Is a Bug Avoidable and How Could It Be Found?
Position Statement

Dan Grossman

March 2005

Using automated tools based on programming-language technologies to find software defects (bugs) is a growing and promising field. As a research area less mature than related ones such as compiler construction, there are not yet widely accepted benchmarks and evaluation techniques. Though neither surprising nor necessarily bad, the lack of evaluation standards begs the question of what criteria should guide the inevitable growth of standards by which research on automated defect detection is judged. I contend that obvious metrics such as number of bugs and seriousness of bugs found are insufficient: We should also consider the technologies that led to the bug existing and whether other existing technologies could have found the bug or rendered it irrelevant.

It is understood that a bug's seriousness is relevant though difficult to quantify. In particular, bugs that have little affect on program behavior, bugs that are more expensive to fix than leave, and bugs for which experts dispute whether the bugs are real should rightfully be devalued. Leaving such issues aside, let us assume an easy-to-use automated tool finds *serious* bugs, the sort that developers acknowledge are errors, want to fix, can fix, do fix, and believe the result is of value. I believe the research community today would deem such work a contribution, but in the future the "bar will be raised" in at least two ways.

First, we will learn to devalue bugs that are easily avoidable with known technologies. For example, suppose a static analysis finds double-free errors in C code, i.e., situations where `free` is called more than once on the same object. Further suppose all applications for which bugs were found were stand-alone desktop applications that ran as fast or faster when recompiled to use conservative garbage collection. Were these bugs worth finding and fixing? Should we require the analysis to prove useful on applications for which manual memory management is considered necessary? As another example, suppose a modified compiler uses sophisticated run-time data structures to track uninitialized memory and abort a program if such memory is accessed. Further suppose the program would run without error or performance loss if the compiler had simply initialized all memory to 0.

Second, we will learn that tools must provide an incremental benefit over the state-of-the-art. Many tools exist; how is a new one better? It should find different bugs, find them in a better way (perhaps faster or with less user intervention), or perhaps use a more elegant approach to finding the same bugs. Would compiling the code with `-Wall` lead to exactly one accurate warning for every bug found? Does the tool work only for ten-line programs that are clearly incorrect when manually examined by an expert? Is the tool strictly less powerful and more complicated than an existing tool?

In general, it may simply be that it is currently easy to succeed with automated tools for defect detection because we assume the lowest possible baseline: We find bugs in whatever code is available, which was typically compiled without compiler warnings enabled and without other tools applied to the code. It seems clear this will change, just as the compiler optimization community must now show that a new optimization is neither made-irrelevant-by nor trivially-encoded-with existing and widely used optimizations.