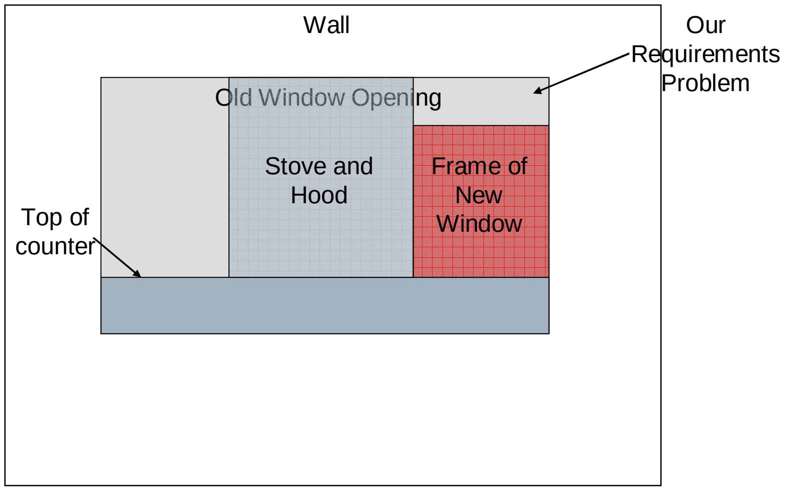
A Requirements Story

- Context: Kitchen remodel
- We wanted big windows
  - Asked how wide they could be
  - Asked how close to the counter we could make them
- Architect drew diagram, we ordered the windows

What We Saw Last Friday



Our Requirements Problem

- Customer wants big windows
  - Customer already has tall windows, but doesn't realize this is non-standard
- Architect discusses window size
  - Mentions that windows will be same height as door
  - Doesn't mention this is lower than our current windows
    - and it isn't obvious because the door is being moved up
- Requirements defect is found during implementation
  - Custom windows already constructed—and you can't take windows back
  - Cost to customer: $3000

I'll be passing a hat after class!

Software Engineering Lessons

- Common misperception: customers don't know what they want
  - Usually deeply wrong, even insulting to customer
  - Customer is the expert in his or her domain
- Real issue: customers can't express what they want without an expert's help
  - We knew we wanted big windows
  - If asked, we would have said we wanted windows as tall as we have them now
  - But we didn't have the expertise to know this was nonstandard

Lessons for Requirements Gathering

- Study customer's current context
  - e.g. what size windows they use right now
  - Ask what they like and don't like about this
  - Ask about any changes you are making!
    - Don't give them smaller windows without asking
- Think about the implications of what customers say
  - They may not know how to express their requirements directly
  - But they will usually give you clues
    - Customer asks about window width, lower side of window—maybe you should ask about window height
- Use prototypes
  - We had mock-up drawings of the kitchen
  - But none of them included side-by-side comparisons of existing window elevations to new window elevations
    - Had there been these, we would have caught the error
- Unfortunately even experts will sometimes get this wrong
  - But they can avoid many mistakes that novices make

Static Analysis

15-654:

Analysis of Software Artifacts

Jonathan Aldrich

Find the Bug!

Source: Engler et al., *Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '00.

```
/* From Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct stripe_head *sh,
                int b_size) {
        struct buffer_head *bh;
        unsigned long flags;

        save_flags(flags);
        cli();
        if ((bh = sh->buffer_pool) == NULL)
                return NULL;
        sh->buffer_pool = bh->b_next;
        bh->b_size = b_size;
        restore_flags(flags);
        return bh;
}
```

disable interrupts

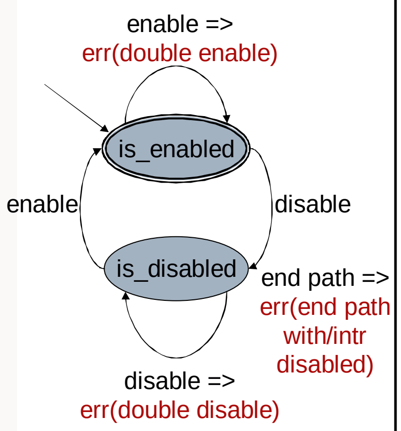
ERROR: returning with interrupts disabled

re-enable interrupts

Metal Interrupt Analysis

Source: Engler et al., *Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '00.

```
{ #include "linux-includes.h" }
sm check_interrupts {
  // Variables
  // used in patterns
  decl { unsigned } flags;

  // Patterns
  // to specify enable/disable functions.
  pat enable = { sti(); }
             | { restore_flags(flags); } ;
  pat disable = { cli(); };

  // States
  // The first state is the initial state.
  is_enabled: disable ==> is_disabled
     | enable ==> { err("double enable"); }
     ;
  is_disabled: enable ==> is_enabled
     | disable ==> { err("double disable"); }
     // Special pattern that matches when the SM
     // hits the end of any path in this state.
     | $end_of_path$ ==>
       { err("exiting w/intr disabled!"); }
     ;
}
```



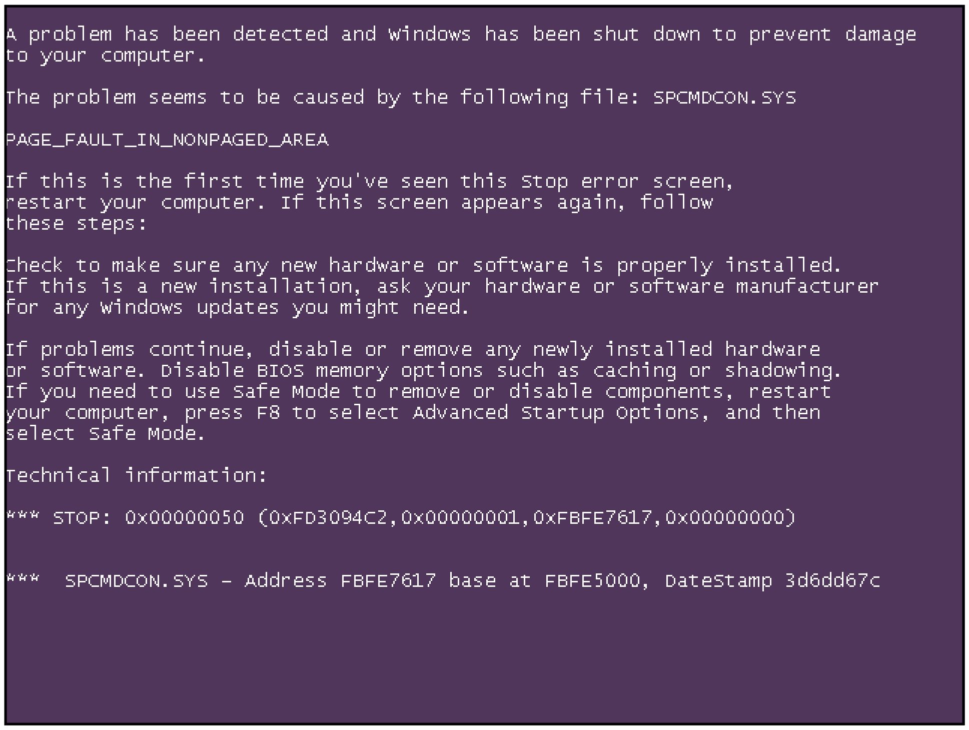
Applying the Analysis

Source: Engler et al., *Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '00.

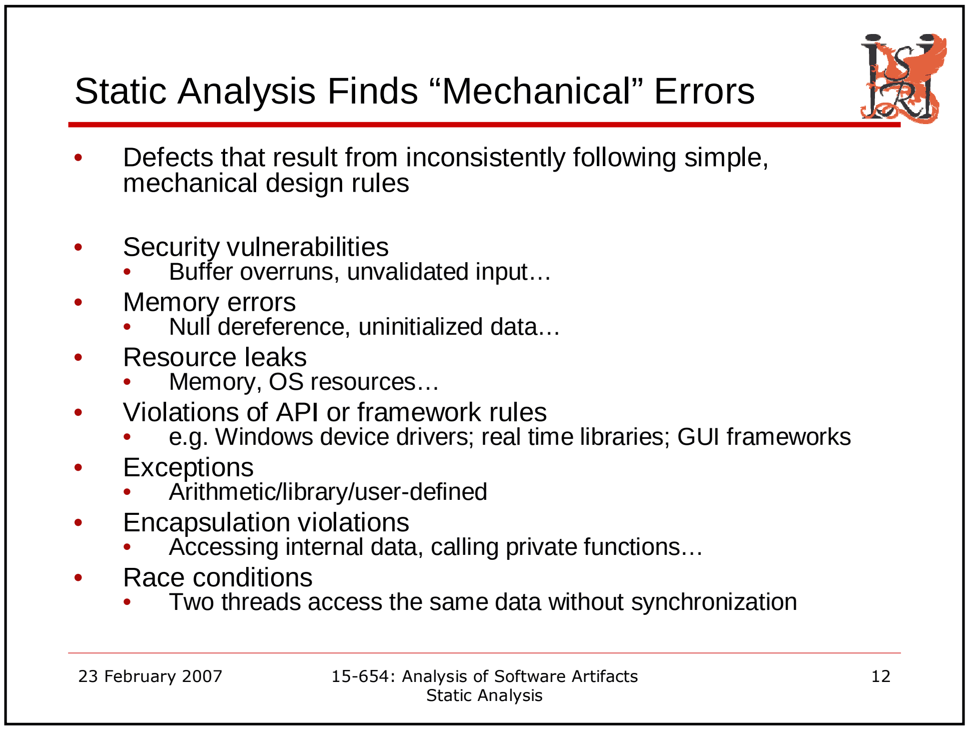
```
/* From Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct stripe_head *sh,      <- initial state is_enabled
                int b_size) {
        struct buffer_head *bh;
        unsigned long flags;

        save_flags(flags);
        cli();                               <- transition to is_disabled
        if ((bh = sh->buffer_pool) == NULL)
                return NULL;                 <- final state is_disabled: ERROR!
        sh->buffer_pool = bh->b_next;
        bh->b_size = b_size;
        restore_flags(flags);                <- transition to is_enabled
        return bh;                           <- final state is_enabled is OK
}
```

