View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Apollo

Software Testing in a Scientific Research Group

Matthew Patrick
Department of Plant Sciences
University of Cambridge
United Kingdom
mtp33@cam.ac.uk

James Elderfield
Department of Plant Sciences
University of Cambridge
United Kingdom
jade2@cam.ac.uk

Richard O. J. H. Stutt
Department of Plant Sciences
University of Cambridge
United Kingdom
rs481@cam.ac.uk

Andrew Rice
Computer Laboratory
University of Cambridge
United Kingdom
andrew.rice@cl.cam.ac.uk

Christopher A. Gilligan
Department of Plant Sciences
University of Cambridge
United Kingdom
cag1@cam.ac.uk

ABSTRACT

Scientific software is more difficult to test than many other software products, but scientists are not usually trained in software engineering techniques. Considering how often software is used to produce scientific results, how can we be sure the predictions made from these results are correct? Software engineering techniques should be useful for computational scientists. The problem is they find it difficult to know how to apply domain-independent techniques to the specific problems they face in their work. Nevertheless, we have discovered scientists use their own intuition to reinvent techniques surprisingly similar to those in software engineering. This seems like a good place to start our training.

CCS Concepts

•**Software and its engineering → Software creation and management; Software testing and debugging;**

Keywords

scientific software; software testing; training and education

1. INTRODUCTION

In a recent survey [7], over 70% of biological, mathematical and physical science researchers said they develop software as part of their job and 80% claimed it would not be possible to conduct their work without such software. Since scientific software is used and developed to answer important research questions, it is crucial that it performs correctly. Yet, only a third of scientists think training in software development is important and more than half admit they do not have a good understanding of software testing [11].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC'16, April 4-8, 2016, Pisa, Italy

Copyright 2016 ACM 978-1-4503-3739-7/16/04…$15.00

http://dx.doi.org/xx.xxxx/xxxxxxx.xxxxxxx

Scientific software is particularly hard to test because it is not always clear what the correct outputs should be [5]. Researchers' perceptions change as their hypotheses are tested and redefined through a process of scientific exploration. In addition to programming mistakes, there are also likely to be errors due to the way in which experimental data are collected and the choice of numerical approximation [5]. Recently, the Software Carpentry Foundation decided to withdraw software testing from its lesson plan because it was difficult to identify suitable tolerances in the values being tested [13]. Programming mistakes can lead to subtle errors that are difficult to detect, as the results are believable.

Sadly, some scientists have found out the importance of software testing the hard way. One research group retracted five papers from top level journals, including Science, because its software had a fault which inverted the protein crystal structures they were investigating [11]. Similarly, nine packages for seismic data processing were found to produce significantly different results due to problems such as off-by-one errors [6]. Other researchers report wasting time trying to improve their models or develop better algorithms, when the real issue was that their software contained faults [4].

It would be tempting to conclude scientists are unwilling to learn the techniques required. However, this paper presents a more positive story. We found that even though scientists are not aware of many testing techniques, they sometimes apply common sense to arrive at similar solutions. It takes considerable time for scientists to learn software testing techniques because they do not have the necessary background in software engineering. We argue that by starting from things scientists already do to ensure their software is correct, we can teach software testing in way that is more accessible.

The contributions of this paper are as follows:

1. We show how researchers use their intuition to find techniques similar to those in software engineering - this could be an effective starting place for education.
2. We explore how to move from scientists' intuitive manual testing towards more systematic automated tests.
3. We use past experiences to seek out ways in which problems with legacy code can be avoided in the future.

2. CASE STUDY

The Epidemiology and Modelling group at the University of Cambridge develops and applies models for the optimisation of disease control. Members of the group are typically trained in mathematics, statistics, biology and physics rather than software engineering. However, it is vital that their results and the predictions made from their models are correct, as they are used to inform policy for government organisations, such as DEFRA and the USDA, as well as multi-government responses to disease outbreaks in Africa.

Figure 1 describes some of the challenges faced in testing software developed by the group to model the stochastic behaviour of complex biological systems: habitat, disease and meteorological data is used over a range of spatiotemporal scales; stochastic interactions occur between species and the models frequently involve highly nonlinear dynamics. We need to separate the unavoidable error in data and mathematical assumptions from any programming mistakes.

