Using Random Search to Identify Fault Characteristics in Software Applications

Matthew Patrick and Manuel Oriol

University of York, UK
{matthew.patrick,manuel.oriol}@cs.york.ac.uk
http://www.scs.york.ac.uk

Abstract. Search-based testing can be difficult for commercial applications of software because of the many potential points of failure. It is useful to have heuristics in order to guide the search. However, the majority of research on test data generation techniques focuses on small test programs that are not representative of the larger systems often used in industry. This research uses random testing to reveal run-time errors in commonly used open source software. The aim is to determine whether the fault characteristics of software differ according to its application and by doing so provide some insight into the problem of testing software applications in general.

Keywords: Random testing; Fault characteristics; YETI

1 Introduction

Random search is one of the most straightforward and inexpensive testing strategies [1]. It is sometimes seen as inferior to other strategies because it does not take into account the syntactic or semantic structure of a program [2]. However, because it covers the whole input-space, it is useful for providing confidence in the general characteristics of software [3]. Its low overhead also means that random search can reveal more faults per unit of time than an exhaustive strategy [4]. Successful automation of random testing requires an oracle that does not need human involvement to verify the results of executing each test case [1]. Unfortunately, it might be just as difficult to produce an oracle that works correctly as to develop the software without any bugs.

The York Extendible Testing Infrastructure (YETI) provides a framework for executing random testing sessions without the need for an explicit oracle. The oracle is implicit in the occurrence of assertion violations and run-time errors, for example an incorrect casting or division by zero. As YETI is programming language independent, it can be used to test software written in a number of different programming languages. However, for this research it will be used to test software written in Java. YETI is able to produce a significant amount of test data in a short space of time. In Java, it is able to make up to 10^6 calls per minute [6]. Therefore, it is very useful for providing thorough fault characteristics for the software under test. This research uses YETI to assess software used for

different purposes. The results are then analysed to provide some insight into how each application area may best be tested.

2 Categories of Software

The aim in defining these categories of software is to work towards a benchmark from which the practices of testing each category can be improved. Software categories can be helpful if they reveal unique challenges that are faced testing examples of their software. Many of the tools developed in research are evaluated using artificial example programs that are too small-scale to be representative of any particular category. Yet in order for a tool to be relevant for industry, it must be able to perform well with much more complicated software. Therefore, we will consider how software that is used in industry may be categorised for the purposes of testing. Considerations can be made for size, the number of modules or complexity of data structures, and the way it is developed, whether the software is open source or object orientated. These categories can be useful, but they focus on the details and structure of the code rather than the purpose of the software.

At the other end of the spectrum lie categorisations of domains in which software is intended to be used. A good example can be found in the Microsoft Asset Inventory Service [5]. Although its categories are clearly defined by purpose, their complexity makes them difficult to distinguish from the perspective of testing. For example, the *Education and Reference* and *Home and Entertainment* categories both involve strong elements of user interaction that must be tested. We propose to find a compromise by considering five distinct forms of behaviour that software can exhibit: functional, open, progressive, user and timed. It is likely that some software will exhibit more than one form of behaviour. For example, a web-based application that includes a database will act progressively, but also feature open and user-type behaviour. Therefore, these categories do not represent the complete behaviour of a software system, but they do offer an insight into the types of behaviour that can be tested in software.

Functional behaviour involves significant numerical calculation and/or algorithmic operations. Examples range from core data structures such as *java.Integer*, up to complex libraries such as the *Bouncy Castle* cryptography suite. This is the kind of behaviour that is often targeted in traditional testing research.

Open behaviour makes use of features that are shared between multiple parties, with potentially different goals. This involves danger from attackers outside the system and the possibility of conflict between different modules inside the system. It is important that the software possesses fault tolerance and is able to resist tampering.

Progressive behaviour maintains and develops a data source over time. When a progressive system starts with a data source in a particular state, it must leave it in a legal state. This can be difficult, as the data source may potentially be infinite, or only bounded by the available memory. Typically, this is addressed by dividing the data source into manageable sections, for example a byte stream.

