Software Testing, Quality Assurance and Maintenance Winter 2015

Lecture 12 — January 30, 2015

Patrick Lam *version 1*

Last time, we talked about some tools. Today, we'll talk about more tools.

Daikon. This tool is a dynamic tool that generates program invariants by observing program executions. An invariant is an expression that relates variables and constants; for instance, `x + y == 5`. Daikon runs the program and examines the values that it computes. Daikon can potentially find formal specifications as well as bugs. It works on Java, C, C++, and Lisp.

`plse.cs.washington.edu/daikon/`

ESC/Java (Compaq). This tool statically verifies that Java programs confirms to specifications written in the Java Modelling Language (JML). Here's an example of such a specification. (We should see more later.)

```
//@ public invariant balance >= 0 && balance <= MAX_BALANCE;
```

`www.hpl.hp.com/downloads/crl/jtk/index.html`

Valgrind. Valgrind is a framework for x86-to-x86 just-in-time compilation. It includes various tools, which can detect errors like memory errors and threading errors. These tools make programming in C/C++ almost tolerable. No C/C++ programmer's toolkit is complete without Valgrind.

memcheck is a Valgrind tool that detects memory errors at runtime. It helps make your programs more correct, by finding: illegal reads & writes; uses of uninitialized values; double frees; copies with overlapping sources and destinations; and memory leaks. **helgrind** is a Valgrind tool to detect thread errors (e.g. races).

`valgrind.org`

cppcheck. Next up, we have an open-source tool that statically checks for out-of-bounds errors; memory leaks; division by zero; null pointer dereferences; calls to obsolete functions; uses of uninitalized variables; etc.

`sourceforge.net/projects/cppcheck`

Flawfinder. This tool is not very powerful; it is as powerful as an enhanced grep, and identifies non-comment calls to bad functions:

- buffer overflow risks:
  `strcpy()`, `strcat()`, `gets()`, `sprintf()`, `scanf()`
- format string problems:
  `[v][f]printf()`, `[v]snprintf()`, `syslog()`
- file system race conditions:
  `access()`, `chown()`, `tmpnam()`, etc.

`www.dwheeler.com/flawfinder`

Clang Static Analyzer (U Illinois). This is an extensible C/C++ compiler front-end which comes with a good static analyzer.

The compiler: `clang-analyzer.llvm.org`.

`clang-analyzer.llvm.org/available_checks.html` reports that it can, among other things, understand function arguments labelled as "nonnull", and statically check for violations of these annotations; plus the usual static checks, including for some memory leaks, division by 0, and null pointer dereferences.

Sparse (Linux). This is a "semantic parser" which was built specifically to find faults in the Linux kernel. In particular, it can find errors where kernel writers mix userspace and kernelspace pointers. Also, it can find the usual: null pointer dereferences, etc.

`https://sparse.wiki.kernel.org/index.php/Main_Page`
`linux.die.net/man/1/sparse`

Splint (U Virginia). This tool was inspired by lint, the original (somewhat lame) static analyzer for C. splint is a better lint, also for C. It checks for security vulnerabilities and coding mistakes, and can use annotations. For instance:

```
1  /*@falsewhennull@*/ bool isNonEmpty
2                        (/*@null@*/ char *x) {
3    return (x != NULL && *x != '\0');
4  }
```

`www.splint.org`

Pex (Microsoft). This tool performs white-box unit testing.

`pexforfun.com`

KLEE: Symbolic Execution Engine (Stanford). The key idea here is to use symbolic execution to automatically generate high-coverage test suites. This is difficult due to the zillions of program paths that exist; KLEE attempts to find the interesting ones. For more information, read the research paper at `www.doc.ic.ac.uk/~cristic/papers/klee-osdi-08.pdf`.

`klee.github.io`

Symbolic Execution. We can also use symbolic execution to improve software testing, as seen in this reference:

"DASE: Document-Assisted Symbolic Execution for Improving Automated Software Testing". In Proceedings of ICSE '15: Wong, Zhang, Wang, Liu, & Tan.

The key idea is to use input constraints automatically extracted from documents to guide symbolic execution to test more effectively.

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; we'll see more about those concepts in the next two lectures as well as the project. Coverity does a lot of work to keep the false positive rate low.

`www.coverity.com`

GrammaTech CodeSonar. GrammaTech is another company that produces static analysis tools.CodeSonar is a static analysis tool for C, C++, and Java. It is, in particular, very good at C/C++. The Java bug finding performance is reportedly similar to FindBugs, although CodeSonar has a better user interface.

This tool aims for high recall (i.e. find all the things!)

`www.grammatech.com/products/codesonar`

Visual Studio (Microsoft). Visual Studio includes a number of testing tools. For instance, one can write constraints that are related to e.g. buffer length,
which the compiler then checks for bugs.

Other commercial tools. Here's a list of more tools.

- PCLint: fast, naïve. `www.gimpel.com/html/pcl.htm`
- PVS-Studio: `www.viva64.com/en/b/0149`
- Fortify: helps find security vulnerabilities in a wide variety of languages + in config files.
- Intel Parallel Studio XE: Static Security Analysis for C++, Fortran.
- Klocwork Insight: finds security issues & bugs in C/C++, Java, C#.

Development-related tools. (suggested by Michael Viana) These don't really help with analyzing code, generally, but are also helpful if you have the problem they're trying to solve.

- ScalaTest: flexible testing framework for Scala.
- ScalaCheck: random test generators, property-based testing.
- Jacoco: code coverage (many others in the same space, e.g. EclEmma.)
- Atlassian Bamboo: continuous integration server

Homework. Draw a decision tree to help a software tester/developer select an appropriate tool (if any) for a given project. (For any problem, more than one tool may be appropriate.) Pick two tools, use them, and compare them.