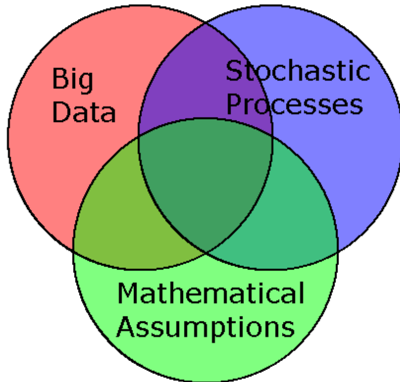

Figure 1: Some Challenges in Testing the Software

Big data causes difficulties for testing because the results depend on large quantities of spatial and temporal information that has a high potential for error, even before we start using it. Spatially explicit simulation tools have been developed within the group to model the spread of disease using a compartmental modelling framework [10]. The time at which hosts move from one compartment to another depends on their proximity to infected hosts. It is challenging to test software that relies on complex spatiotemporal data.

Stochastic processes also cause problems because it is not possible to specify any one correct output. Instead, there is a distribution of potential outputs with varying probabilities. For example, model parameters are determined by statistical inference of historical data, using techniques such as MCMC (Markov Chain Monte Carlo) and ABC (Approximate Bayesian Computation). It is difficult to test whether software is correct when outputs are probabilistic.

Finally, the mathematical assumptions made in models have a significant impact on predictions for the spread of an epidemic. For example, a decision must be made whether to model individuals or to take a mean-field approach [12]. Although these techniques might be expected to give similar results, the details will differ depending on the biological system. Software testing is challenging as we do not know what effect the assumptions made will have on the results.

3. RESULTS

We gave 12 members of the group a survey to find out about the practices they are currently using to ensure the software they develop is correct. We asked them to indicate whether they knew about 10 commonly used software engineering techniques and if they did, whether they used them. For each technique, an explanation was given using non-technical terms, in case the researcher did not know the specific terminology. The aim of this study is to identify areas in which additional training in software engineering techniques might be helpful for a diverse group of modellers with varying backgrounds, working on a range of fundamental and applied problems in epidemiology. It is hoped that by sharing these findings, other researchers developing scientific software might be able to gain from our experiences.

Previous studies have asked questions about scientific software engineering [5][7], but we thought it would be helpful to find out which languages the group members use, as developing and testing scripts is likely to be different from compiled software. Half of the group program in C++, for reasons of efficiency (see Figure 2). However, MATLAB and R are just as popular because they are easy to program and have useful libraries for mathematical modelling. Linux shell scripts are also used, since they allow members to send tasks to the group's computing cluster. The large number of languages used within the group, the wide variety of tasks they are applied to and the diverse range of programming abilities all pose challenges to ensuring software is correct.

Figure 3 shows the software engineering techniques members of the group have heard of and use. The results are representative of researchers with a wide range of backgrounds, from experienced programmers through to people just starting to learn. Some of the techniques were used by all the researchers (e.g. manual testing) and some were used by none (e.g. coverage metrics). The techniques cover topics in black box testing, white box testing and code clarity. We interviewed group members individually after the survey to learn more about the reasons for these results.

By far the most popular method of making sure software is correct was manual testing. By this, researchers meant they run code with particular inputs and then look to see whether the outputs appear to be correct. This is typically achieved using an intuitive understanding of what the results should be, or by comparison with previous results. In one way or another, everyone in the group uses this technique to help ensure the correctness of their programming code. Code clarity techniques, such as modularisation are also widely used, though as we will see later not all the time.

By contrast, more sophisticated techniques such as automated unit tests, coverage metrics and boundary/partition testing are seldom used. Many respondents claim to be using assertions, but interviews later revealed they have a different interpretation of this technique from established literature. Systematic techniques require specialist software engineering knowledge and formal representations of the testing problem, which are typically not available for scientific research. We need to close the gap between software engineering research and scientific computing. That means both sides need to learn more about what the other does.

