Interactive Tool for Iterative Test Suite Construction

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

***Abstract*—We can only test software effectively if we understand how it is intended to behave. For some categories of programs, such as scientific models, it is not obvious what the output of the software should be. New techniques are needed to help domain experts, such as scientists, gather the knowledge they need to construct suitable tests and oracles. This paper introduces a new interactive tool for iterative test suite construction that is based upon the scientific method paradigm that scientists are familiar with. We apply our technique to a deterministic mathematical model, used to predict the spread of disease, and show how it helps scientists uncover situations they had not yet considered. Of the 15 hypotheses originally created by modellers, our technique found discrepancies in all but one, allowing us to refine them into a more rigorous test suite.**

I. Introduction

The Human Oracle Problem [1] can impose a significant obstruction to the quality of software testing. It is relatively easy to produce a large number of test cases using automated techniques, but the effort required to determine whether each of their outputs is correct can be prohibitively expensive. An oracle is a mechanism by which it is determined for each input what the output should be. Testing techniques typically assume the availability of an automated oracle [1]. Yet, many programs belong to a category of software which are said to be 'non-testable' [2]. This means it is just as expensive to develop an automated oracle (and just as hard to make sure it is correct), as it is the software we are testing. The end result is that human experts have to act as the oracle instead, evaluating each output one at a time, to assess if it is correct.

In addition to problems with the throughput of test output evaluations, there are also issues related to accuracy. One research group had to retract five papers from top level journals, such as Science, because its software contained a fault that remained unnoticed [3]. It was later learnt the protein crystal structures they were investigating were being inverted. Similarly, nine packages for seismic data processing were found to produce significantly different results due to problems such as off-by-one errors [4]. The predictions made from the packages would have led people using them to come to different conclusions, potentially leading to $20 million oil wells being drilled in the wrong place. We cannot just rely on experts to look at each output and check it matches their expectation. This approach is likely to miss important errors, since the output may appear reasonable and still be incorrect.

Scientific software is particularly difficult to test, because its correct behaviour is not normally known in advance. The software is developed to investigate new research questions and the perceptions of researchers change as their hypotheses are explored [5]. If we already knew the answers to these questions, there would be no need to create the software in the first place. Intrinsic difficulties occur in evaluating the output of scientific software, due to the stochasticity and uncertainties in the underlying model. Scientific software frequently incorporates complex models with nonlinear dynamics that are difficult to test. Each numerical approximation has the potential to introduce new errors [5], inaccuracies may arise due to the way in which experimental data are collected, and then when the model is implemented on a finite precision computer, some further accuracy is inevitably lost.

In this paper, we address the Human Oracle Problem by adapting the well-known strategy of Iterative Hypothesis Testing to allow domain experts to produce stronger and more reliable oracles for test suites. Iterative Hypothesis Testing is a core component of the scientific research method. Scientists start with initial hypotheses about how the system they are studying is expected to behave. Then, as these hypotheses are tested and explored, new information is gathered that can be used to add to and refine them [6]. Similarly, scientists might have some initial hypotheses as to what the outputs of the software they are developing should be, but they do not at first have sufficient information to create a rigorous set of test cases. Instead, Iterative Hypothesis Testing can be used to find cases in which these hypotheses do not hold, and identify ways in which they can be improved through repeated refinements.

Since our technique operates in a form scientists are already familiar with, it is highly suitable for testing scientific software (it can also be used on a wide range of other software). We apply our tool to an implementation of an epidemiological model. Hypotheses are generated in consultation with epidemiological modellers and then refined using our tool. The rest of this paper is arranged as follows: Section II gives more details about our testing technique and Section III describes the model we are testing; Section IV presents our initial hypotheses and Section V explores the results of applying Iterative Hypothesis Testing to them; Section VI presents some related work; Section VII provides our conclusions and Section VIII describes some opportunities for further research.