User behaviour may be command-line, graphical or even web-based. Run-time errors can occur when an unsolicited GUI event is triggered. It is difficult to test the many paths through software featuring user behaviour because of the many ways that the components of its interface may be manipulated. The biggest challenge is being able to cover a sufficient proportion of the search space to achieve a reasonable degree of confidence in its correctness. This is made even more difficult when we consider there is a human element involved, raising the level of unpredictability.
Timed behaviour includes a real-time performance requirement and typically some form of concurrency. Failures in concurrent systems may result in unexpected deadlock and race conditions. These can be difficult to avoid and expensive to detect because they occur as an interaction between classes.

3 Experimental Method

To determine the software fault characteristics, it is necessary to decide which classes to analyse within YETI. A thorough evaluation requires a large number of classes. However, for this initial research, we will only consider one. The challenge is to choose a class that represents the software well. If it is too specific or too general, we will not gain a good understanding of the application. Figure 1 shows how the largest class will be selected (with some restriction) as it is likely to have a significant role in the software.

This research considers five categories of software application. *Functional* systems involve significant numerical calculation and/or algorithmic operations. *Open* systems have features that are shared between multiple parties, with potentially different goals. *Progressive* systems are used to maintain and develop a data source over time. *User* systems may be command-line, graphical or even web-based. *Timed* systems include a real-time performance requirement and typically some form of concurrency.

4 Experiments

Hundreds of thousands of random test cases were applied for every class to provide a basic analysis of the fault characteristics of each application. YETI was used to test each class for ten minutes and record any relevant failures along with the time they occurred. As this required a considerable amount of memory, it was necessary to extend YETI's working memory allowance up to 500 Megabytes. A relevant failure is a failure that has not been seen before. Although the software may fail many times, we are more interested in the relevant failures because they give a better indication of the software's fault characteristics.

Functional

Functional systems range from core data stuctures such as *java.Integer*, up to complex libraries such as the *Bouncy Castle* cryptography suite. We considered

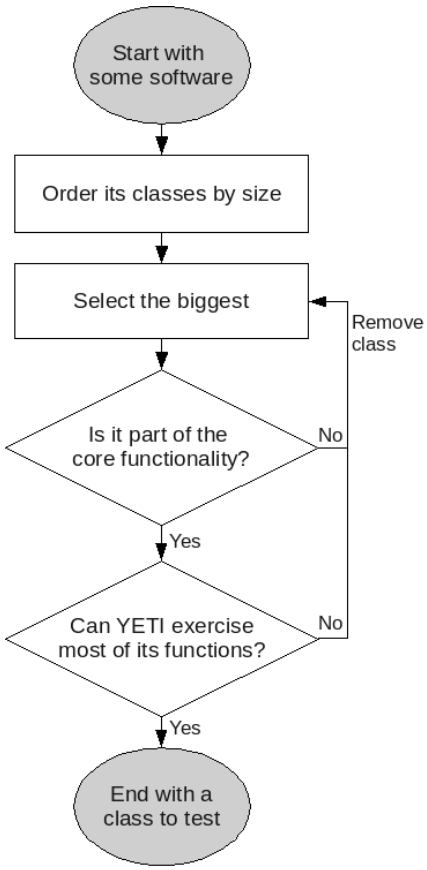

Fig. 1. Method to select which class to test

JBullet, a full 3D physics environment for Java. JBullet has to process numbers quickly and efficiently, whilst at the same time provide accurate calculations. Figure 2 shows only two relevant failures were raised by the class, both very early on in the evaluation. This suggests the biggest danger with this type of software is not the occurrence of run-time errors, but rather incorrect calculations or delays that affect the worst case execution time for particular inputs.

Open

Open systems face dangers from attackers outside the system and the possibility of conflict between different modules inside the system. It is important that the software possesses fault tolerance and is able to resist tampering. Therefore, it is a surprise to see 19 run-time errors when analysing the *jigsaw.ssi.SSIFrame* class of the Jigsaw web server (see figure 3). These errors might be masked with the interaction between classes, but they may still cause problems if left unseen.

