A Note on Rigour and Replicability

Panos Louridas
Greek Research and Technology Network S.A.
and
Department of Management Science and Technology
Athens University of Economics and Business
Athens, Greece
louridas@grnet.gr, louridas@aueb.gr

Georgios Gousios
Technical University of Delft
Makelweg 4
Delft, The Netherlands
G.Gousios@tudelft.nl

ABSTRACT

As any empirical science, Software Engineering research should strive towards better research practices. Replication is regrettably not a priority for Software Engineering researchers and, moreover, not afforded by many published studies. Here we report our experience from our encounter with a recent paper in a flagship Software Engineering conference. Our experience shows that current publication requirements do not guarantee replicability.

Categories and Subject Descriptors

D.2 [**Software Engineering**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Measurement

Keywords

ACM proceedings, empirical software engineering, statistics

1. INTRODUCTION

"Replication—the confirmation of results and conclusions from one study obtained independently in another—is considered the scientific gold standard"; thus started a special section in *Science* last year on Data Replication and Reproducibility [7]. A similar call was made a few months later in *Nature* [6]. In the context of software engineering, it has been known for a while that the quality of empirical studies is not up to the desired level [14, 20]. Replications are much touted [13, 15, 1] but rarely performed [16, 20].

To help others replicate their work, researchers should not only make their data available to others upon publication, but ensure that all their material, comprising code, should also be readily shared, to ensure full replication [12]. As Computer Scientists and Software Engineers we have the advantage of knowing how to do that—share our code, tools, data and documentation—and of living and working in a culture where openness, in the form of Open Source Software, has been the norm for many years now. In the effort to enable full replication, we should lead by our example.

In this note, we would like to draw our attention to problems with replication in our own field, taking as an example a paper that appeared in a recent flagship conference. The paper, "Combining Functional and Imperative Programming for Multicore Software: An Empirical Study Evaluating Scala and Java" appeared in the International Conference on Software Engineering ICSE that took place in Zurich on June 2–9, 2012 [11]. It drew our attention because it compares programming in a popular imperative language against programming in a hot multi-paradigm (imperative and functional) language; it so happens that we have had considerable experience with both languages. The results were quite surprising compared to our experience; unfortunately, we were equally surprised by our inability to check them.

In what follows we describe what we believe is wrong with the presentation of that particular paper. Note that we do not prove that the paper itself is wrong. To do that, or conversely to verify the paper's findings, we should be able to check the authors' methods. We were unable to do that. We believe that this holds valuable lessons for the materials that all authors should make available upon review and publication, and what journal editors and proceedings committees should require from the manuscripts they receive.

2. STATISTICAL RIGOR

The authors state at the outset that they "confirm with statistical significance Scala's claim that Scala code is more compact than Java code, but clearly refute other claims of Scala on lower programming effort and lower debugging effort". They "interpret $p \leq 0.05$ as a strong indication for a difference, which degrades as p increases; $p > 0.1$ is the threshold where the difference becomes insignificant". This is somewhat strange, as the threshold is usually taken to be $p < 0.05$, with strong statistical significance taken at $p \leq 0.01$.

The experiment was carried out by a single population (of size 13) that had to work out a programming project in Scala and in Java in four weeks. Seven of the subjects worked in Scala and six in Java. When the four weeks ended, the languages were reversed so that the six subjects that had worked with Java switched to Scala and the six subjects that had worked with Scala switched to Java.

To compare the effort required to finish the project in Scala and in Java the authors used the Wilcoxon's paired rank sum test. The authors explain that "infomaly speaking, this non-parametric test evaluates whether two populations differ with statistical significance". If this is what the authors intended, then they used the wrong test, as in their study they do not investigate the difference between two populations, but the difference in measurements in the *same* population (while programming in Java and in Scala). The rank sum test is the so-called Wilcoxon, Mann and Whitney test [4, Chapter 4], where "the data consist of two random samples, a sample from the control population and an independent sample from the treatment population" [4, p. 106]. By contrast, the Wilcoxon signed rank test [4, Chapter 3] is used with "pairs of 'pretreatment' and 'posttreatment' observations; here we are concerned with a shift in [the] location [median] due to the application of the 'treatment' " [4, p. 35] which is what the authors want to do. Note that in this discussion we are using the same reference statistics textbook as the authors of the paper.