II. Our Interactive Test Suite Construction Tool

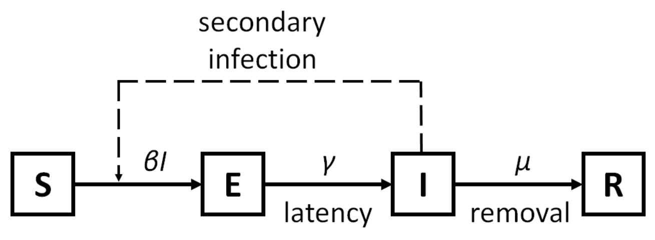
We introduce a new tool that helps domain experts iteratively refine their test suites. An interface is provided in which hypotheses can be entered as predicate relations about the output values produced for particular sets of inputs. The interface can also be used to define relations which describe how the output should change if the input is modified. These 'metamorphic relations' are an established way to test software for which an oracle is not available [7]. However, our tool differs from previous research by enabling the iterative refinement of these relations through a search-based approach.

Our tool uses random testing to identify discrepancies in the output between the values expected by the hypotheses and the values actually produced. It then performs a directed search to identify conditions under which the discrepancies occur. Our tool then proposes a slightly modified form of the hypotheses to address these discrepancies. For example, suppose we incorrectly assumed that one of the outputs of our software was always less than 10. Our technique uses random testing to identify cases in which this does not hold, then explores the search space to conclude it is never greater than 11. Of course, there is no guarantee the new hypotheses will not fail, and they may need to be refined further.

Rather than simply trusting the new hypotheses proposed by our tool, it is often more effective to use it interactively. If our tool cannot find any discrepancies at first, we can increase the number of random test cases it generates and instruct the tool to consider boundary conditions (such as setting values to 0). When the tool indicates particular hypotheses and input/output values that fail, we can look back at the software and consider why this is happening. The information provided by the tool allows a domain expert to determine whether the discrepancy indicates a fault in the software, or if it is caused by a misunderstanding of how it should behave. This helps the expert decide if the software or hypotheses should be changed.

Our tool is built on top of QuickCheck for R [8]. We provide a wrapper interface that transforms hypotheses into parametrised unit tests. These are then tested using automatically generated random test cases. Random testing is a straightforward and inexpensive software testing technique [9]. It can generate a large number of input values in a short amount of time, then verify the results using automatic tests of our hypotheses. Despite its simplicity, random testing is often more effective than advanced testing techniques [10].

Fig. 1. SEIR Model schematic



III. Case Study: The SEIR Model

We evaluate our technique by applying it to an epidemiological *SEIR* model [11]. The *SEIR* model (see Figure 1) tracks the hosts of a pathogen through the following compartments: *Susceptible* (not infected), *Exposed* (infected but neither infectious nor showing symptoms), *Infectious* (infectious and showing symptoms) and *Removed* (no longer infectious, because they are dead or recovered). Hosts may start off susceptible (S), but when they are exposed (E) to infectious hosts, they enter into a latent period before being infectious (I); later the infectious hosts are removed (R) from the population.

We run tests with a fixed total amount of host ($N = 2000$); the amount of infectious host (I) is selected uniformly at random between 0 and $N/2$, and the remaining host is placed into the susceptible (S) compartment. These settings were designed to reflect the conditions under which new infectious material enters the population from outside the model. γ and μ are selected uniformly at random between 0 and 1, β between 0 and $1/N$. Each simulation is run using 100 test cases over 50 time units, with a discrete time step of 0.01 units.

IV. Initial Hypotheses

1) Sanity Checks:

H1: None of the compartments should ever contain a negative amount of host (it is biologically impossible)

H2: The total amount of host should not differ at each time step (our model assumes a closed population)

H3: The amount of susceptible host (S) should never increase (infected hosts do not become susceptible again)

H4: The amount of removed host (R) should never decrease (The R curve should be monotonically increasing)

H5: The exposed host (E) peak should not occur after the infectious host (I) peak (E acts as a buffer for S to I)

2) Metamorphic Relations:

H6: Increasing the infection rate (β) should increase the peak of E (by creating a build-up of host)

H7: Increasing the latent rate (γ) should reduce the time until the peak of I (by allowing host to move faster from E)

H8: Increasing μ should increase the final amount of host in S (by allowing host to move faster from I)

H9: Increasing β should decrease the final amount of host in S (by allowing host to move faster from S)