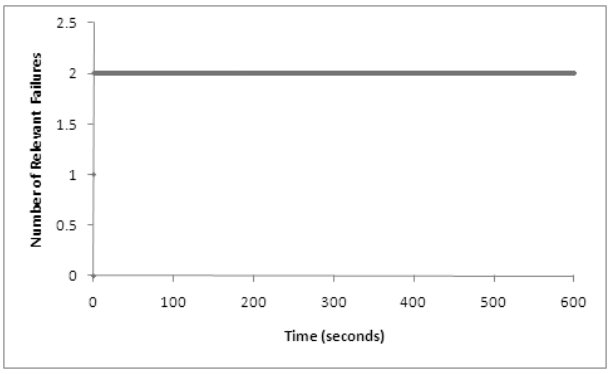

Fig. 2. Functional System Faults

Open systems are particularly prone to problems when there is a heterogeneity of languages, platforms and architectures. This makes it impractical to show software is free from errors. Therefore, it is difficult to make accurate predictions as to the reliability of an open system.

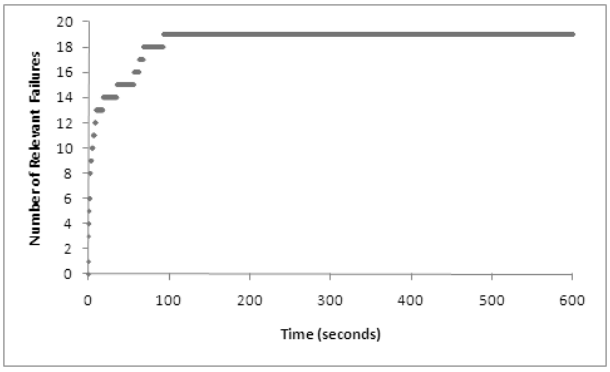

Fig. 3. Open System Faults

Progressive

When a progressive system starts with a data source in a particular state, it must leave it in a legal state. This can be difficult, as the data source may potentially be infinite, or only bounded by the available memory. Typically, this is addressed by dividing the data source into manageable sections, for example a byte stream. We investigated the *Apache Derby* database manager and chose the class *org.apache.derby.impl.sql.SQLParser* to investigate. Figure 4 shows that 47

relevant failures were found in two stages. The first 40 errors were found very quickly, but it took a while longer before the remaining seven could be found. This suggests that when testing progressive systems, it is important not just to consider a few isolated examples, but to make a thorough investigation using complete case models.

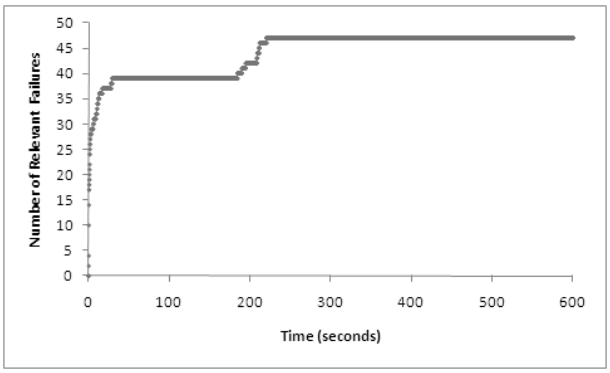

Fig. 4. Progressive System Faults

User

Out of all our experiments, the *javax.swing.JTree* class from Swing generated the most run-time errors. Run-time errors may occur in user systems when an unsolicited GUI event occurs. It is difficult to test the many paths through a user system because of the many ways that interface components can be manipulated. This may be the reason for the steady increase in the number of failures throughout evaluation (see figure 5). More failures may even have been found if the evaluation was left to run. The biggest challenge in testing this kind of system is being able to cover a sufficient proportion of the search space. This is made even more difficult when we consider there is a human element involved, raising the level of unpredictability.

Timed

Finally, we investigated the *javolution.util.FastMap* class of *Javolution*, a real-time library for Java. Failures in timed systems may result in unexpected deadlock and race conditions. These can be difficult to avoid and expensive to detect because they occur as an interaction between classes. Although we were able to find some run-time errors (see figure 6, they might not be representative of those that typically occur in timed systems. The errors we found may be common to any library. Therefore, timed systems on the whole can be unpredictable and difficult to test.

