14.  return bh;
15.}
```

Transition to: is_enabled

Example from Engler et al., *Checking system rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '000

```
1./* from Linux 2.3.99 drivers/block/raid5.c */
2.static struct buffer_head *
3.get_free_buffer(struct stripe_head * sh,
4.                int b_size) {
5.  struct buffer_head *bh;
6.  unsigned long flags;
7.  save_flags(flags);
8.  cli(); // disables interrupts
9.  if ((bh = sh->buffer_pool) == NULL)
10.    return NULL;
11.  sh->buffer_pool = bh -> b_next;
12.  bh->b_size = b_size;
13.  restore_flags(flags); // re-enables interrupts
14.  return bh;
15.}
```



Example from Engler et al., *Checking system rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '000

Defects of interest…

- Are on uncommon or difficult-to-force execution paths.
  - Which is why it's hard to find them via testing.
- Executing (or interpreting/otherwise analyzing) all paths concretely to find such defects is infeasible.
- **What we really want to do is check the entire possible state space of the program.**

Defects *Static* Analysis can Catch

- **Defects that result from inconsistently following simple, mechanical design rules.**
  - **Security:** Buffer overruns, improperly validated input.
  - **Memory safety:** Null dereference, uninitialized data.
  - **Resource leaks:** Memory, OS resources.
  - **API Protocols:** Device drivers; real time libraries; GUI frameworks.
  - **Exceptions:** Arithmetic/library/user-defined
  - **Encapsulation:** Accessing internal data, calling private functions.
  - **Data races:** Two threads access the same data without synchronization

Key: check compliance to simple, mechanical design rules

Definition: software analysis

The systematic examination of a software artifact to determine its properties.

Principle techniques

- **Dynamic:**
  - **Testing:** Direct execution of code on test data in a controlled environment.
  - **Analysis:** Tools extracting data from test runs.
- **Static:**
  - **Inspection:** Human evaluation of code, design documents (specs and models), modifications.
  - **Analysis:** Tools reasoning about the program without executing it.

Fundamental concepts

- **Abstraction.**
  - Elide details of a specific implementation.
  - Capture semantically relevant details; ignore the rest.
- **The importance of *semantics*.**
  - We prove things about analyses with respect to the semantics of the underlying language.
- **Implementation**
  - You do not understand analysis until you have written several.

The Bad News: Rice's Theorem

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

Henry Gordon Rice, 1953

OK, so?

- If you could infallibly statically tell if any program had a non-trivial property (never dereferences null, always releases all file handles, etc, etc), you could also 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), or will not provably terminate.

Proof by contradiction (sketch)

Assume that you have a function that can determine if a program *p* has some nontrivial property (like `divides_by_zero`):