H10: Increasing the number of susceptible host (S_0) should increase the peak of I (more host to be infected)

3) Mathematical Derivations:

H11: I should be increasing when $\gamma E > \mu I$, otherwise it should be decreasing (from rate equation for I)

H12: If $I = E = 0$, the state of the model should not change (every term in the equations contains the value I or E)

H13: Exact analytical solutions are available when $\gamma = 0$ ($I_t = I_0 e^{(-\mu t)}$ and $S_t = S_0 e^{\frac{\beta I_0 (e^{-\mu t} - 1)}{\mu}}$)

H14: Exact analytical solutions are available when $\beta = 0$ ($I_t = \frac{e^{-\gamma t}(E_0 \gamma (e^{t(\gamma-\mu)} - 1) + I_0 (\gamma - \mu) e^{t(\gamma-\mu)})}{\gamma - \mu}$)

H15: A final size equation can determine the value of S when $t = \infty$ ($\ln(\frac{S_0}{S_\infty}) = \frac{\beta}{\mu}(N - S_\infty)$)

V. Exploring and Refining the Hypotheses

We applied our interactive tool to explore and refine the initial hypotheses presented in Section IV. Our tool found discrepancies in all but one of the hypotheses (**H4**). In many cases, the hypotheses tests can be refined automatically, through simple changes (e.g. the introduction of an edge case or the selection of suitable tolerance thresholds). However, it is sometimes necessary for a domain expert to pick the most appropriate change. In this section, we consider some examples of how our tool was able to help refine the hypotheses.

A. Complexities of the Model

Some discrepancies occurred due to edge cases in the model behaviour, not taken into account by the initial hypotheses. It is likely these edge cases would be identified by domain experts, if allowed a sufficient amount of time. However, it is much easier to find these cases automatically using our tool. For example, in **H5** we reasoned that since host pass through the E compartment on their way to the I compartment, the peak of E would be before that of I in time. Our tool found a case in which this did not hold, using the following input parameters: $I_0 = 477$, $\beta = 4.95 \times 10^{-4}$, $\gamma = 0.285$, $\mu = 0.986$. The discrepancy can be seen in Figure 2: the peak of E does not come before the peak of I because I is monotonically decreasing (i.e. it has no peak). A similar problem occurs with **H7**: increasing γ cannot make the peak of I come earlier if it is already monotonically decreasing. These hypotheses tests were refined by adding exceptions for the edge cases.

In some cases, it is only possible to identify discrepancies in the hypotheses by using specific boundary values. Random testing is inefficient at finding these cases because it explores the input domain evenly, so our tool also selects values from the edge of the input domain. We found boundary tests to be particularly useful for the situations in which parameters or compartment values are zero. For example, when $S = 0$, changing the value of β has no effect on the peak of E, because there is no host to move into E from S. We therefore added an exception for this situation in **H6**. Similarly in **H8**, increasing μ has no effect on the final amount of host in S when $\beta = 0$ because host can never leave the S compartment. By finding and addressing these edge cases, our tool helps to refine the hypotheses tests and make them usable in practice.

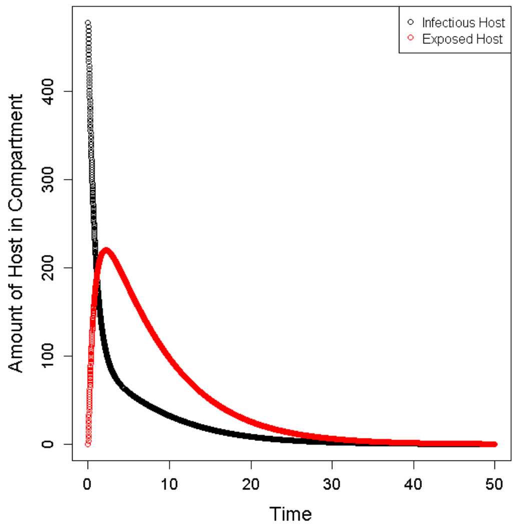


Fig. 2. Peak in E but not in I (H5)

