Principles of Software Construction: Objects, Design and Concurrency

Static Analysis

Christian Kästner   Charlie Garrod

© 2012-13 C Garrod, C Kästner, J Aldrich, and W Scherlis
Recap Frameworks
Recap Frameworks

Total Tweets

- thomyorke: 8,398
- AndroidPolice: 4,997
- Fazrulz: 1,753

Tweets vs. Time of Day

Followers Over Time
Map-Reduce Framework
Map Reduce Tasks

- Searching and Filtering
- Counting Words
- Computing average response times from logs
- Precomputing common-friends lists
- Building inverted indexes
- Distributed simulation
- Counting unique items
- Cross-correlation (who buys X also buys Y)
- Pagerank

many more, e.g.: http://highlyscalable.wordpress.com/2012/02/01/mapreduce-patterns/
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
/* 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;
}
Limits of Inspection

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

So, let’s automate inspection!
{ #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!"); } ;
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) {
        // load data
        // ...
    }

    reader.close();
}
Mental Model for Analysing "Freed Resources"

close =>
err(double close)

free

close

allocated

allocate

der close

er(end path without close)
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

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
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?

```java
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 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
class X {
    Logger logger;
    public void foo() {
        ...
        if (logger.inDebug()) {
            logger.debug("We have " + conn + "connections.");
        }
    }
}
class Money {
    String currency;
    int amount;

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

    public boolean equals(Object o) {
        if (o instanceof Money)
            return (((Money) o).amount == this.amount) &&
                    ((Money) o).currency.equals(this.currency);
        return false;
    }
}

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

This class overrides equals(Object), but does not override hashCode(), and inherits the implementation of 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 must 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
}
```
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?

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

bar(x, y) {
    print(x / y);
}
```
public Boolean decide() {
    if (computeSomething()==3)
        return Boolean.TRUE;
    if (computeSomething()==4)
        return false;
    return null;
}

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
<table>
<thead>
<tr>
<th></th>
<th>Error exists</th>
<th>No error exists</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Error Reported</strong></td>
<td>True positive (correct analysis result)</td>
<td>False positive</td>
</tr>
<tr>
<td><strong>No Error Reported</strong></td>
<td>False negative</td>
<td>True negative (correct analysis result)</td>
</tr>
</tbody>
</table>

**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

Complete Analysis

Unsound and Incomplete Analysis
The Bad News: Rice's Theorem

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

- Every decidable static analysis is necessarily incomplete or unsound (or both)
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. }

Abstraction (before statement)

2-4: enabled
5: enabled
6: disabled
7: disabled
8: disabled
9: disabled
11: disabled
13: unknown

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!
Example: Extended Interrupt Problem

1. int foo() {
2.     cli();
3.     if (a) {
4.         ...
5.         restore_flags();
6.     } else {
7.         ...
8.     }
9.     if (b)
10.        return;
11.     if (c)
12.        restore_flags();
13.     return rv;
14. }

end
public int foo() {
    doStuff();
    return 3;

    doMoreStuff();
    return 4;
}

end
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 paths in the program (as opposed to all paths in the method)

• 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

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

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

Abstraction (before statement)

<table>
<thead>
<tr>
<th>Line</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>not-null</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>not-null,</td>
<td>maybe-null</td>
</tr>
<tr>
<td>5</td>
<td>not-null,</td>
<td>maybe-null</td>
</tr>
<tr>
<td>6</td>
<td>not-null,</td>
<td>not-null</td>
</tr>
<tr>
<td>8</td>
<td>not-null,</td>
<td>null</td>
</tr>
<tr>
<td>9</td>
<td>not-null,</td>
<td>null</td>
</tr>
<tr>
<td>10</td>
<td>not-null,</td>
<td>not-null</td>
</tr>
<tr>
<td>12</td>
<td>maybe-null,</td>
<td>not-null</td>
</tr>
</tbody>
</table>

Error: may have null pointer on line 12
Example: Method calls

1. int foo() {
2.     Integer x = bar();
3.     Integer y = baz();
4.     Integer z = noNullsAllowed(x, y);
5.     return z.intValue();
6. }

7. Integer noNullsAllowed(Integer x, Integer y) {
8.     int z;
9.     z = x.intValue() + y.intValue();
10.    return new Integer(z);
11. }

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 ESC/Java
Example: Method calls

```java
1. int foo() {
2.     Integer x = bar();
3.     Integer y = baz();
4.     Integer z = noNullsAllowed(x, y);
5.     return z.intValue();
6. }

7. @NonNull Integer noNullsAllowed(@NonNull Integer x, @NonNull Integer y) {
8.     int z;
9.     z = x.intValue() + y.intValue();
10.    return new Integer(z);
11. }

12. @NonNull Integer bar();

13. @Nullable Integer baz();
```
Another Data-Flow Example

```java
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

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

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

ESC/Java is 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
thread1() {
    read x;
}

thread2() {
    lock();
    write x;
    unlock();
}

Interleaving 1: OK
Interleaving 2: OK
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
JSure Demo
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
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

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

Error on line 5:
Redundant comparison to null
“False” Positives

1. public class Constants {
2.     static int myConstant = 1000;
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
Methods to increase precision

• Ignore highly unusual cases
• Make abstraction less abstract
• Add specifications
• Make code more analyzable
  ▪ Remove aliases
  ▪ Remove pointer arithmetic
  ▪ Programming without side effects
  ▪ Clean up control flow
Hard problems

• Side effects
  ▪ Often difficult to specify precisely
    ▪ In practice: ignore (unsafe) or approximate (loses accuracy)

• Modularity
  ▪ Specifications
  ▪ Not just performance issue
  ▪ Don’t have to analyze all the code
  ▪ Reduces interactions between people

• Aliasing and pointer arithmetic

• Termination

• Precision
Counter c = new Counter();
Counter d = doSomething(c);
d.value = null;
c.value.inc();
Aliasing

- Two variables point to the same object
- A variable might change underneath you during a seemingly unrelated call
- Multi-threaded: change at any time!
- Makes analysis extremely hard

**Solutions**
- Can add more specifications
  - Unique, Immutable, Shared...
- Analysis can assume no aliases
  - Can miss issues from aliasing
- Analysis can assume the worst case
  - Can report issues that don’t exist
Pointer arithmetic in C

- Very difficult to analyze
- Many tools will gladly report an issue, even if there is none
- May be a good idea to avoid
  - Rationale: It might be correct, but it’s ugly and makes problems more difficult to find
Termination

• How many paths are in a program?
• Exponential # paths with if statements
• Infinite # paths with loops / recursion
• How could we possibly cover them all?
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
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