PEST: Programs to Evaluate Software Testing Tools and Techniques

James R. Lyle
National Institute of Standards and Technology
100 Bureau Drive Stop 8970
Gaithersburg, MD 20899-8970
(301) 975-3270
jlyle@nist.gov

Mary T. Laamanen
National Institute of Standards and Technology
100 Bureau Drive Stop 8970
Gaithersburg, MD 20899-8970
(301) 975-3260
mary.laamanen@nist.gov

Neva M. Carlson
National Institute of Standards and Technology
100 Bureau Drive Stop 8970
Gaithersburg, MD 20899-8970
(301) 975-3296
neva.carlson@nist.gov

Abstract

PEST is a collection of reference materials for the empirical evaluation and comparison of software testing techniques. Often the publication of a new testing technique or strategy includes a theoretical analysis and an ad hoc empirical evaluation. Because each researcher usually uses a different set of programs for an empirical evaluation, there is little basis for comparison between different techniques.

The project objective is to develop and make available to software testing researchers and tool vendors a set of reference materials for the empirical evaluation and comparison of software testing techniques. This set of reference materials includes a diverse suite of program modules that can be the subject of a testing technique, testing support tools and examples of using PEST.

Each module contains a program specification, a correct implementation in C that can be used as an oracle and several faulty C implementations, each seeded with a single fault from a commonly available fault taxonomy. The programs are designed such that a common test harness can be used to execute each faulty variant over test data for comparison against the oracle. This allows for the computation of metrics to compare the relative effectiveness of test data generated by different techniques.

Keywords

software testing, software metrics

1 Introduction

PEST is a collection of reference materials for the empirical evaluation and comparison of software testing techniques. Often the publication of a new testing technique or strategy includes a theoretical analysis and an ad hoc empirical evaluation. Because each researcher usually uses a different set of programs for an empirical evaluation, there is little basis for comparison between different techniques. A common suite of faulty programs would remedy this problem.

The project objective is to develop and make available to software testing researchers and tool vendors a set of reference materials for the empirical evaluation and comparison of software testing techniques. This set of reference materials includes a diverse suite of program modules that can be the subject of a testing technique, testing support tools and examples of using PEST. The PEST materials can be used to evaluate and compare both white box and black box testing techniques. In addition, the PEST materials can be used by testing tool vendors and as a supplement to class materials in academic courses or a professional training environment.

This short paper describes the PEST reference materials. The heart of the collection is a suite of modules, corresponding to a single programming project. Each module contains a program specification, a correct (at least we do our best to make it so) implementation in C that can be used as an oracle and several faulty C implementations, each seeded with a single fault from a commonly available fault taxonomy[1]. The programs are designed such that a common test harness can be used to execute each faulty variant over test data for comparison against the oracle. This allows statistics to be collected for the computation of metrics to compare the relative effectiveness of test data generated by different techniques.

PEST currently contains materials created at NIST; however we expect to expand the collection with contributions from the testing community. Contributions can be in many forms, such as modules in other languages (Java, C++), ex-

perience reports, alternate fault taxonomies, or descriptions of metrics that can be used to compare testing techniques.

This paper also includes examples of how the PEST materials can be used to compare testing techniques, investigate a given technique or be used in teaching about testing.

2 Reference Materials

The PEST reference materials are divided into four categories:

1 Testable modules
2 Support Tools
3 Usage examples
4 Miscellaneous items

2.1 Modules

Each testable module is a simple programming project located in its own directory. Each directory contains a specification file (either plain text, Word or LaTeXformat) describing what the program should do, a `main` control program file and several faulty program version files. The control program, oracle, and the faulty versions are structured so that it is easy to verify if the input data has caused the fault to be revealed. The control program calls a `verify` function to compare the results of the oracle with the faulty version. The control program has the following structure:

```
get_input(...){...}
oracle(...){...}
verify(...){...}
main(...){...
get_input(...);
bug(...);
oracle(...);
return verify(...);}
```

Each faulty version is a different implementation of the `bug` function in a separate source file. The file names are keyed to the category of the fault from Beizer's fault taxonomy. This makes it possible to characterize the effectiveness of test cases against particular fault classes.

To execute a set of test cases on a particular faulty variant, the variant is compiled and then linked with the control program. The control program begins execution through the `main` procedure. The `main` procedure gets the test data, calls both the faulty and the correct versions and returns the results of a comparison to the invoking environment for tabulation. For some modules, the oracle can be eliminated if the `verify` function can determine the correctness of the computation from the input and output. An example of this might be a numerical analysis routine that can be checked by substitution of results into an inverse function that returns the original input.

Modules are being developed from a variety of application areas, including system utility software, text processing, simulation, financial and games.

2.2 Fault Taxonomy

PEST uses an adapted version of Beizer's fault taxonomy[1] as a guide for creating faulty variants. Beizer's taxonomy is more than just *bugs* and *faults*, it is a detailed classification scheme for *why is there a problem with this program*. This includes everything from misunderstanding requirements to using the wrong hardware for the test. This is too general as a guide for inserting faults. We use items from only four of Beizer's nine categories. We ignore *requirements, standards, system architecture, test execution and other* categories but use *functionality, structural, data and integration* as a source for fault insertion.

2.3 Support Tools

PEST currently has several support tools: a test harness that can be used to run and tally results for multiple data sets over all the variants of a module, tailored random number generators and a module browser.

The PEST browser is a graphical user interface (GUI) tool written as a Java Swing application for browsing the contents of the database. The tool allows the user to view the files stored in PEST by navigating through the particular bug taxonomy. The mutant program files are categorized according to bug classes contained in the chosen taxonomy. If a specific program is selected then the user is able to simultaneously compare code between the oracle and mutant version. Changes between the two programs are marked by color highlights.