Outline

- Why static analysis?
  - The limits of testing and inspection
- What is static analysis?
- How does static analysis work?
- What are key issues for analysis of OO systems?
- What tools are available?
- How does it fit into my organization?



```
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
```



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

Difficult to Find with Testing, Inspection

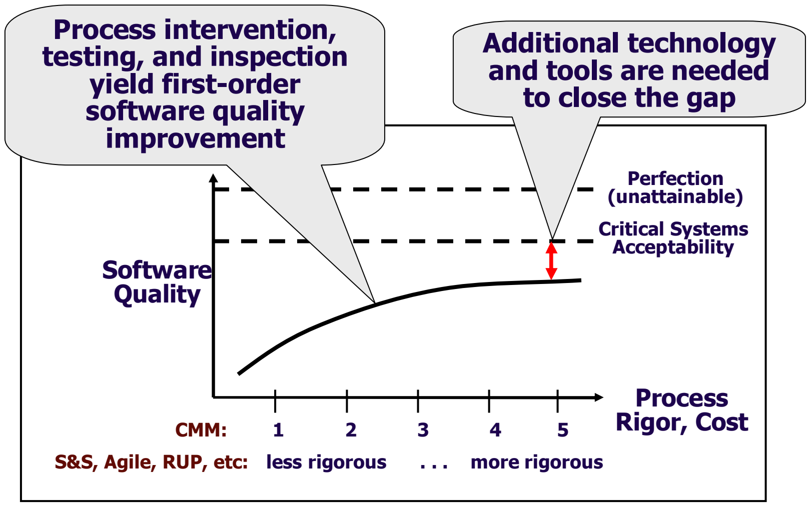
- Non-local, uncommon paths
  - Security vulnerabilities
  - Memory errors
  - Resource leaks
  - Violations of API or framework rules
  - Exceptions
  - Encapsulation violations
- Non-deterministic
  - Race conditions

Quality Assurance at Microsoft (Part 1)

- 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
  - More on this later

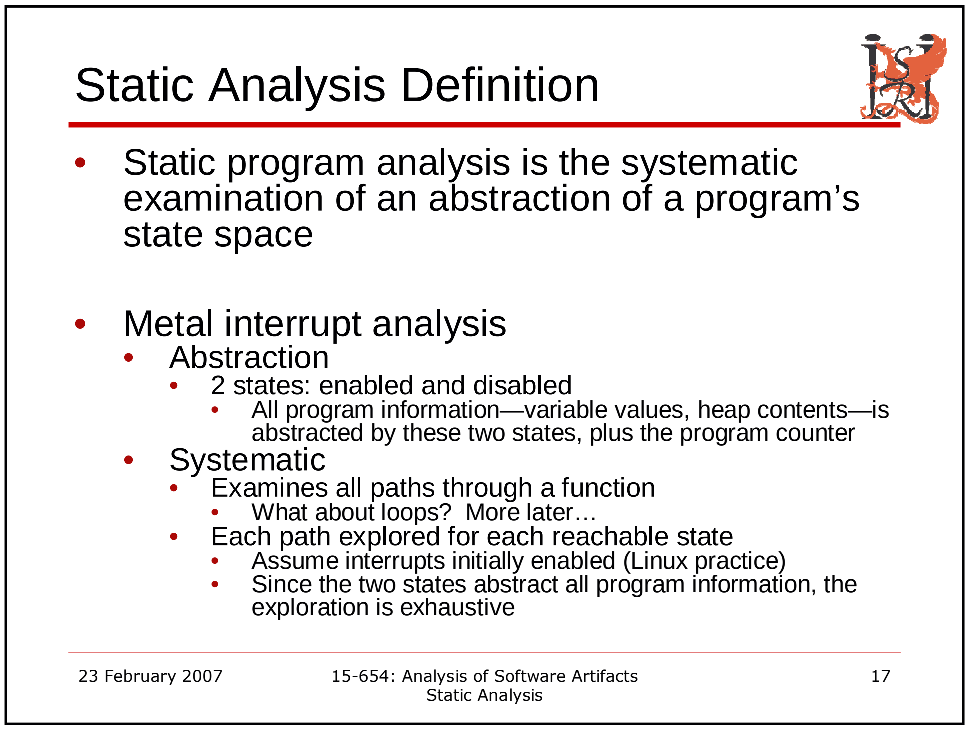
Process, Cost, and Quality

Slide: William Scherlis



Outline

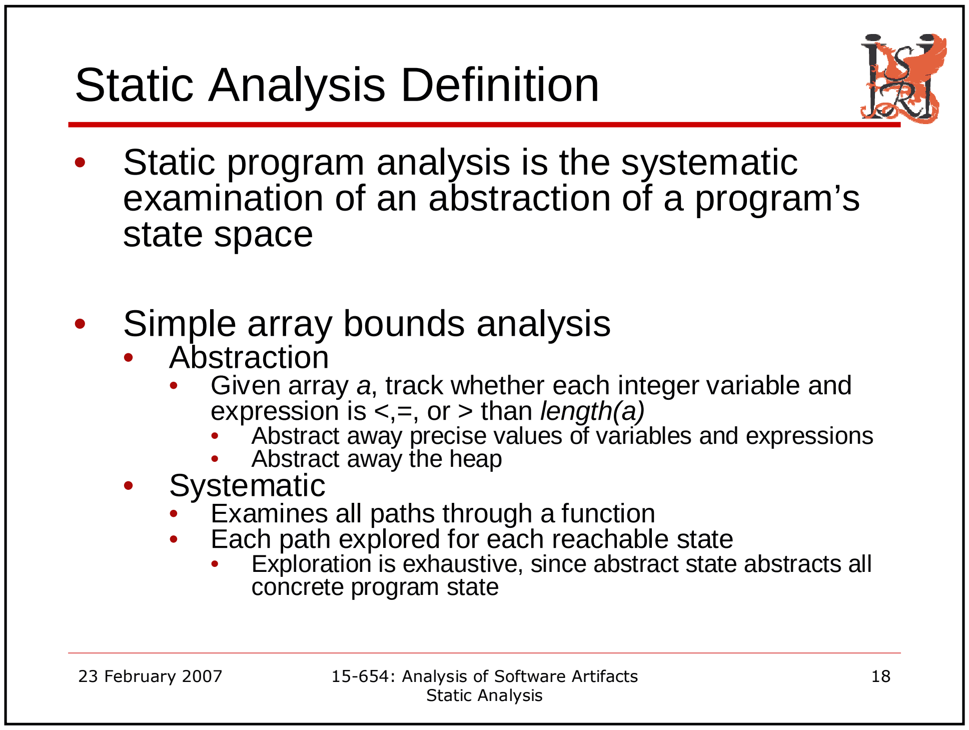
- Why static analysis?
- What is static analysis?
  - Abstract state space exploration
- How does static analysis work?
- What are key issues for analysis of OO systems?
- What tools are available?
- How does it fit into my organization?



