Ensuring Software Quality with Static Analysis Tools

Jonathan Aldrich

Assistant Professor

Carnegie Mellon University

CMU - Los Alamos seminar, May 14, 2008

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

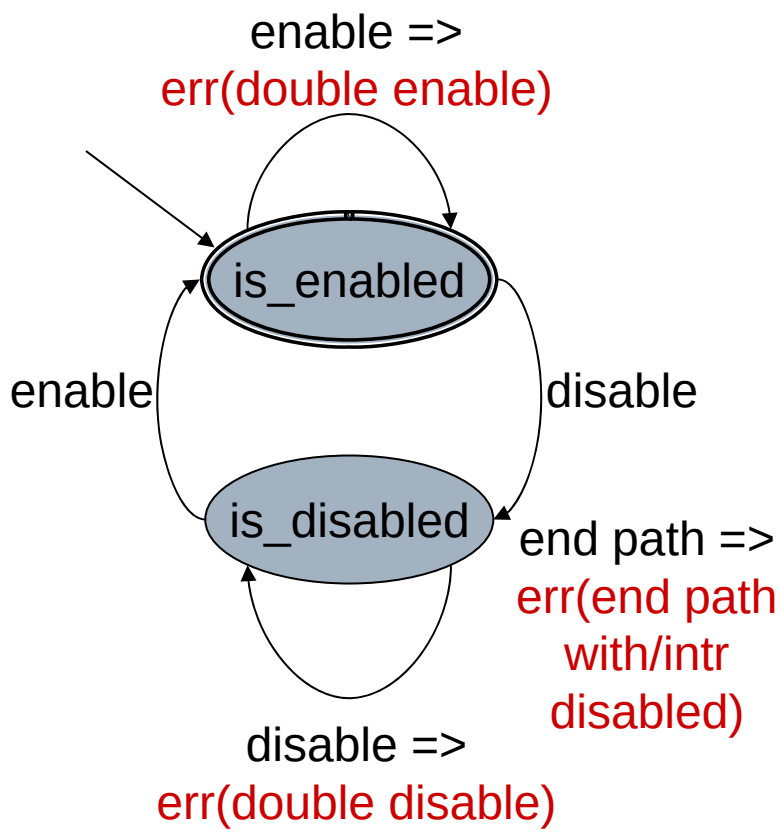
ERROR: returning with interrupts disabled

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.