```
1.  int silly(program p, input i) {
2.    p(i);
3.    return 5/0;
4.  }
5.  bool halts(program p, input i) {
6.   return divides_by_zero(silly(p,i));
7.  }
```

	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

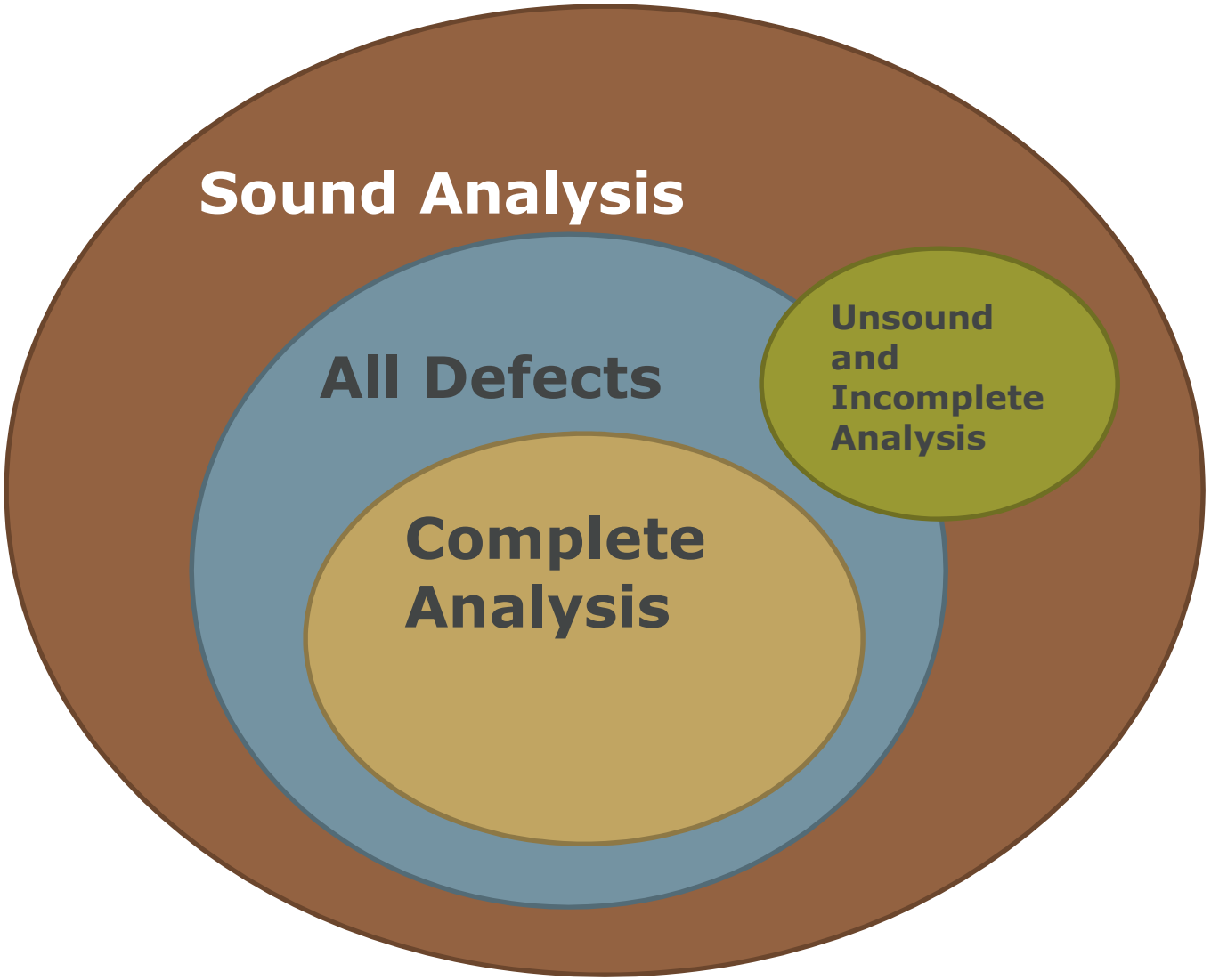
typically overapproximated

Complete Analysis:

every reported defect is an actual defect

-> no false positives

typically underapproximated



HOW THE CLASS WILL WORK

Language definitions

- Concrete syntax: The rules by which programs can be expressed as strings of characters.
  - Use finite automata and context-free grammars, automatic lexer/parser generators
- Abstract syntax: a subset of the parse tree of the program.

WHILE abstract syntax

- Categories:
  - $S \in$ **Stmt** statements
  - $a \in$ **Aexp** arithmetic expressions
  - $x, y \in$ **Var** variables
  - $n \in$ **Num** number literals
  - $P \in$ **BExp** boolean predicates
  - $l \in$ **labels** statement addresses (line numbers)
- Syntax:
  - S ::= $x := a$ | skip | S_1 ; S_2 | if P then S_1 else S_2 | while P do S
  - a ::= x | n | a_1 op_a a_2
  - op_a ::= + | - | * | / | …
  - P ::= true | false | not P | P_1 op_b P_2 | $a1$ op_r $a2$
  - op_b ::= and | or | …
  - op_r ::= < | ≤ | = | > | ≥ | ...

Concrete syntax would be similar, but would add things like (parentheses) for disambiguation during parsing

Example WHILE program

```
y := x;
z := 1;
while y > 1 do
   z := z * y;
   y := y - 1
```

Exercise: Building an AST

```
y := x;
z := 1;
while y > 1 do
    z := z * y;
    y := y - 1
```

WHILE3ADDR: An Intermediate Representaiton

- Simpler, more uniform than WHILE syntax
- Categories:
  - *I* ∈ **Instruction** instructions
  - *x, y* ∈ **Var** variables
  - *n* ∈ **Num** number literals
- Syntax:
  - I ::= $x := n \mid x := y \mid x := y\ op\ z$ $\mid$ goto $n \mid$ if $x\ op_r\ 0$ goto n
  - op_a ::= + | - | * | / | …
  - op_r ::= < | ≤ | = | > | ≥ | ...
  - P ∈ **Num** → *I*

Exercise: Translating to While3Addr

- Categories:
  - *I* ∈ **Instruction** instructions
  - *x, y* ∈ **Var** variables
  - *n* ∈ **Num** number literals

- Syntax:
  - I ::= x := n | x := y | x := y op z
    | goto n | if x op_r 0 goto n
  - op_a ::= + | - | * | / | …
  - op_r ::= < | ≤ | = | > | ≥ | ...
  - P ∈ **Num** → *I*

WHILE3ADDR Extensions (more later)

- Syntax:
  - I ::= $x := n$ | $x := y$ | $x := y \; op \; z$
    | goto n | if x op_r 0 goto n
    | $x := f(y)$
    | return x
    | $x := y.m(z)$
    | $x := \&p$
    | $x := *p$
    | $*p := x$
    | $x := y.f$
    | $x.f := y$

Syntactic Analysis

- Walks a program representation, searching for errors
  - Example: bad shift analysis

```
For each instruction I in the program
  if I is a shift instruction
     if (type of I's left operand is int
          && I's right operand is a constant
          && value of constant < 0 or > 31)
        warn("Shifting by less than 0 or more
              than 31 is meaningless")
```

Practice: String concatenation in a loop

- Write pseudocode for a simple syntactic analysis that warns when string concatenation occurs in a loop
  - In Java and .NET it is more efficient to use a StringBuffer
  - Assume any appropriate AST elements

For next time

- Get on Piazza and Canvas
- Read our lecture notes and the course syllabus