Biophys Rev (2015) 7:343–352
DOI 10.1007/s12551-015-0177-3

REVIEW

How to test bioinformatics software?

Amir Hossein Kamali[1,2] · **Eleni Giannoulatou**[1,3] · **Tsong Yueh Chen**[4] · **Michael A. Charleston**[5] · **Alistair L. McEwan**[2] · **Joshua W. K. Ho**[1,3]

Received: 10 June 2015 / Accepted: 24 July 2015 / Published online: 13 August 2015

Abstract Bioinformatics is the application of computational, mathematical and statistical techniques to solve problems in biology and medicine. Bioinformatics programs developed for computational simulation and large-scale data analysis are widely used in almost all areas of biophysics. The appropriate choice of algorithms and correct implementation of these algorithms are critical for obtaining reliable computational results. Nonetheless, it is often very difficult to systematically test these programs as it is often hard to verify the correctness of the output, and to effectively generate failure-revealing test cases. Software testing is an important process of verification and validation of scientific software, but very few studies have directly dealt with the issues of bioinformatics software testing. In this work, we review important concepts and state-of-the-art methods in the field of software testing. We also discuss recent reports on adapting and implementing software testing methodologies in the bioinformatics field, with specific examples drawn from systems biology and genomic medicine.

✉ Joshua W. K. Ho
j.ho@victorchang.edu.au

[1] Victor Chang Cardiac Research Institute, Darlinghurst, NSW 2010, Australia

[2] School of Electrical and Information Engineering, The University of Sydney, Sydney, NSW 2006, Australia

[3] St. Vincent's Clinical School, The University of New South Wales, Sydney, NSW 2010, Australia

[4] Department of Computer Science and Software Engineering, Swinburne University of Technology, Melbourne, VIC, Australia

[5] School of Physical Sciences, The University of Tasmania, Hobart, TAS, Australia

Keywords Software testing · Bioinformatics · Quality assurance · Automated testing · Cloud-based testing

Introduction

Nowadays, the use of computer programs is pervasive in many areas of biomedical sciences, especially in biophysics, genomics, proteomics, biotechnology and medicine. Rapid accumulation of high-throughput datasets and an increasing focus on systems-level biological modelling increase the size and complexity of bioinformatics programs. This poses a great challenge in developing a good testing strategy to ensure the reliability of the design and implementation of these programs (Chen et al. 2009).

Nonetheless, the mechanism of scrutinising the software implementation of these programs is often far less comprehensive than the rigorous peer review process of the research articles that describe the programs' applications (Check Hayden 2015). The potentially widespread problem of errors or misuses of scientific computing in biology and medicine is highlighted by recent news and commentary articles in top-tier journals such as *Nature* and *Science* (Joppa et al. 2013; Merali 2010), and the problem could be attributed to a lack of proper software verification and validation (Alden and Read 2013; Check Hayden 2013). Incorrect computed results may lead to wrong biological conclusions and may misguide downstream experiments. In some cases, it may lead to retraction of scientific papers (Check Hayden 2015).

This problem is especially critical if these programs are to be used in a translational clinical setting, such as the analysis of whole genome sequencing data for identifying genetic variants in a patient's DNA sample. In a genetic variant calling pipeline, one must have high confidence that the resulting variant calls have high sensitivity and specificity. A recent

comparison of five commonly used variant-calling pipelines demonstrates that the overall concordance of the variant calls was low (only 57.4 % for single nucleotide variants and 26.8 % for indels were concordant across the tested pipelines) (O'Rawe et al. 2013). The study found that a large portion of pipeline-specific variant calls could be validated by independent means, suggesting that each pipeline may be missing a lot of genuine genetic variants. This is particularly troubling, since although false positive variant calls can be distinguished from true positives through external validation, it is almost impossible to systematically distinguish false negatives from the vast amount of true negatives.

Besides variant calling, the use of different variant annotation programs and transcript annotation files can also make a substantial difference in annotation results that are not commonly appreciated. McCarthy et al. (2014) recently examined the effect of using different transcript annotations and different variant annotation programs. They found that a non-trivial proportion of variants were annotated differently due to the use of different transcript annotations, or different programs. These troubling reports highlight the need to ensure that bioinformatics pipelines are subjected to better verification and validation.

Software testing is an important step towards developing high quality software. It has been challenging for software developers, especially for developers of scientific software. In a recent survey of nearly 2000 scientists, it was found that in the past five years 45% of scientists spent more time developing software. Nonetheless, less than half of them had a good understanding of software testing (Hannay et al. 2009; Merali 2010). Performing proper software testing can be a time-consuming task, accounting for up to 50 % of the total software development time (Myers et al. 2011). Therefore, it is especially important to make sure we use effective and systematic software testing strategies.

Many efficient software testing concepts and techniques have been developed over the years (Myers et al. 2011). Recently, some groups, including ours, are beginning to adopt state-of-the-art software testing techniques to test scientific software (Baxter et al. 2006; Murphy et al. 2009a), including bioinformatics software (Chen et al. 2009). This review begins by outlining several key concepts in software testing, followed by discussing state-of-the-art testing techniques. Furthermore, we review recent case studies that have applied various software test strategies to verify or validate bioinformatics software.