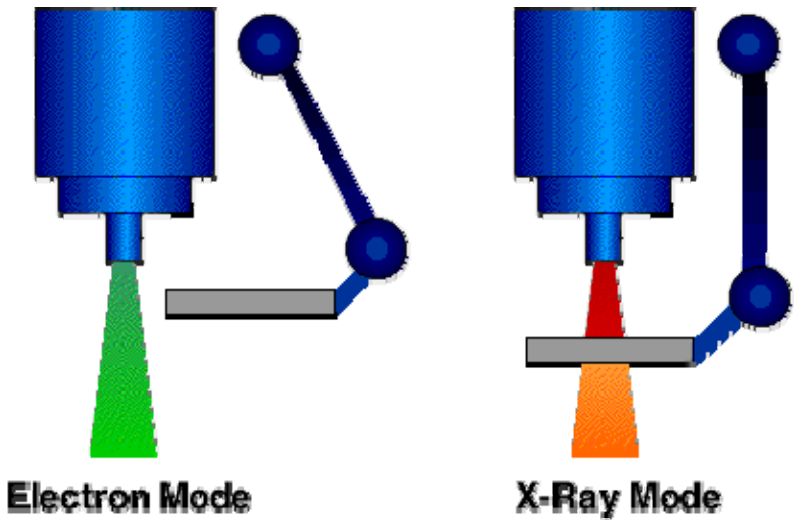
```
/* From Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct stripe_head *sh,        <-- initial state is_enabled
                int b_size) {
        struct buffer_head *bh;
        unsigned long flags;

        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 does the future hold?
- What tools are available?
- How does it fit into my organization?

Software Disasters: Therac-25

- Delivered radiation treatment
- 2 modes
  - Electron: low power electrons
  - X-Ray: high power electrons converted to x-rays with shield
- Race condition
  - Operator specifies x-ray, then quickly corrects to electron mode
  - Dosage process doesn't see the update, delivers x-ray dose
  - Mode process sees update, removes shield
- Consequences
  - 3 deaths, 3 serious injuries from radiation overdose



from http://www.netcomp.monash.edu.au/cpe9001/assets/readings/HumanErrorTalk6.gif

source: Leveson and Turner, An Investigation of the Therac-25 Accidents, *IEEE Computer*, Vol. 26, No. 7, July 1993.

Software Disasters: Ariane 5

- $7 billion, 10 year rocket development
- Exploded on first launch
  - A numeric overflow occurred in an alignment system
  - Converting lateral velocity from a 64 to a 16-bit format
  - Guidance system shut down and reported diagnostic data
  - Diagnostic data was interpreted as real, led to explosion
- Irony: alignment system was *unnecessary* after launch and should have been shut off
- Double irony: overflow was in code reused from Ariane 4
  - Overflow impossible in Ariane 4
  - Decision to reuse Ariane 4 software, as developing new software was deemed too risky!

from http://www-user.tu-chemnitz.de/~uro/teaching/crashed-numeric/ariane5/

source: Ariane 501 Inquiry Board report

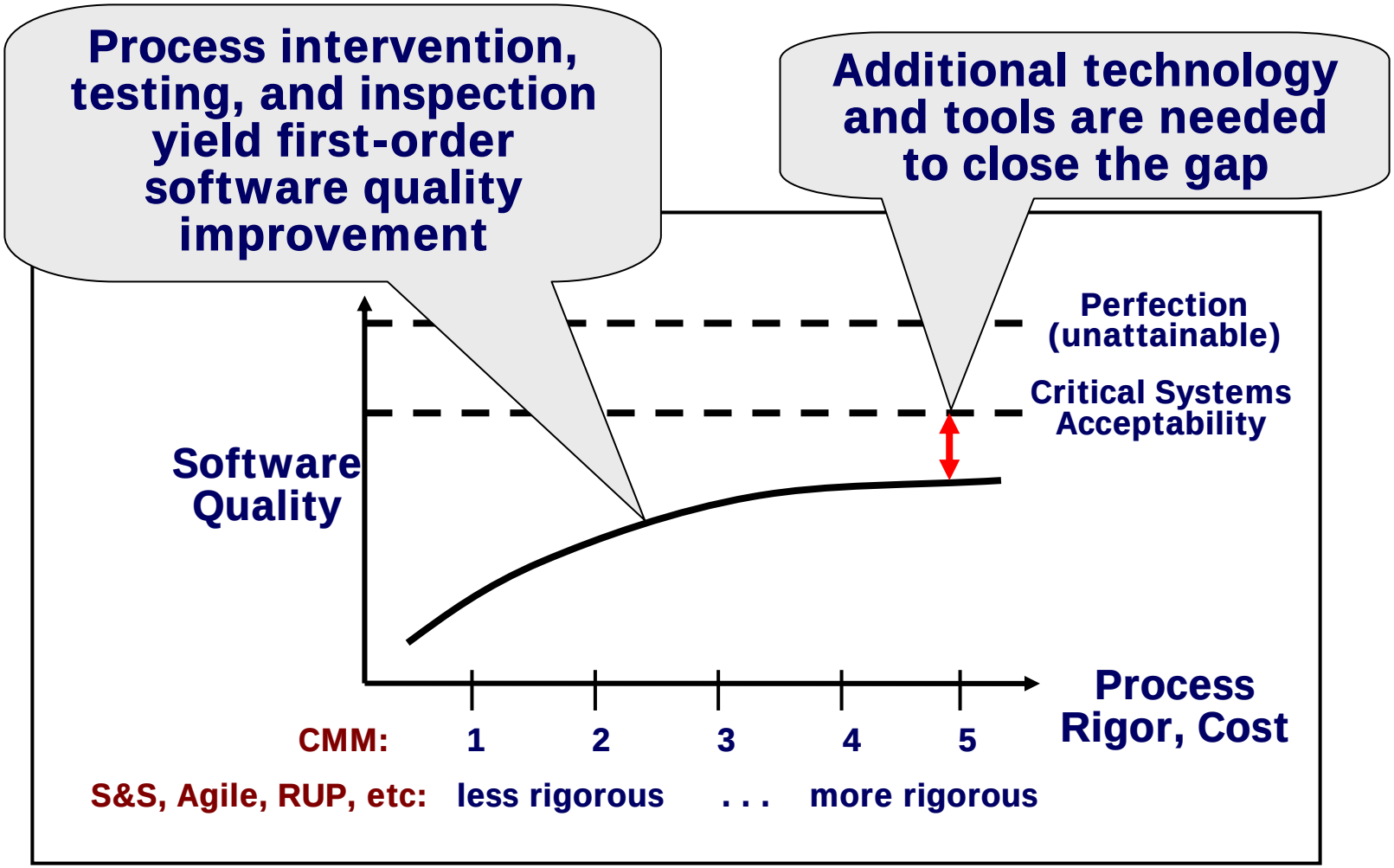
Software Disasters

Market Drivers for Quality

- Security
  - News: credit cards compromised
- Business-critical
  - How much do Amazon, eBay lose if their sites goes offline?
- Safety-critical
  - Control software for vehicles, aircraft
- Regulations
  - HIPAA, others: legal requirements on software

Process, Cost, and Quality

Slide: William Scherlis



Root Causes of Errors

- Requirements problems
  - Don't fit user needs

Static Analysis Contributions

- Design flaws
  - ★ Lacks required qualities

Hard

Does design achieve goals?

Is design implemented right?

- Implementation errors
  - Assign
  - Checking
  - Algorithm
  - ★ Timing
  - ★ Interface
  - ★ Relationship

Security

Hard

Is data initialized?

Is dereference/indexing valid?

Are threads synchronized?

Are interface semantics followed?

Are invariants maintained?

Taxonomy: [Chillarege et al., Orthogonal Defect Classification]

Existing Approaches

- Testing: *is the answer right?*
  - Verifies features work
  - Finds algorithmic problems
- Inspection: *is the quality there?*
  - Missing requirements
  - Design problems
  - Style issues
  - Application logic

- Limitations
  - Non-local interactions
  - Uncommon paths
  - Non-determinism
- Static analysis: *will I get an answer?*
  - Verifies non-local consistency
  - Checks all paths
  - Considers all non-deterministic choices

Errors Static Analysis can Find

- 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
- Race conditions

Theme: consistently following rules throughout code

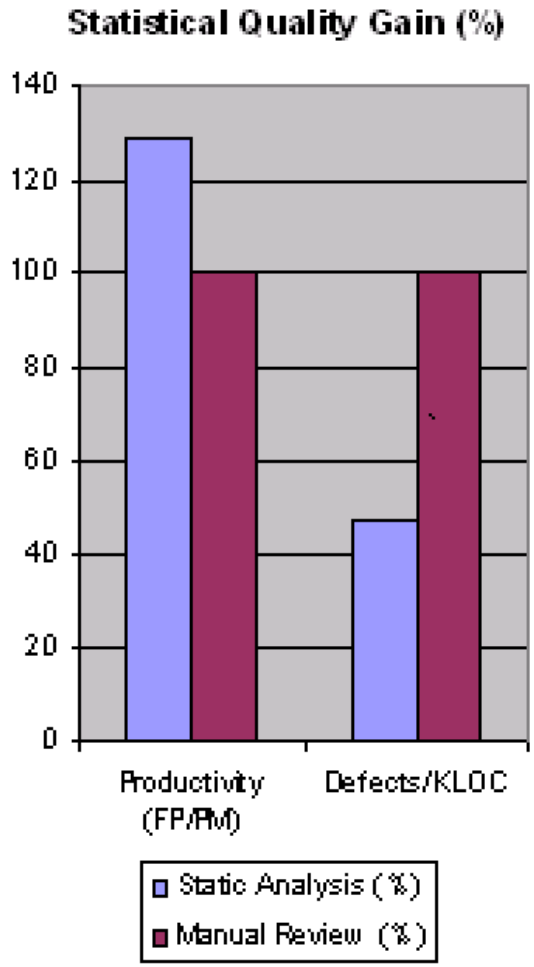
Empirical Results on Static Analysis

- Nortel study [Zheng et al. 2006]
  - 3 C/C++ projects
  - 3 million LOC total
  - Early generation static analysis tools

- Conclusions
  - Cost per fault of static analysis 61-72% compared to inspections
  - Effectively finds assignment, checking faults
  - Can be used to find potential security vulnerabilities

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]

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 Longhorn/Vista release of Windows
    - Release still pending
- Early 2000s: add static analysis
  - More on this later

Outline

- Why static analysis?
- What is static analysis?
  - Abstract state space exploration
- How does static analysis work?
- What does the future hold?
- 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

Outline

- Why static analysis?
- What is static analysis?
- How does static analysis work?
  - Termination, Soundness, and Precision
- What does the future hold?
- What tools are available?
- How does it fit into my organization?

How can Analysis Search All Paths?

- Exponential # paths with if statements
- Infinite # paths with loops
- 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. }
```

