15-214

15-214

toad

Spring 2013

School of Computer Science

institute for SOFTWARE RESEARCH

Principles of Software Construction: Objects, Design and Concurrency

Static Analysis

Jonathan Aldrich Charlie Garrod

The four course themes

- Threads and Concurrency
  - Concurrency is a crucial system abstraction
  - E.g., background computing while responding to users
  - Concurrency is necessary for performance
  - Multicore processors and distributed computing
  - Our focus: application-level concurrency
  - Cf. functional parallelism (150, 210) and systems concurrency (213)
- Object-oriented programming
  - For flexible designs and reusable code
  - A primary paradigm in industry – basis for modern frameworks
  - Focus on Java – used in industry, some upper-division courses
- Analysis and Modeling
  - Practical specification techniques and verification tools
  - Address challenges of threading, correct library usage, etc.
- Design
  - Proposing and evaluating alternatives
  - Modularity, information hiding, and planning for change
  - Patterns: well-known solutions to design problems

Static Analysis

- Analyzing the code, without executing it
- Find bugs / proof correctness

- Many flavors

Find the Bug!

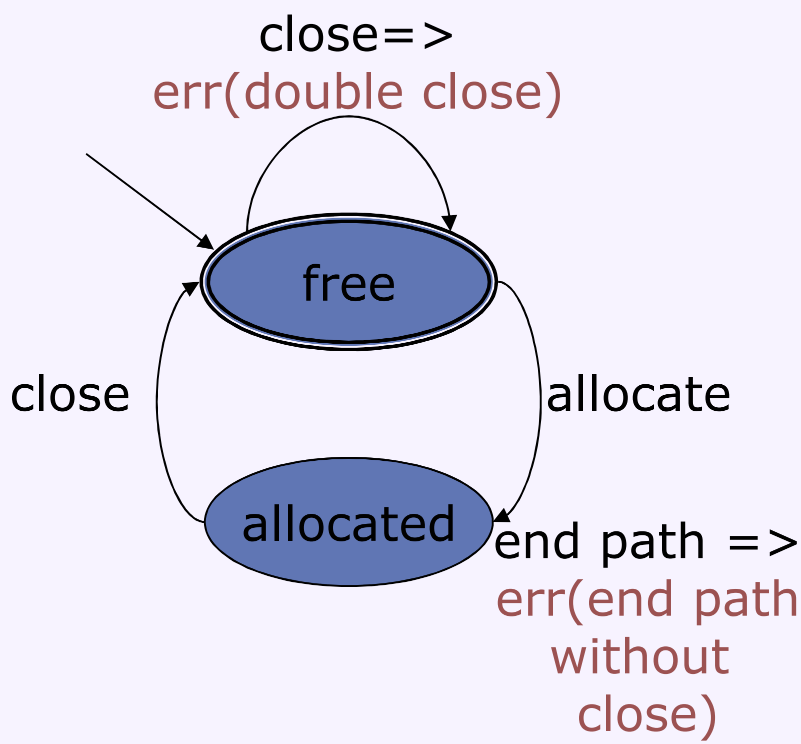
```
public void loadFile(String filename) throws IOException {
      BufferedReader reader =
            new BufferedReader(
                  new FileReader(filename));

      if (reader.readLine().equals("version 1.0"))
            return; // invalid version

      String c;
      while ((c = reader.readLine()) != null) {
            processData(c)
      }

      reader.close();
}
```

Limits of Inspection

- People
- …are very high cost
- …make mistakes
- …have a memory limit

So, let's automate inspection!

Mental Model for Analyzing "Freed Resources"



Find the Bug!

```
public void loadFile(String filename) throws IOException {
      BufferedReader reader =                 ← initial state free
            new BufferedReader(               ← transition to allocated
                  new FileReader(filename));

      if (reader.readLine().equals("version 1.0"))
            return;          ←——————— final state allocated: ERROR!

      String c;
      while ((c = reader.readLine()) != null) {
            processData(c)
      }

      reader.close();                         ← transition to free
}                  ←——————————————————— final state free is OK
```

Static Analysis Finds "Mechanical" Errors

- Defects that result from inconsistently following simple, mechanical design rules

- Security vulnerabilities
  - Buffer overruns, unvalidated input…
- Memory errors
  - Null dereference, uninitialized data…
- Resource leaks
  - Memory, OS resources…
- Violations of API or framework rules
  - e.g. Windows device drivers; real time libraries; GUI frameworks
- Exceptions
  - Arithmetic/library/user-defined
- Encapsulation violations
  - Accessing internal data, calling private functions…
- Race conditions
  - Two threads access the same data without synchronization

FindBugs

findbugs.sourceforge.net

Example Tool: FindBugs

- Origin: research project at U. Maryland
  - Now freely available as open source
  - Standalone tool, plugins for Eclipse, etc.
- Checks over 250 “bug patterns”
  - Over 100 correctness bugs
  - Many style issues as well
  - Includes the two examples just shown
- Focus on simple, local checks
  - Similar to the patterns we’ve seen
  - But checks bytecode, not AST
    - Harder to write, but more efficient and doesn’t require source
- http://findbugs.sourceforge.net/