Software testing definitions and concepts

Software testing is the process of actively identifying potential faults in a computer program. Software testing can be *static* or *dynamic. Static* testing involves code review or inspection, whereas *dynamic* testing involves execution of the Program Under Test (PUT) using a given set of test cases. The rest of the review focuses on techniques for dynamic software testing. In dynamic software testing, the PUT can be thought of as implementation of a (mathematical or computational) function $f(x) = y$ where x represents all valid input from the input domain and y represents all possible outputs. The goal of verification is to show that for a given implementation f_{PUT} of PUT, $f_{PUT}(x) = f(x)$ for all possible x from the input domain. An input, $x_{failure}$ is a *failure-causing input* if $f_{PUT}(x_{failure}) \neq f(x_{failure})$, and the PUT is deemed to contain a failure. A failure reveals an underlying fault in the implementation of the program, which in turn is a manifestation of an error introduced by the programmer (Lanubile et al. 1998). Common terminologies used in the software testing field are summarised in Table 1.

A PUT may fail because of incorrect implementation of the algorithm (i.e., the verification problem), or a mismatch between the algorithm and the intended behaviour (i.e., the validation problem). In other words, verification asks, "Are we building the software right?" whereas validation seeks to answer, "Are we building the right software?" In addition to the limitations of the algorithm and implementation, failure can also be caused by incorrect expectations of the intended use of a program (Joppa et al. 2013), and runtime hardware or system failure. For the rest of this review, we mainly focus on methods that test the limitations of the implementation of the program, i.e., verification, but some ideas can be extended to validation as well.

Why is bioinformatics software testing difficult?

There are two main challenges in testing scientific software, especially bioinformatics software: the *oracle problem* and the *test case selection problem*.

The oracle problem

In dynamic software testing, an *oracle* is a mechanism that decides if the output of the PUT is correct given any possible input. This mechanism is most useful if it is computationally simpler than the algorithm of the PUT. For example, for a program that implements a sorting algorithm with complexity of $O(n\log n)$ for sorting n numbers, a possible oracle is to use a simple $O(n)$ algorithm that traverses through the output sorted sequence to check the condition: the number in the $(i+1)^{th}$ position is always greater or equal to the number in the i^{th} position. When such a test oracle exists, we can apply a large number and variety of test cases to test the PUT since the correctness of the output can be verified using the oracle.

Table 1 Definition of commonly used terms in software testing

Key term	Definition
Validation	"The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements." (IEEE 1990)
Verification	"The process of evaluating a system or component to determine whether the products of given development phase satisfy the conditions imposed at the start of that phase." (IEEE 1990)
Quality control	"A set of activities designed to evaluate the quality of developed or manufactured products." (IEEE 1990)
Quality assurance	"A planned and systematic pattern of all actions necessary to provide adequate confidence that an item or product conforms to established technical requirements." (IEEE 1990)
Test case	"A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement." (IEEE 1990)
Test suite	"A set of several test cases for a component or system under test, where the post condition of one test is often used as the precondition for the next one." (ISTQB 2015)
Test reliability	"A set of test data T for a program P is reliable if it reveals that P contains an error whenever P is incorrect. It is important to note that it has been proven that there is no testing strategy that can check the reliability of all programs." (Howden 1976)
Regression testing	"Testing of a previously tested program following modification to ensure that defects have not been introduced or uncovered in unchanged areas of the software, as a result of the changes made. It is performed when the software or its environment is changed." (ISTQB 2015)
Oracle	"A mechanism, which can systematically verify the correctness of a test result for any given test case." (Liu et al. 2014)
Test oracle problem	"The oracle problem occurs when either an oracle does not exist, or exists but is too expensive to be used." (Liu et al. 2014)
Black-box testing	"Testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions." (IEEE 1990)
White-box testing	"Testing that takes into account the internal mechanism of a system or component. Types include branch testing, path testing, statement testing." (IEEE 1990)
Test coverage	"The degree to which a given test or set of tests addresses all specified requirements for a given system or component." (IEEE 1990)
Fault	"Fault – concrete manifestation of an error within the software. One error may cause several faults, and various errors may cause identical faults." (Lanubile et al. 1998)
Error	"Defect in the human thought process made while trying to understand given information, solve problems, or to use methods and tools. In the context of software requirements specifications, an error is a basic misconception of the actual needs of a user or customer." (Lanubile et al. 1998)
Failure	"Departure of the operational software system behavior from user expected requirements. A particular failure may be caused by several faults and some faults may never cause a failure." (Lanubile et al. 1998)
Successful test	"A test that cannot reveal any error in the implemented software using given test case." (Chen et al. 1998)
Static testing	"Testing of a component or system at specification or implementation level without execution of that software, e.g. reviews or static analysis." (ISTQB 2015)
Dynamic testing	"Testing that requires the execution of the test item." (IEEE 2013)

Majority of these terms are defined in IEEE Standard Glossary 610.12-1990 (IEEE 1990) and International Software Testing Qualification Board Glossary (ISTQB 2015) and ISO/IEC/IEEE 29119 (IEEE 2013).

Without a tangible oracle, the choice of test cases is greatly limited to those *special test cases* where the expected outputs are known or there exists a way to easily verify the correctness of the testing results. In particular, an *oracle problem* is said to exist when: (1) "there does not exist an oracle" or (2) "it is theoretically possible, but practically too difficult to determine the correct output" (Chen et al. 2003; Weyuker 1982). The existence of a practical oracle is essential when performing systematic program testing. Many bioinformatics programs suffer from the oracle problem since they often deal with large input and output data, and implement complex algorithms. Consider the situation of the molecular dynamic simulator or a short read sequence aligner. In both cases, the correctness of the output is very hard to verify.

The test case selection problem

In dynamic software testing, the main approach is to execute a set of test cases. If a failure is detected after executing a test case, a fault is identified. However, not all test cases can trigger a failure even if the PUT contains one or more faults. Therefore, an effective software testing strategy often aims

to identify the smallest set of test cases that reveals as many different faults as possible. A test case is simply an input drawn from the input space of the software. Many bioinformatics programs have a large input space, and it is often computationally challenging to automatically survey this space efficiently to identify the most failure-revealing test cases. This is the test case selection problem. A good software testing methodology often makes use of some knowledge of the likely position of failure-causing input to select potentially fault-revealing input as test cases.