Path 1 (before stmt): true/no loop

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

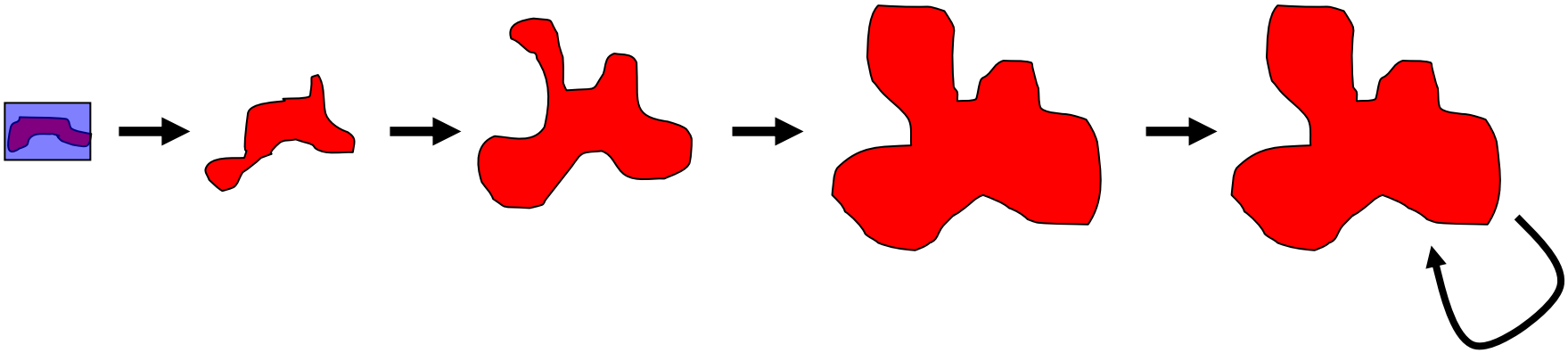
all of state space has been explored

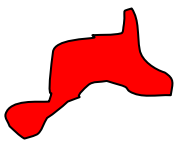
Sound Analyses

- 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

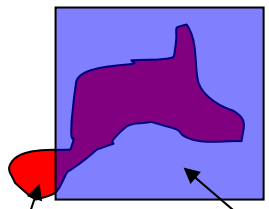
Soundness and Precision





Program state covered in actual execution

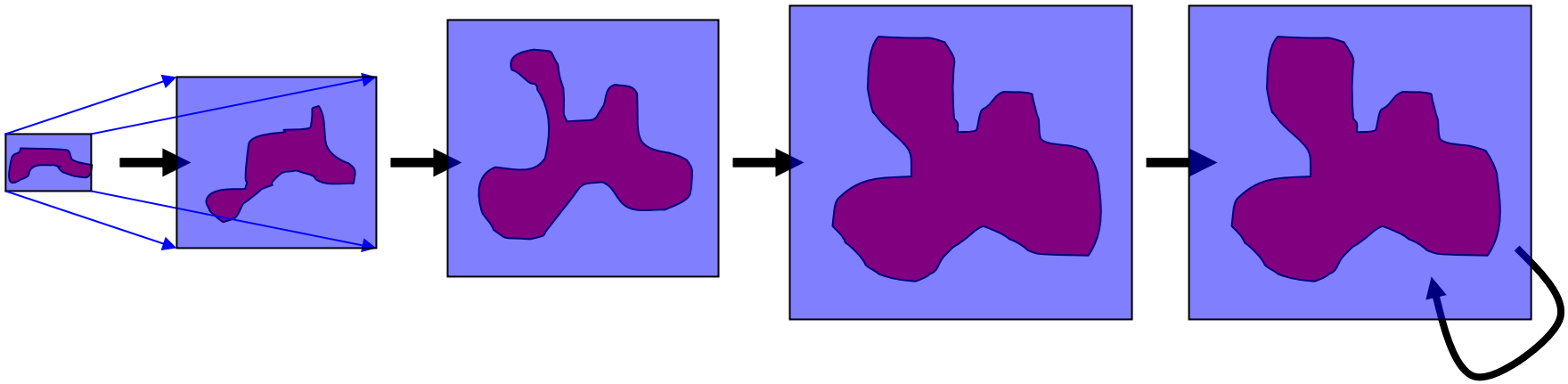
Program state covered by abstract execution with analysis

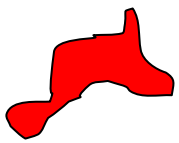


unsound
(false negative)

imprecise
(false positive)