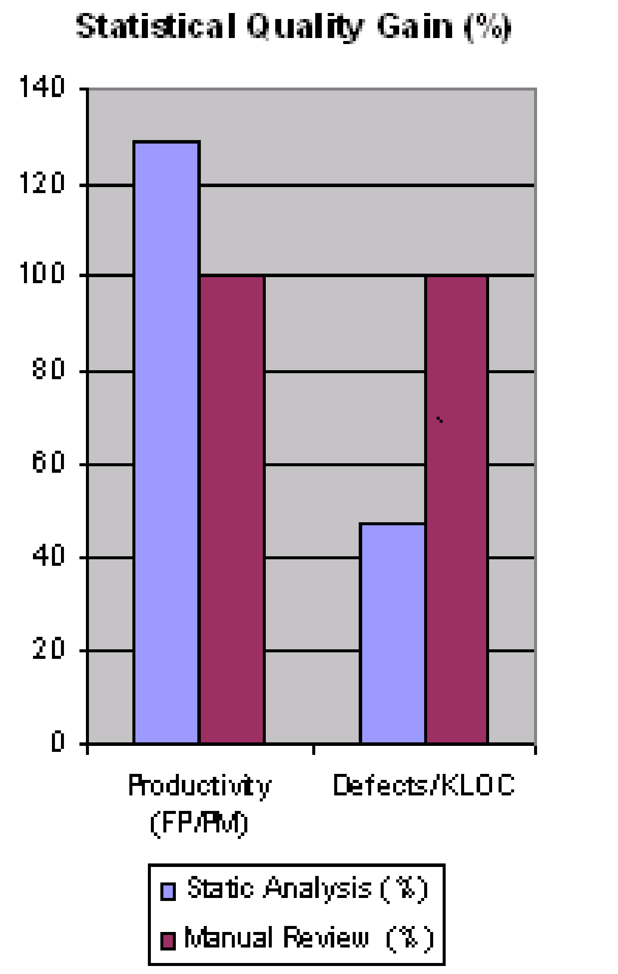
Example FindBugs Bug Patterns

- Correct equals()
- Use of ==
- Closing streams
- Illegal casts
- Null pointer dereference
- Infinite loops
- Encapsulation problems
- Inconsistent synchronization
- Inefficient String use
- Dead store to variable

Demonstration: FindBugs

Empirical Results on Static Analysis

- InfoSys study [Chaturvedi 2005]
  - 5 projects
  - Average 700 function points each
  - Compare inspection with and without static analysis
- Conclusions
  - Fewer defects
  - Higher productivity



Adapted from [Chaturvedi 2005]

Outline

- Why static analysis?
  - Automated
  - Can find some errors faster than people
  - Can provide guarantees that some errors are found
- How does it work?
- What are the hard problems?
- How do we use real tools in an organization?

Outline

- Why static analysis?
- How does it work?
  - Systematic exploration of program abstraction
  - Many kinds of analysis
    - AST walker
    - Control-flow and data-flow
    - Type systems
    - Model checking
  - Specifications frequently used for more information
- What are the hard problems?
- How do we use real tools in an organization?

Abstract Interpretation

- Static program analysis is the **systematic examination** of an **abstraction of a program's state space**
- Abstraction
  - Don't track everything! (That's normal interpretation)
  - Track an important abstraction
- Systematic
  - Ensure everything is checked in the same way
- Let's start small…

AST Analysis

A Performance Analysis

What’s the performance problem?

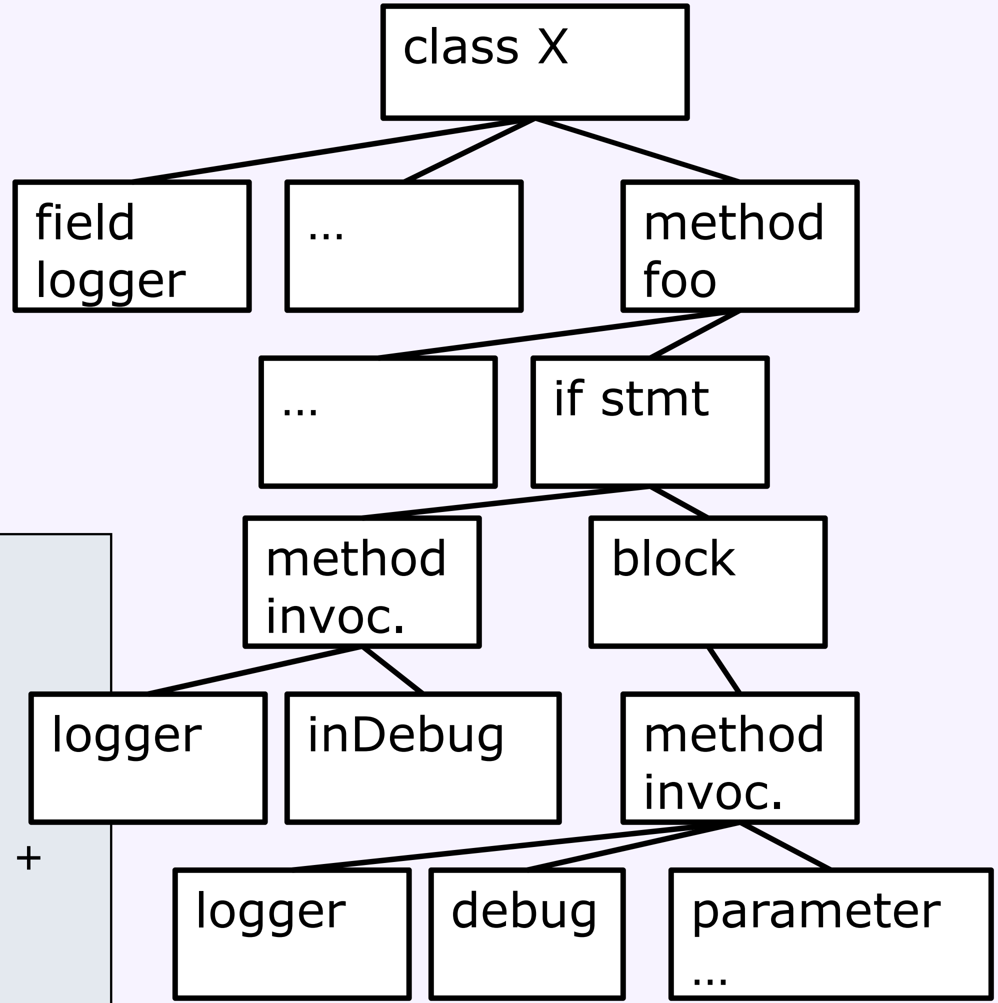
```
public foo() {
  …
  if (logger.inDebug()) {
    logger.debug(“We have ” + conn + “connections.”);
  }
}
```

Seems minor…
but if this performance gain on 1000 servers means we need 1 less machine, we could be saving a lot of money

A Performance Analysis

- Check that we don't create debug strings outside of a Logger.inDebug check
- Abstraction
  - Look for a call to Logger.debug()
  - Make sure it's surrounded by an if (Logger.inDebug())
- Systematic
  - Check all the code
- Known as an **Abstract Syntax Tree (AST) walker**
  - Treats the code as a structured tree
  - Ignores control flow, variable values, and the heap
  - Code style checkers work the same way
    - you should never be checking code style by hand
  - Simplest static analysis: grep

Abstract Syntax Trees




```
class X {
  Logger logger;
  public void foo() {
    …
    if (logger.inDebug()) {
      logger.debug(“We have ” +
conn + “connections.”);
    }
  }
}
```

```java
class Money {
    String currency;
    int amount;

    public Money(String currency, int amount) {
        this.currency = currency;
        this.amount = amount;
    }

    public boolean equals(Object o) {
```

```
}
```

Problem | Javadoc | Declarati | Search | Console | Coverag | History | Bug Info | Bug Expl

A.java: 77

Navigation

Bug: Money defines equals and uses Object.hashCode()

This class overrides `equals(Object)`, but does not override `hashCode()`, and inherits the implementation `hashCode()` from `java.lang.Object` (which returns the identity hash code, an arbitrary value assigned to the object by the VM). Therefore, the class is very likely to violate the invariant that equal objects mu have equal hashcodes.