Software testing methodologies

Many methods have been developed in the software testing field. Many of them are designed to address the oracle and the test case selection problems. In the following sections, we review several testing methods that are commonly used in the area of scientific computing, with a focus on their advantages and limitations. This is not an extensive list of methods, but rather a selection of methods that illustrate important concepts and considerations when developing a software testing strategy. These methods are illustrated in Fig. 1 and summarised in Table 2.

Special test case testing

Special test case testing is perhaps the most widely used approach in software testing. In special test case testing, the program's functionality is tested over a predefined set of (input, output) pairs known as the special test cases, which can be used to verify the correctness of the program. For example, to test the correctness of the program P that computes $sin(x)$ for any given x, some values like $x = \pi/2$ and $x = \pi/6$ can be considered as special cases since the result of P for these inputs is well known (1 and 0.5, respectively). Special test case testing is a useful strategy for performing testing in the absence of an oracle. The method has the advantages of being intuitive and easy to implement. Using this approach, any inconsistency between the program's output and the expected output is considered to be a failure, which directly suggests an underlying fault. Nonetheless, the biggest limitation of the approach is that the choice of test cases is often very limited, which prohibits the application of more systematic testing strategies. It does not solve the oracle problem or the test case selection problem, but it serves as a good point of reference for a comparison with other testing methodologies.

N-version programming

In *N*-version programming (NVP), the correctness of a program is checked by comparing the outputs generated by multiple independent implementations of the same algorithm (or the same general requirement) against the same set of inputs (Chen and Avizienis 1978). It is expected that the outputs obtained from these implementations will be the same for all test cases. At the end of a test round, a tester can conclude whether the outputs are concordant or discordant. To increase the effectiveness of this method, it is recommended that different developers implement these different versions (Chen and Avizienis 1978; Knight and Leveson 1986). For example NVP was used to discover the low concordance of variant calling results produced by five commonly used variant calling pipelines (O'Rawe et al. 2013). Compared to the use of simple test cases, one major advantage of *N*-version programming is that it enables any input to be used as a test case. In other words, this method can perform software testing on the entire input domain without the need of an oracle. This approach is readily implementable if multiple versions of the same program already exist. The main disadvantage of this approach is that it cannot decide which individual version/program contains a fault if the outputs of multiple versions do not agree. Also, this approach is expensive and may not always be feasible in practice.

Metamorphic testing

Metamorphic testing (MT) alleviates the oracle problem by using some problem domain-specific properties, namely metamorphic relationships (MRs), to verify the testing outputs. The central idea is that although it is impossible to directly test the correctness of any given test case, it may be possible to verify the *expected relationships* of the outputs generated by multiple executions of a program over the *source* and *follow-up test cases* by comparing their corresponding outputs against the MRs (Chen et al. 1998; Zhou et al. 2004). In other words, MT tests for properties that users expect of a correct program. If a MR is violated, for any pair of source and follow-up test cases, the tester reports a failure in the program. MT has been successfully applied to test many different types of software, such as numerical programs (Zhou et al. 2004), embedded software (Kuo et al. 2011), analysis of feature models (Segura et al. 2010), machine learning (Murphy et al. 2008; Xie et al. 2011), testing service oriented applications (Chan et al. 2007), and big data analytics (Otero and Peter 2015).

A simple and classical example of MT is to test the correctness of an implementation of a program that computes the $sin(x)$ trigonometric function, using some well-known mathematical properties of the function as MRs (Table 3). These MRs express the expected relationships between outputs from the source test cases (left side of the equations), and outputs from the follow-up test cases (right side of the equations). For example, we may design a source test case of $x_1 = 1.345$. Based on MR2, a possible follow-up test case is $x_2 = 1.345 + \pi$. The output of