The aggregated effort statistics, in terms of hours spent by each subject on each project, show that "on average it takes 20 hours (38%) longer to complete the Scala project. The populations differ significantly with $p = 0.059$". As we mentioned above, this p-value could very well be taken as non-significant. We tried to investigate that further. We took the effort numbers from the figure included by the authors in the paper, and ran the appropriate statistical tests ourselves. We used the R package for statistical computing. Our results are strikingly different from the authors, in that they do confirm their contention with a much more statistically significant p-value. In particular,

```
wilcox.test(scala.hours, java.hours, pair=TRUE)
```

produced a p-value of 0.002, with the warning that the exact p-value could not be computed due to ties. To get an exact calculation of the p-value we used the R coin library, where

```
wilcoxsign_test(scala.hours ~ java.hours,
  distribution=exact())
```

produced a p-value of 0.00049.

We proceeded to verify the paper's "multivariate analysis of variance (MANOVA) that analyzes the impact of Java and Scala skills (beginner/expert) on the Java and Scala effort of each subject". Unfortunately we were stymied, as the input data for MANOVA cannot be found in the paper or any attached figures. The results "show that expert skills lead to lower effort in comparison to beginners", which is not surprising. The authors validated their results based on the Box test for the equality of variance-covariance matrices. However, they do not mention how they validated the other assumptions required by analysis of variance, such as normality of sample populations. As the effort statistics were investigated using a non-parametric method (the Wilcoxon test), we would expect to find a justification for using a parametric test for the impact of skills on effort, but we did not, nor were we able to test the MANOVA assumptions ourselves.

Further statistical claims are made regarding Scala's code compactness. It is argued by proponents of the language that Scala code is more compact than Java and less prone to boilerplate. The authors performed a "Wilcoxon rank sum" test on their measurements. This would probably mean the Wilcoxon Mann and Whitney test, which would be suitable in this case, as we do not have "pre-" and "post-" condition measurements, but measurements from two independent samples. It is argued that "the paired Wilcoxon rank sum test on each subject's solution shows support ($p = 0.078$) that Scala code is more compact" in terms of Lines of Code (LOC)—a p-value that usually is counted as not statistically significant. A comparison in terms of characters finds that "the statistical support is weaker ($p = 0.094$)"—an even less statistically significant p-value.

3. EXPERIMENTAL DESIGN

Developer Experience. The empirical study involved 13 subjects, Master's students close to their graduation, on average in their fourth year of Computer Science studies. They reported "an average of four years of Java experience and no Scala experience". They were given four weeks of training with Java and Scala. Java training "covered parallel programming with shared-memory". Scala training "included functional programming and parallel programming with actors". Following the training, the subjects were asked to solve the Dining Philosophers problem in both Scala and Java. Their measure of proficiency in both languages was measured. In Java, seven subjects were classified as experts and six as beginners. In Scala, seven subjects were classified as experts and six as beginners. No details are given on how the classification was performed.

In computer science education literature, programming expertise classification usually follows the five stage scale (Novice, Advanced Beginner, Competent, Proficient, Expert) proposed by Dreyfus and Dreyfus [2]. It is generally agreed that it takes roughly ten years for a novice to become an expert programmer [19]. However, this does not mean that every two years a developer automatically gains an experience level; factors such as the complexity of tasks exposed to and competitiveness of the environment come into play. In popular programming culture, estimates range from from six months [17] to ten years [8].

In light of this evidence, it is surprising that six Master's students, close to their graduation, with four years of Java experience, were classified as still being beginners in the language. It is equally surprising that seven students were classified as experts in Scala, after only four weeks of training.

A way to measure language proficiency would be against familiarity with language constructs. Scala is a complex language, with many features. The creator of Scala has proposed a categorization of knowledge levels for the language [9]:

- Level A1: Beginning application programmer
- Level A2: Intermediate application programmer
- Level A3: Expert application programmer
- Level L1: Junion library designer
- Level L2: Senior library designer
- Level L3: Expert library designer

Each level comes with an brief summary (bullet points) of the skills required for it. The categorization has been adopted to classify the material in a recent book on Scala [5]. It does not appear that it was used in the classification of the subjects in the experiment. Unfortunately, there is no information on how exactly the classification was done, and in which way exactly the students had achieved expert status.

The difference in expertise may explain the subjects' statements regarding ease of programming in the two languages and their productivity with them: "Only 30% say that adapting to Scala's programming model was easy, compared to 100% for Java" (after four years of experience) and "92% of subjects feel productive in Scala, compared to 100% in Java" (which seems to be a solid endorsement of Scala, taking into account the short training period).