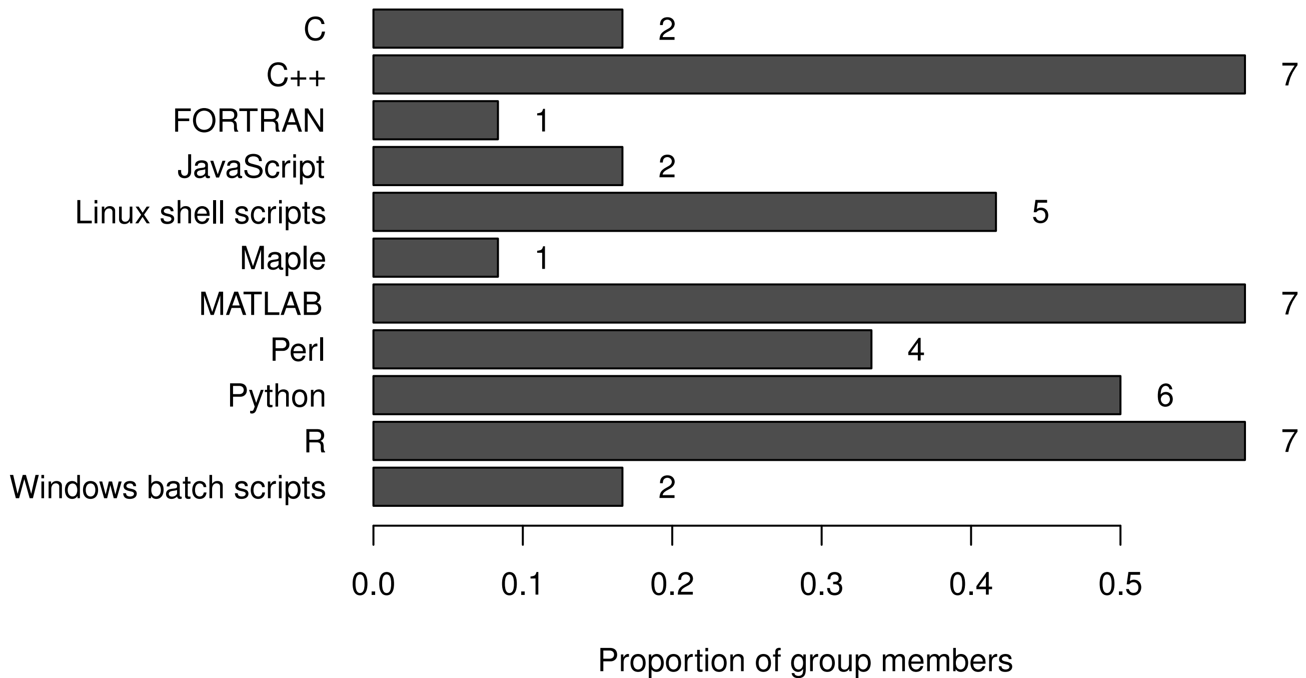

Figure 2: Programming languages used by 12 members of the Epidemiology and Modelling group