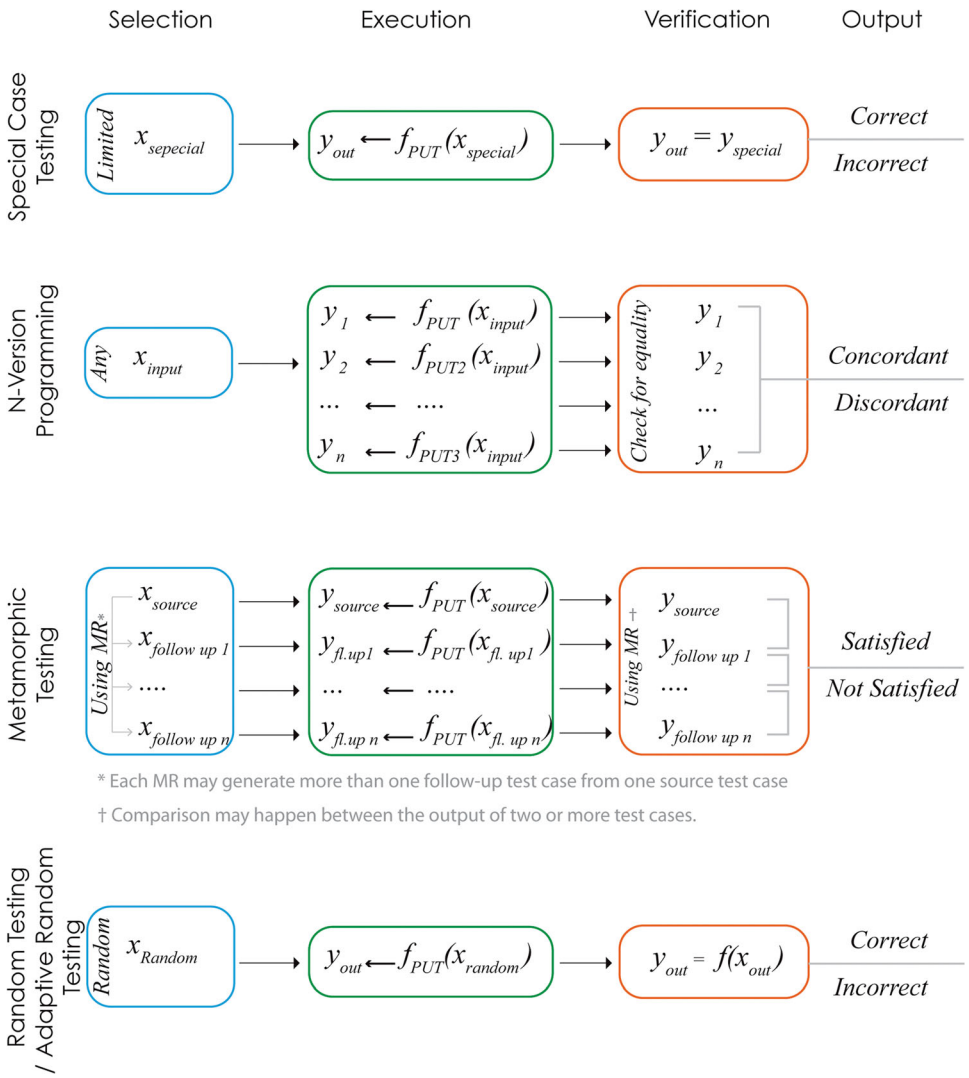

Fig. 1 Comparison of different testing techniques

the source and follow-up test cases are then compared to check whether MR2 is satisfied, i.e., $sin(x_1) = -sin(x_2)$. It should be noted that this follow-up test case can then be used as a source test case to generate additional follow-up test cases, such as $x_3 = 1.345 + 2\pi$.

As illustrated in the $sin(x)$ example, an MR is used for two purposes: (1) to generate additional follow-up test cases by modifying the source input, and (2) to check the relationship between the outputs produced by the execution of the source and follow-up test cases. It should be noted that in general many follow-up test cases can be derived from a single source test case input based on one MR. It is important to note that satisfying an MR does not necessarily imply the program is correct. Nonetheless, violation of an MR does imply the presence of a fault.

Table 2 Comparison of advantages and disadvantages of testing techniques

Methodology	Test case selection	Test case coverage	Testing output	Requires oracle?	Alleviates the oracle.problem?
Special case testing	Predefined	Limited	Faulty/not faulty	No	No
NVP	Input space	Input space	Concordant/discordant	No	Yes
RT/ART	Random	Input space	Faulty/not faulty	Yes	No
MT	Based on relationships	Nearly all input space	Satisfied/not satisfied	No	Yes

NVP N-version programming; *RT* random testing; *ART* adaptive random testing; *MT* metamorphic testing

Table 3 Metamorphic relationships for $sin(x)$

MR	MT relationship
MR1	$sin\ (x) = sin\ (x + 2\pi)$
MR2	$sin\ (x) = -sin\ (x + \pi)$
MR3	$sin\ (x) = -sin\ (-x)$
MR4	$sin\ (x) = sin\ (\pi - x)$
MR5	$sin\ (x) = sin\ (x + 4\pi)$

Compared to NVP, MT can directly test an individual program without the need to compare to other independently developed programs. Also, test cases can possibly be drawn from the entire input space, if there is no special restriction placed on the MRs. Not all MRs have the same effectiveness to reveal failures in a program. Recent empirical evidence suggests that a small number of MRs may be sufficient to create an effective test, given that the MRs are diverse (Liu et al. 2014). The main challenge in applying MT for automated testing includes identification and selection of effective MRs, and generation of diverse test cases based on the MRs.

Random testing

If an oracle exists or if the correctness of the output can be evaluated by techniques such as NVP or MT, one can select any input as a test case. In this case, the main challenge is to develop a mechanism to select a set of inputs to be used as test cases—the test case selection problem. The main idea of the problem is to identify the smallest set of test cases that can reveal the maximum number of faults in a program.

Arguably, the simplest method for selecting test cases is to select them randomly from the input space. This is the basic idea of random testing (RT). This approach starts by identifying the input domain, then randomly samples test cases independently from the input domain. These randomly chosen test cases are then executed by the PUT, and the results are checked by an oracle or other mechanisms (Hamlet 1994). RT is perhaps the simplest and most intuitive approach of test case selection, and it is often used as the 'reference' when investigating the performance of test case selection methods.

The advantage of this approach is that it is much easier to implement than carefully 'hand picking' special test cases. It is generally quick to generate a large number of random test cases that cover the input space widely in an automated fashion. Hook and Kelly (2009) conducted an experiment to compare the effectiveness of 105 hand-picked test cases and 1,050 random test cases from the valid input space. Surprisingly, they found that randomly picked test cases were more effective than hand-picked test cases (Gray and Kelly 2010). Their results suggest that random test case selection, especially when it is combined with some 'hand-picked' test cases, could be an effective technique for revealing failures (Gray and Kelly 2010).

RT has some limitations. Most notably, this method does not 'select' test cases per se. It simply generates test cases randomly from the input domain. This method does not use any information about the program structure, execution path, structure of the input domain, or knowledge of common faults. Therefore, it is conceivable that many "good" test cases (such as boundary conditions) are ignored. One solution suggested to overcome this issue is to use RT along with other testing methods such as special case testing and keep track of executions in branches of the program. Another limitation of this method relates to its dependency on an oracle to verify the output of program for random input. Therefore it cannot be used for testing programs in which a practical oracle does not exist.

