Miranda Edwards
mirae
1340721

Software Development Difficulties

Difficulty: Debugging in general can be difficult without the use of a debugger. Debuggers themselves are useful although using them can be tedious: the repetitive act of pausing the code execution before the problematic point, then stepping in or through each line of code slows the programmer down. The general fallback here for programmers is to debug using print statements to verify the state of execution at certain points of time and running the program normally. Adding print statements of every interesting variable is slow and requires manual inspection of the printed values, which can be prone to human error. Additionally, this litters the executed code and makes it less readable. Once the bug has been fixed, the programmer must go back through their code removing all print statements to check in their code. This process is repeated the next time a bug is found.

Current tooling that addresses this difficulty are debuggers in general, which fall short in the scenarios described above. Using logging statements instead of print statements can allow some of the debugging code to be useful in the future and left in production code, but still risks littering the logs.

I propose adding assertions to the execution of the debugger, linked to the code being debugged. Specifically, the ability to say, "Run until one of my assertions are not true, then pause execution" allows a programmer to investigate the program state at the time of execution. In other words, this is essentially a breakpoint dependent on some piece of logic at a specific location, not just a given line number. The difference between this and current tooling or methods is that these assertions lay outside of the actual code being debugged. Instead of needing to manually inspect the state of variables, the debugger validates them, reducing the allowance for human error.

Similar solutions exist in the form of 'watchpoints', for example in Visual Studio or gdb, which pause execution anywhere in code when a variable changes value. My proposition would be different in that the assertions are more in the form of a precondition, tied to a specific line of code.

Difficulty: Race conditions in multi-threaded code can be difficult to debug. Debugging a race condition requires knowledge of the ordering of code execution that caused the bug, and an ability to recreate that ordering. It's also difficult to mark a race condition as 'solved' because there's no guarantee of the ordering of threads running in the order that cause the original bug.

Tools that help find and fix race conditions exist for individual languages. Java PathFinder searches for deadlocks by executing all possible statement sequences and scheduling combinations for objects shared between threads. The Go Race Detector

watches for unsynchronized access to shared variables, which points to the affected values without giving information about the context and ordering of execution of the threads.

Traditionally, techniques for avoiding race conditions involve avoiding sharing data between threads and using mutexes aggressively. Sharing data is not always avoidable. Mutexes, by definition, limit concurrent processes with respect to a given resource and can slow down the execution of code if used incorrectly, or worse, can lead to deadlocks caused by threads attempting to acquire the same mutex.

I propose a monitoring tool that tracks the ordering of execution of processes and writes the result to the file. This output could ideally be used to recompile the trouble code and force the same code execution ordering. By being able to reproduce the ordering, tracking down the bugs caused by interweaved processes becomes simpler.

Difficulty: Writing tests can be tedious, but maintaining those tests can be even more tedious. Upon refactoring a piece of code, it's necessary to update or even rewrite the corresponding tests. It's easy enough to look for the references to a changed piece of code in tests but is harder when you're dealing with tests that indirectly rely on the changed piece of code. To express it more concretely, consider 4 functions: A(), B(), TestA(), and TestB(), and let B() call A(). If you change the behavior of A(), you may need to update the code of B(), TestA() and TestB(). Because the behavior in TestB() *indirectly* depends on the behavior of A(), it can be easy to miss updating TestB() solely via a search for references.

I propose adding a tool that tracks indirect references to changed code and highlights the potentially impacted tests. This could be color-coded and match with the original code changes, so in the case that a programmer changes 2 different pieces of code, they can identify which change impacts which tests.

Current solutions don't really consider this a problem, as far as I've found. In theory, a programmer would continually run a suite of unit tests between code changes so they never run into the case where they've changed several things and every test is broken. However, with large tests suites that are slow to build and run, this does not happen in practice, forcing the programmer to backtrack and undo each change. By mapping code changes to individual affected tests, the programmer can reduce their search space in finding which change broke which test.