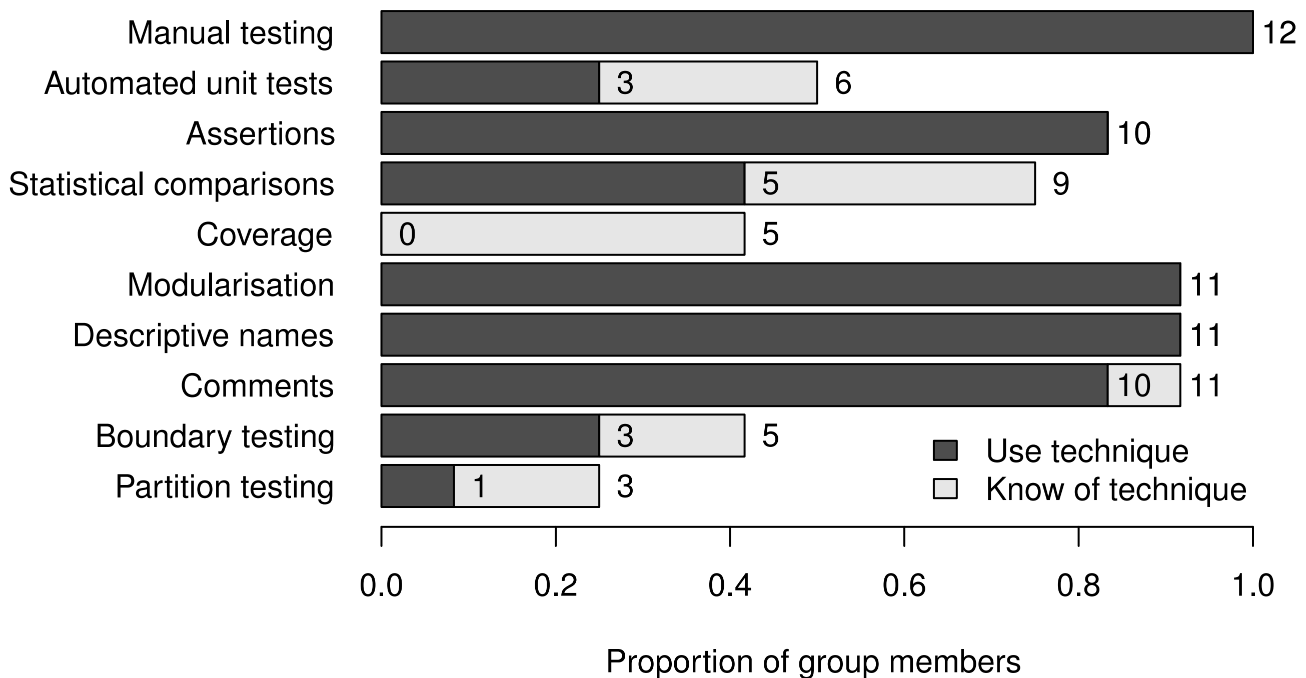

Figure 3: Software Engineering techniques used by 12 members of the Epidemiology and Modelling group

3.1 Trying not to reinvent the wheel

It was clear from the survey that members of the group were less familiar with the more sophisticated techniques. Fewer than knew what boundary testing or partition testing were and less than a quarter actually uses them. No-one in the group is applying coverage metrics to evaluate their test cases. Furthermore, it was later discovered one respondent who claimed to use boundary testing was actually confused about the difference between this and interaction testing. These results are consistent with a lack of education on software engineering in the wider scientific community [5].

However, even though members of the group tend to approach software testing intuitively rather than methodically, their intuition sometimes leads to techniques similar to those used in software engineering. Consider for example the challenge of determining whether test outputs are correct. In software engineering this is known as the oracle problem [1]. It is particularly difficult for scientific software developed to find out the answer to a research question; such programs are known as 'non-testable' [14], because the oracle is just as hard to write as the software. In line with software engineering, group members have addressed this by seeking various forms of partial and pseudo oracle to evaluate their outputs.

One option is to start by building functionality that already exists and has results that have been published in the literature. It is then possible to check whether the outputs are similar to those indicated by published graphs or tables, before extending the code with new functionality. Another option is to use a toy data set for which the expected behaviour can be produced using a simpler methodology. For example, the model implemented by a spatial simulator can be tested against the output of a non-spatial simulation if the spatial landscape is reduced to a single grid square. In some cases, an analytical solution may be used to check whether the stochastic simulation is correct, as it should give approximately the same result. In each case, the goal is to take a difficult testing problem and collapse it down to a simple case for which the expected output is easily obtainable.

For situations in which it is not possible to identify an appropriate pseudo-oracle, researchers are often able to describe how the output is expected to change when the input is adjusted. For example, if the average distance spores can travel is increased, we would expect to see the pattern of disease to become more dispersed. This type of approach is known in software testing research as a metamorphic relation [1]. Another strategy is to look for strange and unexpected results. For example, some group members identify thresholds above or below which output values would seem suspicious and then check whether or not these conditions occur. In other cases, it may be possible to identify outputs that are internally inconsistent, such as when a set of frequencies do not add up to one. It is interesting to note that even though the researchers who were interviewed did not realise it at the time, this is a form of boundary testing [8].