Adaptive random testing

Adaptive random testing (ART) is a simple approach that takes advantage of the simplicity of RT, and incorporates additional information about the failure-causing input regions to minimise the number of test cases required to detect the same number of failures. The main observation is that failure-causing inputs are not randomly distributed in the input space, but are usually clustered together to form distinct failure regions (Chan et al. 1996). Chan et al. (2004) categorised failure-causing inputs into three types of patterns: block pattern, point pattern and strip pattern (Fig. 2). In block or strip patterns, all the failure-causing inputs are clustered in one or a few regions in the input space. In contrast, point pattern consists of possibly many distinct failure-causing inputs that are scattered across the whole input domain. They found that most real-life failure-causing inputs in programs form block or strip patterns, which means failure-causing inputs tend to cluster together in the input space. The implication is that non-failure regions are also contiguous; therefore, after the execution of a non-failure-causing input x_i, one should select a random test case that is the furthest away from x_i in the input space. This is the basis of ART (Chen et al. 2005, 2010).

The simplest implementation of ART, the fixed size candidate set (FSCS) approach, involves first generating a random set of candidate test cases in the input domain. At first, one test case is randomly selected for execution. If execution of this test case does not cause any failure, the candidate test case which is most different from the executed test cases is selected for the next execution, and so on. This process continues until a pre-defined number of failures are discovered or until all the input test cases have been successfully executed. ART provides a simple and rational approach to automatically generate diverse test cases.

Theoretical and empirical studies have shown that ART can be up to 50% more effective than traditional RT in terms of failure detection ability (Chen and Merkel 2008). The improved effectiveness stems from utilising the knowledge of the most likely failure-causing patterns of a program. The

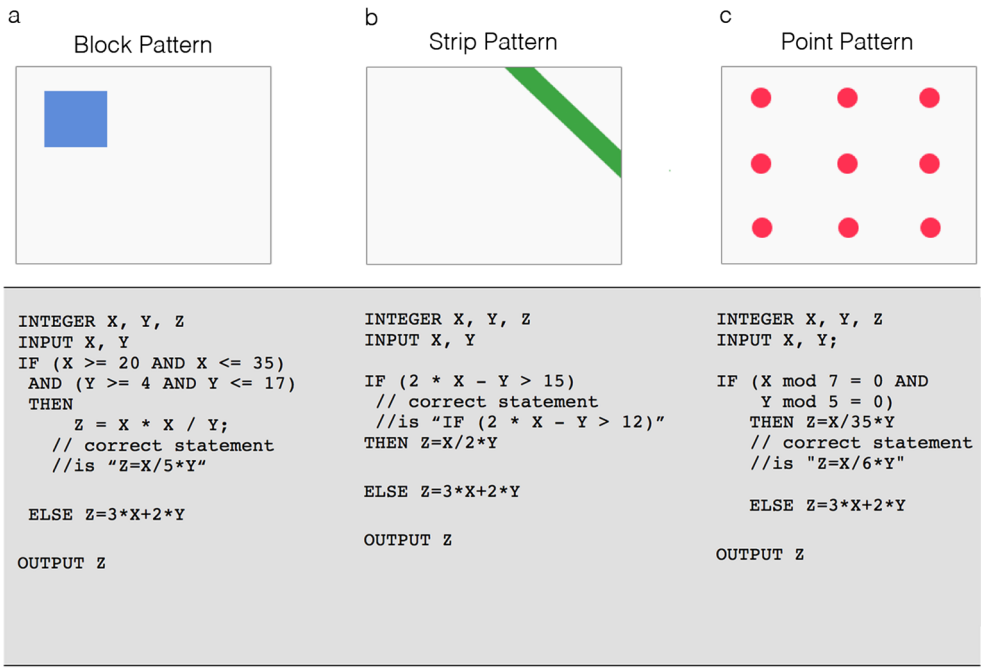

Fig. 2 Illustration of different types of failure-causing input patterns, with corresponding example source codes

additional computation involved in selecting the next test cases can be reduced by various methods, so ART remains a practical method for performing testing in real life (Chen et al. 2010). One challenge of ART is that it requires a meaningful distance metric to be defined in the input space, which may be non-trivial for programs that take non-numerical inputs.

In vivo testing: continuous software testing in an operational environment

In vivo testing has been introduced to perform software testing not only at the testing stage of the software development cycle, but also in the software deployment stage when the program is being executed in its operational environment (Chu et al. 2008; Murphy et al. 2009b). Since not all faults can be revealed using test cases in the software development stage, concurrent software testing on user input data while the software instance is running on the users' machines is an effective solution to detect more hidden faults. Similarly, in bioinformatics software, testers might not be familiar with some specific uses of the bioinformatics programs and their test cases may not identify all test faults in the software correctly.

In this method, the code for executing the test cases is embedded inside the main source code of the program. Therefore, testing is executed in parallel to the execution of the real input from the users in an independent duplicated environment (Murphy et al. 2009b). This feature allows software testers to test their program using real inputs as test cases under realistic parameters and hardware environments. In vivo testing has three main advantages. Firstly, it can detect faults that may otherwise be hard to detect in a 'clean state' in a testing environment. Secondly, in vivo testing can guarantee that the testing process will be continued even after the software is released. Finally, inputs collected from the real world scenarios have a better chance of revealing faults than randomly chosen inputs (Dai et al. 2010).