Static Analysis Definition

- Static program analysis is the systematic examination of an abstraction of a program's state space

- Metal interrupt analysis
  - Abstraction
    - 2 states: enabled and disabled
      - All program information—variable values, heap contents—is abstracted by these two states, plus the program counter
  - Systematic
    - Examines all paths through a function
      - What about loops? More later…
    - Each path explored for each reachable state
      - Assume interrupts initially enabled (Linux practice)
      - Since the two states abstract all program information, the exploration is exhaustive



Static Analysis Definition

- Static program analysis is the systematic examination of an abstraction of a program's state space

- Simple array bounds analysis
  - Abstraction
    - Given array *a*, track whether each integer variable and expression is <,=, or > than *length(a)*
      - Abstract away precise values of variables and expressions
      - Abstract away the heap
  - Systematic
    - Examines all paths through a function
    - Each path explored for each reachable state
      - Exploration is exhaustive, since abstract state abstracts all concrete program state

Array Bounds Example

```
1.  void foo(unsigned n) {
2.      char str = new char[n+1];
3.      int idx = 0;
4.      if (n > 5)
5.              idx = n
6.      else
7.              idx = n+1
8.      str[idx] = ‘c’;
9.  }
```

Path 1 (before stmt): then branch

2: ∅
3: n↦<
4: n↦<, idx↦<
5: n↦<, idx↦<
8: n↦<, idx↦<
9: n↦<, idx↦<

no errors

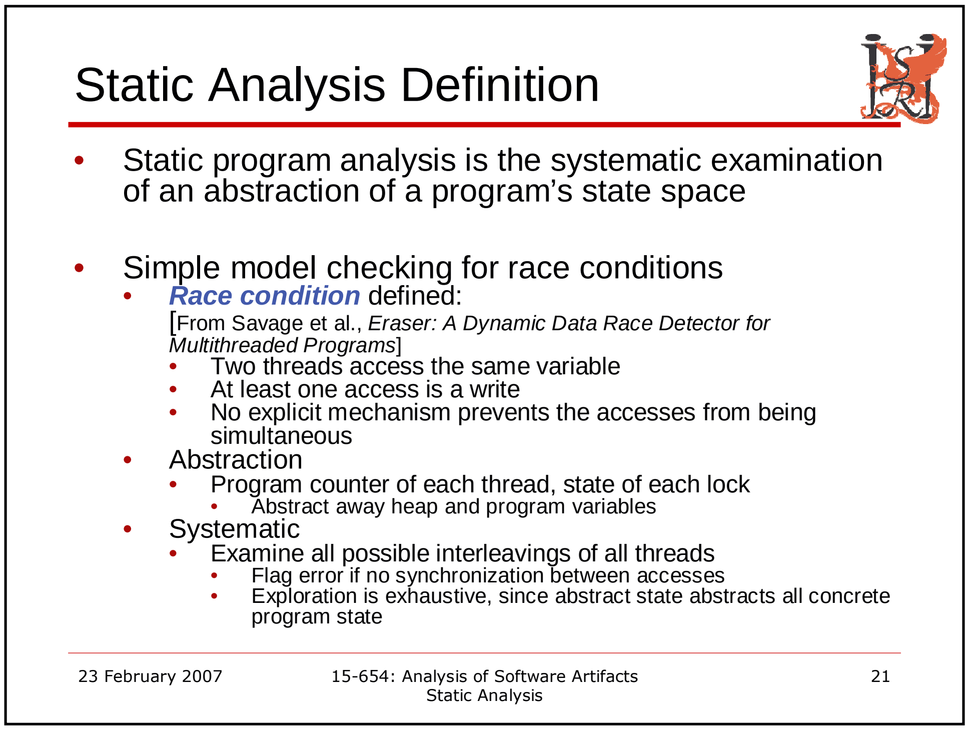
Array Bounds Example

```
1.  void foo(unsigned n) {
2.      char str = new char[n+1];
3.      int idx = 0;
4.      if (n > 5)
5.              idx = n
6.      else
7.              idx = n+1
8.      str[idx] = ‘c’;
9.  }
```

Path 1 (before stmt): else branch

2: ∅
3: n↦<
4: n↦<, idx↦<
7: n↦<, idx↦<,=
8: n↦<, idx↦<,=
9: n↦<, idx↦<,=

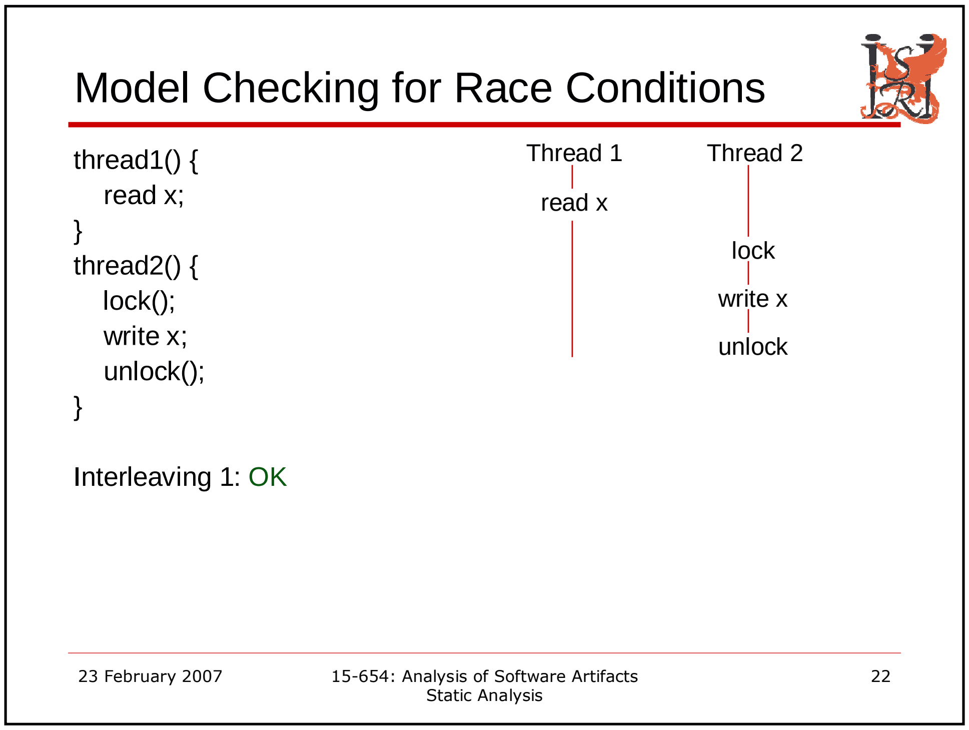
error: array out of bounds at line 8



Static Analysis Definition

- Static program analysis is the systematic examination of an abstraction of a program's state space
- Simple model checking 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