B. Complexities of the Implementation

In addition to edge cases in the model, our tool can also be used to explore the differences between mathematical behaviour and the behaviour of the model when implemented on a finite precision computer. Disparities are addressed by introducing a tolerance threshold into each hypothesis test. The optimum value of the threshold depends on the hypothesis being tested. For example we found the optimum threshold for **H1** to be around 1×10^{-6}, whereas for **H2** it was around 2×10^{-8}. Our tool identifies the optimum threshold by incrementally increasing its value and counting the number of times the hypotheses fail. Hypotheses tests fail on every run if the threshold is set too low, but if it is set too high, the hypotheses will never fail (even if there is a fault in the software). Our tool therefore sets the threshold to the smallest value at which there are no failures during 100 runs.

Finite precision computation also has an effect due to the discrete time steps that are used. For example, our tool found that **H11** does not hold because it incorrectly identifies the points at which I transitions from increasing to decreasing (see Figure 3). Just before the transition point, $\gamma E < \mu I$, so I should decrease. However, our tool found the next value can be higher, because the mathematical minimum occurs between the time steps and then the value of I increases. The same problem occurs when calculating the transition point at the maximum of I. Using our tool, we were able to address this by only checking up until the point before the transition.

VI. Related Work

There have been other attempts to help make the construction of test oracles easier. For example, Staats et al. [12] use mutation testing and Loyola et al. [13] use network centrality analysis as a predictive measure to rank variables for their ability to detect faults. These approaches are useful for guiding testers towards parts of the code that deserve greater consideration, but they are not designed to help scientists in the iterative process of refining their test suite.

Salari and Knupp [14] suggest a number of tests that are useful for scientific software. For example, the Method of Manufactured Solutions creates tests by solving a partial differential equation backwards, i.e. inverting the equation to determine the inputs needed to create the correct solution. They also proposed trend tests (varying the input parameters and checking the overall pattern), symmetry tests (e.g. changing the order of inputs and checking that the results are the same) and comparison tests (using pseudo-oracles) [14]. Our work differs from this in that in addition to proposing a set of tests, we provide a tool for automatically refining them.

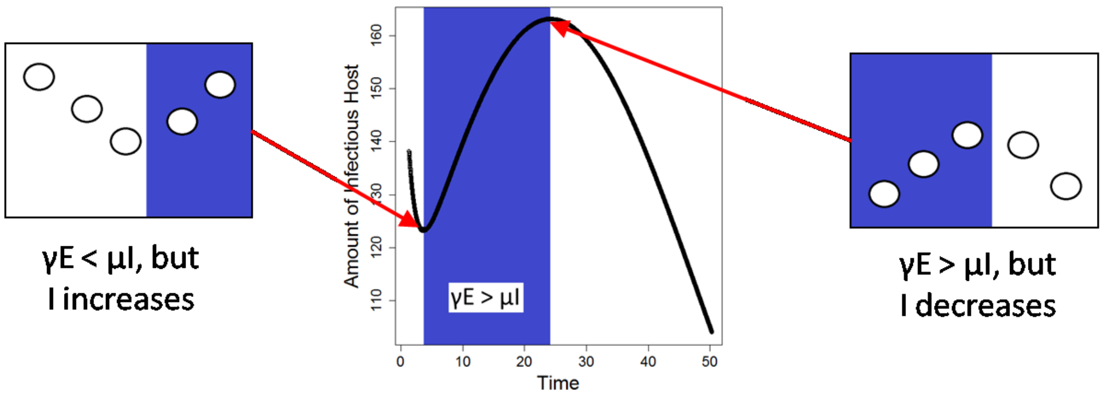


Fig. 3. Incorrectly Identified Increases and Decreases in I (H11)

VII. Conclusions

We presented a new interactive tool for iterative test suite construction that is easy for scientists to use because it is based on a methodology they are already familiar with (Iterative Hypothesis Testing). We have shown our tool can find discrepancies resulting from unforeseen complexities of the model and implementation. Although these issues might have been identified manually, if given enough time, it is faster and easier for scientists to create some initial hypotheses and then use our tool to identify the unexpected cases for which they need refining. The information gathered by our tool can be used to improve the tests through a mixture of automatic and interactive refinement. As a result, scientists can easily use our tool to create rigorous test suites for their software.