The fact that scientific researchers are already using some of the techniques advocated for software testing (albeit with a lesser degree of rigour), provides a natural route for educating them about software engineering. We can build upon the tactics researchers already know and are using to teach them how they relate to established techniques. For example, partition testing [8] is an effective way to cover the input domain thoroughly, whilst at the same time reducing the number of test cases that have to be evaluated. Members of the group seemed reluctant to apply partition testing in their work, as they could not envision how it could be applied. In particular, they were unclear as to how to identify suitable partitions for a continuous input domain.

There are, however, instances in which researchers already use partitions without realising it. For example, the basic reproduction number (R_0) is used in epidemiology to determine whether or not a pathogen will be able to spread through the population [3]. Researchers know that if R_0 is greater than one, they should expect the pathogen and hence the disease to spread, whereas if R_0 is less than one the disease will die out. They look intuitively for these differences in output behaviour when performing manual testing, but do this in an ad hoc way without regard to the established practices for partition testing. Similarly, the case where $R_0 = 1$ provides a natural starting point for teaching about boundary testing, as it lies directly between these two partitions. Of course, it is worth considering whether these testing techniques will be effective for the application domain. Not every technique is appropriate in every situation.

3.2 From manual tests to unit tests

Although half of the group had heard of automated unit tests, less than a quarter actually use them. Many see automated tests as too much effort, without being sufficiently helpful for their work. By contrast, one of the key motivations for automated unit testing is that, after the initial outlay of effort to set up the tests, it can provide a significant reduction in human effort compared with manual testing [2]. Rather than checking the output values by hand, or stepping through the program to see whether it behaves as expected, unit tests can be made to check automatically whether the output or intermediate variables meet certain criteria.

The main difficulty in creating unit tests is in specifying what the software is expected to do. Scientists are able to test software manually by checking the output, but they find it difficult to describe these tests in a more formal way. It is often not possible to know what the results will be before the software is run, but scientists are intuitively able to spot results that seem incorrect. Manual testing will always be an important part of scientific software engineering, especially early in the development process. However, if we can record this intuition in automated unit tests, this will enable more rigorous and systematic testing of established code.

How can we determine which differences in behaviour we need to be aware of in the unit tests and how can we make sure the software is able to answer the research questions correctly (both now and in the future)? We need an iterative way to improve the tests as we find out more about the system we are researching. Initially we can create tests of basic functionality, but then refine those tests to include more sophisticated measures of correctness. We would be building a set of unit tests that can be run regularly, much in the same way as regression testing [2], except rather than just using some criteria to ensure behaviour does not change, we would assess whether those criteria should be improved.

Assertions are a convenient way to check the behaviour of software at various points in its execution. Although some group members have started to use assertions with automated test frameworks, the majority use ‘if’ statements to check for errors, then output a warning to the screen. The problem with this approach is that the error messages can be missed amongst the other text produced. To combat this problem, researchers manually introduce the ‘if’ statements during development to check whether particular parts of the code are working correctly, but then remove them once testing is complete. Such checks are helpful, but they are difficult to automate and information is lost when they are removed. It would be better to integrate the assertions into unit tests, so we can have a record of the errors that occurred and the tests that found them. Not only will this help us to find faults more quickly in the future, but it serves as a starting point for the iterative refinement of our tests.

An issue arises over knowing whether we have tested the software sufficiently. Structural coverage metrics [2] assess the quality and robustness of unit tests to potential sources of failure. However, considering very few people have written automated tests, it is not surprising they are yet to use coverage metrics. One respondent did not think coverage metrics are useful, because even if all the code is covered, the tests might still be useless. Although it is true coverage is only useful if there is a suitable oracle for each test, metrics are a practical way to encourage developers to write more effective tests. Beyond simple control coverage, we might consider more sophisticated metrics, such as data flow or mutation analysis [2]. It may also be possible to devise new metrics for scientific research, that consider how thoroughly we have tested our answers to the research questions.