3 Examples

This section describes some example uses of PEST to compare testing techniques and teach about testing. While both the PEST module selected and the testing techniques used are trivial, the examples demonstrate how more elaborate PEST modules can be used to compare more sophisticated testing techniques.

3.1 Triangle Module

This example could be used with students to illustrate basic testing principles. The testing task is to try several testing strategies to the specification of the classic triangle classification program adapted from Myers[2].

> The procedure is given a character string containing three integers separated by spaces, tabs or one sign character (plus or minus) optionally preceded by spaces or tabs. An integer contains

the digits 0-9 with an optional plus (+) or minus (-) sign. The procedure returns an error code if the input is not three integers. The three values are interpreted as representing the lengths of the sides of a triangle. The procedure returns a code indicating whether the triangle is scalene, isosceles, equilateral or not a triangle.

The triangle module has 17 faulty variants over 9 fault classes as in the following table:

Faults in Triangle		
Code	Description	N
231	Missing case	4
232	Extra case	1
3128	Control flow predicate	1
3141	Loop initial value	2
3142	Loop terminal value	1
3143	Loop increment	2
32221	Expression operator	4
32222	Expression parentheses	1
32223	Expression sign	1

Code and Description are the fault category from Beizer's taxonomy, N is the number of faulty versions of the triangle program created for the fault category.

3.2 Testing with Random Numbers

The first strategy investigated used 250 random triples uniformly distributed from 0 up to 100. The first 35 triples triggered faults in 8 of the 17 faulty versions. After 145 triples, two more faults were found. No more faults were found by the remaining triples.

After examining the random data it was observed that most triples were not valid triangles. This should have been expected if the probability of generating valid triangles had been considered. For example, the chance of a random equilateral triangle is about 1 in 10,000. To generate data with more valid triangles a second strategy was tried. The random distribution was reduced in range to from 0 up to 10. We should expect about 2 equilateral triangles from this distribution and we actually got 3.

The results of the second strategy were a little better, we found more faults with fewer test cases: 11 faults after 34 random triples.

3.3 Testing with Random Triangles

A third strategy of generating random triangles from each class rather than triples of random numbers was tried. First, a class was selected with equal probability from equilateral, isosceles, scalene or not a triangle, then random side lengths of up to 100 were generated in the required relationship.

This strategy found 11 faults after 10 random triangles, another reduction in effort.

The reduction of maximum side length in the second strategy had been helpful, so the final strategy was to reduce the side length to 10 for the random triangles. This strategy found one more fault, but 67 triangles were needed to detect the last fault.

3.4 Comparison to Myers

We also generated test data based on Myers as a comparison. The following table gives the results for each faulty version against each testing strategy. R100 and R10 are the random number triples, T100 and T10 are the random triangles and M is the data generated from Myers. Except for Myers, 50 data sets were generated and tried. The value in the table indicates how many data sets triggered the fault. There was one Myers data set that contained 29 triples.

Fault Class vs Strategy					
Code	R100	R10	T100	T10	M
231 v1	0	44	50	50	1
231 v2	44	50	50	47	1
231 v3	49	49	50	49	1
231 v4	50	50	50	50	1
232 v1	0	0	0	36	1
3128 v1	22	48	50	46	1
3141 v1	0	0	0	0	1
3141 v2	0	0	0	0	1
3142 v1	0	0	0	0	1
3143 v1	0	0	0	0	1
3143 v2	0	0	0	0	0
32221 v1	50	50	50	50	1
32221 v2	50	49	49	50	1
32221 v3	49	50	50	50	1
32221 v4	49	50	50	50	1
32222 v1	22	47	50	47	1
32223 v1	50	50	50	50	1

The faults in version 231v1 and 232v1 show a fundamental weakness of random testing. A fault may require a relationship among several data components that is unlikely to be generated at random. An equilateral triangle is required to trigger 231v1. This had a chance of about 1 in 10,000 for R100. The fault in version 232v1 required a right triangle to trigger, another low likelihood event.

3.5 Summary

The triangle module is useful for demonstrating how PEST can be used to investigate testing techniques, but is not very useful for an actual investigation. A more typical module (although still small at 500 lines and 300 statements) is the linker module. It is a subsystem of an automatic `makefile` generator.

From the description of a set of object files, the linker module identifies for each main procedure the subset of object files that unambiguously define procedures referenced (by an unbroken chain of references back to the main procedures object file) from within this subset of object files. The linker module also identifies procedures referenced within this subset of object files but defined more than once or not at all.

Other modules in development include a text editor, a finance application, a simulator for a simple CPU and components from game software.

4 Contributions

There are a number of areas where contributions to PEST would be useful.

1. An improved taxonomy of software faults that also considered object oriented programming would be helpful.
2. Additional modules, especially written in Java, would allow PEST to be used on object oriented program testing. Modules need to strike a balance between being large enough to yield useful results and small enough to be manageable. A module that takes several months to test would not be very useful.
3. The metrics used in the triangle example are very simple. More sophisticated metrics that take into account multiple modules would be very useful.

5 Conclusions

As PEST grows over time by contributions from the testing community it will become a valuable testing resource for researchers, tool vendors and educators.

The Information Technology Laboratory (ITL) at NIST responds to industry and user needs for objective, neutral tests for information technology. ITL works with industry, research and government organizations to develop and demonstrate tests, test methods, reference data, proof of concept implementations and other infrastructural technologies. Tools developed by ITL provide impartial ways of measuring information technology products so that developers and users can evaluate how products perform and assess their quality based on objective criteria.

References

[1] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold International Company Limited, New York, second edition, 1990.

[2] G. J. Myers. *The Art of Software Testing*. Wiley-Interscience, New York, 1979.