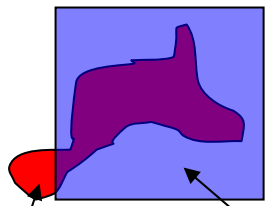
Soundness and Precision





Program state covered in actual execution

Program state covered by abstract execution with analysis



unsound
(false negative)

imprecise
(false positive)

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

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

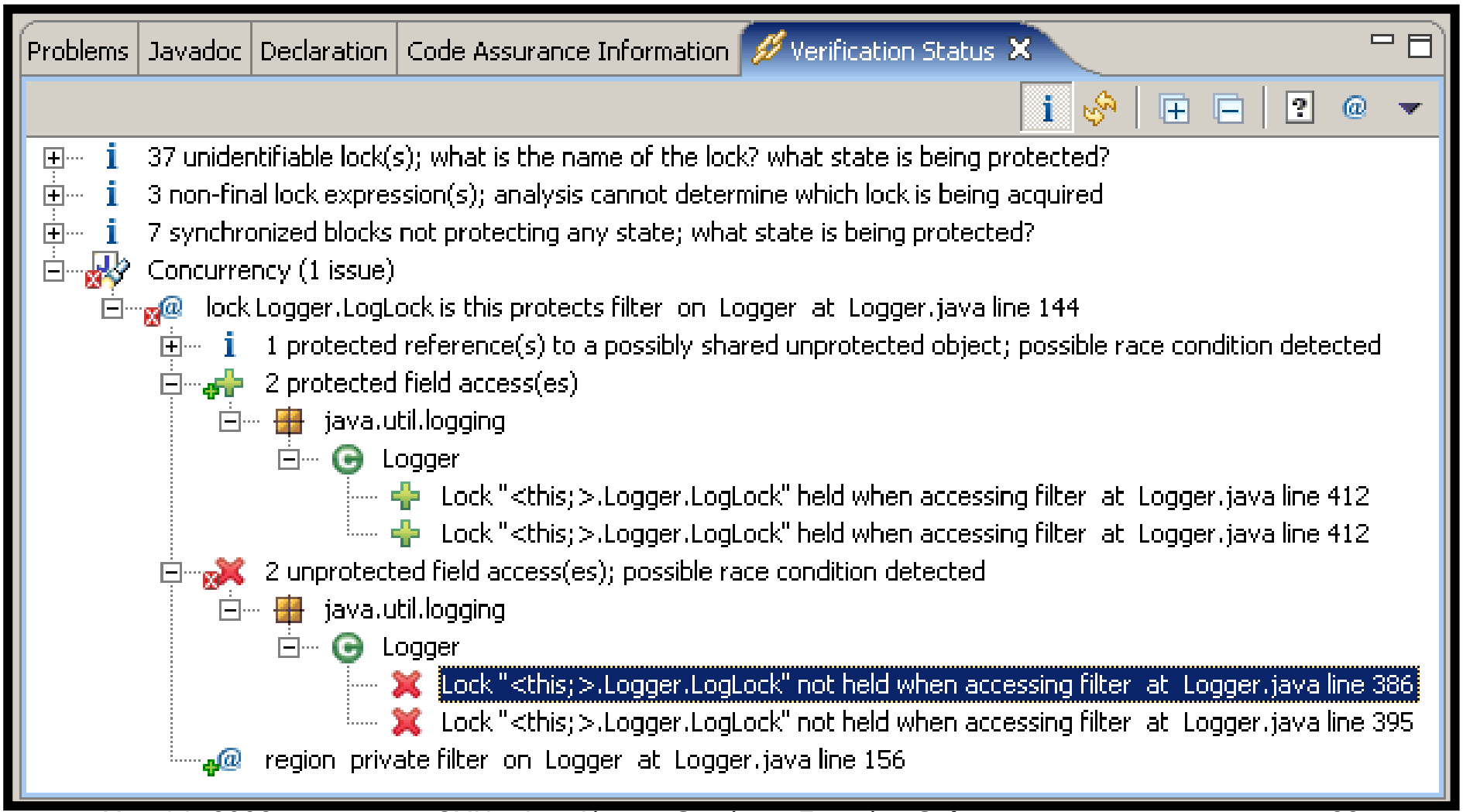
Soundness Tradeoffs

- 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

Outline

- Why static analysis?
- What is static analysis?
- How does static analysis work?
- What does the future hold?
  - Current CMU Analysis Research
- What tools are available?
- How does it fit into my organization?