Cloud-based software testing

The cloud platform is the latest revolution in information technology which provides on-demand access to a large and scalable amount of computing and storage resources without limiting developers to specific hardware restrictions (Parveen and Tilley 2010). This feature can be beneficial to reduce execution time of testing, especially in terms of automated software testing (Riungu-Kalliosaari et al. 2012).

Cloud testing is one of the applications of cloud computing, and is poised to take software testing to the "next level" (Candea et al. 2010). Testing as a Service (TaaS) is one of the outcomes of cloud testing that is considered to provide on-demand software testing activities for given computer programs based on the cloud infrastructure (Gao et al. 2011). TaaS can be used for different purposes, such as *testing of Service as a Service (SaaS) applications*; *testing of the cloud* which provides testing to assure functionality of the cloud from an external and end-user perspective; *testing inside a cloud,* which provides testing cloud infrastructures and the integrations of different components of cloud along with management and security testing; and finally *testing applications over a cloud*, which provides a service for software developers

to test specific software applications using the highly scalable and distributed environment offered by the cloud (Gao et al. 2011).

Cloud-based testing provides several benefits compared to traditional software testing. Firstly, it enables large-scale and on-demand testing with a large number and variety of input data using many compute instances in a short period of time. This facility alleviates the need to invest in large high-performance computing infrastructure for testing, and also allows us to request different hardware or operating system environments for testing, which enables testing in a realistic situations. Moreover software testing may need a considerably large amount of disk space and storage, and an isolated environment to perform the testing (Parveen and Tilley 2010). For instance, in vivo testing may require independent disks or virtual machines to execute the program against test cases with different configurations in parallel without affecting each other's program state.

The cloud platform also provides several advantages in comparison with conventional platforms and systems. Firstly, it provides an on-demand and online access for users, which is cost-effective and users will not be charged when they are not using the resource (Buyya et al. 2009). Secondly, the cloud can be accessible from anywhere and enables collaboration between developers and users from different locations. Finally, the cloud is adaptive and scalable; this means it can provide scalable hardware resources for different tasks that can be helpful to reduce the cost of information technology (Leavitt 2009). We anticipate that the next generation of software testing using TaaS will become more popular as it provides easier and more comprehensive software testing for software developers.

Besides all of the benefits of the cloud, the scalability feature itself in the cloud can pose a great challenge and an underestimate of the scale ratio could lead to heavy costs. This issue could become worse if the scalability ratio for scaling up or reducing hardware resources is incorrectly estimated with an automated algorithm. Latency of network data transfer is another issue that reduces the transfer and access speed during tests. This is due to the nature of remote existence of cloud. Latency becomes more important when the testing environment or task depends on another system from a different region or outside of cloud (Leavitt 2009).

Mutation analysis: evaluation of the effectiveness of software testing methodologies

Mutation analysis was introduced to quantify the effectiveness of testing methods (Hamlet 1977). The main idea of mutation analysis is the generation of *mutants* by injecting artificial faults into the program's source code, which can be compiled and executed. Mutants are generated using simple syntactic rules, known as *mutation operators*. It is important to check that the mutant can generate different outputs compared to the original program when given the same inputs. If the mutant and the original program produce the same outputs given the same inputs, this mutant is considered as an equivalent mutant. An equivalent mutant may arise due to having the seeded fault in a section of the program that cannot be reached by the execution path. The following discussion involves the analysis of non-equivalent mutants. In general, a non-equivalent mutant should satisfy three characteristics: *reachability*, *necessity* and *sufficiency* (DeMilli and Offutt 1991). Reachability means the mutated part of the program should be accessible in the program flow. Necessity means the mutated part of the program should produce a different internal state compared to the original version. Finally, sufficiency means that the error should be propagated to the program output.

Once a set of non-equivalent mutants is generated, a set of test cases, generated by a testing methodology, is applied to test the mutants. We can then determine how many test cases reveal a fault in the mutants. A mutant that is identified by a test method to contain a fault is called a *killed* mutant, whereas a mutant that is not detected to contain a fault is called an *alive* mutant. The proportion of the killed mutants to all non-equivalent mutations is called the mutation score. The process of generating mutants can also be either manual or automated (Jia and Harman 2011), and the process of testing mutants can be automated.

Applications of software testing in bioinformatics

During our review of the bioinformatics literature, we only found a few reports that attempt to adopt state-of-the-art software testing methods to verify or validate bioinformatics software, including reports from the authors of this review. Here we summarise some of their results.

Biological network simulators

Bergmann and Sauro (2008) performed a comparison of twelve biological network simulators that are compatible with Systems Biology Modeling Language (SBML). In their study, they simulated the same 150 curated SBML models from the BioModels database using the twelve simulators and compared their results. Their approach is akin to *N*-version programming. They showed that only six packages could return a result for all their models (all other packages failed to simulate some of the models). They also observed that among all the simulators, only two of them had complete agreement with each other across all models. In a separate study, Evans et al. (2008) developed a test suite for testing stochastic simulators.

Their approach is essentially special test case testing—they evaluated the simulation output from multiple executions, and checked that the outputs from these distributions fell within the expected range of values. They showed that this test suite could be very helpful for simulator developers to test the correctness of their implementations. Chen et al. (2009) used MT to test a gene network simulator, GNLab. They identified ten MRs for this program. In this study, they found violation of one MR, which specified that adding a new edge with zero weight should not affect the simulation results. It turned out that this problem is due to a mis-specification of the algorithm.

