Cite this paper as follows: Bertrand Meyer, Ilinca Ciupa, Lisa (Ling) Liu, Manuel Oriol, Andreas Leitner, Raluca Borca-Muresan: *Systematic evaluation of test failure results*, to appear in to appear in Workshop on Reliability Analysis of System Failure Data (RAF 2007), Cambridge (UK), 1-2 March 2007.

Systematic evaluation of test failure results

Bertrand Meyer[1], Ilinca Ciupa[1], Lisa (Ling) Liu[1], Manuel Oriol[1], Andreas Leitner[1], Raluca Borca-Muresan[2]

[1]Chair of Software Engineering, ETH Zurich, Switzerland
{firstname.lastname}@inf.ethz.ch

[2]Department of Computer Science, Technical University of Cluj-Napoca, Romania
ralucab@student.ethz.ch

Abstract

Systematic software testing provides an important source of software failure analysis. The field suffers, however, from insufficiently reproducible results, lack of standard credible data, and insufficiently explicit assumptions. The present article attempts to provide an objective basis for failure analysis through an automatic testing framework (AutoTest) for contract-equipped software. We present five principles for scientific failure analysis, a set of reproducible test results, and a first analysis of their consequences for software development.

1 Testing strategies and their purpose

The area of software testing provides one of the best possible illustrations of the lack of credible large-scale failure analysis highlighted by the call for proposals for this workshop. While research in software testing has made considerable advances in recent years, and succeeded in rehabilitating an approach that used to arise considerable suspicions caused by the proponents of formal software development, it still suffers from insufficient credibility of its results. In particular:

- There are several criteria for assessing the effectiveness of testing strategies, such as number of faults uncovered, number of tests to first fault, time to first fault uncovered, code coverage, ratio of fault-revealing test cases out of all test cases run. None has achieved universal acceptance.
- The examples used to assess effectiveness of testing strategies are often unsatisfactory. For example, they frequently involve artificial cases, or existing programs into which faults have been seeded. Sometimes they come from programs written by students, since this makes it easier to set up experiments in an academic environment. Conclusions drawn from such examples raise doubts as to their applicability to actual industrial developments.
- All too often the supporting elements — test results, test data, program source and binary code, testing tool source and binary code — are known only through published article, and not available for people who might want, in the usual practice of experimental scientific research as accepted in disciplines other than computer science, to reproduce and verify the results.

It is not surprising then that many of the explicit and implicit assumptions common in the world of software testing lack a sound basis. The situation was already criticized by Hamlet [1] several years ago. For example, while all textbook presentations of testing include a discussion of coverage measure (instruction, branch, path coverage etc.) there is not enough connection with actual measures of software quality.

2 Criteria for testing strategies

We are engaged in a project to help remedy this state of affairs by uncovering hard evidence about testing strategies through analysis of failure data. This work is based on five principles: reproducibility, realism, objectivity, explicitness and quality.

Reproducibility: all of our results should be entirely reproducible by others. This means that all the code — both of the programs being tested and of the testing tools — is publicly available under an open-source license. (In the future it may become useful to include other people’s code, in particular example programs, with specific license status, but we will always focus on reproducibility.)

Realism: while artificial examples may play a useful pedagogical role, our focus is on test examples from production code. So far our work has used Eiffel libraries such as EiffelBase, which are used in thousands of actual applications, and other systems in production, originating with us or with other sources.

Objectivity: we submit all our hypotheses to experimental validation.

Explicitness: the criteria for such validation are stated explicitly, and are themselves subject to objective assessment.

Quality: any assessment criterion must be justified by evidence supporting its relevance to the general issue of software quality, since any testing strategy must, in the end, help towards this goal.

3 A framework

Our current work on contract-based testing has provided first steps for software analysis of test failure data. AutoTest [2] is a framework for automatic testing of contract-equipped software components. Here “automatic” is taken to mean more than in the usual application of this term to testing: AutoTest not only automates the testing process, but also removes the need for test data (by generating all objects, routine calls and argument values automatically) and test oracles (by using contracts, as present in Eiffel but also in JML and Spec#, as oracles).