Fluid: Concurrency Analysis



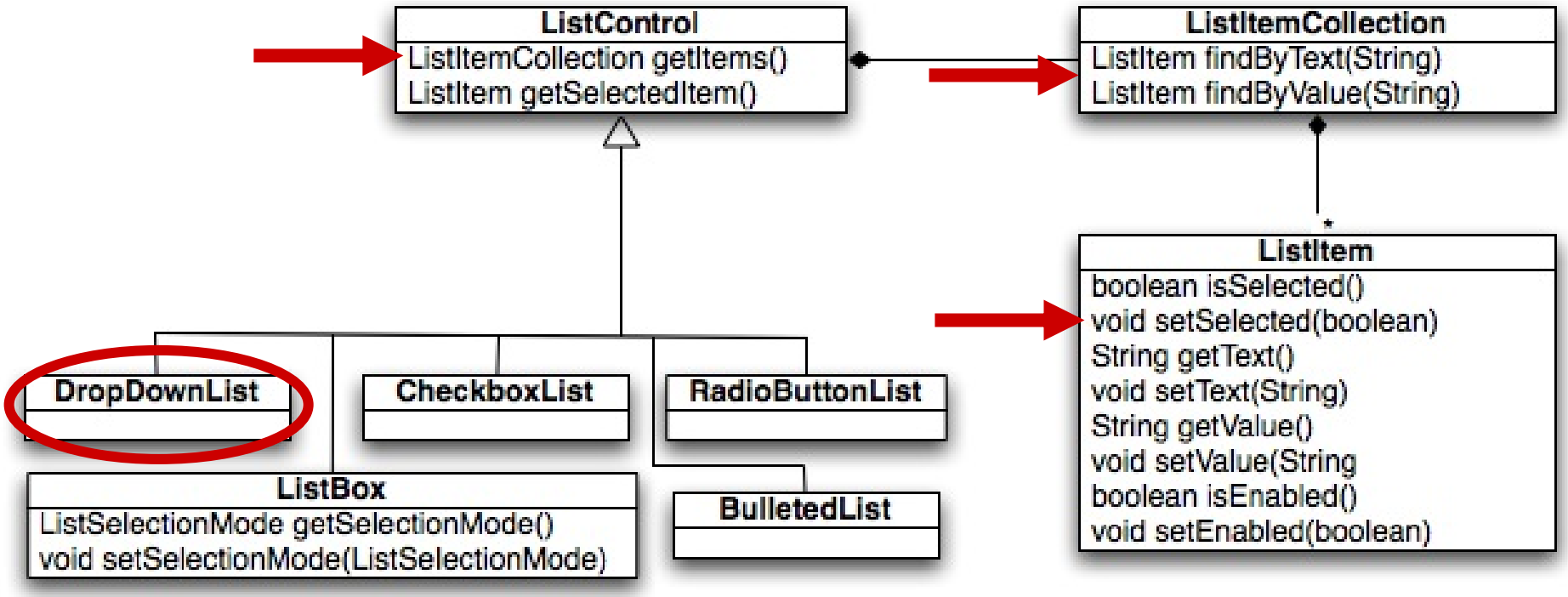
Principal Case Study Results

- (top-10 SV software vendor – Vendor V)
  - 300KLOC concurrent Java – productized production code
    - Highly multi-threaded container code
  - 3 days modeling time
  - 45 Lock models
    - Models: requiresLock, aggregate, unshared, mapinto, etc.
    - 1800 "**+**", 230 "**X**", 1700 "**i**"
  - Several major architectural issues detected
    - Deadlocks and races
  - About 25 faults detected and corrected in code base
  - Vendor staff developed many models themselves
    - UI "natural" for developers
    - Highly interactive use
  - Tool identified areas for code review

Framework Usage: DropDownList

- Can add drop down lists to a web page
- Can change the selection programmatically
- Only one item is selected at a time

Class diagram



Let's change the selection

```
String searchTerm = …;
ListItem newItem;
DropDownList ctrl = getControl("myList");




newItem = ctrl.getItems().findByValue(searchTerm);
newItem.setSelected(true);
```

Cannot have multiple items selected in a DropDownList.

Stack Trace:

```
[HttpException (0x80004005): Cannot have multiple items selected in a DropDownList.]
   System.Web.UI.WebControls.DropDownList.VerifyMultiSelect() +133
   System.Web.UI.WebControls.ListControl.RenderContents(HtmlTextWriter writer) +206
   System.Web.UI.WebControls.WebControl.Render(HtmlTextWriter writer) +43
   System.Web.UI.Control.RenderControlInternal(HtmlTextWriter writer, ControlAdapter adapter) +74
   System.Web.UI.Control.RenderControl(HtmlTextWriter writer, ControlAdapter adapter) +291
```

Correct code

```
String searchTerm = …;
ListItem newItem, oldItem;
DropDownList ctrl = getControl(“myList”);

oldItem = ctrl.getSelectedItem();
oldItem.setSelected(false);

newItem = ctrl.getItems().findByValue(searchTerm);
newItem.setSelected(true);
```

Our Solution

- Track relationship between objects — *“child(oldItem, ctrl)”*
- Track object state — *“selected(oldItem)”*, *“no_selection(ctrl)”*
- Enforce constraints — *“must remove old selection before setting new selection”*

Cooperative Permissions

- Lightweight verification of architectural protocols
  - Naturally express coordination among pointers
    - Follows engineering intuition
  - Immediate benefits and consistency over time
    - Assure correct usage of libraries, frameworks; find defects
  - Works with OO designs
    - Supports aliasing, recursion, and inheritance
  - Proved sound, validated on small but real examples
    - OOPSLA 2007 paper, Modular Typestate Checking for Aliased Objects
    - upcoming OOPSLA 2008 paper – extension to concurrency