Sequence alignment programs

Short read sequence alignment programs are popular software programs in bioinformatics, and are widely used to analyse next-generation sequencing data. Popular alignment programs such as BWA (Li and Durbin 2009), BOWTIE and BOWTIE2 (Langmead and Salzberg,2012) are widely used among bioinformaticians. These programs have been tested and evaluated using many reference test data by the developers. In a recent study by Giannoulatou et al. (2014) these sequence aligner programs were tested using MT. In their approach, they identified nine MRs. As an example, one of the MRs stated that random permutation of the input should not affect the alignment results. Surprisingly, this is one of the MRs that was violated by one of the aligners. This result is unexpected since the order of the input data is not supposed to affect alignment results. None of the tested aligners satisfied all nine MRs. This result further supports the importance of testing bioinformatics programs, especially these widely used programs that have a potential to be used in a translational clinical setting in genomic medicine. The usefulness of metamorphic testing for such a type of software was clearly demonstrated.

Concluding remarks

In this paper, we discussed the needs of proper software testing in bioinformatics. The main problem is related to the amount of data and complexity of algorithms in bioinformatics software, which makes it hard to verify the output data and to select many diverse test cases. We have also reviewed several popular and state-of-the-art software testing techniques, and discussed their applications. The key concepts illustrated by these methods include multiple executions of the same program or related programs, using diverse test cases in the input space, testing after deployment, and enabling scalable and parallelised testing using cloud technology. It is important to mention that there are many other software testing techniques (Beizer 1990; Myers et al. 2011), but our main focus of this review was to discuss those techniques that have been used or are suitable for testing bioinformatics programs. Further research is required to quantify and compare the effectiveness of different methodologies, and make software testing much more systematic and automatable. We believe additional testing activities will improve the reliability of bioinformatics software, and therefore the reliability of scientific research results.

Compliance with ethical standards

Funding This work was supported in part by funds from the New South Wales Ministry of Health, a New South Wales Genomics Collaborative Grant, an Australian Research Council Grant, and a Microsoft Azure Research Award.

Conflict of interests All authors (AHK, EG, TYC, MAC, ALME and JWKH) declare that they do not have any conflict of interest.

Ethical approval This article does not contain any studies with human or animal subjects performed by any of the authors.

References

Alden K, Read M (2013) Computing: Scientific software needs quality control. Nature 502:448

Baxter SM, Day SW, Fetrow JS, Reisinger SJ (2006) Scientific software development is not an oxymoron. PLoS Comput Biol 2, e87

Beizer B (1990) Software testing techniques. Van Nostrand Reinhold, New York

Bergmann FT, Sauro HM (2008) Comparing simulation results of SBML capable simulators. Bioinformatics 24:1963–1965

Buyya R, Yeo CS, Venugopal S, Broberg J, Brandic I (2009) Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. Futur Gener Comput Syst 25:599–616

Candea G, Bucur S, Zamfir C (2010) Automated software testing as a service. In: Proceedings of the 1st ACM Symposium on Cloud Computing. ACM, Indianapolis, Indiana, USA, pp 155–160

Chan FT, Chen TY, Mak IK, Yu YT (1996) Proportional sampling strategy: guidelines for software testing practitioners. Inf Softw Technol 38:775–782

Chan WK, Cheung S, Leung KR (2007) A metamorphic testing approach for online testing of service-oriented software applications. Int J Web Serv Res 4:61–81

Check Hayden E (2013) Mozilla plan seeks to debug scientific code. Nature 501:472

Check Hayden E (2015) Journal buoys code-review push. Nature 520: 276–277

Chen L, Avizienis A (1978) N-Version Programming: A fault-tolerance approach to reliability of software operation. In: Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8), Toulouse, France, June 1978, pp 3–9

Chen TY, Merkel R (2008) An upper bound on software testing effectiveness. ACM Trans Softw Eng Methodol 17:16

Chen TY, Cheung SC, Yiu S (1998) Metamorphic testing: a new approach for generating next test cases. Technical Report HKUST-CS98-01, Dept. of Computer Science, Hong Kong University of Science and Technology

Chen TY, Tse T, Zhou ZQ (2003) Fault-based testing without the need of oracles. Inf Softw Technol 45:1–9

Chen TY, Merkel RG, Eddy G, Wong P (2004) Adaptive Random Testing Through Dynamic Partitioning. In: Proc Fourth Int Conf Quality Software, Braunschweig, Germany, Sept 2004, pp 79–86

Chen TY, Leung H, Mak I (2005) Adaptive random testing. In Proceedings of the 9th Asian Computing Science Conference: Higher-Level Decision Making (ASIAN 2004), Springer-Verlag, Berlin Heidelberg, pp 320–329

Chen TY, Ho JW, Liu H, Xie X (2009) An innovative approach for testing bioinformatics programs using metamorphic testing. BMC Bioinformatics 10, 24.
This paper provides the first case study of using metamorphic testing to test bioinformatics programs.

Chen TY, Kuo F-C, Merkel RG, Tse T (2010) Adaptive random testing: The ART of test case diversity. J Syst Softw 83:60–66

Chu M, Murphy C, Kaiser G (2008) Distributed in vivo testing of software applications. In: Proc 1st Int Conf Software Testing, Verification, and Validation, Lillehammer, Norway, 2008, pp 509–512

Dai H, Murphy C, Kaiser G (2010) Confu: Configuration fuzzing testing framework for software vulnerability detection. Int J Secure Softw Eng 1:41–55

DeMilli R, Offutt AJ (1991) Constraint-based automatic test data generation. IEEE Trans Softw Eng 17:900–910

Evans TW, Gillespie CS, Wilkinson DJ (2008) The SBML discrete stochastic models test suite. Bioinformatics 24:285–286

Gao J, Bai X, Tsai W-T (2011) Cloud testing-issues, challenges, needs and practice. Software Engineering: An International Journal 1:9–23

