Using an Existing Suite of Test Objects: Experience from a Testing Experiment

B. Lindström  
Department of Computer Science  
University of Skövde  
Box 408, S-54128, Skövde, Sweden  
birgitta@ida.his.se

M. Grindal  
Department of Computer Science  
University of Skövde  
Box 408, S-54128, Skövde, Sweden  
magr@ida.his.se

J. Ofutt  
ISE Department, 4A4  
George Mason University  
Fairfax, VA 22030-4444 USA  
ofut@ise.gmu.edu

ABSTRACT
This workshop paper presents lessons learned from a recent experiment to compare several test strategies. The test strategies were compared in terms of the number of tests needed to satisfy them and in terms of faults found. The experimental design and conduct are discussed, and frank assessments of the decisions that were made are provided. The paper closes with a summary of the lessons that were learned.

Categories and Subject Descriptors
D.2 [Software]: Software Engineering; D.2.5 [Software Engineering]: Testing and Debugging

General Terms
Experimentation, Verification

Keywords
Software testing, fault seeding, representativity, independence

1. INTRODUCTION
There are many practical difficulties with designing and carrying out experiments in software engineering [2]. This paper explores some of these problems by describing and discussing experience from setting up and executing an experiment [3] to compare different test strategies. In particular, we explain the consequences, in terms of representativity and independence, of using an already existing suite of test objects as opposed to implementing our own test objects.

Section 2 gives a short overview of the experiment. Section 3 explains what we mean by representativity and independence and how our experiment is affected by these concepts. The actual choice of the test object suite is covered in section 4 and in section 5 we explain why and how we tailored the used suite of test objects before we conclude with our main observations in section 6.

2. OVERVIEW OF EXPERIMENT
The main goal with our experiment was to evaluate several black-box test strategies by comparing efficiency and effectiveness. The experiment measured number of test cases, number of faults found, and code coverage achieved.

The experiment was conducted by automating as much as possible. First, each test strategy was automated by constructing a script to select test cases. Then, natural language specifications of the test objects were manually translated to a tabular notation. The translation was based on equivalence partitioning, which divides inputs into equivalence classes, and values were stored into the table. These tables were then fed into the test case selection scripts, which generated a unique test suite for each pair of test strategy and test object. After generating all test suites, the next step was to execute the test cases on faulty versions of each test object. Faults were seeded by making copies of the test object, each containing a single fault, and the original version was considered to be correct. Outputs from the correct version were used as expected outputs. Actual outputs from executing test suites on faulty versions were then compared with the expected output, and if differences were found, the fault was considered to be detected. Code coverage information was also collected.

3. ISSUES
Two major issues that have to be considered when designing experiments are representativity and independence. Merriam-Webster [7] defines representativity as “serving as a typical or characteristic example.” The higher the representativity is, the more likely it is that the results from an experiment are valid and useful in the world outside the laboratory. Thus, this is a subject of external validity. Merriam-Webster [7] defines independence as “the quality or state of being independent.” Independent is defined as “not determined by or capable of being deduced or derived from or expressed in terms of members (as axioms or equations) of the set under consideration.” An experiment is of course, always controlled to some extent but it is important to limit the control as much as possible so that the results
are primarily caused by the investigated parameters. Maintaining a high level if independence in an experiment allows a high degree of confidence in the results. This, of course, is a subject of internal validity.

3.1 Representativity

Representativity in our experiment is affected by the choice of test objects, selection of faults, and choice of test cases. In software, representativity is usually impossible to guarantee or even discuss in general terms. For example, how can we decide if a program is representative of the programs made by company X or if a set of faults is typical of a certain type of systems? While this problem clearly limits the generality of any results, representativity still plays an important role in the experimental set-up.

One or more test objects must be used to evaluate a set of different test strategies. Selection of such test object is done according to a set of different criteria. Apart from conforming to these criteria, one consideration is whether a tester is likely to use the investigated test strategies on that test object. If no tester would ever use the strategies for this kind of test object, the results would be useless.

Another question of representativity is with the choice of faults. It can be argued that the most interesting faults from a results perspective are detected by some, but not all of the investigated test strategies. These faults can demonstrate differences between two test strategies. Thus, a representative set of faults should also exhibit a large variance in the type of faults rather than in the number of faults of a certain type.

The test strategies used in our experiment are all based on equivalence partitioning. Consideration of representativity regarding the test cases is especially important when defining equivalence classes. When we define the equivalence classes we actually make an assumption about potential faults. We assume that all test cases that belong to the same class will reveal the same set of faults and the accuracy of this assumption depends on the actual faults. Moreover, the resulting set of equivalence classes depends on the tester and his experiences. Different testers are likely to derive different equivalence classes. Hence, it is desirable to define the equivalence classes in such a way that they are representative both with respect to faults and testers. We consider this particular form of representativity very hard to validate and the design of our experiment explicitly tries to reduce its importance. The same set of values is used for all strategies and any differences in the results that depend on the level of representativity of the values will be the same for all compared strategies. Hence, as long as we use an experienced tester for deriving the equivalence classes, the results are useful.

3.2 Independence

A crucial aspect of designing an experiment to compare test strategies is to select a test object that maintain independence between the implementation and the experimental set-up. A priori knowledge of the test strategies can bias the choice in such a way to cause some test strategies to be favored. Thus, it is important that the test object is implemented by a programmer who does not know the test strategies to be used.