If you don't think instances of this class will ever be inserted into a HashMap/HashTable, the recommended `hashCode` implementation to use is:

```java
public int hashCode() {
  assert false : "hashCode not designed";
  return 42; // any arbitrary constant will do
  }
```

Confidence: Normal, **Rank**: Of Concern (16)

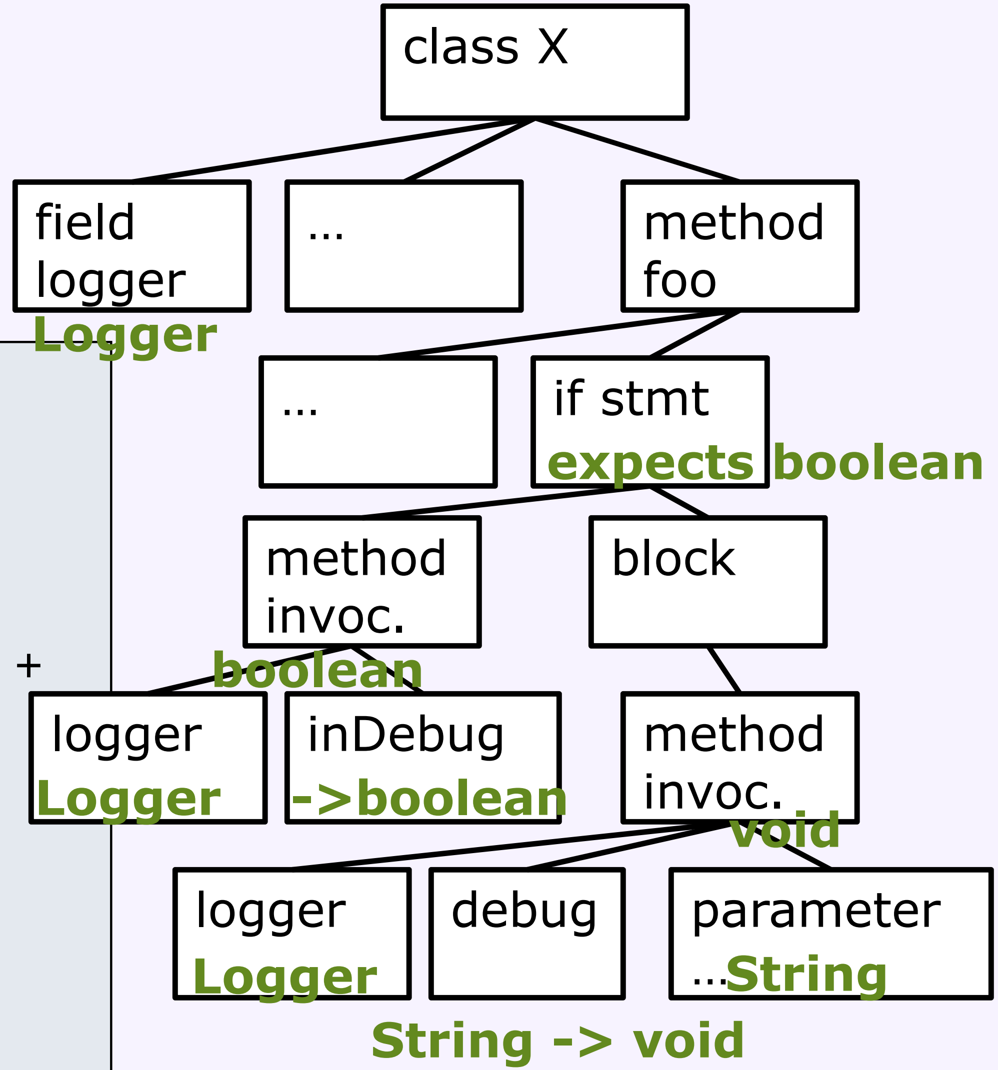
Type Checking

```java
public void foo() {
    int a = computeSomething();

    if (a == "5")
        doMoreStuff();
}
```

Type Checking

- Classifying values into types
- Checking whether operations are allowed on those types
- Detects a class of problems at compile time, e.g.
  - Method not found
  - Cannot compare int and boolean

```
class X {
  Logger logger;
  public void foo() {
    …
    if (logger.inDebug()) {
      logger.debug("We have " +
conn + "connections.");
    }
  }
}
class Logger {
   boolean inDebug() {…}
   void debug(String msg) {…}
}
```



Typechecking in different Languages

- In Perl…
  - No typechecking at all!
- In ML, no annotations required
  - Global typechecking
- In Java, we annotate with types
  - Modular typechecking
  - Types are a specification!
- In C# and Scala, no annotations for local variables
  - Required for parameters and return values
  - Best of both?

```
foo() {
   a = 5;
   b = 3;
   bar(“A”, “B”);
   print(5 / 3);
}

bar(x, y) {
   print(x / y);
}
```

Bug finding

```java
public Boolean decide() {
    if (computeSomething()==3)
        return Boolean.TRUE;
    if (computeSomething()==4)
        return false;
    return null;
}
```

Problem | Javadoc | Declarati | Search | Console | Coverag | History | **Bug Info** | Bug Expl

A.java: 69

Navigation

Bug: FBTest.decide() has Boolean return type and returns explicit null

A method that returns either Boolean.TRUE, Boolean.FALSE or null is an accident waiting to happen. This method can be invoked as though it returned a value of type boolean, and the compiler will insert automatic unboxing of the Boolean value. If a null value is returned, this will result in a NullPointerException.