If real proficiency is reflected in the efficiency of the programs the subjects wrote, then the speedup analysis presented in the paper, comparing the speed gains by running the Scala and Java programs on multiple threads, would provide a clue. Again, we do not have access to the measurement data; the only available information is the paper text plus the figures. These show that the average speedup of Scala programs is about two and of Java programs about three. There are a few outliers with significant speedups. It is these programs that we would expect to be the work of the subjects with real expert status.

Functional vs. Imperative. Apart from evaluating programmer proficiency, the paper evaluates and uses as a basis for comparison the use of functional vs. imperative programming language constructs. To do that, the authors count keywords and methods: “`var`, `object`, `array`, `while`, `for`, `abstract`, `import java`, etc. indicate an imperative style. By contrast, constructs such as `val`, `list`, `map`, `filter`, `flatMap`, `foreach`, `:::` (list concatenation), `::` (list cons operator) indicate a functional style”. In each program the number of occurrences of imperative and functional programming language was measured and then a percentage figure of imperative vs. functional style was derived. The actual measurements are not provided by the paper; to validate the results the authors asked a Software Engineer to count manually the methods and classify them as using a functional or imperative style. The resulting functional / imperative percentages confirmed the automated counting method, with a median divergence of 20%; there is no statistical significance attached to that figure.

Statistics apart, we are not aware of a standard way to judge adherence to a functional programming style. In a Scala style guidelines proposal, functional programming includes “case classes as algebraic data types, options, pattern matching, partial functions, destructuring bindings, lazyness, call by name, `flatMap`” [3]. Or we could include function literals and closures, recursion, tail calls, lists maps and sets, traversal, mapping, filtering, folding and reducing, options, pattern matching, partial functions, currying, and so on [18, Chapter 8].

To drive the point home, we tried to replicate the authors’ method with a published example. Consider the following code showing the usual imperative code for matrix multiplication [10, Example 6.15.2].

```
def matmul(xss: Array[Array[Double]],
           yss: Array[Array[Double]]) = {

  val zss: Array[Array[Double]] =
    new Array(xss.length, yss(0).length)

  var i = 0
  while (i < xss.length) {
    var j = 0
    while (j < yss(0).length) {
      var acc = 0.0
      var k = 0
      while (k < yss.length) {
        acc = acc + xss(i)(k) * yss(k)(j)
        k += 1
      }
      zss(i)(j) = acc
      j += 1
    }
    i += 1
  }
  zss
}
```

Counting the keywords and language constructs we find that the code is 7% functional and 93% imperative. By contrast, consider the following code [10, Example 6.19.2], in which the creator of Scala explains how the above operation could be written in a functional way:

```
def transpose[A](xss: Array[Array[A]]) = {
  for (i <- Array.range(0, xss(0).length)) yield
    for (xs <- xss) yield xs(i)
}

def scalprod(xs: Array[Double], ys: Array[Double]) = {
  var acc = 0.0
  for ((x, y) <- xs zip ys) acc = acc + x * y
  acc
}

def matmul(xss: Array[Array[Double]],
           yss: Array[Array[Double]]) = {
  val ysst = transpose(yss)
  for (xs <- xss) yield
    for (yst <- ysst) yield
      scalprod(xs, yst)
}
```

Counting using the same method we find that the code is 6% functional and 94% imperative, i.e., less functional than the imperative version. Measuring code “functional-ness” automatically by means of keywords does not really work. More research needs to be done towards that direction, but conclusions cannot be drawn from the method presented by the authors.

4. A PROPOSAL

We believe that it is time for the Software Engineering community to raise the bar on what can be expected from an empirical study. ICSE and other top Software Engineering conferences can play an active role on that front. We would have encountered none of the problems outlined above if published papers included:

- All measurement data.
- All interviews, questionnaire, research protocols, and other related data derived from subjects, anonymized if necessary.
- Full details on the statistical methods used. These should include scripts and programs, so that it is easy for other researchers to run them. If statistical frameworks are used (e.g., R or SPSS), full details on the versions and libraries should be provided as well.
- Any other code that has been used in the publication’s research.
- Documentation for all of the above.

To enforce the above, conferences could require from authors to open up their data and their data manipulation tools under a license that enables everybody to use them. Sharing of data should happen in an organized way; for example, conference organization committees could create a shared repository where researchers can upload their data and tools along with instructions to use them. To enable full replication, researchers could provide virtual machine images with the full environment and data they used. Moreover, conferences and journals can describe a formal redress procedure; should an error is found in a paper, authors should be required to reply to the error claim.