23 February 2007 | 15-654: Analysis of Software Artifacts Static Analysis | 21

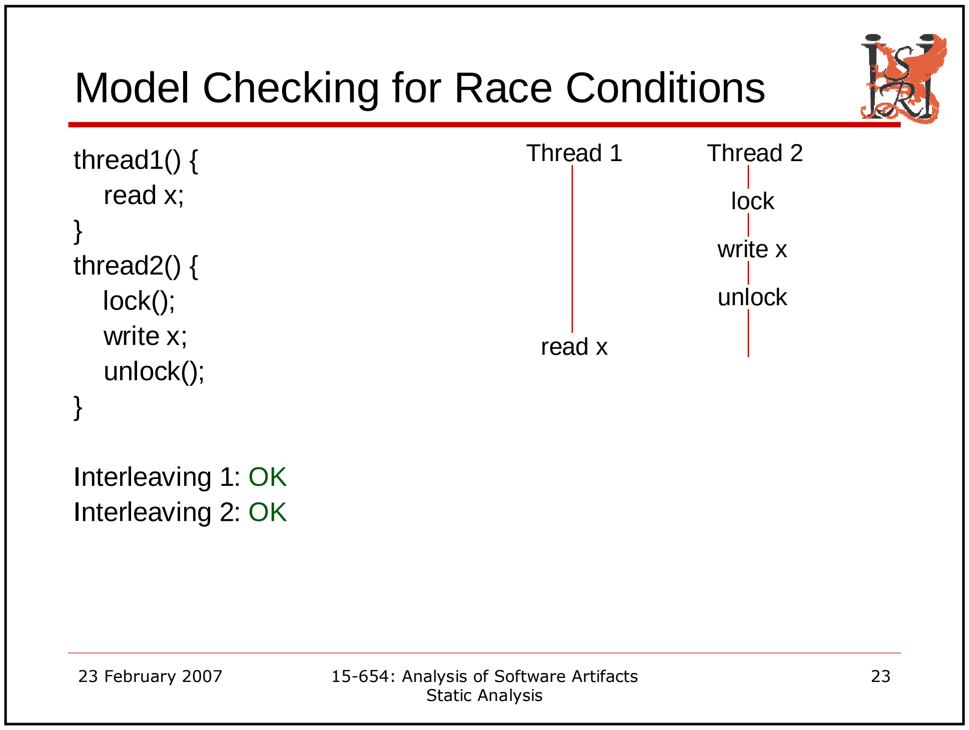


Model Checking for Race Conditions

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

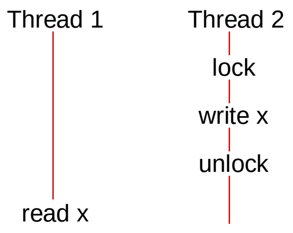
Interleaving 1: OK

23 February 2007 | 15-654: Analysis of Software Artifacts Static Analysis | 22

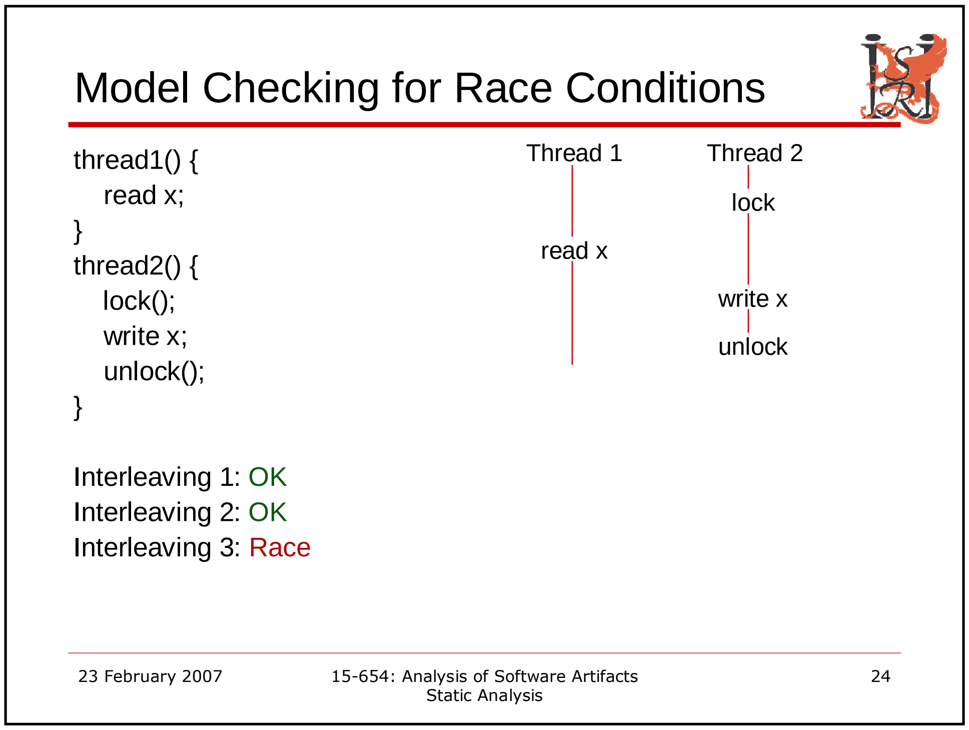


Model Checking for Race Conditions

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

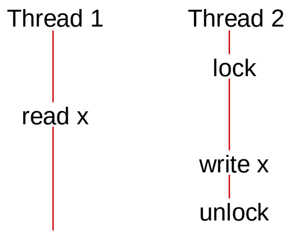


Interleaving 1: OK
Interleaving 2: OK



Model Checking for Race Conditions

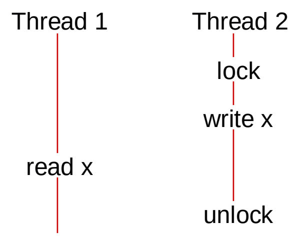
```
thread1() {
    read x;
}
thread2() {
    lock();
    write x;
    unlock();
}
```



Interleaving 1: OK
Interleaving 2: OK
Interleaving 3: Race

Model Checking for Race Conditions

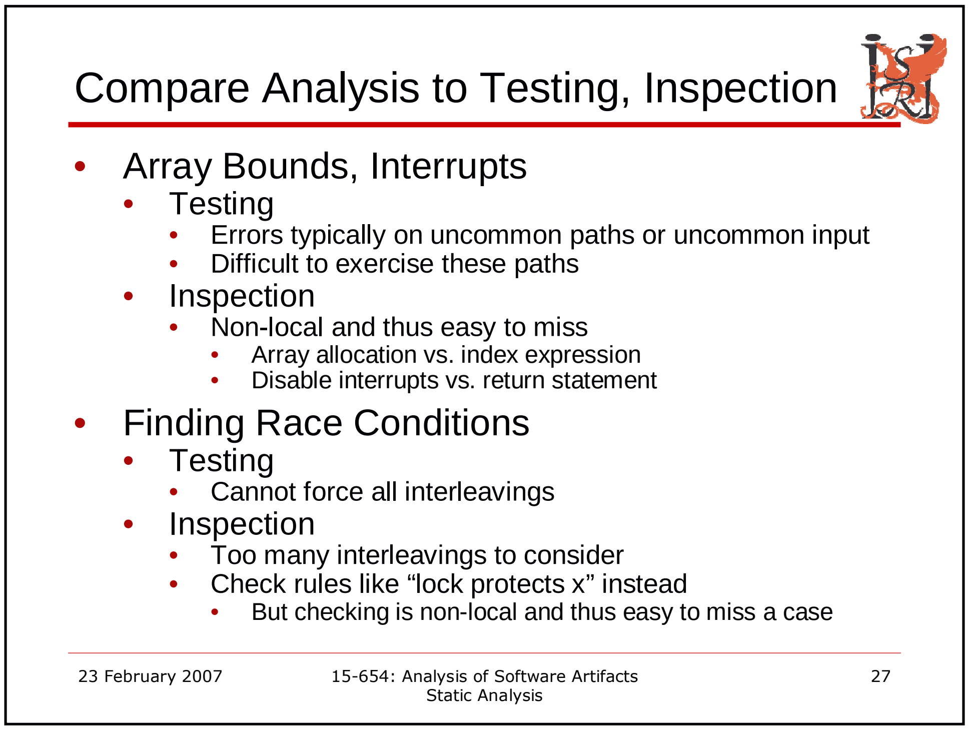
```
thread1() {
    read x;
}
thread2() {
    lock();
    write x;
    unlock();
}
```



Interleaving 1: OK
Interleaving 2: OK
Interleaving 3: Race
Interleaving 4: Race

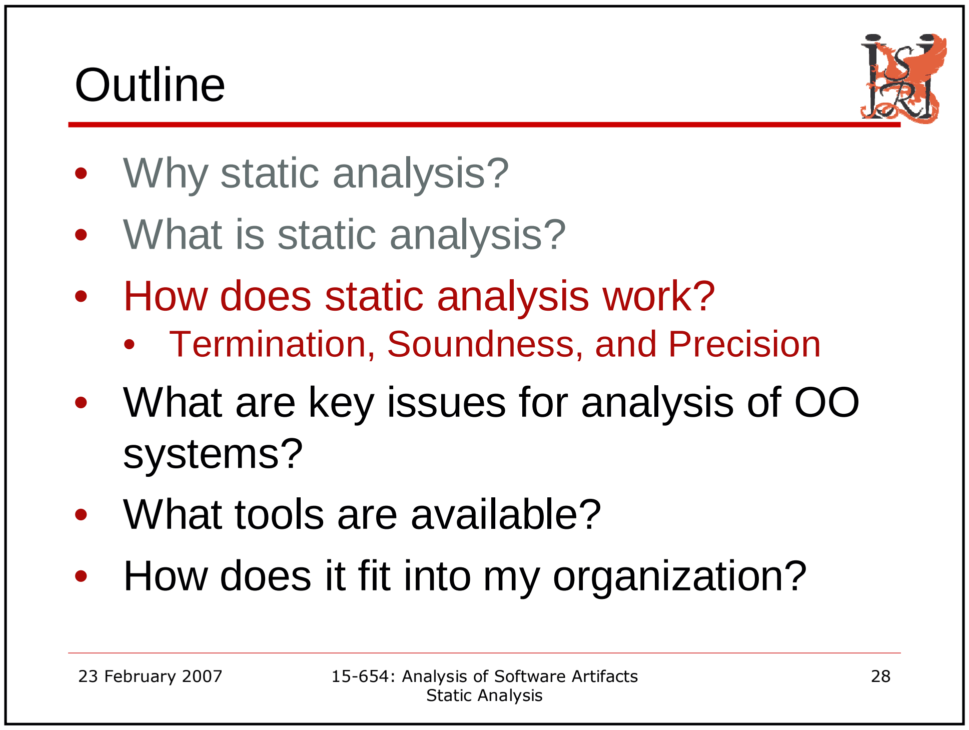
Compare Analysis to Testing, Inspection

- Why might it be hard to test/inspect for:
  - Array bounds errors?
  - Forgetting to re-enable interrupts?
  - Race conditions?