Confidence: Normal, **Rank**: Troubling (14)
Pattern: NP_BOOLEAN_RETURN_NULL
Type: NP, **Category**: BAD_PRACTICE (Bad practice)

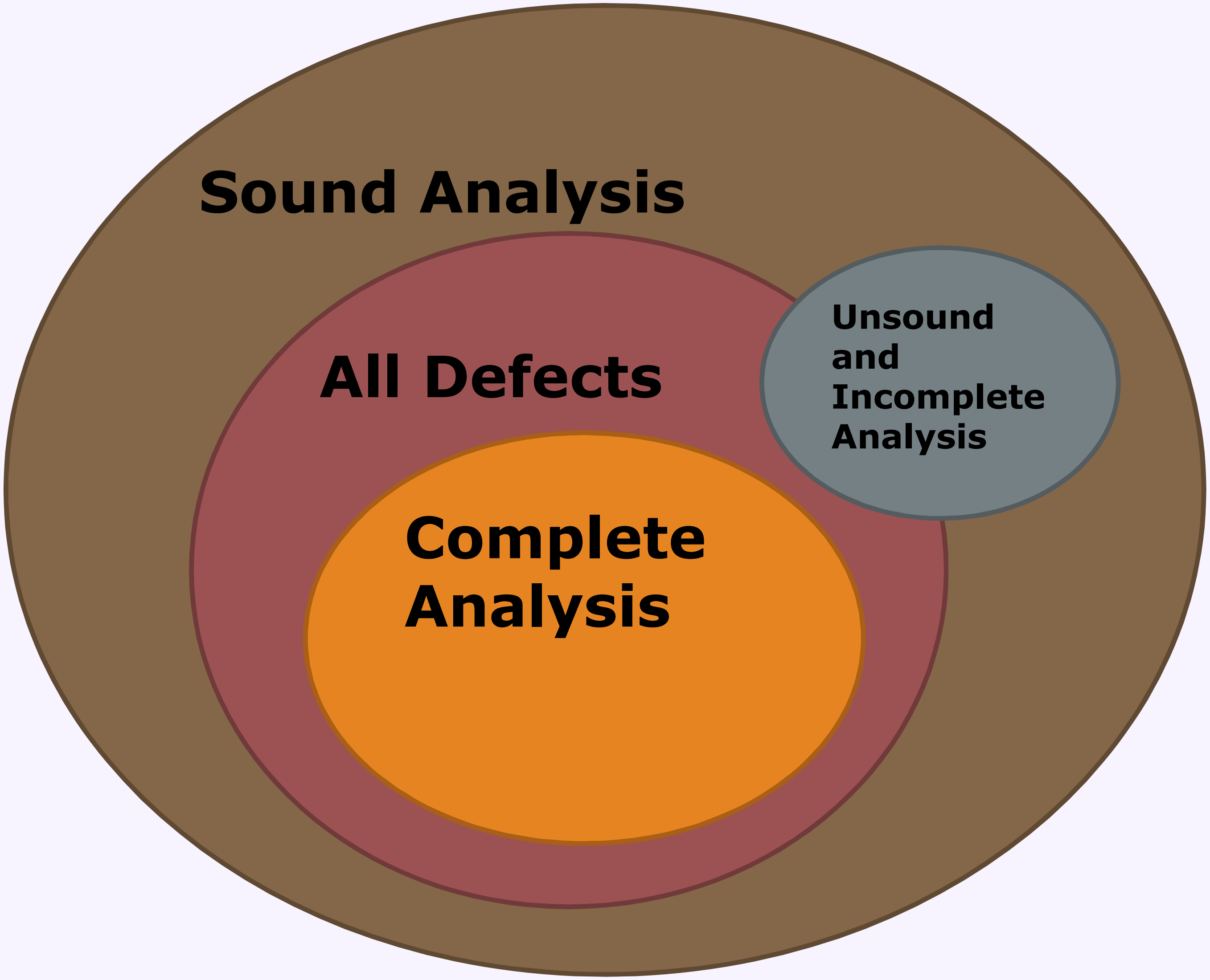
Intermission: Soundness and Completeness

	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 does testing relate? And formal verification?


Sound Analysis
All Defects
Unsound
and
Incomplete
Analysis
Complete
Analysis

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)

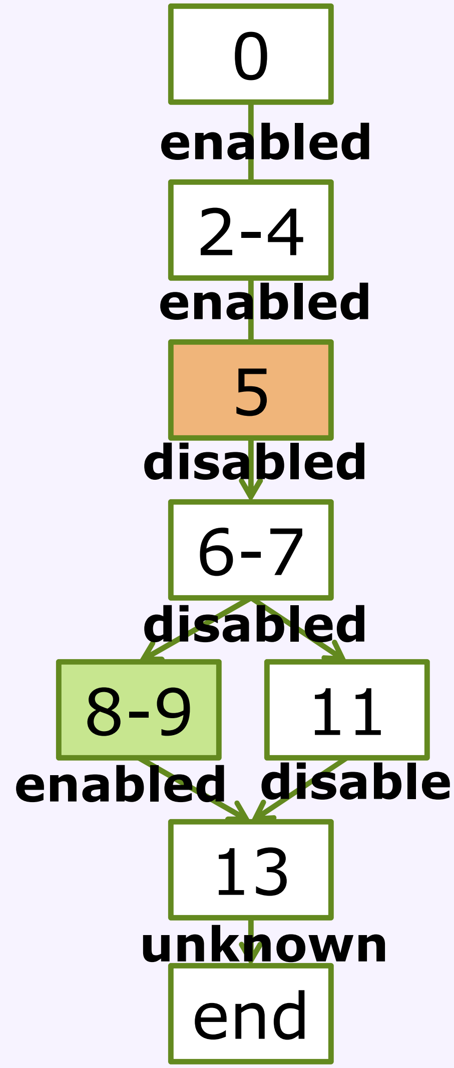
Control-Flow Analysis

An interrupt checker

- Check for the interrupt problem
- Abstraction
  - 2 states: enabled and disabled
  - Program counter
- Systematic
  - Check all paths through a function
- Error when we hit the end of the function with interrupts disabled
- Known as a **control flow analysis**
  - More powerful than reading it as a raw text file
  - Considers the program state and paths

Example: Interrupt Problem

```
1. int foo() {
2.     unsigned long flags;
3.     int rv;
4.     save_flags(flags);
5.     cli();
6.     rv = dont_interrupt();
7.     if (rv > 0) {
8.         do_stuff();
9.         restore_flags();
10.    } else {
11.        handle_error_case();
12.    }
13.    return rv;
14.}
```



Error: did not reenable interrupts on some path

Adding branching

- When we get to a branch, what should we do?
  - 1: explore each path separately
    - Most exact information for each path
    - But—how many paths could there be?
    - Leads to an exponential state explosion
  - 2: join paths back together
    - Less exact
    - But no state explosion
- Not just conditionals!
  - Loops, switch, and exceptions too!