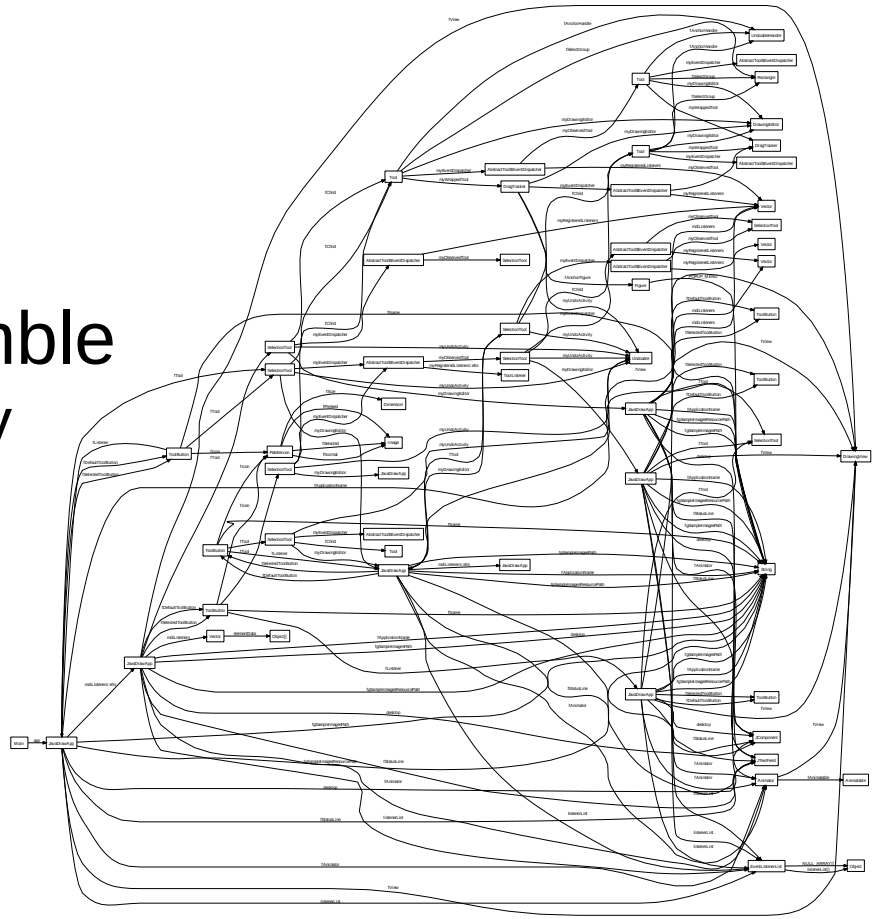
Ownership Object Graphs

with Marwan Abi-Antoun

- 5-year vision: a big picture you can trust
  - Summarize million LOC systems in 1 page
  - Zoom in to any level of detail
    - Look inside a top-level component
  - Assure that the 1-page summary is complete
    - Communication integrity

Case Study: JHotDraw

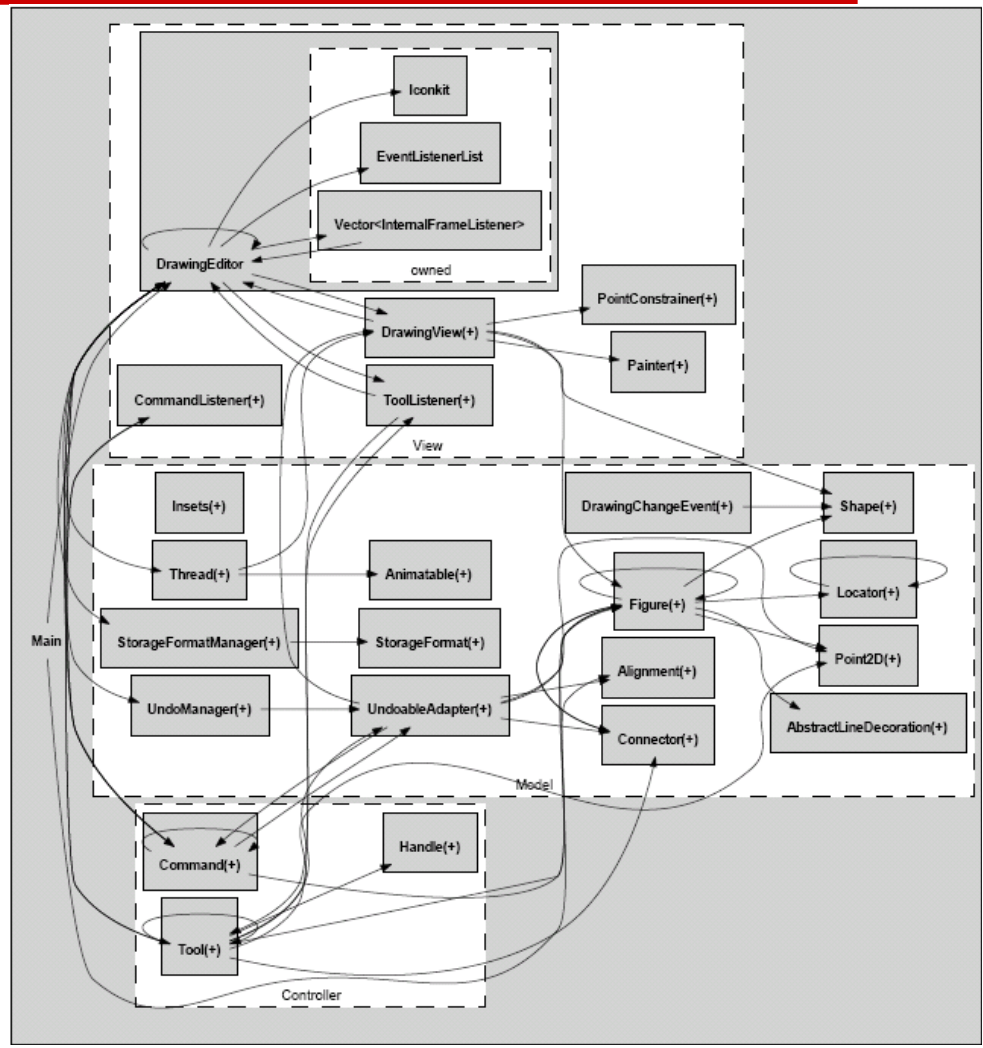
- JHotDraw v. 5.3
  - 195 classes
  - 15,000 LOC
- Object graph from Womble
  - Shows details effectively
  - Does not scale to high-level view
  - Unsound



Output of Womble (Jackson and Waingold [JW01])

Case Study: JHotDraw

- Architectural structure
  - Shows object relationships
  - Displays Model-View-Controller design intent
  - Fits on a page
- Characteristics
  - Hierarchical
    - Top level objects are summaries
    - Can zoom in to details
  - Sound
    - Represents all objects
    - Shows all field links



Outline

- Why static analysis?
- What is static analysis?
- How does static analysis work?
- What are current tools like?
- What does the future hold?
- What tools are available?
- How does it fit into my organization?