Giannoulatou E, Park S-H, Humphreys DT, Ho JW (2014) Verification and validation of bioinformatics software without a gold standard: a case study of BWA and Bowtie. BMC Bioinformatics 15:S15

Gray R, Kelly D (2010) Investigating test selection techniques for scientific software using Hook's mutation sensitivity testing. Procedia Comput Sci 1:1487–1494

Hamlet RG (1977) Testing programs with the aid of a compiler. IEEE Trans Softw Eng SE-3:279–290

Hamlet R. (1994) Random testing. Encyclopedia of Software Engineering. Wiley, New York, pp 970–978

Hannay JE, MacLeod C, Singer J, Langtangen HP, Pfahl D, Wilson G (2009) How do scientists develop and use scientific software? In: Proceedings of Workshop on Software Engineering for Computational Science and Engineering., pp 1–8

Hook D, Kelly D (2009) Mutation sensitivity testing. Comput Sci Eng 11: 40–47

Howden WE (1976) Reliability of the path analysis testing strategy. IEEE Trans Softw Eng SE-2:208–215

IEEE (1990) IEEE Standard Glossary of Software Engineering Terminology. IEEE Std 610.12-1990 1–84.

IEEE (2013). Software and systems engineering — Software testing — Part 1: Concepts and definitions. ISO/IEC/IEEE 29119 1–84.

ISTQB I (2015) Glossary of Testing Terms. ISTQB Glossary http://www.istqb.org/downloads/finish/20/193.html.

Jia Y, Harman M (2011) An analysis and survey of the development of mutation testing. IEEE Trans Softw Eng 37:649–678

Joppa LN, McInerny G, Harper R, Salido L, Takeda K, O'Hara K, Gavaghan D, Emmott S (2013) Troubling trends in scientific software use. Science 340:814–815

Knight JC, Leveson NG (1986) An experimental evaluation of the assumption of independence in multiversion programming. IEEE Trans Softw Eng 12:96–109

Kuo F-C, Chen TY, Tam WK (2011) Testing embedded software by metamorphic testing: A wireless metering system case study. In: 36th IEEE Conference on Local Computer Networks (LCN), Bonn, Germany, 2011, pp 291–294

Langmead B, Salzberg SL (2012) Fast gapped-read alignment with Bowtie 2. Nat Methods 9:357–359

Lanubile F, Shull F, Basili VR (1998) Experimenting with error abstraction in requirements documents. In: Proceedings of Fifth International Software Metrics Symposium, Bethesda, Maryland, 2008, pp 114–121

Leavitt N (2009) Is Cloud Computing Really Ready for Prime Time? Computer 42:15–20

Li H, Durbin R (2009) Fast and accurate short read alignment with Burrows–Wheeler transform. Bioinformatics 25:1754–1760

Liu H, Kuo F-C, Towey D, Chen TY (2014) How effectively does metamorphic testing alleviate the oracle problem? IEEE Trans Softw Eng 40:4–22

McCarthy DJ, Humburg P, Kanapin A, Rivas MA, Gaulton K, Cazier J-B, Donnelly P (2014) Choice of transcripts and software has a large effect on variant annotation. Genome Med 6:26

Merali Z (2010) Computational science: Error, why scientific programming does not compute. Nature 467:775–777

Murphy C, Kaiser GE, Hu L (2008) Properties of machine learning applications for use in metamorphic testing. In: Proceedings of the 20th International Conference on Software Engineering and Knowledge Engineering (SEKE, San Francisco, CA, pp 867–872

Murphy C, Shen K, Kaiser G (2009a) Automatic system testing of programs without test oracles. In: Proceedings of the eighteenth international symposium on Software testing and analysis, Chicago, IL, 2009, pp 189–200

Murphy C, Kaiser G, Vo I, Chu M (2009b) Quality assurance of software applications using the in vivo testing approach. In: Proceedings of International Conference on Software Testing Verification and Validation, Denver CO, 2009, pp 111–120

Myers GJ, Sandler C, Badgett T (2011) The art of software testing. Wiley, New York

O'Rawe J, Jiang T, Sun G, Wu Y, Wang W, Hu J, Bodily P, Tian L, Hakonarson H, Johnson WE (2013) Low concordance of multiple variant-calling pipelines: practical implications for exome and genome sequencing. Genome Med 5:28.
This paper shows there is a low concordance among five widely used variant calling pipelines, suggesting the importance of improving the quality of these pipelines

Otero C, Peter A (2015) Research Directions for Engineering Big Data Analytics Software. IEEE Intell Syst 30:13–19

Parveen T, Tilley S (2010) When to migrate software testing to the cloud? In: Proceedings of Third International Conference on Software Testing, Verification, and Validation Workshops (ICSTW), Washington, DC, 2010. pp 424–427

Riungu-Kalliosaari L, Taipale O, Smolander K (2012) Testing in the cloud: Exploring the practice. IEEE Softw 29:46–51

Segura S, Hierons RM, Benavides D, Ruiz-Cortés A (2010) Automated test data generation on the analyses of feature models: A metamorphic testing approach. In: Proceedings of the Third International Conference on Software Testing, Verification and Validation (ICST), Washington, DC, 2010, pp 35–44

Weyuker EJ (1982) On testing non-testable programs. Comput J 25:465–470

Xie X, Ho JW, Murphy C, Kaiser G, Xu B, Chen TY (2011) Testing and validating machine learning classifiers by metamorphic testing. J Syst Softw 84:544–558

Zhou ZQ, Huang D, Tse T, Yang Z, Huang H, Chen T (2004) Metamorphic testing and its applications. In: Proceedings of the 8th International Symposium on Future Software Technology (ISFST 2004), Xi'An, China, 2004, pp 346–351