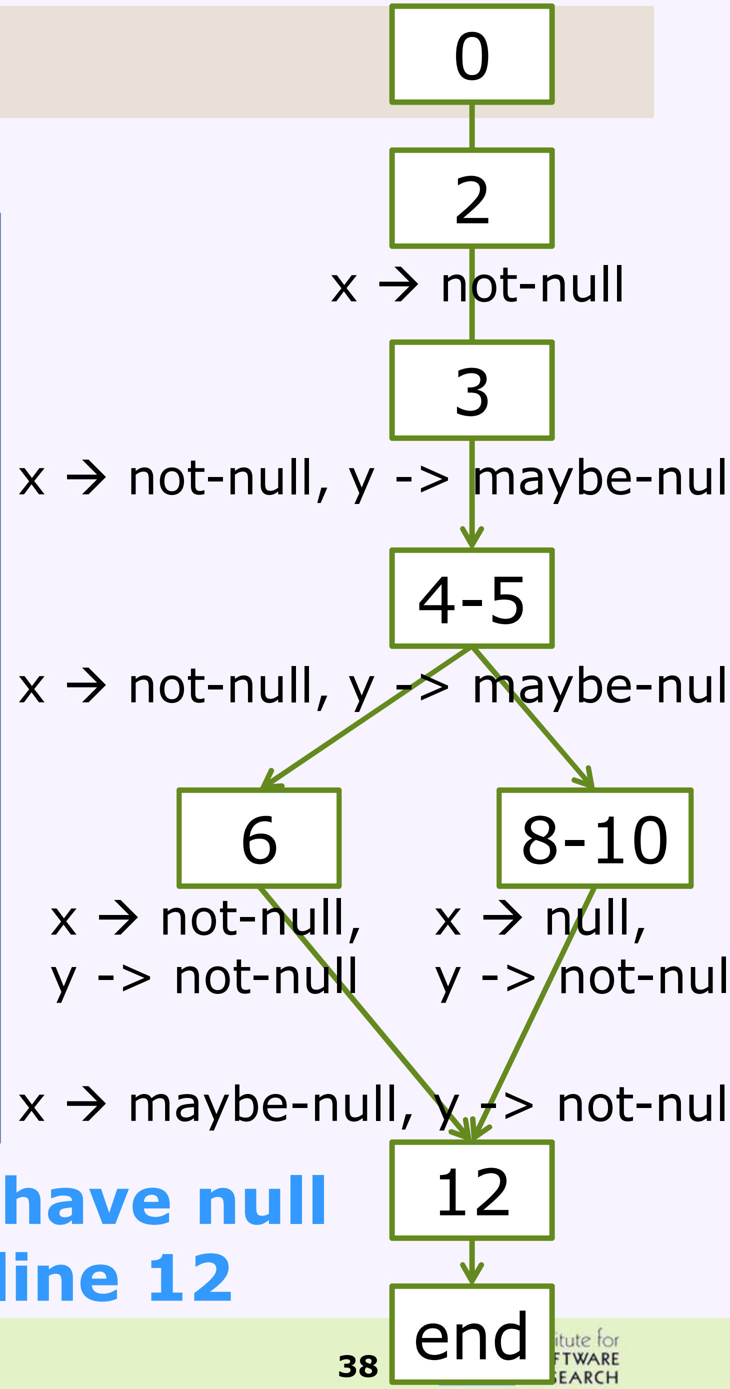
Data-Flow Analysis

A null pointer checker

- Prevent accessing a null value
- Abstraction
  - Program counter
  - 3 states for each variable: null, not-null, and maybe-null
- Systematic
  - Explore all data values for all variables along all paths in a method or program
- Known as a **data-flow** analysis
  - Tracking how data moves through the program
  - Very powerful, many analyses work this way
  - Compiler optimizations were the first
  - Expensive

Example: Null Pointer Problem

```
1.  int foo() {
2.      Integer x = new Integer(6);
3.      Integer y = bar();
4.      int z;

5.      if (y != null)
6.         z=x.intVal()+y.intVal();
7.      else {
8.         z = x.intVal();
9.         y = x;
10.        x = null;
11.     }
12.     return z + x.intVal();
13. }
```



Error: may have null pointer on line 12

Example: Method calls

```
int foo() {
    Integer x = bar();
    Integer y = baz();
    Integer z = noNullsAllowed(x, y);
    return z.intValue();
}

Integer noNullsAllowed(Integer x, Integer y) {
    int z;
    z = x.intValue() + y.intValue();
    return new Integer(z);
}
```

Two options:
1. Global analysis
2. Modular analysis with specifications

Global Analysis

- Dive into every method call
  - Like branching, exponential without joins
  - Typically cubic (or worse) in program size even with joins
- Requires developer to determine which method has the fault
  - Who should check for null? The caller or the callee?