We are aware that exact replication is not always possible; others believe that it may be unattainable [1, Chapter 4]. We are also aware that sharing of tools and data may be restricted due to reasons not having to do with the developer’s will to shared them; licensing and NDA compliance may be two of them. What we propose can be a best effort approach: by default, submissions should be accompanied by datasets and tools; if these are not available due to *force majeure*, it should be up to the editor/conference chair to decide on the submission.

5. CONCLUSIONS

The purpose of this note is not to point fingers, but to raise the issue of the dangers of inadequate reproducibility. We were drawn to this particular article and use it as an example mostly because some of the findings contradict our own experience. Other articles in the same conference are equally opaque with regards to replication and verification. However, we believe that publication-time availability of experimental data, tools and experiment replication documentation ("lab packs" as Basili [13] describes them) should be a requirement for publication. Our proposal, if adopted, might be a first step in this direction.

6. REFERENCES

[1] A. Brooks, M. Roper, M. Wood, J. Daly, and J. Miller. Replication's role in software engineering. In Forrest Shull, Janice Singer, and Dag I. K. Sjøberg, editors, *Guide to Advanced Empirical Software Engineering*, pages 365–379. Springer London, 2008. 10.1007/978-1-84800-044-5_14.

[2] H.L. Dreyfus and S.E. Dreyfus. *Mind over machine.* Free Press, 1988.

[3] Marius Eriksen. Effective Sscala. `http://twitter.github.com/effectivescala`, 2012.

[4] Myles Hollander and Douglas A. Wolfe. *Nonparametric Statistical Methods.* John Wiley & Sons, Inc., 2nd edition, 1999.

[5] Cay S. Horstmann. *Scala for the Impatient.* Addison-Wesley, 2012.

[6] Darrel C. Ince, Leslie Hatton, and John Graham-Cumming. The case for open computer programs. *Nature*, 482:485–488, 23 February 2012.

[7] Barbara R. Jasny, Gilbert Chin, Lisa Chong, and Sacha Vignieri. Again, and again, and again.... *Science*, 334:1225, 2 December 2011.

[8] Peter Norvig. Teach yourself programming in ten years. `http://norvig.com/21-days.html`, 2001.

[9] Martin Odersky. Scala levels: beginner to expert, application programmer to library designer. `http://www.scala-lang.org/node/8610`, 2011.

[10] Martin Odersky. The Sscala language specification version 2.9. Technical report, Programming Methods Laboratory, EPFL, Switzerland, May 24 2011.

[11] Victor Pankratius, Felix Schmidt, and Gilda Garretón. Combining functional and imperative programming for multicore software: An empirical study evaluating Scala and Java. In *Proceedings of te 34^{th} International Conference on Software Engineering*, Zurich, June 6–9 2012.

[12] Roger D. Peng. Reproducible research in computational science. *Science*, 334:1226–1227, 2 December 2011.

[13] Victor R.Basili, Forrest Shull, and Filippo Lanubile. Building knowledge through families of experiments. *IEEE Transactions on Software Engineering*, 25(04):456–473, 1999.

[14] M. Shaw. Writing good software engineering research papers. In *Proceedings of the 25th International Conference on Software Engineering, 2003.*, pages 726–736, May 2003.

[15] Forrest Shull, Jeffrey Carver, Sira Vegas, and Natalia Juristo. The role of replications in empirical software engineering. *Empirical Software Engineering*, 13:211–218, 2008. 10.1007/s10664-008-9060-1.

[16] Dag I.K Sjøberg, J.E. Hannay, O. Hansen, V.B. Kampenes, A. Karahasanovic, N.-K. Liborg, and A.C. Rekdal. A survey of controlled experiments in software engineering. *IEEE Transactions on Software Engineering*, 31(9):733–753, Sept. 2005.

[17] Bjarne Stroustrup. Posting to comp.lang.c++. `http://www2.research.att.com/~bs/blast.html`, December 1994.

[18] Dean Wampler and Alex Payne. *Programming Scala.* O' Reilly, 2009.

[19] Leon E. Winslow. Programming pedagogy—a psychological overview. *SIGCSE Bull.*, 28(3):17–22, September 1996.

[20] Carmen Zannier, Grigori Melnik, and Frank Maurer. On the success of empirical studies in the International Conference on Software Engineering. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 341–350, New York, NY, USA, 2006. ACM.