Our standard testbed for AutoTest is not artificial examples but existing programs and libraries, where AutoTest regularly uncovers actual faults.

We have recently extended AutoTest to run extensive test campaigns using cluster computer architectures, allowing far more extensive testing than usually conducted in testing research.

In accordance with the principles above, all our software is freely available [3] and all our experiments are designed to be reproducible by others.

4. First results

While much remains to be done to provide an answer to the ambitious goals stated above, the application of AutoTest in line with the stated guidelines provides a first set of conclusions summarized below.

First, we can define some credible criteria for testing strategies on the EiffelBase library captured in a particular snapshot, for example the version of February 2006. This is both a realistic example, used in production applications, and an imperfect piece of software since AutoTest finds faults. Using always the same older version is obviously more accurate to compare results because when AutoTest finds a fault the maintainers of EiffelBase correct it. We can also pretend that we know “all the faults” in that library: if we ever find a new one, we simply add it to the fault base and update all the previous experiments. This leads to precisely defined criteria of any proposed testing strategy, or any claimed improvement to existing strategies:

- *How many* of the faults it finds.
- *How fast* it finds them. We believe this criterion is more significant than “number of tests to first fault”.

Next, we have a credible database of failure results, produced through exhaustive automatic testing, which we can submit to human analysis. This has been performed on a first set of results, leading to a tentative classification of faults. Note the combination of automatic mechanisms (exhaustive automatic testing through AutoTest) and the necessary human interpretation of the relevant parts of the result.

The classification of the types of faults we propose is two-fold: it addresses the two questions *Where?* and *Why?*. The first criterion for this classification is the location of the fault (*Where?*): either a contract or the implementation. We hence have the following categories, with percentages of corresponding faults found in EiffelBase mentioned in parentheses:

- Specification-induced fault (52.2%)
- Implementation-induced fault (46.4%)
- Impossible to judge (we cannot tell if the problem originates in the specification or in the implementation) (1.4%)

The second criterion that we use for classifying faults is the reason of the failure (*Why?*). A particular case of failure occurs in routines which (directly or indirectly) depend on routines containing faults. We call this a *supplier-induced fault*. We have two categories of supplier-induced faults, determinded by whether the supplier routine is called from the contract or from the implementation of the client routine. Other categories are problems induced by the use of inheritance, wrong export status (visibility faults), feature call attempted on a void target, and faults appearing in external routines. The results of the experiment are summarized below:

- Specification supplier induced fault (6.6%)
- Implementation supplier induced fault (22.1%)
- Inheritance-induced fault (8.8%)
- Wrong export status (14%)
- Feature call on void target (3.6%)
- Failure of an external routine (4.4%)
- Other (40.5%)

5. Summary and perspective

We are currently working on expanding the results obtained so far. In particular, we use cluster computing, as noted, to extend the scope of AutoTest execution to the equivalent of hundreds of hours of computer time; using the framework described here, we systematically evaluate the benefits of proposed testing strategies such as Adaptive Random Testing [4]; we compare the effectiveness of automated and human testing through carefully controlled experiments. We also expect to extend the approach to other libraries and programs.

We welcome the workshop’s focus on applying a scientific approach to failure analysis; by providing a framework for reproducible results, this work presents a contribution in the important area of failures found by testing.

REFERENCES

[1] R. Hamlet, Random testing, in J. Marciniak, editor, Encyclopedia of Software Engineering, pages 970-978, Wiley, 1994.

[2] I. Ciupa, A. Leitner, Automatic testing based on design by contract, Proceedings of Net.ObjectDays, pages 545-557, 2005.

[3] A. Leitner and I. Ciupa, AutoTest, http://se.ethz.ch/people/leitner/auto_test, 2006

[4] T. Chen, H. Leung, and I. Mak, Adaptive random testing, In M. J. Maher, editor, Advances in Computer Science - ASIAN 2004: Higher-Lever Decision Making. 9th Asian Computing Science Conference. Proceedings. Springer-Verlag GmbH 2004.