Modular Analysis w/ Specifications

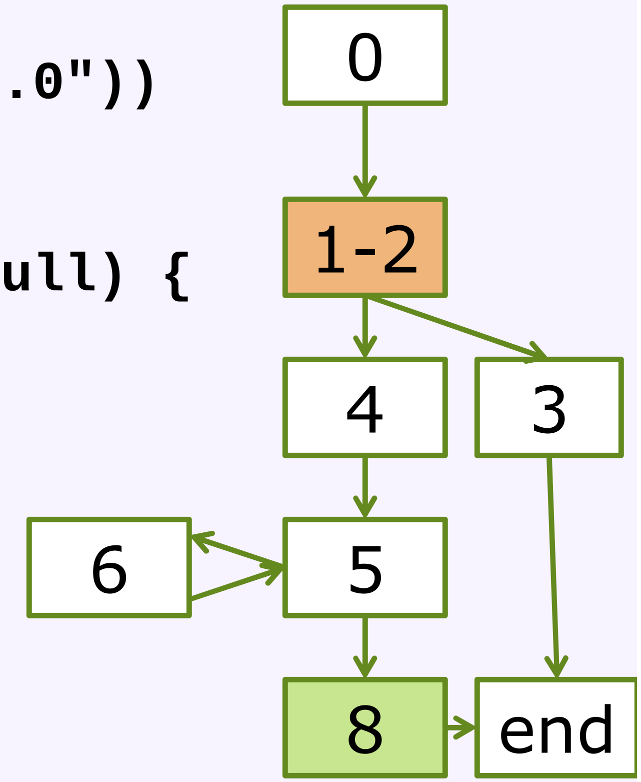
- Analyze each module separately
- Piece them together with specifications
  - **Pre-condition** and **post-condition**
- When analyzing a method
  - Assume the method's precondition
  - Check that it generates the postcondition
- When the analysis hits a method call
  - Check that the precondition is satisfied
  - Assume the call results in the specified postcondition
- See formal verification and Dafny

Example: Method calls

```
int foo() {
    Integer x = bar();
    Integer y = baz();
    Integer z = noNullsAllowed(x, y);
    return z.intValue();
}

@Nonnull Integer noNullsAllowed(
                          @Nonnull Integer x,
                          @Nonnull Integer y) {
    int z;
    z = x.intValue() + y.intValue();
    return new Integer(z);
}

@Nonnull Integer bar();

@Nullable Integer baz();
```

Another Data-Flow Example

```
public void loadFile(String filename)
   BufferedReader reader = …;
   if (reader.readLine().equals("ver 1.0"))
      return; // invalid version
   String c;
   while ((c = reader.readLine()) != null) {
      // load data
   }
   reader.close();
}
```



abstractions:
needs-closing, closed, unknown

Recap: Class invariants

- Is always true outside a class's methods
- Can be broken inside, but must always be put back together again

```
public class Buffer {
  boolean isOpen;
  int available;
  /*@ invariant isOpen <==> available > 0  @*/

  public void open() {
    isOpen = true;
    //invariant is broken
    available = loadBuffer();
  }
}
```

ESC/Java, Dafny are a kind of static analysis tool

Other kinds of specifications

- Class invariants
  - What is always true when entering/leaving a class?
- Loop invariants
  - What is always true inside a loop?
- Lock invariant
  - What lock must you have to use this object?
- Protocols
  - What order can you call methods in?
    - Good: Open, Write, Read, Close
    - Bad: Open, Write, Close, Read

Model Checking

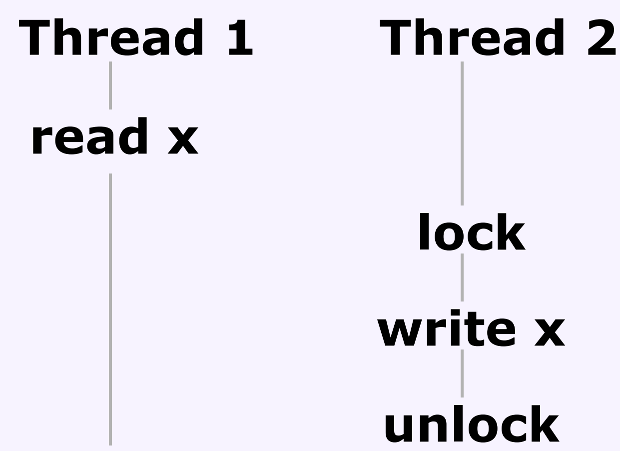
Static Analysis for Race Conditions

- ***Race condition*** defined:

  [From Savage et al., *Eraser: A Dynamic Data Race Detector for Multithreaded Programs*]
  - Two threads access the same variable
  - At least one access is a write
  - No explicit mechanism prevents the accesses from being simultaneous
- Abstraction
  - Program counter of each thread, state of each lock
    - Abstract away heap and program variables
- Systematic
  - Examine all possible interleavings of all threads
    - Flag error if no synchronization between accesses
    - Exploration is exhaustive, since abstract state abstracts all concrete program state
- Known as *Model Checking*

Model Checking for Race Conditions

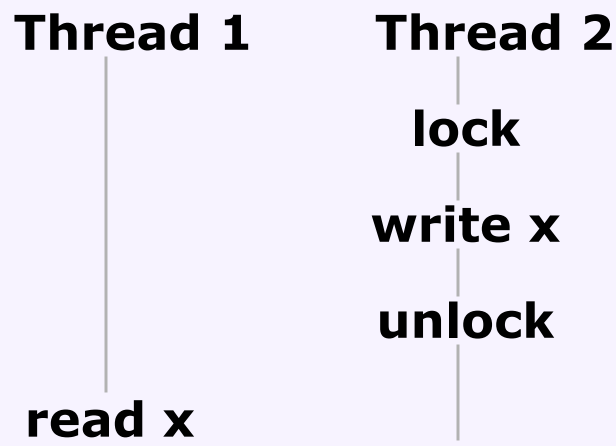
```
thread1() {
    read x;
}
thread2() {
    lock();
    write x;
    unlock();
}
```



Interleaving 1: OK

Model Checking for Race Conditions

```
thread1() {
    read x;
}
thread2() {
    lock();
    write x;
    unlock();
}
```

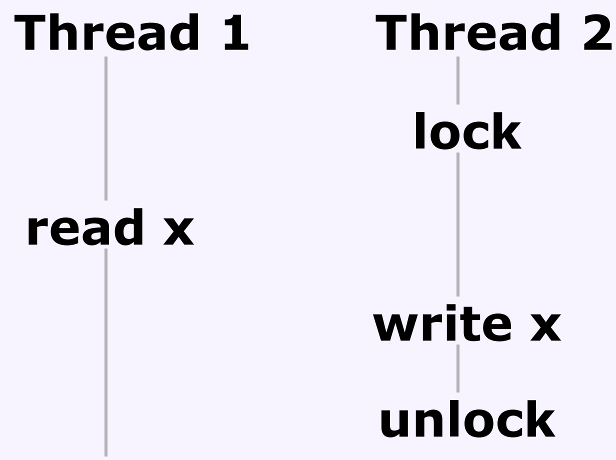


Interleaving 1: OK

Interleaving 2: OK

Model Checking for Race Conditions

```
thread1() {
    read x;
}
thread2() {
    lock();
    write x;
    unlock();
}
```

