Software Testing, Quality Assurance and Maintenance Winter 2017

Lecture 32 — March 24, 2017

Patrick Lam version 2

We've seen two tools so far: PMD and jshint. These tools both operate on Abstract Syntax Trees and ensure relatively shallow program properties. Out of the box, PMD guarantees generic properties (but we also talked about writing our own checkers).

jshint, PMD, and other tools (like FindBugs, below) enforce generic rules that all programs should satisfy. You can read a comparison of different tools in this paper:

`www.cs.umd.edu/~jfoster/papers/issre04.pdf`

FindBugs. This tool is somewhat deeper than PMD. FindBugs is an open-source static bytecode analyzer for Java out of the University of Maryland. A key difference is that it performs static analysis at Java bytecode level rather than AST level. It's therefore harder to write FindBugs rules.

`findbugs.sourceforge.net`

FindBugs finds bug patterns like:

- off-by-one;
- null pointer dereference;
- ignored `read()` return value;
- ignored return value (immutable classes);
- uninitialized read in constructor;
- and more…

Such patterns are typically easier to evaluate at bytecode level because the variability of the AST has been compiled away.

False positives. FindBugs, like all static analysis tools, gives some false positives. (The course project has a question about false positives.) In general, they occur because the analysis tool is not powerful enough. Because of the halting problem, there can be no all-powerful tool.

Consider this case:

```
try { socket.close(); }
catch (Exception ignore) {}

try { reader.close(); }
catch (Exception ignore) {}
```

FindBugs, of course, declares "this method might ignore an exception", but it's fine in this case, since there's nothing that the program needs to do about failed close actions.

Here are some techniques to help avoid false positives:

`patricklam.ca/papers/14.msr.saa.pdf`

Beyond hard-coded rules

Inferring specifications: Coverity Static Analyzer. This industrial-strength tool (statically) identifies bugs in C/C++, Java, and C# codebases. It claims to scale to "hundreds of users, thousands of defects, and millions of lines of code in a single analysis." It does so by inferring must-beliefs and may-beliefs from the code base, as we've discussed in Lecture 22. Coverity does a lot of work to keep the false positive rate low. Your project reproduces some of the key technology behind Coverity.

Coverity is a commercial product which can find many bugs in large (millions of lines) programs; it is therefore a leading company in building bug detection tools. Clients (900+) include organizations such as Blackberry, Yahoo, Mozilla, MySQL, McAfee, ECI Telecom, Samsung, Siemens, Synopsys, NetApp, Akamai, etc. These include domains including EDA, storage, security, networking, government (NASA, JPL), embedded systems, business applications, operating systems, and open source software. We have access to Coverity for this course, but there's also a free trial:

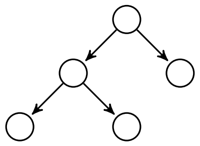
`http://softwareintegrity.coverity.com/FreeTrialWebSite.html`

Developer-provided specifications

A number of research-quality tools aim to enforce developer-provided specifications. Developers should be aware of how their software works. Specification languages enable developers to express this knowledge in a machine-readable way and tools can enforce that the software satisfies its specification. We'll discuss some specification-based tools below.

Korat (University of Illinois). Key Idea: Generate Java objects from a representation invariant specification written as a Java method.

For instance, here's a binary tree:



One characteristic of a binary tree:

- left & right pointers don't refer to same node.

We can express that characteristic in Java as follows:

```java
boolean repOk() {
  if (root == null) return size == 0;                 // empty tree has size 0
  Set visited = new HashSet(); visited.add(root);
  List workList = new LinkedList(); workList.add(root);
  while (!workList.isEmpty()) {
    Node current = (Node)workList.removeFirst();
    if (current.left != null) {
      if (!visited.add(current.left)) return false; // acyclicity
      workList.add(current.left);
    }
    if (current.right != null) {
      if (!visited.add(current.right)) return false; // acyclicity
      workList.add(current.right);
    }
  }
  if (visited.size() != size) return false;          // consistency of size
  return true;
}
```

Korat then generates all distinct ("non-isomorphic") trees, up to a given size (say 3). It uses these trees as inputs for testing the `add()` method of the tree (or for any other methods.)

`korat.sourceforge.net/index.html`