VIII. Further Work

We applied our tool to a simple model, so we could explain it more clearly. When discrepancies were found, it was relatively straightforward to identify the source of the error and update our hypotheses appropriately. However, this might not be so easy with complex models or other forms of scientific software. We need to make sure our tool can be used to identify the cause of each discrepancy, so we can respond efficiently. It would therefore be useful to evaluate our tool on other, more complex, software. We might also like to consider alternative strategies for exploring hypotheses, such as those based on advanced search-based software testing techniques or symbolic analysis of the differential equations.

Another issue arises over how we can know whether our approach is sufficiently rigorous. We will evaluate the effectiveness of metrics for coverage assessment, such as those based on control and data-flow criteria. It might be also be worthwhile to create new metrics specifically designed for scientific software, measuring coverage at the level of the experiments and hypotheses. Since scientific software recreates events that happen in the real world, we can also use techniques such as lab and field experiments to verify our results. We will investigate how best to use these in parallel with software development, to iteratively improve the tests.

IX. 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”. We thank Richard Stutt and James Elderfield for useful discussions.

References

[1] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, “The Oracle Problem in Software Testing: A Survey,” *IEEE Trans. Software Engineering*, vol. 41, no. 5, pp. 507–525, 2015.

[2] E. J. Weyuker, “On Testing Non-testable Programs,” *The Computer Journal*, vol. 25, no. 4, pp. 465–470, 1982.

[3] Z. Merali, “Computational science: Error, why scientific programming does not compute,” *Nature*, vol. 467, no. 7317, 2010.

[4] L. Hatton and A. Roberts, “How accurate is scientific software?” *Software Engineering, IEEE Transactions on*, vol. 20, no. 10, pp. 785–797, 1994.

[5] J. C. Carver, M. S. Starkville, R. P. Kendall, S. E. Squires, and D. E. Post, “Software Development Environments for Scientific and Engineering Software: A Series of Case Studies,” in *Proc. 29th Int. Conf. Software Engineering*, 2007, pp. 550–559.

[6] J. Hannay, C. MacLeod, J. Singer, H. Langtangen, D. Pfahl, and G. Wilson, “How Do Scientists Develop and Use Scientific Software?” in *Soft. Eng. for Computational Science and Eng., ICSE*, 2009. [Online]. Available: http://dx.doi.org/10.1109/SECSE.2009.5069155

[7] Z. Q. Zhou, D. H. Huang, T. H. Tse, Z. Yang, H. Huang, and T. Y. Chen, “Metamorphic Testing and its Applications,” in *Proc. 8th Int. Symp. Future Software Technology*, 2004.

[8] A. Piccolboni, “quickcheck,” 2015. [Online]. Available: https://github.com/RevolutionAnalytics/quickcheck

[9] J. W. Duran, “An Evaluation of Random Testing,” *IEEE Trans. Software Engineering*, vol. 10, no. 4, pp. 438–444, 1984.

[10] S. Shamshiri, J. M. Rojas, G. Fraser, and P. McMinn, “Random or Genetic Algorithm Search for Object-Oriented Test Suite Generation?” in *Proc. GECCO*, 2015, pp. 1367–1374.

[11] M. Y. Li, J. R. Graef, L. Wang, and J. Karsai, “Global dynamics of a SEIR model with varying total population size,” *Mathematical Biosciences*, vol. 160, no. 2, pp. 191–213, 1999.

[12] M. Staats, G. Gay, and M. P. E. Heimdahl, “Automated Oracle Creation Support, or: How I Learned to Stop Worrying about Fault Propagation and Love Mutation Testing,” in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 870–880.

[13] P. Loyola, M. Staats, I.-Y. Ko, and G. Rothermel, “Dodona: Automated Oracle Data Set Selection,” in *Proc. Int. Symp. Softw. Testing Anal.*, 2014, pp. 193–203.

[14] K. Salari and P. Knupp, “Code Verification by the Method of Manufactured Solutions,” Sandia National Laboratories, Tech. Rep. SAND2000-1444, June 2000.