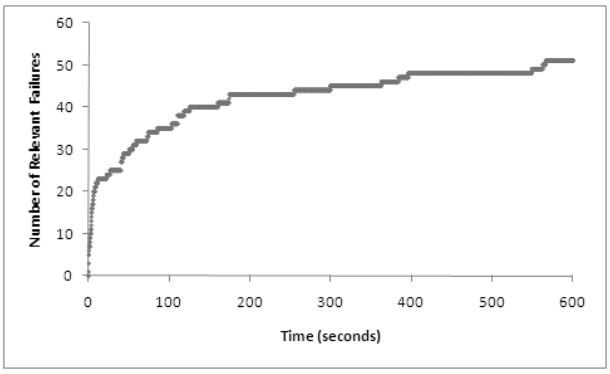

Fig. 5. User System Faults

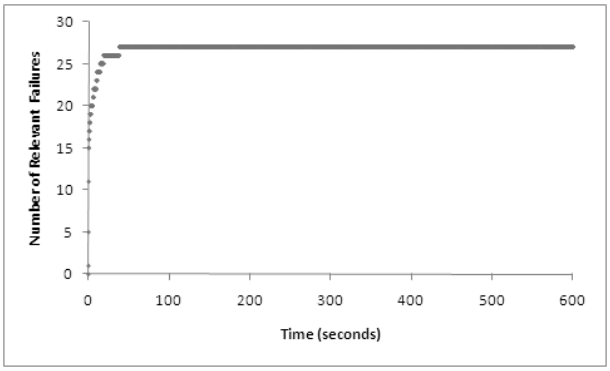

Fig. 6. Timed System Faults

5 Limitations and Future Work

We have already mentioned the difficulties involved in finding classes to represent the software under analysis. Some classes are overly specialised and others too generalised to be relevant for this purpose. If a class involves considerable interaction with other classes (for example in a timed system), the inter-class dependencies may be too great for the isolated results from one class to be useful on their own. If this technique is to be used in determining relevant fault characteristics for such software, it will be necessary to adapt it to involve the analysis of more than one class. There is already some support for this in YETI. The next step is to design a process by which a subset of classes may be chosen and the way in which they are to interact decided.

As YETI does not use an explicit oracle, it does not have any real insight as to whether software performs the correct operations. This may be a problem for functional systems, which rely heavily on the notion of correctness. If this is a concern, it should be possible to make use of bug repositories to analyse which

particular errors have occurred in the past. However, these may be subjective and time-consuming. One of the main reason for using YETI is that it is able to analyse software for which there is no oracle or for which the specification is inaccurate.

6 Conclusions

In conclusion, this technique has been able to identify an interesting characteristic about each category of software. At present it appears unsuitable for certain categories (i.e. functional and timed). However, this may be corrected by including more classes in the analysis. Although random testing may struggle to find a particular failure that occurs only for a small subset of the input domain, it can run many test cases quickly and should be able find most of the failures with relatively little computational expense. The biggest challenge for the future will be to determine how to choose which combinations of classes should be exercised together for optimum results. If this can be achieved, random search could be an excellent choice for identifying the failure characteristics of software.

References

1. Ince, D., The automatic generation of test data. The Computer Journal. 30:1, 63–69 (1987)
2. Myers, G., Badgett, T., Thomas, T. and Sandler, C., The Art of Software Testing, Wiley, New York (2004)
3. Hamlet, R., Maciniak, J., Random Testing, Encyclopedia of Software Engineering, Wiley, New York, 970–978 (1994)
4. Mankefors, S., Torkar, R. and Boklund, A., New quality estimations in random testing, Proceedings of the 14th International Symposium on Software Reliability Engineering, 468–478 (2003)
5. Microsoft Corporation, Software Categories, Microsoft TechNet, http://technet.microsoft.com/en-us/library/bb852143.aspx (2010)
6. Oriol, M., Testing .NET Code with YETI, Proceedings of the 15th IEEE International Conference on Engineering of Complex Computer Systems (2010)