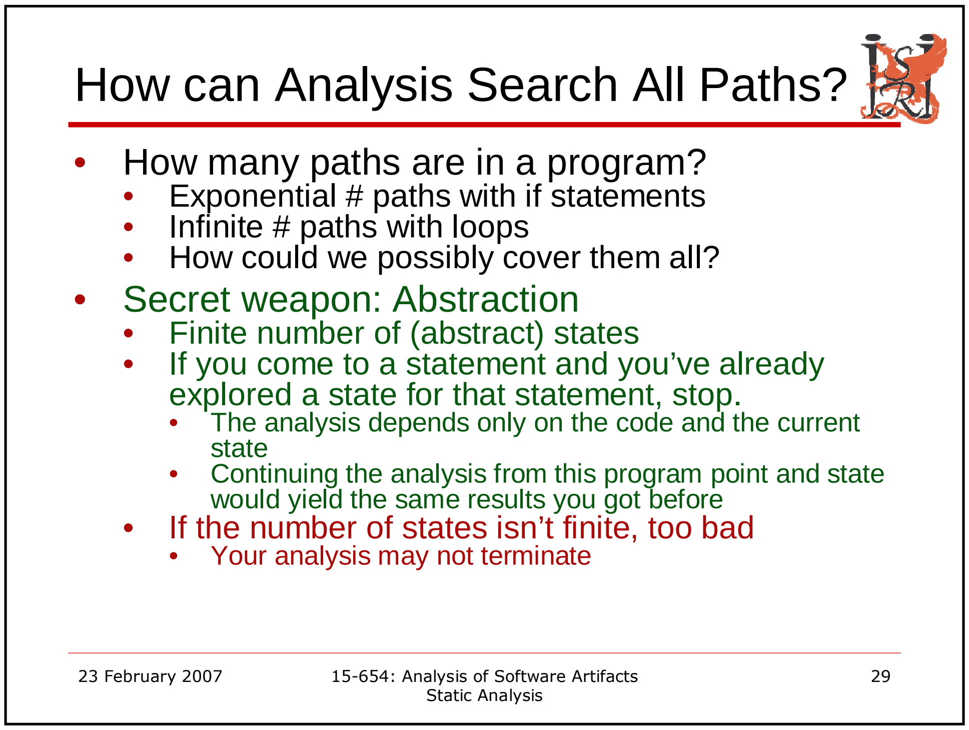
Compare Analysis to Testing, Inspection

- Array Bounds, Interrupts
  - Testing
    - Errors typically on uncommon paths or uncommon input
    - Difficult to exercise these paths
  - Inspection
    - Non-local and thus easy to miss
      - Array allocation vs. index expression
      - Disable interrupts vs. return statement
- Finding Race Conditions
  - Testing
    - Cannot force all interleavings
  - Inspection
    - Too many interleavings to consider
    - Check rules like “lock protects x” instead
      - But checking is non-local and thus easy to miss a case



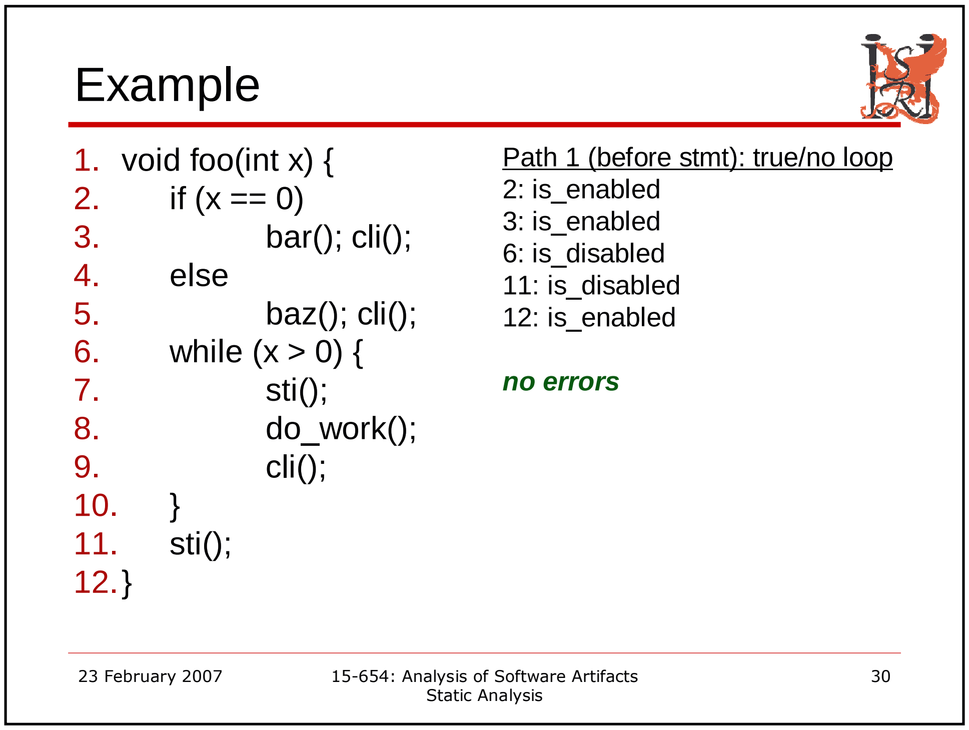
Outline

- Why static analysis?
- What is static analysis?
- How does static analysis work?
  - Termination, Soundness, and Precision
- What are key issues for analysis of OO systems?
- What tools are available?
- How does it fit into my organization?



How can Analysis Search All Paths?

- How many paths are in a program?
  - Exponential # paths with if statements
  - Infinite # paths with loops
  - How could we possibly cover them all?
- Secret weapon: Abstraction
  - Finite number of (abstract) states
  - If you come to a statement and you've already explored a state for that statement, stop.
    - The analysis depends only on the code and the current state
    - Continuing the analysis from this program point and state would yield the same results you got before
  - If the number of states isn't finite, too bad
    - Your analysis may not terminate



Example

```
1.  void foo(int x) {
2.      if (x == 0)
3.              bar(); cli();
4.      else
5.              baz(); cli();
6.      while (x > 0) {
7.              sti();
8.              do_work();
9.              cli();
10.     }
11.     sti();
12. }
```

<u>Path 1 (before stmt): true/no loop</u>
2: is_enabled
3: is_enabled
6: is_disabled
11: is_disabled
12: is_enabled

no errors

Example

```
1.  void foo(int x) {
2.      if (x == 0)
3.              bar(); cli();
4.      else
5.              baz(); cli();
6.      while (x > 0) {
7.              sti();
8.              do_work();
9.              cli();
10.     }
11.     sti();
12. }
```

Path 2 (before stmt): true/1 loop

2: is_enabled
3: is_enabled
6: is_disabled
7: is_disabled
8: is_enabled
9: is_enabled
11: is_disabled

already been here

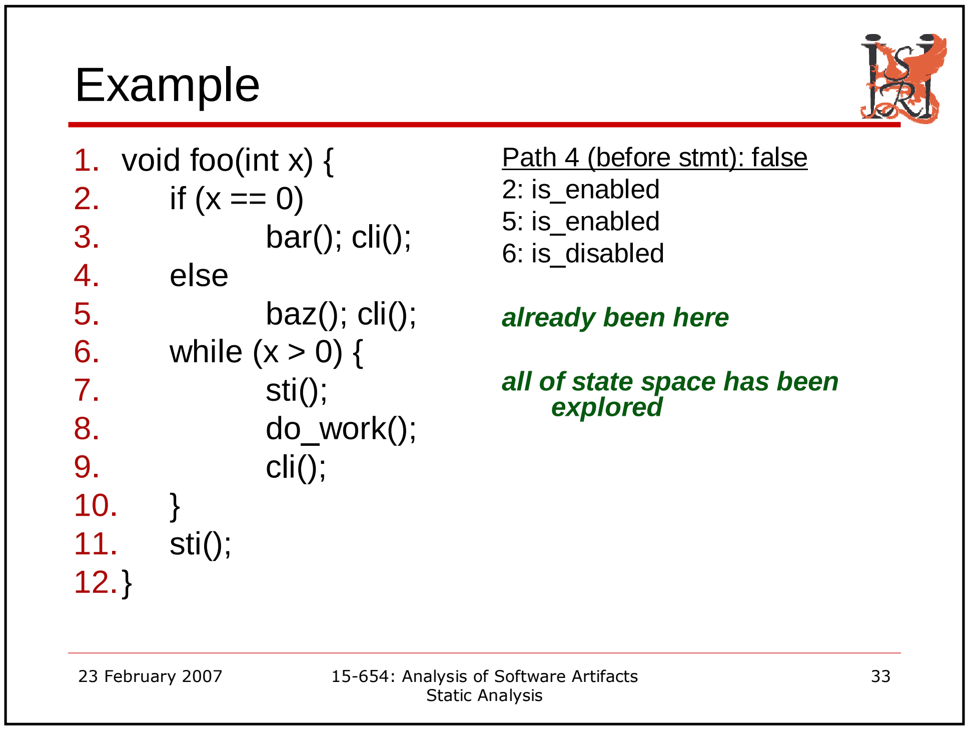
Example

```
1.  void foo(int x) {
2.      if (x == 0)
3.              bar(); cli();
4.      else
5.              baz(); cli();
6.      while (x > 0) {
7.              sti();
8.              do_work();
9.              cli();
10.     }
11.     sti();
12. }
```