3.3 Avoiding the perils of legacy code

The Epidemiology and Modelling group has had problems with legacy code. When researchers leave the group, the people taking over their work describe the code as being difficult to understand because it contains arbitrary constants and the purpose of each function/variable is not always clear. It can take a long time to reproduce the results of the previous research before even starting to extend its work. Researchers should be encouraged to improve the clarity of their code as it benefits other people in the group and makes it possible for their software to be reused.

It can also be difficult to be confident that software which contains legacy code is correct. Even if advanced testing techniques are used to identify errors in the output, this does not mean the developer will be able to identify the faults correctly. If legacy code is being used the developer does not understand, there is a danger he or she will respond to errors and failing test cases by fixing the symptom of the problem rather than the cause. It is important the programming code is clear and easy to read, as this makes it more likely for mistakes to be spotted and corrected appropriately.

One characteristic of programming code that has a significant impact on its clarity is the way in which it is divided into modules. Group members typically start writing code as a single large module, then extract sections of it into functions if it represents a clear independent feature, or is used repeatedly by the program. There is a danger this produces some large complex modules. Practitioners recommend modules should be small and should only do one thing [9]. This simplicity at the module level makes it easy to see what the code is doing. However, it does require planning. One group member used to develop code into packages for R, so it can be easily reused, but no longer does this as he finds it takes too long. Similarly, some researchers admitted using copy and paste for reasons of speed. We need to find a trade-off between actions that take less time in the short term, but may cause longer term problems in maintenance.

On the whole, there is no consistent naming system within the group. Researchers use a combination of CamelCase and underscores, sometimes within the same program. It surprised me to find that a couple of researchers were using a form of Hungarian notation, as most people consider this to be old-fashioned and unhelpful [9]. Yet, the particular naming system being used is probably not as important as the names themselves being meaningful. Many of the researchers interviewed said that they try to use meaningful names, but few were able to give a clear description of what it means for a name to be meaningful. Two key points are that names should be intention revealing (it should be immediately clear why the variable exists, what it does and how it is used) and avoid disinformation (taking care with entrenched meanings and making sure the names are not too similar to tell apart) [9]. It should not be necessary to use a comment to reveal the purpose of a variable or module.

Comments should be used to explain implementation decisions and to inform the reader of concepts underlying the code [9]. The group members who were interviewed had a clear understanding of what comments should and should not be used for. One researcher uses comments to specify the equation or model and other researchers use comments to indicate the existence of edge cases. Group members said they try not to make comments too lengthy or confusing and avoid including redundant information that is already clear from the code. However, when we examined the group’s version control repository, we found that researchers do not always follow their own guidelines. Large sections of the code remained uncommented and comments were often used to disable old code rather than to provide useful information.

The survey results suggest members of the group know how to write code that is clear and easy to read. However, in practice they find this takes too much time. In research environments, there is often pressure to spend more time writing publications than developing software. As long as the code performs the necessary research task, it is often considered good enough. How then can we encourage researchers to write code that is usable by other people? Recommended techniques include code reviews and pair programming [9]. It can also be helpful to make the code open source. Yet people are reluctant to take time out of their research to engage in these activities, it may seem daunting for others to scrutinise their code and they may be worried about ownership. Perhaps the most effective strategy is to focus on the benefits for using their own code again. Copying and pasting large sections of code is error prone and further changes are not propagated across the various copies. By contrast, suitable modularisation, commenting and naming systems help improve research productivity with regards to reuse.

4. THREATS TO VALIDITY

The study presented in this paper was conducted with a small number of participants (12 members of the Epidemiology and Modelling group). More could be learnt by involving participants from groups in other research areas. The survey contained a limited number of questions, but the responses were followed up by interviews. A longer term study could be used to investigate changes in testing practices over time.

5. CONCLUSIONS

Scientific software is difficult to test because it is not always clear what the correct outputs should be. There are also challenges from big data, stochastic processes and mathematical assumptions. Yet it is important the results are correct, as they are used to inform government policy. Group members are not trained in software engineering techniques and they tend not to test programs methodically. Instead, they determine whether or not their code is working by looking at the output and checking it matches what they expect.

However, we have also found group members use intuition to discover techniques similar to those in software engineering. For example, to address the oracle problem, group members have sought various forms of partial and pseudo oracle. They also use metamorphic relations to describe how the output is expected to change when the input is adjusted. We believe software testing can be taught more effectively if we start with the techniques scientists are already using.