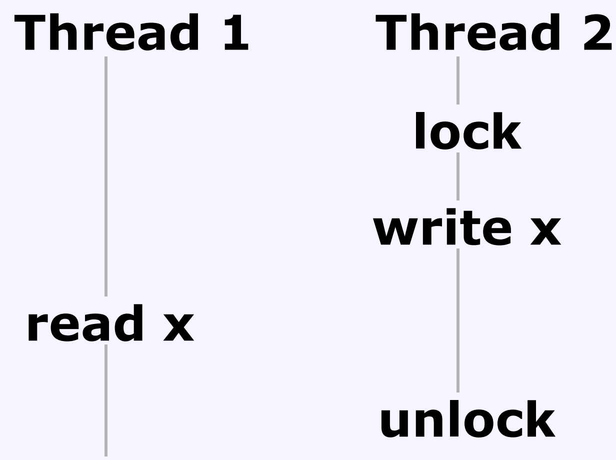


Interleaving 2: OK

Interleaving 3: Race

Model Checking for Race Conditions

```
thread1() {
    read x;
}
thread2() {
    lock();
    write x;
    unlock();
}
```



Interleaving 3: Race

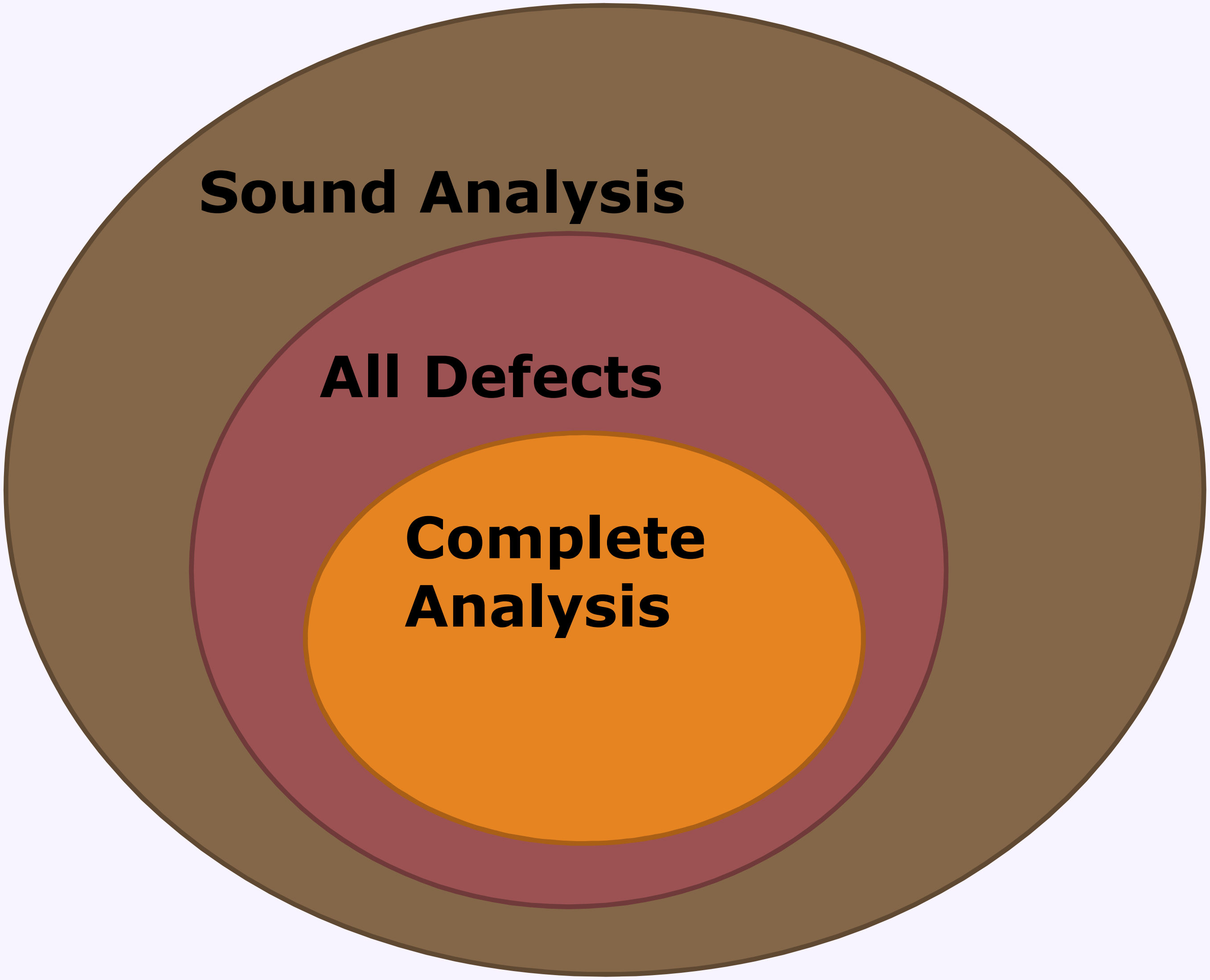
Interleaving 4: Race

Outline

- Why static analysis?
- How does it work?
- What are the important properties?
  - Precision
  - Side effects
  - Modularity
  - Aliases
  - Termination
- How do we use real tools in an organization?

Tradeoffs

- You can’t have it all
  1. No false positives
  2. No false negatives
  3. Perform well
  4. No specifications
  5. Modular
- You can’t even get 4 of the 5
  - Halting problem means first 3 are incompatible (Rice's theorem)
  - Modular analysis requires specifications
- Each tool makes different tradeoffs


Sound Analysis
All Defects
Complete
Analysis

Soundness / Completeness / Performance Tradeoffs

- Type checking does catch a specific class of problems, but does not find all problems
- Data-flow analysis for compiler optimizations must err on the safe side (only perform optimizations when sure it's correct)
- Many practical bug-finding tools analyses are unsound and incomplete
  - Catch typical problems
  - May report warnings even for correct code
  - May not detect all problems
- Overwhelming amounts of false negatives make analysis useless
- Not all "bugs" need to be fixed

“False” Positives

```
1.  int foo(Person person) {
2.      if (person != null) {
3.          person.foo();
4.      }
5.      return person.bar();
6.  }
```

Error on line 5: Redundant comparison to null

- Is this a false positive?
- What if that branch is never run in practice?
- Do you fix it? And how?