Path 3 (before stmt): true/2+ loops

2: is_enabled
3: is_enabled
6: is_disabled
7: is_disabled
8: is_enabled
9: is_enabled
6: is_disabled

already been here



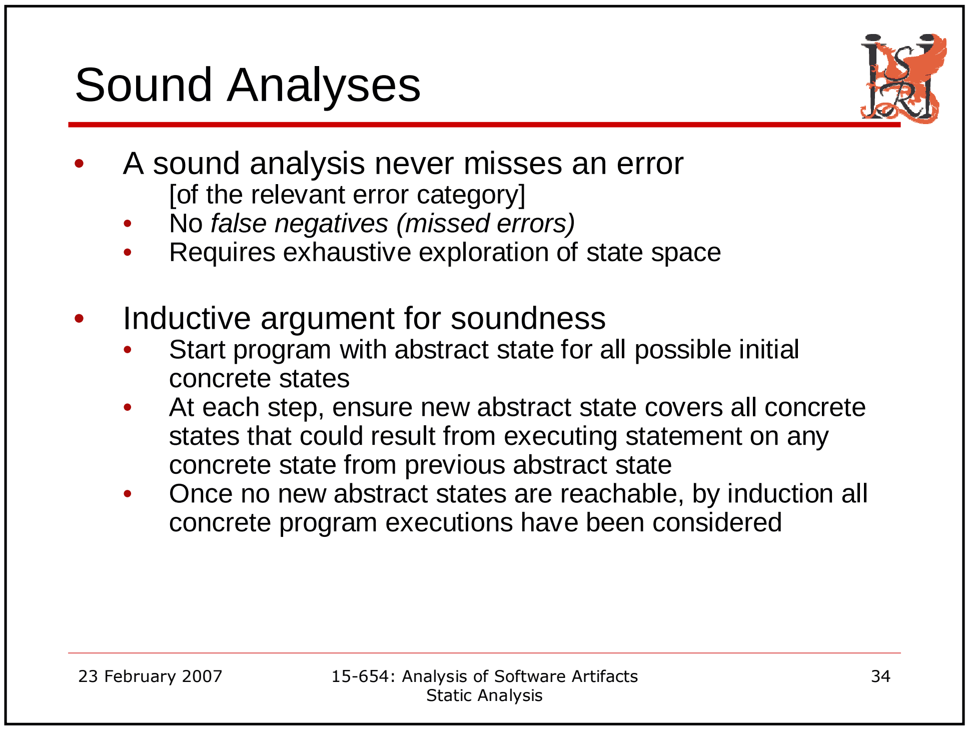
Example

```
1.  void foo(int x) {
2.      if (x == 0)
3.          bar(); cli();
4.      else
5.          baz(); cli();
6.      while (x > 0) {
7.          sti();
8.          do_work();
9.          cli();
10.     }
11.     sti();
12. }
```

Path 4 (before stmt): false
2: is_enabled
5: is_enabled
6: is_disabled

already been here

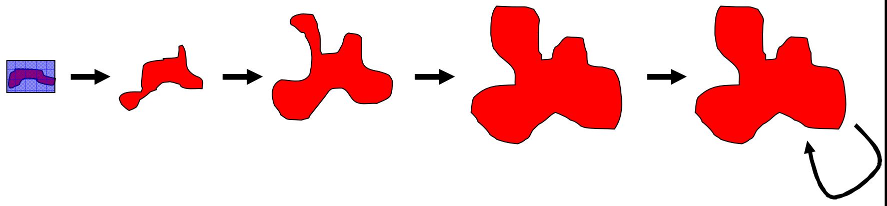
all of state space has been explored



Sound Analyses

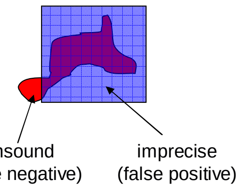
- A sound analysis never misses an error
  [of the relevant error category]
  - No *false negatives (missed errors)*
  - Requires exhaustive exploration of state space
- Inductive argument for soundness
  - Start program with abstract state for all possible initial concrete states
  - At each step, ensure new abstract state covers all concrete states that could result from executing statement on any concrete state from previous abstract state
  - Once no new abstract states are reachable, by induction all concrete program executions have been considered

Soundness and Precision

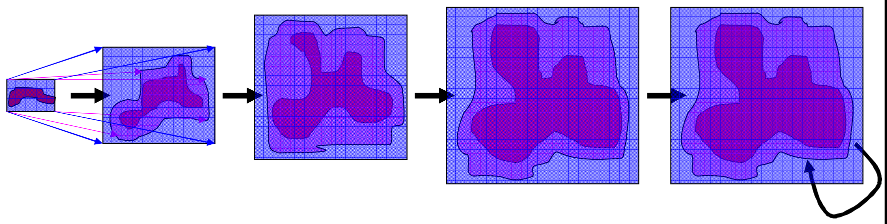


🟥 Program state covered in actual execution

🟦 Program state covered by abstract execution with analysis



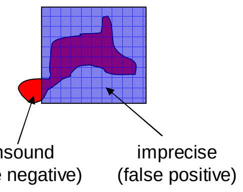
Soundness and Precision



🟥 Program state covered in actual execution

🟪 Program state covered by executing from abstract state

🟦 Program state covered by abstract execution with analysis



Abstraction and Soundness

- Consider “Sound Testing”
  [testing that finds every bug]
  - Requires executing program on every input
    - (and on all interleavings of threads)
  - Infinite number of inputs for realistic programs
    - Therefore impossible in practice

- Abstraction
  - Infinite state space → finite set of states
  - Can achieve soundness by exhaustive exploration

Array Bounds Precision

```
1.  void foo(unsigned n) {
2.      char str = new char[n+1];
3.      int idx = n-1;
4.      idx = idx+1;
5.      str[idx] = ‘c’;
6.  }
```

What will be the result of static analysis?

Path 1 (before stmt):
2: $\emptyset$
3: $n \mapsto <$
4: $n \mapsto <$, $idx \mapsto <$
5: $n \mapsto <$, $idx \mapsto <,=$
6: $n \mapsto <$, $idx \mapsto <,=$

error: array out of bounds at line 5
False positive! (not a real error)

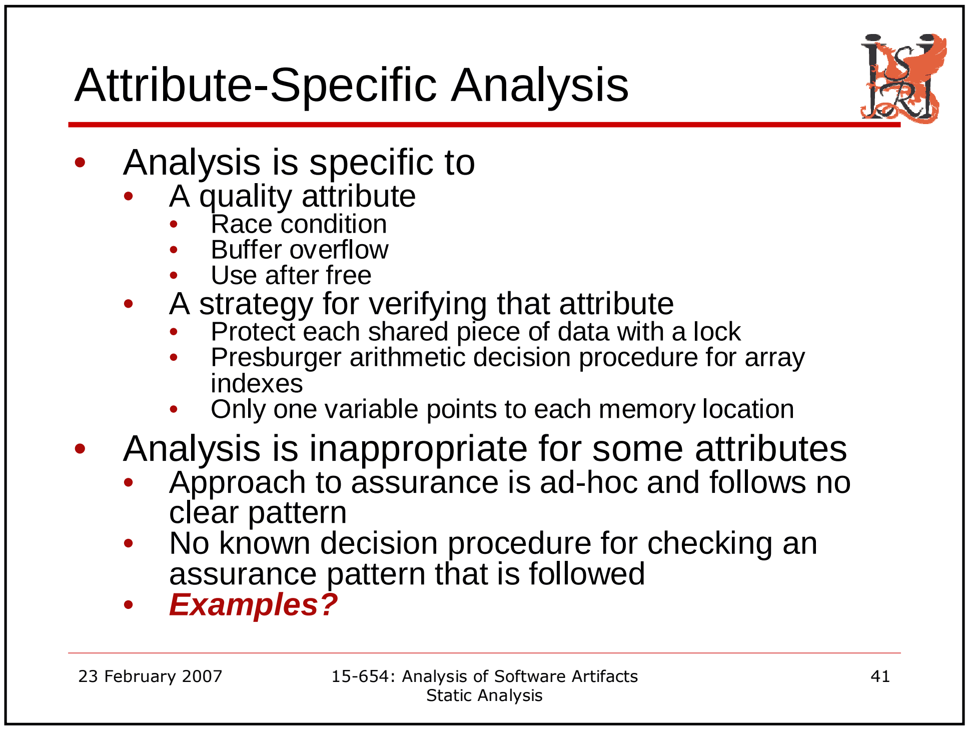
What went wrong?
- At statement 4 we only know idx < length(str)
- We need to know idx < length(str)-1

Regaining Array Bounds Precision

- Keep track of exact value of index
  - Infinite states
    - or 2^{32}, close enough
- Add a <-1 state
  - Not general enough
- Track formula relating expressions to arrays
  - Undecidable for arbitrary formulas
- Track restricted formulas
  - Decent solution in practice
    - Presburger arithmetic