Error Taxonomy (incomplete list)

- **Concurrency**
  - race conditions
  - deadlock
  - data protected by locks
  - non-lock concurrency (e.g. AWT)
- **Exceptional conditions**
  - integer over/underflow
  - division by zero
  - unexpected exceptions
  - not handling error cases
  - type conversion errors
- **Memory errors**
  - array bounds / buffer overrun
  - illegal dereference (null, integer, freed)
  - illegal free (double free, not allocated)
  - memory leak
  - use uninitialized data
- **Resource/protocol errors**
  - calling functions in incorrect order
  - failure to call initialization function
  - failure to free resources
- **Input validation**
  - command injection
  - cross-site scripting
  - format string
  - tainted data
- **Other security**
  - privilege escalation
  - denial of service
  - dynamic code
  - malicious trigger
  - insecure randomness
  - least privilege violations
- **Design and understanding**
  - dependency analysis
  - heap structure
  - call graph
- **Code quality**
  - metrics
  - unused variables

Microsoft Tools

- Static Driver Verifier (was SLAM)
  - http://www.microsoft.com/whdc/devtools/tools/sdv.mspx
  - Part of Windows Driver Kit
  - Uses model checking to catch misuse of Windows device driver APIs
- PREfast and the Standard Annotation Language
  - Ships with Visual Studio 2005 (team edition) and Windows SDK
    - http://windowssdk.msdn.microsoft.com/
  - Standard Annotation Language
    - Lightweight code specifications
    - Buffer size, memory management, return values, tainted data
  - PREfast
    - Symbolically executes paths to find memory errors
    - Lightweight version of PREfix analysis used internally at Microsoft
    - Verifies SAL specifications
  - Blogs on getting started with SAL
    - http://blogs.msdn.com/michael_howard/archive/2006/05/19/602077.aspx
    - http://blogs.msdn.com/michael_howard/archive/2006/05/23/604957.aspx
  - Microsoft docs
    - http://msdn2.microsoft.com/en-us/library/ms182025.aspx
    - http://msdn2.microsoft.com/en-us/library/y8hcsad3.aspx
- If you use Microsoft tools, use these!

FindBugs

- findbugs.sourceforge.net
- Focus: bug finding
- Language: Java
- Open source project
  - Free
  - Large community
  - Easy to adapt and customize
  - Many defect detectors
  - Eclipse plugin support
  - Mostly searches for localized bugs

- **Memory errors**
  - array bounds / buffer overrun
  - illegal dereference (null, integer, freed)
  - double free
  - memory leak
  - use uninitialized data
- **Input validation**
  - command injection
  - tainted data
- **Concurrency**
  - race conditions
  - deadlock
  - data protected by locks
- **Resource/protocol errors**
  - failure to free resources
- **Exceptional conditions**
  - integer over/underflow
  - not handling error cases
  - type conversion errors
- **Code quality**
  - unused variables

Coverity Prevent/Extend

- www.coverity.com
- Focus: bugs and security
- Languages: C, C++, Java
- OS: Windows, Linux, OS X, NetBSD, FreeBSD, Solaris, HPUX
- Builds on the Metal static analysis research project at Stanford
- Open source analysis project
  - http://scan.coverity.com
- Selling points
  - Low false positive rates
  - Scales to 10 MLOC+
  - Statistical bug finding approach
  - Extensibility with Extend
  - Seamless build integration

- **Memory errors**
  - array bounds / buffer overrun
  - illegal dereference (null, integer, freed)
  - double free
  - memory leak
  - use uninitialized data
- **Input validation**
  - command injection
  - cross-site scripting
  - format string
  - tainted data
- **Concurrency**
  - race conditions
  - deadlock
- **Resource/protocol errors**
  - calling functions in incorrect order
  - BSTR library usage (Microsoft COM)
  - failure to free resources
- **Exceptional conditions**
  - not handling error cases

GrammaTech CodeSonar

- www.grammatech.com
- Focus: bug finding
- Languages: C, C++
- OS: Windows, Linux, Solaris
- Company founded by Tim Teitelbaum of Cornell and Tom Reps of U. Wisc. Mad.
- Selling points
  - Strong coverage of C/C++ errors
  - Binary analysis technology under development
  - Support for custom checks
  - Easy integration with build
  - CodeSurfer program understanding tool
- **Memory errors**
  - array bounds / buffer overrun
  - illegal dereference (null, freed)
  - illegal free (double free, not allocated)
  - memory leak
  - use uninitialized data
- **Input validation**
  - format string
  - tainted data
- **Concurrency**
  - race conditions
  - deadlock
- **Exceptional conditions**
  - integer over/underflow
  - not handling error cases
  - division by zero
  - type conversion errors
- **Design and understanding**
  - navigation
  - dependency analysis
  - ASTs, CFGs, pointer analysis
  - heap structure
  - call graph

Klocwork K7 Development Suite

- www.klocwork.com
- Focus: security and bugs
- Languages: C, C++, Java
- OS: Windows, Linux, Solaris
- Selling points
  - Strong focus on both bugs and vulnerabilities
  - Built-in extensibility
  - Enterprise/process support
    - track quality over time
  - Architectural visualization support

- **Memory errors**
  - array bounds / buffer overrun
  - illegal dereference (null, integer, freed)
  - illegal free (double free, not allocated)
  - memory leak
  - use uninitialized data
- **Input validation**
  - command injection
  - cross-site scripting
  - format string
  - tainted data
- **Concurrency**
  - race conditions
- **Resource/protocol errors**
  - calling functions in incorrect order
- **Exceptional conditions**
  - not handling error cases
- **Other security**
  - insecure randomness
  - least privilege violations
- **Design and understanding**
  - dependency analysis

Fortify Source Code Analysis Suite

- www.fortify.com
- Focus: security
- Languages: C, C++, .NET family (C#, VB), Java, ColdFusion, TSQL, PLSQL, XML
  - OO support from the beginning
- Windows, Linux, OS X, Solaris, AIX
- Sponsor of FindBugs, fully integrated FindBugs support
- Selling points
  - Strong focus on security
  - Built-in extensibility
  - Good coverage of security errors
- Runtime products from same rule set:
  - Fortify Tracer— transform black box tests into white box results
  - Fortify Defender—detect attacks

- **Memory errors**
  - array bounds / buffer overrun
  - illegal dereference (null, freed)
  - double free
  - memory leak
  - use uninitialized data
- **Input validation**
  - command injection
  - cross-site scripting
  - format string
  - tainted data
- **Concurrency**
  - race conditions
  - deadlock
- **Resource/protocol errors**
  - calling functions in incorrect order
  - failure to call initialization function
  - failure to free resources
- **Exceptional conditions**
  - integer over/underflow
  - unexpected exceptions
  - not handling error cases
- **Code quality**
  - metrics (attack surface, etc.)

