CSE 503: Assignment 1
Nate Yazdani (`nyazdani`)
Spring 2017

Difficulty 1: Selecting Units and Cases for Testing

Description of the problem. When writing software, I am rarely certain what sort of features should be considered a "unit" worthy of independent testing. Testing each method or procedure individually seems reasonable, but how many ways should each be tested? Coverage of lines or paths is a common metric, but this may result in "unnatural" test cases that won't help the programmer's intuition during later (inevitable) fault diagnosis. Moreover, the compositional behavior of methods/procedures needs testing, and coverage metrics (particularly of paths) do not scale well. It's not just about detecting when the code is buggy; ideally, well-selected test cases should "paint a picture" of what is wrong with the code, in terms of functionality, assumptions, invariants, *etc.*

Significance of the problem. Everyone should test their code well, but it's unclear how to come up with good tests, as described above. I expect that this is more of an issue for junior developers than senior ones. Still, the inability of junior developers to select high-quality tests is everyone's concern, especially if their selected tests are insufficient to catch or effectively diagnose critical bugs.

Challenges for a potential solution. The definition of "good" or high-quality tests is unclear. An automated system for test selection may need deep understanding of the code in order to be effective. If a programmer must annotate their code at the level of Hoare-style verification, then such a hypothetical tool is unlikely to be worth anyone's time. Even if such a system were usable and generated good tests, the programmer would need to develop a familiarity with them in order to use them effectively. This could be mitigated if the system could name tests descriptively, but that is another hard problem.

Lack of acceptable solutions. Automatic test generation is a well-studied problem, and it still has not been satisfactorily solved. I know of two existing systems, Randoop and Iorek, but I am unsure what their technical strengths and weaknesses are. I suspect that lack of adoption is in large part due to lack of exposure, unwillingness to invest in learning new frameworks, and hesitation to trust and rely on these systems long-term.

Possible approaches. Common approaches to this problem rely on randomness, mutation, and constraint solving. Perhaps experience would change this opinion, but I think that a test generation system based on occasional program annotations of loop invariants, preconditions, postconditions, *etc.*, could be very useful. Such a system could integrate with Iorek — which uses constraint solving to find valid test cases — to generate tests that check these assertions in a variety of conditions.

Difficulty 2: Designing Isolated Tests

Description of the problem. After choosing what to test, implementing the tests in a way that minimizes interference from other components is a further challenge. Mocking frameworks effectively address this issue (as well as some others) for common system libraries, but minimizing interference from the programmer's own code is also critical. If tests are not well-isolated, then fault localization becomes a challenging, time-intensive process. This concern may prompt developers to avoid testing methods or procedures in combination, but tests relative to an initial program state do not necessarily generalize to all reachable states of the program.

Significance of the problem. Everyone should have high-quality tests. this problem is particularly concerned with the aspect of test isolation, the quality that a single logical fault should cause a fairly small, distinctive set of tests to fail. This quality facilitates effective debugging, because the programmer can quickly diagnose what's wrong with their code. If a large proportion of test fails in the presence of a bug, then the tests have done little to help the programmer figure out the issue (though its existence is at least known).

Challenges for a potential solution. It's unclear what interaction is essential or superfluous to a given test, so a solution would need to receive some sort of test specification from the user that indicates the "intent" of the test. The quality of the mechanism for test specification will significantly impact the

usability and utility of an overall system, so careful thought ought to be given to it. Even with that, the process of test generation is entirely non-trivial and may prove intractable if not sufficiently restricted.

Lack of acceptable solutions. I'm not aware of any tool that attempts to address this problem in the general case. Mocking frameworks do effectively address common aspects of this problem (with wide adoption), so perhaps this is adequate for the majority of developers. It could also be the case that this issue is not actually as pervasive as I had thought, or its consequences are easily identified and handled by experienced developers in practice.

Possible approaches. Given my background, this sounds like an optimal synthesis problem, *i.e*, synthesize a test equivalent to some reference that minimizes interaction with the code under test. However, program synthesis is a difficult problem and may be overkill for this problem. More simply, a framework or library for custom, application-state mocking could alleviate many aspects of the problem, but this approach also has many open-ended questions, such as how to express valid states. In turn, that issue could be handled by observing program states in a set of random executions, but the programmer may not have even intended every observed state to be valid in the first place. If the programmer is willing to provide an invariant, then it seems feasible to apply constraint solving to derive a set of particular program states to test against.

Difficulty 3: Debugging in a Domain-Specific Language

Description of the problem. I often struggle to effectively debug programs written in a domain-specific language (DSL). Since most DSLs are small-scale projects, language-specific debugging tools rarely exist. When a DSL is embedded, the host language's debugging facilities can help to some degree, but relating error reports at the level of the host language back to the source program is often challenging. Those who use Racket-embedded DSLs know this pain all too well; even constructs like `for` loops are implemented with macros (like most Racket DSLs), and the Racket interpreter reports the source of runtime exceptions within such a loop at no finer granularity than the loop itself. At the very least, DSLs should express error messages in terms of the source program and provide a usable stepping debugger.

Significance of the problem. The field of computing is increasingly trending towards DSLs and solutions. Prominent examples of successful DSLs (in no particular order) include OpenCL, CUDA, Hadoop, Spark, SQL, D3, Vega, and Rosette. As developers rely on these DSLs more and more, the availability of effective tools for debugging and quality assurance will take on increasing importance. This point is particularly acute for DSLs with unusual semantics, like Rosette, that are difficult to reason about without the aide of debugging tools.

Challenges for a potential solution. There is a broad spectrum of implementation approaches for DSLs, and capturing even a substantial fragment of them in a single solution is unlikely. Given the variety of DSLs, the requirements for a satisfactory solution are also unclear.

Lack of acceptable solutions. Those DSLs that are well established or supported by large corporations sometimes do have specialized debugging tools built for them (*e.g.*, the case for CUDA), though at great expense of time and resources. I suspect that most language implementors see this problem only from the context of their own project and are thus more interested in building powerful, specialized debugging tools for their language than for attacking the problem for DSLs in general.

Possible approaches. The most straightforward approach is to sink more time and resources into developing specialized debugging tools for each DSL, but this approach scales very poorly and is too resource-intensive in practice. A more promising approach is to examine common host languages and try to construct some sort of common framework for debugging in its embedded DSLs. Racket is a particularly good candidate for such a host language, given its strong tradition for embedding DSLs. In particular, this strong tradition gives a clear focus for integration, the macro system. In general, one could imagine a debugging framework for DSLs that language implementors could use to add debugging functionality to their language implementations, but this may require far-reaching assumptions about the languages and their implementations. At the same time, the debugging functionality provided may be too weak or hard to use for language projects to bother adopting the framework in the first place.