Analysis as an Approximation

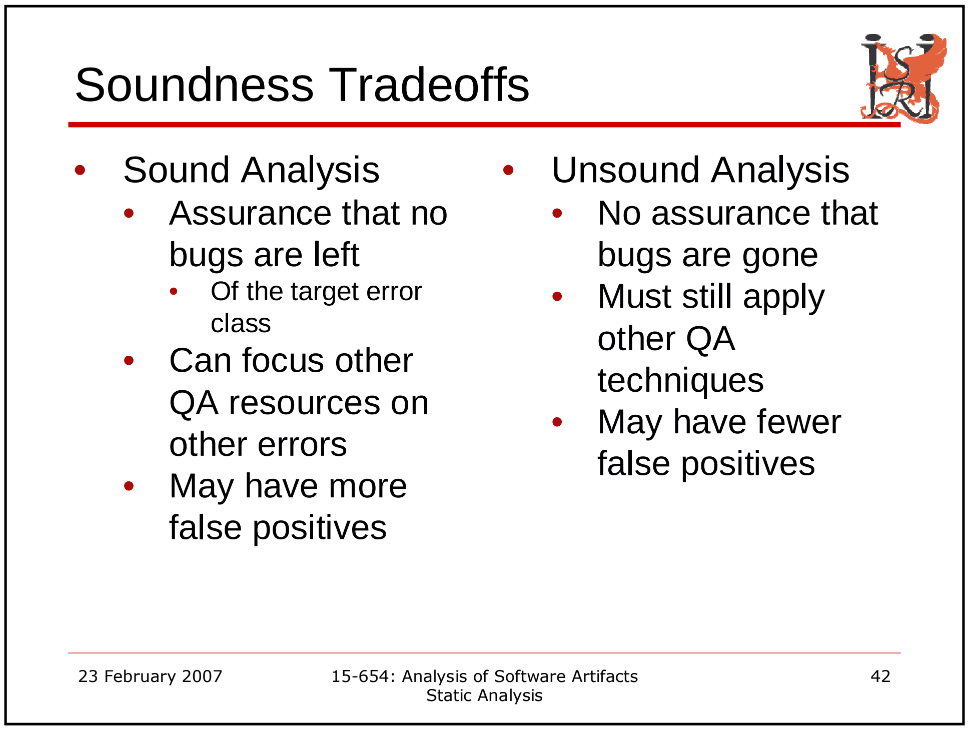
- Analysis must approximate in practice
  - May report errors where there are really none
    - False positives
  - May not report errors that really exist
    - False negatives
  - All analysis tools have either false negatives or false positives
- Approximation strategy
  - Find a pattern P for correct 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



Attribute-Specific Analysis

- Analysis is specific to
  - A quality attribute
    - Race condition
    - Buffer overflow
    - Use after free
  - A strategy for verifying that attribute
    - Protect each shared piece of data with a lock
    - Presburger arithmetic decision procedure for array indexes
    - Only one variable points to each memory location
- Analysis is inappropriate for some attributes
  - Approach to assurance is ad-hoc and follows no clear pattern
  - No known decision procedure for checking an assurance pattern that is followed
  - ***Examples?***

23 February 2007 — 15-654: Analysis of Software Artifacts Static Analysis — 41



Soundness Tradeoffs

- Sound Analysis
  - Assurance that no bugs are left
    - Of the target error class
  - Can focus other QA resources on other errors
  - May have more false positives
- Unsound Analysis
  - No assurance that bugs are gone
  - Must still apply other QA techniques
  - May have fewer false positives

23 February 2007 — 15-654: Analysis of Software Artifacts Static Analysis — 42

Which to Choose?

- Cost/Benefit tradeoff
  - Benefit: How valuable is the bug?
    - How much does it cost if not found?
    - How expensive to find using testing/inspection?
  - Cost: How much did the analysis cost?
    - Effort spent running analysis, interpreting results – includes false positives
    - Effort spent finding remaining bugs (for unsound analysis)
- Rule of thumb
  - For critical bugs that testing/inspection can't find, a sound analysis is worth it
    - As long as false positive rate is acceptable
  - For other bugs, maximize engineer productivity

Outline

- Why static analysis?
- What is static analysis?
- How does static analysis work?
- What features make analysis practical?
  - Design intent, modularity, incrementality, and immediate reward
  - Illustration: Microsoft's PreFAST tool
- How to fit analysis into an organization?

Standard Annotation Language (SAL)

- A language for specifying contracts between functions
  - Intended to be lightweight and practical
  - More powerful—but less practical—contracts supported in systems like ESC/Java or Spec#
- Preconditions
  - Conditions that hold on entry to a function
  - What a function expects of its callers
- Postconditions
  - Conditions that hold on exiting a function
  - What a function promises to its callers
- 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

Demonstration

Buffer/Pointer Annotations

- Format
  - Leading underscore
  - List of components below

_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

Other Useful Annotations

__checkReturn	Callers must check the return value.

- .Net Annotation Format
  - Pre/Post attribute with arguments for the pre or postcondition
  - Surrounded in brackets
  - Alternative for the annotations above
  - Required for Tainted

[SA_Pre(Tainted=SA_Yes)]	This argument is tainted and cannot be trusted without validation.
[SA_Pre(Tainted=SA_No)]	This argument is not tainted and can be trusted
[SA_Post(Tainted=SA_No)]	Same as above, but useful as a postcondition

Other Supported Annotations

- How to test if this function succeeded
- How much of the buffer is initialized?
- Is a string null-terminated?
- Is an argument reserved?
- Is this an overriding method?
- Is this function a callback?
- Is this used as a format string?
- What resources might this function block on?
- Is this a fallthrough case in a switch?

SAL: the Benefit of Annotations

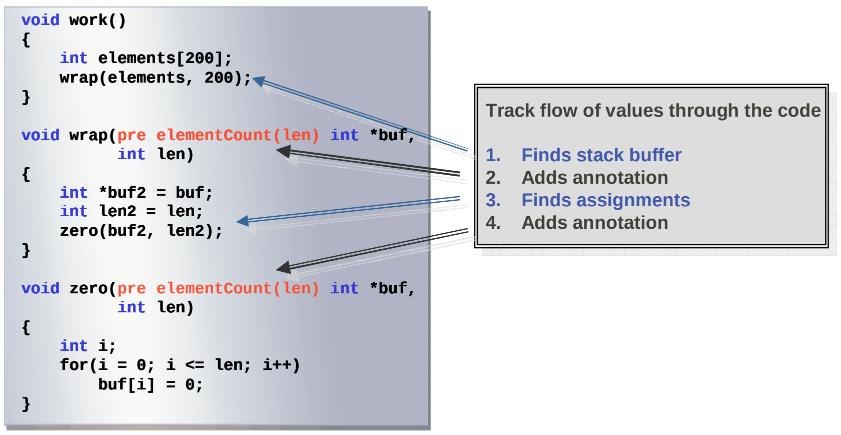
- Annotations express *design intent*
  - How you intended to achieve a particular quality attribute
    - e.g. never writing more than N elements to this array
- As you add more annotations, you find more errors
  - Get checking of library users for free
  - Plus, those errors are less likely to be false positives
    - The analysis doesn't have to guess your intention
- Annotations also improve scalability
  - 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?
- Microsoft approach
  - 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
      - May need to change annotations
      - Hopefully saves work overall
    - Unfortunately not yet available outside Microsoft

Case Study: SALinfer

[Source: Manuvir Das]



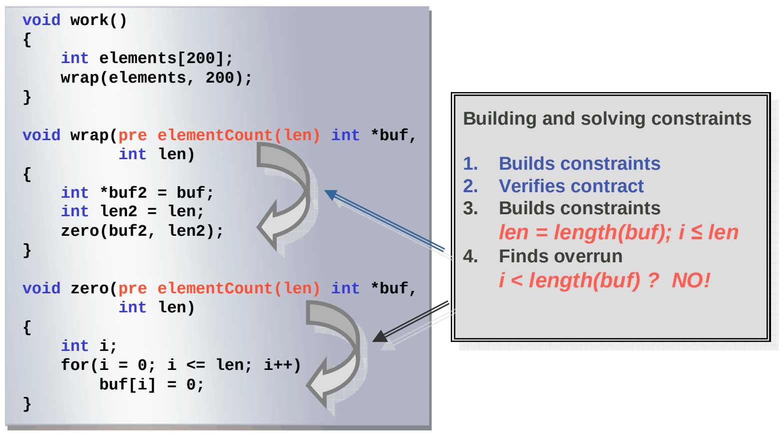
```
void work()
{
    int elements[200];
    wrap(elements, 200);
}

void wrap(pre elementCount(len) int *buf,
          int len)
{
    int *buf2 = buf;
    int len2 = len;
    zero(buf2, len2);
}

void zero(pre elementCount(len) int *buf,
          int len)
{
    int i;
    for(i = 0; i <= len; i++)
        buf[i] = 0;
}
```

Track flow of values through the code

1. **Finds stack buffer**
2. **Adds annotation**
3. **Finds assignments**
4. **Adds annotation**

Case Study: SAL verification

[Source: Manuvir Das]



```
void work()
{
    int elements[200];
    wrap(elements, 200);
}

void wrap(pre elementCount(len) int *buf,
          int len)
{
    int *buf2 = buf;
    int len2 = len;
    zero(buf2, len2);
}

void zero(pre elementCount(len) int *buf,
          int len)
{
    int i;
    for(i = 0; i <= len; i++)
        buf[i] = 0;
}
```

Building and solving constraints

1. Builds constraints
2. Verifies contract
3. Builds constraints
   len = length(buf); i ≤ len
4. Finds overrun
   i < length(buf) ? NO!

