2005 SIGPLAN Workshop on the Evaluation of Software Defect Detection Tools

Bug Specimens are Important

Jaime Spacco, David Hovemeyer, William Pugh
Dept. of Computer Science
University of Maryland
College Park, MD, 20742 USA
{jspacco,daveho,pugh}@cs.umd.edu

June 10, 2005

> science *n. The observation, identification, description, experimental investigation, and theoretical explanation of phenomena.*

To perform a scientific experimental investigation of software defects, we need bugs.

Lots of them. Thousands only begins to cover it.

Pinned to the wall, embedded in blocks of clear plastic, with a whole cabinet full of note cards about them.

So, how are we going to make that happen?

Studying Software Defects

Many papers on software reliability don't even address the issue of software defects, and instead report on numbers such as the size of the points-to relation, with a hope that the numbers will be relevant to software defect detection.

But when papers do report on number of bugs found, often they only report the "number of bugs found".

Unfortunately, companies now now feel that the exact bugs they can identify with their tools is a trade secret, and no longer make such information publicly available.

Even the exact nature of the bugs identified by research papers is often unclear: many different things can be classified as a null pointer or synchronization error, and without being able to examine which bugs are being classified as which, it is very difficult to compare two different analysis techniques. The use of heuristics to eliminate false positives or unimportant bugs makes it hard even to compare bug counts.

Work on array data dependence analysis showed that many things other than proposed techniques can substantially results presented in a paper. For example, the kind of induction variable recognition used often had a far greater impact on the accuracy of the analysis than the actual array data dependence analysis technique used. I would not be surprised to find the same of software defect detection techniques.

OK, fine, so we need to come up with some benchmarks.

But I'm not sure if benchmarks will cut it.

Most benchmarks are lousy. Benchmarks can be OK for allowing you to compare two different techniques. But conventional benchmarks can be really lousy as a basis for trying to build an experimental understanding of a phenomena. If we put together a benchmark of 10 programs with a total of 14 null pointer dereferences, what does that tell us? Not much.

The problem is that we don't understand software defects enough to craft small benchmarks that characterize the types of problems we want to be able to detect with software defect detection tools. Some special cases, maybe (e.g., format string vulnerabilities).

But if we want to understand the larger universe of software defects, we need huge collections of software defects. This, of course, means a huge collection of software. And open source repositories can be a good source of software for research.

But we also need to have some kind of ground truth as to what is a bug, and how important the bug is. One possible source for this is bug databases for open source projects. These bug databases can be pretty noisy, and matching the bug reports with the actual lines of code containing the defect or the lines changed in order to fix the defect is a challenging research

problem.

To use open source projects as a basis for studying software defects, we need

- Results on specific and available versions of open-source software artifacts.
- Detailed bug reports published in a easy, machine readable format.

To be truly successful, our community also needs to choose some specific versions of specific open-source software artifacts. This would allow

- Artifacts to be examined by multiple researchers.
- Development of standard interchange format for labeling defects and warnings in the artifacts.
- Some kind of shared information about confirmed defects found in the artifact. These confirmed defects would arise from bug databases, change logs, unit tests, or manual inspections.

The Marmoset Project

In addition to using open source, widely used, software artifacts for benchmarking, at the University of Maryland we are using student code as an experimental basis for studying software defects. This work is taking place as part of the Marmoset project, which is an innovative system for student programming project submission and testing. The Marmoset system provides significant technical, motivational and pedagogical advantages over traditional project submission and grading systems, and in particular helps students strengthen their skills at developing and using testing.

But of more relevance, the Marmoset system allows us to have students participate in research studies where we collect every save of every file as they develop their programming projects. These each are added to a database, and we compile and run all of the unit tests on each compilable snapshot. We collect code coverage information from each run, and also apply static analysis tools such as FindBugs to each snapshot. Figure 1 shows some data from two semesters of our second semester OO programming course (taught in Java).

We have found the tests results to be uniquely valuable. It starts letting us have a good approximation for ground truth. But more importantly, we can look for defects we weren't expecting. We didn't expect to see so many ClassCast and StackOverflow errors. By sampling the snapshots where those errors occurred, we were able to discover new bug patterns, implement them as static analysis rules, validate them on student code and use them to find dozens of serious bugs in production code.

# snapshots	108,352
# compilable	84,950
# unique	68,226
# test outcomes	939,745
# snapshots with exception:	
NullPointer	11,527
ClassCast	4,705
IndexOutOfBounds	3,345
OutOfMemory	2,680
ArrayIndexOutOfBound	2,268
StringIndexOutOfBound	2,124
NoSuchElement	2,123
StackOverflow	2,023

Figure 1: Marmoset Data from CMSC 132

We still have lots of work to do; we've only started to collect code coverage information from our test runs, and still have to integrate it into our analysis. The correspondence between defects and exceptions under test isn't as direct as we would like, because one fault can mask other faults, or faults can occur in code not covered by any test.

Still, we are very excited about the data we are collecting via the Marmoset project and would love to talk to other researchers about sharing the data we've collected and rolling the Marmoset system out to other schools to allow them to start collecting data for the Marmoset project as well.