A similar problem exists when seeding faults into a program. This is a tricky problem since generating test cases before seeding faults means the experimenter may know about the test cases when creating the faults, yet seeding the faults before generating the tests means the experimenter may know about the faults when generating the tests. In both cases there is an obvious risk of an unwanted bias. In order to maintain a high level of independence, it is therefore necessary to use a set of faults created by a different experimenter, who does not know how the tests are created.

Our experiment used this approach; dividing up the tasks. Furthermore, all test cases were generated before studying the faults. When tests were generated, the test generator did not even know how many faults had been created.

4. CHOICE OF TEST OBJECT

As mentioned previously, representativity and independence plays a key role in selecting test objects to use in an experiment. However, other criteria may also affect this choice. This section discusses some of the criteria used in our experiment.

To conduct our experiment we needed our test objects to satisfy certain criteria: (1) Specifications must exist to derive test cases from; (2) An implementation in some suitable language must exist; and (3) A number of faults must be known and documented.

Our two main options were to create our own test objects or to find already existing test objects. The main argument for making our own test object suite was that we would have total control over factors of test objects such as size, type of application, types of faults, source code language etc. However, as already explained in section 3 creating our own test object would result in significant independence and representativity problems.

Thus, we searched for an already existing test object. An Internet search yielded a complete experimental package containing a “Repeatable Software Experiment”, including a number of different test objects by Kamsties and Lott [5, 4]. The purpose of the package was to compare the effectiveness of one black-box test strategy, one white-box test strategy, and one code reviewing technique. Although this purpose was somewhat different from ours it was still close enough to provide suitable test objects.

Using an already existing test object also had the advantages of time savings and an increased possibility of cross-experiment comparisons. Basili et al. listed as an important factor for advancing the knowledge in computer science by experimentation [1]. However, since the purpose of the experiment of Kamsties and Lott was somewhat different than ours we ended up having to tailor the test objects in some respects. The following section describes our changes.

5. TAILORING THE TEST OBJECT SUITE

The experiment was large enough to make it necessary to automate the generation, execution, and results evaluation of the test cases. Automation caused some of the original faults in the test suite to become undetectable. One example was a fault that sent error information to stdout instead of stderr. This could not be detected because the automation script sent all output from the test object to the same file. Thus, our automation decreased the number of faults could be used.

5.1 Fault seeding
Our initial data indicated that the total original number of faults (33) was not large enough to yield significant differences between the test strategies that we compared. Thus, we decided to increase the number of faults in the test objects. However, all the people involved so far in the experiment had extensive knowledge of both the algorithms of the test strategies and the contents of each test suite. If one of the existing members of the project seeded more faults, that would seriously jeopardize the independence, biasing the results. Our solution was to incorporate a new member into the group to create more faults.

This person was given printouts of all the source code and asked to come up with more faults. As described in section 3.2 it is difficult to assess the representativity of faults, so this “fault seeder” was given carte blanch with respect to the type of faults that should be constructed. A program mutation-like approach resulted in 98 new faults.

The original test suites were then run on the new faults. Out of the 131 total faults, 120 were detectable and the fault detection by the investigated test strategies varied from 108 to 119. Although the difference was still relatively small, we decided to stop at this point. A major reason for not increasing the amount of faults further to obtain greater differentiation is the question of representativity. Even if we could achieve greater differences in the fault detection results we would not know if these results are general or due to an over representation of a certain type of fault.

With this somewhat discouraging insight we instead proceeded by very carefully analyzing the faults that were only discovered by a subset of the test strategies. This analysis indicated that we could classify the faults into different types and that for some types of faults one test strategy would detect faults by chance while other methods would guarantee the detection of all the faults of that type. Our conclusion from this is that the diversity of faults is more useful than the number of faults in this type of experiment.

5.2 Separating the faults

In the original experiment of Kamsties and Lott there was only one version of each test object, which contained all faults. To simplify the results analysis we separated the faults into different versions of the programs. Specifically, when a program failed, we knew exactly which fault caused the failure. Representativity is the main argument for keeping all faults in the same file since this is the case in a real testing situation. However, as was shown in section 3.1, the question of general representativity is difficult to handle. Thus, the choice to trade representativity to simplify results analysis was deemed not to hurt the experimental design.

6. CONCLUSIONS

Using an already existing suite of test objects in our experiment saved time, and increased both independence and repeatability. However, due to the limited amount of existing suites of test objects we did not find a perfect set of test objects for our purposes. Thus, we had to modify the test objects. These modifications proved challenging both with respect to representativity and independence.

Despite the hundreds of experiments that have been performed in software engineering over the past decades, very few collections of test objects are available. A recent attempt to remedy this problem is SEEWeb, a repository of software experiments [6]. SEEWeb is currently under construction.

One major lesson about software testing experimentation from this experiment is that it is impossible to assess the general representativity of a test object and its faults with respect to the real world. However, representativity still plays an important role in the design of an experiment. A second lesson is that more than one person needs to be involved to ensure independence. It is important to isolate the knowledge of test strategies and their implementation from the knowledge of the faults to avoid bias. A third lesson is that when trying to compare and differentiate test strategies, the number of faults used is probably not at important as diversity of faults. A challenging question is how to quantify “fault diversity.”

7. REFERENCES