Available as part of Microsoft Visual Studio 2005

Recommendations

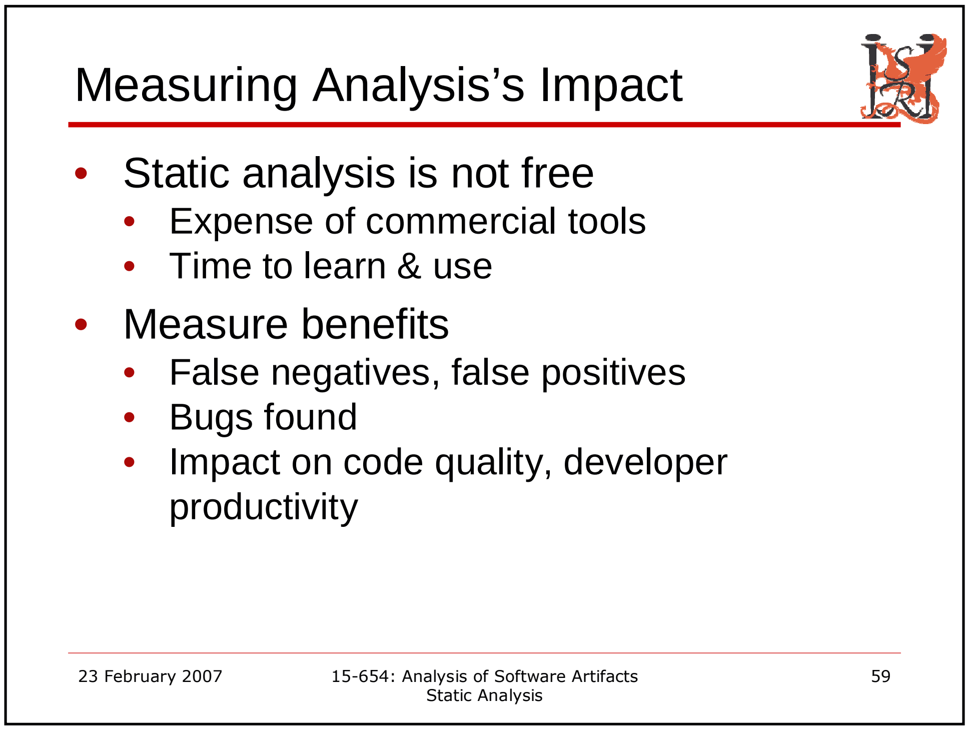
- If you use Microsoft's tools…
  - Turn on /analyze
  - Annotate all functions that write to buffers
  - Annotate all library functions
  - Annotation other functions as possible

Outline

- Why static analysis?
- What is static analysis?
- How does static analysis work?
- What are key issues for analysis of OO systems?
- What tools are available?
- How does it fit into my organization?
  - Lessons learned at Microsoft: Introduction, measurement, refinement, check in gates
    - Source for Microsoft experience: Manuvir Das

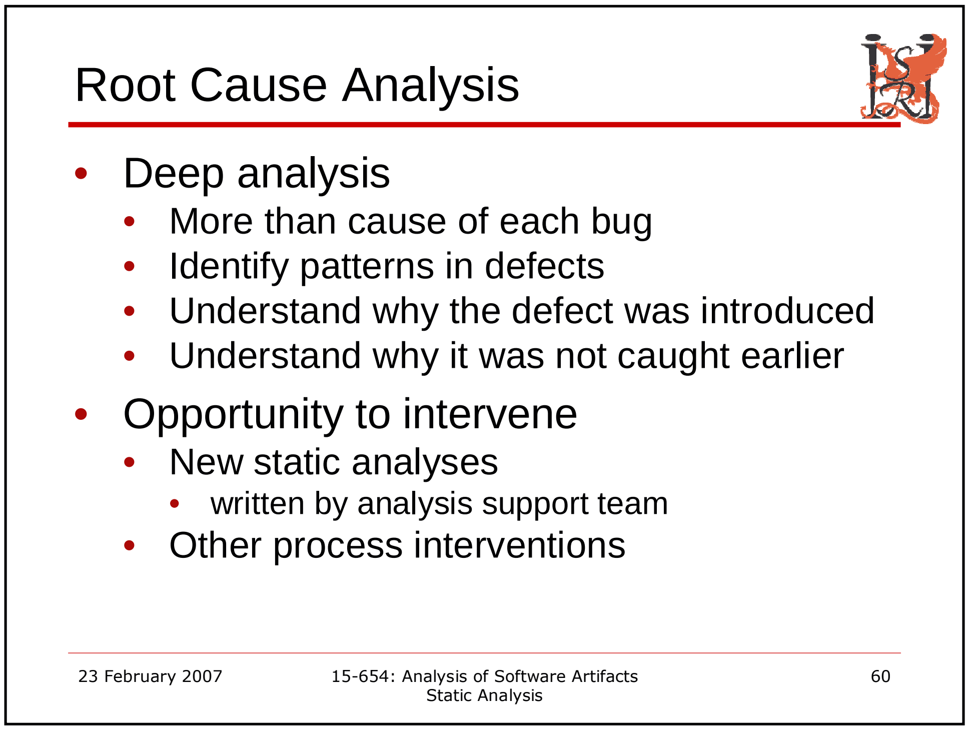
Introducing Static Analysis

- Incremental approach
  - Begin with early adopters, small team
  - Use these as champions in organization
- Choose/build the tool right
  - Not too many false positives
  - Good error reporting
    - Show error context, trace
  - Focus on big issues
    - Something developers, company cares about
  - Ensure you can teach the tool
    - Suppress false positive warnings
    - Add design intent for assertions, assumptions
  - Bugs should be fixable [Manuvir Das]
    - Easy to fix, easy to verify, robust to small changes
- Support team
  - Answer questions, help with tool



Measuring Analysis's Impact

- Static analysis is not free
  - Expense of commercial tools
  - Time to learn & use
- Measure benefits
  - False negatives, false positives
  - Bugs found
  - Impact on code quality, developer productivity



Root Cause Analysis

- Deep analysis
  - More than cause of each bug
  - Identify patterns in defects
  - Understand why the defect was introduced
  - Understand why it was not caught earlier
- Opportunity to intervene
  - New static analyses
    - written by analysis support team
  - Other process interventions

Check-in Gates

- Microsoft practice
  - Cannot check in code unless analysis suite has been run and produced no errors
    - Test coverage, dependency violation, insufficient/bad design intent, integer overflow, allocation arithmetic, buffer overruns, memory errors, security issues
- Requirements for success
  - Low false positives
  - A way to override false positive warnings
    - Typically through inspection
  - Developers must buy into static analysis first

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

Contrasting Case Study

- Web company
  - Security, server reliability issues crucial
- Contrast to Microsoft
  - Agile process—no room for extra QA stage
  - Fewer resources for in-house development
- Choice: FindBugs
  - Customize Eclipse plugin to local environment
  - Pick and choose most important analyses
  - Put on developers' desktops
  - Quality gates for lifecycle transitions
  - Writing analyses to capture common issues

Analysis Maturity Model

Caveat: not yet enough experience to make strong claims

- Level 1: use typed languages, ad-hoc tool use
- Level 2: run off-the-shelf tools as part of process
- Level 3: integrate tools into process
  - check in quality gates, milestone quality gates
  - integrate into build process, developer environments
  - pick and choose analyses which are most useful
  - use annotations/settings to teach tool about internal libraries
- Level 4: customized analyses for company domain
  - extend analysis tools to catch observed problems
- Level 5: continual optimization of analysis infrastructure
  - mine patterns in bug reports
  - gather data on analysis effectiveness
  - tune analysis based on observations

The Analysis Revolution

- Analysis is revolutionizing QA practices in leading companies today
- Exhibit A: Microsoft
  - Comprehensive analysis is centerpiece of QA for Longhorn (Windows)
  - Now affects every part of the engineering process
- Analysis technology
  - Enables organizations to increase quality while enhancing functionality
  - Will differentiate tomorrow's leaders in the market
- Now is the time to leverage analysis for QA

Questions?