Ounce Labs 5

- www.ouncelabs.com
- Focus: security
- Languages: Java, C, C++, C#, other web
- OS: Windows, Solaris, Linux, AIX
- Selling points
  - Sound detects critical classes of errors
  - Reports when error can be confirmed
  - Scales to 50 MLOC+
  - Broad coverage of security errors
  - Total portfolio risk management
- **Memory errors**
  - array bounds / buffer overrun
- **Input validation**
  - command injection
  - cross-site scripting
  - format string
  - tainted data
- **Concurrency**
  - race conditions
- **Resource/protocol errors**
  - failure to free resources
- **Other security**
  - privilege escalation
  - denial of service
  - dynamic code
  - malicious trigger

PolySpace

- www.polyspace.com
  - (now part of MathWorks)
- Focus: embedded system defects
- Languages: C, C++, Ada
  - UML Rhapsody, Simulink models
- OS: Windows, Linux, Solaris
- Selling points
  - Focus on embedded systems
  - Mathematically verifies code with proof engine
    - Assured code shown in green
    - Errors in checked classes cannot occur
- **Memory errors**
  - array bounds / buffer overrun
  - illegal dereference (null, integer, freed)
  - use uninitialized data
  - reference to non-initialized class members
- **Exceptional conditions**
  - integer over/underflow
  - division by zero
  - arithmetic exceptions
  - type conversion errors

SureLogic JSure

- www.surelogic.com
- Focus: concurrency, architecture, API usage
- Language: Java
- Selling points
  - Focus on Java concurrency
  - Immediate return on investment
  - Professional services
    - End-to-end support for FindBugs analysis
  - Sound analysis – shows assured code w/ green plus
    - Errors in checked classes cannot occur
- **Concurrency**
  - race conditions
  - data protected by locks
  - non-lock concurrency (e.g. AWT)
- **Architecture compliance**
  - module structure
- *Full disclosure: I have a stake in SureLogic as a consultant and potential technology provider*

Lattix LDM

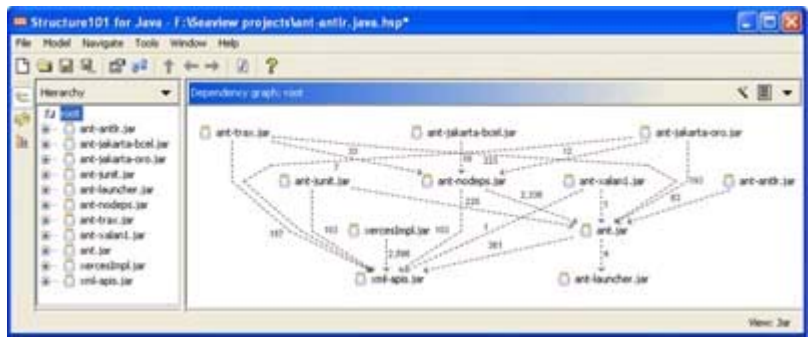
- www.lattix.com
- Focus: architectural structure
- Languages: C, C++, Java, .NET
- OS: Windows, Linux, Mac OS X
- Published in OOPSLA 2005
- Selling points
  - Focus on architectural structure
  - Design Structure Matrix representation
    - Built automatically from code
    - Analysis extracts layered architecture
  - Checks design rules
  - Downloadable trial version

- **Design and understanding**
  - dependency analysis
  - impact analysis
  - architecture violations

Source: OOPSLA 2005 paper

Headway Software Structure 101

- www.headwaysoftware.com
- Focus: architectural structure
- Languages: Java, Ada
- OS: Windows, Linux, Mac OS X
- Selling points
  - Focus on architectural structure
    - Supports design structure matrices, other notations
  - Structural analysis
    - dependencies
    - impact of change
    - architectural evolution
  - Downloadable trial version
- **Design and understanding**
  - dependency analysis
  - impact analysis
  - architectural violations
  - complexity metrics



Source: Headway Software web site

Outline

- Why static analysis?
- What is static analysis?
- How does static analysis work?
- What are current tools like?
- What does the future hold?
- What tools are available?
- How does it fit into my organization?
  - Lessons learned at Microsoft & eBay: Introduction, measurement, refinement, check in gates
    - Microsoft source: Manuvir Das
    - eBay source: Ciera Jaspan

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

Tool Customization

- Tools come with many analyses
  - Some relevant, some irrelevant
  - eBay example [Jaspan et al. 2007]
    - Dead store to local is a critical performance bug if the dead code is a database access
- Process
  - Turn on all defect detectors
  - Estimate value of reports, false positives
  - Assign each detector a priority
    - Tied to enforcement mechanism, e.g. prohibited on check-ins

Cost/Benefit Analysis

- Costs
  - Tool license
  - Engineers internally supporting tool
  - Peer reviews of defect reports
- Benefits
  - How many defects will it find, and what priority?
- Experience at eBay [Jaspan et al. 2007]
  - Evaluated FindBugs
  - Found less severe bugs than engineer equivalent
  - Clearly found more bugs than engineer equivalent
  - Ultimately incorporated tool into process
  - See OOPSLA 2007 practitioner report, Understanding the Cost of Program Analysis Tools, Tuesday 2pm

Enforcement

- Microsoft: check in gates
  - 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
- eBay: dev/QA handoff
  - Developers run FindBugs on desktop
  - QA runs FindBugs on receipt of code, posts results
    - High-priority fixes required
- Requirements for success
  - Low false positives
  - A way to override false positive warnings
    - Typically through inspection
  - Developers must buy into static analysis first

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

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

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
  - pick and choose analyses which are most useful
- Level 3: integrate tools into process
  - check in quality gates, milestone quality gates
  - integrate into build process, developer environments
  - 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 for new analyses
  - gather data on analysis effectiveness
  - tune analysis based on observations

Summary

- Analysis is changing QA practices in leading organizations today
- Exhibit A: Microsoft
  - Comprehensive analysis was centerpiece of QA for Windows Vista
  - 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

Questions?