- **Due 29 May by 14:00**

Assignment 3

Model Based Testing

(DAT262 / DIT848)

Property-based Testing

Group 1

Supriya Supriya (supe@student.chalmers.se)
Himanshu Chuphal (guschuhi@student.gu.se)
Ephraim Eckl (guseckep@student.gu.se)

Property-based Testing

Introduction

Property-based specification of parts of the ND4j.

Junit-Quickcheck

It is a library that supports writing and running property-based tests in JUnit, inspired by QuickCheck for Haskell.

Property-based tests capture characteristics, or "properties", of the output of code that should be true given arbitrary inputs that meet certain criteria. For example, imagine a function that produces a list of the prime factors of a positive integer `n` greater than 1. Regardless of the specific value of `n`, the function must give a list whose members are all primes, must equal `n` when all multiplied together, and must be different from the factorization of a positive integer `m` greater than 1 and not equal to n.

Rather than testing such properties for all possible inputs, junit-quickcheck and other QuickCheck kin generate some number of random inputs, and verify that the properties hold at least for the generated inputs. This gives us some reasonable assurance over time that the properties hold true for any valid inputs.

junit-quickcheck library https://github.com/pholser/junit-quickcheck

Part 1. Design of properties

Q1. Which parts did you select to write properties for and why are they more likely to be good targets for property-based testing?

Q2. Did you find parts of the SUT which would be hard to test with PBT and why would they be hard?

Q3. Describe all of the properties you wrote. Include the purpose with each property, the code/implementation of the property as well as specific generators or setup code you needed.

Part 2. Manual mutation testing

Q4. Describe the mutations that you had to do and which property(ies) each mutation triggered as well as the output of the tool.

Q5. For at least 3 properties/mutations, analyse to what extent the output from the PBT tool would have helped you find the problems and debug it

Part 3. Comparison to MBT from Assignment 2

Q6. Investigate to what degree the introduced mutations from part 2 would have been found by the MBT testing you did in assignment 2. The focus should be on practical experiments here but if you could not create a large "overlap" between what was tested in assignments 2 and 3 you need to discuss theoretically if similar mutations that you introduced here (in part 2) could have been found by your testing in assignment 2.

Bonus Part: Run PITest on your library

Use the automated mutation testing tool PITest on your library and evaluate to what degree your PBT testing in assignment 3 and your MBT testing in assignment 2 constitutes "good" testing (in terms of mutation scores reported by PITest).

References

1. http://graphwalker.github.io/
2. http://www.cse.chalmers.se/edu/year/2017/course/DAT261/_static/modeljunit2_docs/nz/ac/waikato/modeljunit/package-summary.htm
3. https://deeplearning4j.org/docs/latest/nd4j-overview
4. http://graphwalker.github.io/Tests_execution/
5. Utting, Mark; Legeard, Bruno: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006. ISBN 0123725011.
6. Pimont, S. and Rault, J. (1976). A software reliability assessment based on a structural and behavioral analysis of programs. In International Conference on Software Engineering (ICSE), pages 486–491.
7. De Bruijn, N. G. (1946). A combinatorial problem. Koninklijke Nederlandse Akademie v. Wetenschappen, 49:758–764
8. https://inf.mit.bme.hu/sites/default/files/materials/category/kateg%C3%B3ria/education/software-verification-and-validation/18/SWVV-2018_L19_Model-based_testing.pdf