"False" Positives

```
1.public class Constants {
2.  static int myConstant = 100;
3.}
```

- Is this a false positive?
- What if it's in an open-source library you imported?
- What if there are 1000 of these?

Error on line 3: field isn't final but should be

Outline

- Why static analysis?
- How does it work?
- What are the important properties?
- How do we use real tools in an organization?
  - FindBugs @ eBay
  - SAL @ Microsoft
  - Coverity

True Positives

- Technical Defn: An issue that could result in a runtime error
- True Positives *that we care about*
  - 1. Any issue that the developer intends to fix
  - 2. (more subtle) Any issue that the developer wants to see, regardless of whether it is fixed
  - Varies between projects and people
- Soundness and completeness are defined technically
  - But sometimes don't exactly match what people want

FindBugs at eBay

- eBay wants to use static analysis
- Need off the shelf tools
- Focus on security and performance
- Had bad past experiences
  - Too many false positives
  - Tools used too late in process
- Help them choose a tool and add it to the process

How important is this issue?

```
1.  void foo(int x, int y)
2.      int z;
3.      z = x + y;
4.  }
```

Line 3: Dead store to local

How about this one?

```
void foo(int x, int y)
  List dataValues;
  dataValues = getDataFromDatabase(x, y);
}
```

Significant overhead, and not caught any other way!

Tool Customization

- Turn on all defect detectors
- Run on a single team's code
- Sort results by detector
- Assign each detector a priority
- Repeat until consensus (3 teams)

Priority = Enforcement

- Priority must mean something
  - (otherwise it's all "high priority")
- High Priority
  - High severity functional issues
  - Medium severity, but easy to fix
- Medium Priority
  - Medium severity functional issues
  - Indicators to refactor
  - Performance issues
- Low Priority
  - Only some domain teams care about them
  - Stylistic issues
- Toss
  - Not cost effective and lots of noise

Cost/Benefit Analysis

- Costs
  - Tool license
  - Engineers internally supporting tool
  - Peer reviews of defect reports
- Benefits
  - How many defects will it find?
  - What priority?
- Compare to cost equivalent of testing by QA Engineers
  - eBay's primary quality assurance mechanism
  - Back of the envelope calculation
  - FindBugs discovers significantly more defects
    - Order of magnitude difference
    - Not as high priority defects

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

PREFast at Microsoft

- Concerned with memory usage
- Major cause of security issues
- Manpower to developer custom tool

Standard Annotation Language (SAL)

- A language for specifying contracts between functions
  - Intended to be lightweight and practical
  - Preconditions and Postconditions
  - More powerful—but less practical—contracts supported in systems like JML or Spec#
- Initial focus: memory usage
  - buffer sizes
  - null pointers
  - memory allocation

SAL is checked using PREfast

- Lightweight analysis tool
  - Only finds bugs within a single procedure
  - Also checks SAL annotations for consistency with code
- To use it (for free!)
  - Download and install Microsoft Visual C++ 2005 Express Edition
    - http://msdn.microsoft.com/vstudio/express/visualc/
  - Download and install Microsoft Windows SDK for Vista
    - http://www.microsoft.com/downloads/details.aspx?familyid=c2b1e300-f358-4523-b479-f53d234cdccf
  - Use the SDK compiler in Visual C++
    - In Tools | Options | Projects and Solutions | VC++ Directories add C:\Program Files\Microsoft SDKs\Windows\v6.0\VC\Bin (or similar)
    - In project Properties | Configuration Properties | C/C++ | Command Line add /analyze as an additional option

Buffer/Pointer Annotations

_in	The function reads from the buffer. The caller provides the buffer and initializes it.
_inout	The function both reads from and writes to buffer. The caller provides the buffer and initializes it.
_out	The function writes to the buffer. If used on the return value, the function provides the buffer and initializes it. Otherwise, the caller provides the buffer and the function initializes it.
_bcount(size)	The buffer size is in bytes.
_ecount(size)	The buffer size is in elements.
_opt	This parameter can be NULL.

PREfast: Immediate Checks

- Library function usage
  - deprecated functions
    - e.g. gets() vulnerable to buffer overruns
  - correct use of printf
    - e.g. does the format string match the parameter types?
  - result types
    - e.g. using macros to test HRESULTs
- Coding errors
  - = instead of == inside an if statement
- Local memory errors
  - Assuming malloc returns non-zero
  - Array out of bounds

SAL: the Benefit of Annotations

- Annotations express **design intent**
  - How you intended to achieve a particular quality attribute
    - e.g. never writing more than N elements to this array
- As you add more annotations, you find more errors
  - Get checking of library users for free
  - Plus, those errors are less likely to be false positives
    - The analysis doesn't have to guess your intention
  - Instant Gratification Principle
- Annotations also improve **scalability** through modularity
  - PreFAST uses very sophisticated analysis techniques
  - These techniques can't be run on large programs
  - Annotations isolate functions so they can be analyzed one at a time

SAL: the Benefit of Annotations

- How to motivate developers?
  - Especially for millions of lines of unannotated code?
- Require annotations at checkin
  - Reject code that has a char* with no __ecount()
- Make annotations natural
  - Ideally what you would put in a comment anyway
    - But now machine checkable
    - Avoid formality with poor match to engineering practices
- Incrementality
  - Check code ↔ design consistency on every compile
  - Rewards programmers for each increment of effort
    - Provide benefit for annotating partial code
    - Can focus on most important parts of the code first
    - Avoid excuse: I'll do it after the deadline
- Build tools to infer annotations
  - Inference is approximate
  - Unfortunately not yet available outside Microsoft

Impact at Microsoft

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

Static Analysis in Engineering Practice

- A tool with different tradeoffs
  - Soundness: can find all errors in a given class
  - Focus: mechanically following design rules
- Major impact at Microsoft and eBay
  - Tuned to address company-specific problems
  - Affects every part of the engineering process