Getting it Right: System Testing of Scientific Software

Date: 15 May 2024
Presented by: Myra Cohen (Iowa State University)

(The slides are available via the link in the sidebar of the page linked above.)

Q: It would be interesting to hear what you have to say about the use of machine learning and how this impacts on the system testing. The ML seems to expand our search space a lot. Is there hope? 🙂 My issue is that some of the algorithms are sensitive to the input, for example, classification trees…

A: Machine learning does increase the challenges for system testing, but there is hope. We do need to consider both the input models and the software behavior when we model the system for testing since they both have an impact on the outcome. This is a similar problem to configurability. It increases our search space exponentially, but if we create models, we can still measure coverage across those models and sample the space during testing. We should also consider the parameter tuning of ML algorithms since that can also impact results. An interesting challenge is to be able to model the input/data in a way that we can group inputs into equivalence classes and this likely requires domain knowledge. Of course, the other challenge is the oracle problem. Metamorphic and differential testing may be able to help with that part.

Q: Are there any specific system testing techniques/tools for scientific software?

A: I think most techniques are used for all types of software and can be adapted to the scientific domain. There has been work on testing probabilistic or stochastic programs that can apply to many scientific systems, and the issue of precision, overflow, and compiler transformations of data when software is moved to different machines and/or operating systems is a big problem. Heavy numerical calculations are probably used more in the scientific domain, hence this is an important issue as well.

This arxiv article may be of interest. It was the outcome of the most recent DOE/NSF workshop on Correctness in Scientific Computing and it has many references.
https://arxiv.org/pdf/2312.15640

Q: Do you know of any tools for the coverage for R language?

A:

- Chip Jackson (attendee): covr should do what you want for R: https://covr.r-lib.org/
- Myra Cohen (speaker): thank you for finding this tool. It looks like this tool came out of a discussion of a consortium of R developers (https://github.com/RConsortium/Code-Coverage-WG?tab=readme-ov-file) interested in

this topic. There are also a couple of (unit) testing frameworks for R. One is RUnit (https://cran.r-project.org/web/packages/RUnit/index.html) for unit testing. Another is Testit (https://github.com/yihui/testit)

Q: Are there any resources to learn about system testing for scientific software?

A: A general tutorial on software testing would be a good place to start. The BSSw website has some resources. And some conferences have tutorials. US-RSE '23 had a talk session on software testing of scientific software. And SC 2023 had a Birds of a Feather session on Software Testing for Scientific Software: https://ssecenter.cc.gatech.edu/sc-23-bof/

Q: What are some strategies for "good" input generation/selection for testing? Through software generation (e.g. "random" tests), or perhaps manually designed cases?

A: There are two basic categories for input generation. Random testing (https://randoop.github.io/randoop), fuzzing, or search-based (such as evosuite https://www.evosuite.org/) all work from the code and generate tests based on 'what is there'. This is great if you want automated ways to cover the code. Random testing and search-based approaches tend to generate smaller test suites (and sometimes provide possible oracles – which can be manually checked) so that you can add oracles. Fuzzing generates larger numbers of tests so it is dependent on more obvious faults such as crashes or memory overruns. (Note - the two tools I pointed to are only for Java but are just meant to be representative of the class of tools.)

The second category is related to whether or not the system specifications are satisfied - i.e. 'Is the behavior of the software correct?' This requires generating tests based on the specifications. For this, you have to model the input space and then there are some automated approaches to combine/generate tests (such as the combinatorial approach I showed in my talk). Input models can be based on the grammar, configurations, or system parameters. For each, sets of equivalence classes (or partitions) are needed, and then representative data from within each partition can be chosen as concrete tests.

Q: Thanks for your great talk! I have two questions: (1) How can we conduct model/specification-driven testing when the model/specification of the target software system is not available? (2) Could you please comment on the major differences between testing scientific software and regular software?

A:

(1) For software without specifications, they need to be 'inferred'. This can be done using existing test cases as a starting point and filling in as testing is increased, using domain knowledge, or as bugs are reported – which often indicate missing specifications or at least tell you how users are expecting the system to behave. Other ways to do this is using code (or higher level invariants) – these are facts about code elements (e.g. daikon-like invariant code inference) or manually inferred based on the test cases and

system behavior. Some researchers have studied specification inference from documentation or by building documentation from code using natural language processing, and now even using Large Language Models.

(2) In my opinion, the major differences between testing scientific software and regular software are:

- Scientific software often lacks oracles
- Scientific software often uses models as inputs and these models may be imprecise or incorrect
- Scientific software is often probabilistic or stochastic
- The computation in scientific software often can only be done on high-performance computing systems limiting our ability to reason manually about much of the behavior except for very small cases
- Scientific software relies heavily on mathematical approximations which can lead to issues with precision

Of course, all of these issues/challenges are also present/challenges for other types of software, but I think they are more characteristic of scientific software in general, and usually more than one of these issues is present.

Q: Is creating Metamorphic relations another challenge for the developers? It sounds like a very time-consuming process.

A: The software engineering community has been working on some ways to help with this task such as mining documentation and more recently using LLMs. I do think that we need domain expertise in the form of a developer (or domain expert) to ensure that the relations are valid, but hopefully, automation can help develop

Q: There's a lot of talk now about C++ being memory-unsafe. Do you think good, practically realizable testing can alleviate this concern?

A: There are a lot of techniques for finding memory problems such as fuzzing. There are also program analysis techniques that can help tag issues in code.

Q: Does anyone have suggestions for software testing tools/platforms with containers like Docker or Singularity? VScode with extensions has the ability but has some compatibility issues with 32bit architecture. TotalView doesn't work with containers.

A. I don't have a good answer for this one. It may depend on what type of tests you are writing. Have you looked at BuildTest? The documentation says it can support containers although it is experimental. https://buildtest.readthedocs.io/en/devel/

Q: As per testing, we normally try to find out if the program gives us the right answers for things that we know. But there can be 1000 different algorithms that give us the right answers. The

program that we want is supposed to predict things that we have no answer. If we have no answers, how do we test?

A: This is a great question and the point of the two techniques I showed (differential and metamorphic) testing. With differential testing, we can evaluate if two different implementations that predict the same thing give us the same answer. That does not guarantee the answer is correct but gives us some confidence. In metamorphic testing, the idea is to use relations between sets of tests. It is often easier to define how something should change (get bigger, smaller, stay the same, be negated) than to know the exact answer. If we can build up enough relations (in particular ones related to domain knowledge) then we can also gain confidence that our answers are correct.