A key part of this training is to help researchers transform intuitive tests into systematic tests that can be applied automatically. Yet, since perceptions of correctness change over time, we need an iterative way to improve the tests. Another barrier is that people are under pressure to produce results quickly and they fear this may take too much time. We think this can be addressed by explaining the techniques in terms of reusability. If the software is well tested and divided into appropriate modules, it is much easier for the researcher or other people to save time by using their code again.

6. FURTHER WORK

The results of our survey suggest further training in software engineering techniques might be helpful. In many cases, the group members have not heard of techniques for ensuring their software is correct. Introducing them to these techniques might encourage more systematic testing. We will devise experiments to determine when each technique would be useful. Where no appropriate technique exists, we will create new techniques to handle specific challenges (big data, stochastic processes and mathematical assumptions etc.).

Since scientific software recreates events that happen in the real world, we can use techniques, such as lab and field experiments, not available to most software engineers. In particular we will investigate how to use them in parallel with the software development, so as to iteratively improve the tests. We will also evaluate coverage metrics and devise new metrics at the level of the experiments and hypotheses. Finally, we will extend our experiments and repeat this survey with other research groups within to see if the same trends hold, or if there are differences between fields.

7. ACKNOWLEDGMENTS

This work was supported by the University of Cambridge/ Wellcome Trust Junior Interdisciplinary Fellowship "Making scientific software easier to understand, test and communicate through modern advances in software engineering".

8. REFERENCES

[1] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The Oracle Problem in Software Testing: A Survey. *Software Engineering, IEEE Transactions on*, 41(5):507–525, 2015.

[2] I. Burnstein. *Practical Software Testing: A Process-Oriented Approach*. Springer, New York, NY, 2003.

[3] O. Diekmann, J. Heesterbeek, and J. Metz. On the Definition and the computation of the basic reproduction ratio R_0 in models for infectious diseases in heterogeneous populations. *Journal of Mathematical Biology*, 28(4):365–382, 1990.

[4] P. Dubois. Testing scientific programs. *Computing in Science & Engineering*, 14(4):69–73, 2012.

[5] J. E. Hannay, C. MacLeod, J. Singer, H. P. Langtangen, D. Pfahl, and G. Wilson. How Do Scientists Develop and Use Scientific Software? In *Soft. Eng. for Computational Science and Eng., ICSE*, 2009.

[6] L. Hatton and A. Roberts. How accurate is scientific software? *Software Engineering, IEEE Transactions on*, 20(10):785–797, 1994.

[7] S. Hettrick, M. Antonioletti, L. Carr, N. Chue Hong, S. Crouch, D. De Roure, I. Emsley, C. Goble, A. Hay, D. Inupakutika, M. Jackson, A. Nenadic, T. Parkinson, M. I. Parsons, A. Pawlik, G. Peru, A. Proeme, J. Robinson, and S. Sufi. UK Research Software Survey 2014. https://zenodo.org/record/14809.

[8] P. C. Jorgensen. *Software Testing: A Craftsman's Approach*. CRC Press, Boca Raton, FL, 4 edition, 2013.

[9] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, Upper Saddle River, NJ, 2008.

[10] R. Meentemeyer, N. Cunniffe, A. Cook, J. Filipe, R. Hunter, D. Rizzo, and C. Gilligan. Epidemiological Modeling of Invasion in Heterogeneous Landscapes: Spread of Sudden Oak Death in California (1990-2030). *Ecosphere*, 2(2), 2011.

[11] Z. Merali. Computational science: Error, why scientific programming does not compute. *Nature*, 467(7317), 2010.

[12] M. Renton. Shifting focus from the population to the individual as a way forward in understanding, predicting and managing the complexities of evolution of resistance to pesticides. *Pest Management Science*, 69(2):171–175, 2013.

[13] Software Carpentry Foundation. Why We Don't Teach Testing (Even Though We'd Like To), 2014.

[14] E. J. Weyuker. On Testing Non-testable Programs. *The Computer Journal*, 25(4):465–470, 1982.