

Keynote Address

Auditing Informal Software Testing and Analysis Processes

William E. Howden
CSE, University of California at San Diego, La Jolla, CA

1. Introduction

The two most common software evaluation processes are testing and informal analysis. In order for these processes to be effective, and their results to be accepted, it must be confirmed that they were actually carried out, and that they met some reasonable standard. We will review some of the kinds of evidence that can be used to satisfy auditing procedures, and describe ongoing work in this area.

In the case of testing, we can audit the evaluation process by requiring that test sets be delivered along with the code, and that the results of the tests be documented. However, we know that not just any set of tests should be used, and we want to confirm that an attempt was made to use a good set of tests. In the statistical approach, we evaluate the acceptability of a set of tests on the basis of its size and the pattern of failures and successes the software exhibited over those tests. Alternatives involve the use of some kind of code coverage monitor, namely branch or statement coverage. We will review these methods, some of the more sophisticated extensions such as the use of data flow couples, and introduce a new method called statistical coverage.

In the case of analysis, we can use checklists, as in inspections, but there is no easy way of knowing that the checklists have actually been used, or used rigorously. One approach is to require the programmer to document reasoning or program understanding processes in the form of stylized comments that can be analyzed for consistency. The presence of the comments documents the evaluation process, and the analysis of the comments documents their consistency. We will review a method called QDA that has been applied to assembly language, COBOL and Ada programs.

2. Auditing of Software Testing

2.1 Statistical testing

Two approaches to testing will be identified: statistical and guided. In statistical testing, it is necessary to know a program's operational distribution, i.e. the frequency of occurrence of different possible input cases. When this is available, tests are chosen randomly according to this distribution. The failure density for a program is the distribution weighted size of the input domain over which the program will fail. It is possible to construct mathematical formulae which allow us to conclude, on the basis of a set of tests, that we can have confidence C that the failure density is less than b , where b is between 0 and 1.

The difficulty with the statistical approach is the large numbers of tests that are needed. For example, to be 99% confident that a program's failure density is less than .01, we would have to observe more than 400 tests over which the program operated correctly.

2.2 Guided testing and coverage metrics

In functional testing, we confirm that a program contains required functionality. We may extend this to the testing of design and implementation functions to further confirm that subfunctions operate as expected. After confirming that a program contains required functionality, it is common practice to look for specific kinds of faults. For example, if a program searches a list for some kind of element, then we should make sure that we test for the case where the last element of the list is the special kind, since this kind of input case is associated with a disproportionately large number of faults.

We may require the creation of documented test plans and procedures as part of a test audit, but this will not necessarily allow us to audit the

effectiveness of the tests that were used. To compensate for this, coverage metric tools are often used. We may interpret statement coverage as confirming, in an approximate way, that all program functions and subfunctions have been tested. Branch coverage confirms that different data subcases have been explored. More sophisticated coverage measures include data flow couple coverage, in which it is necessary to test together pairs of statements where the first statement prepares data used by the second. This approach can be interpreted as a more refined attempt to ensure that all program subfunctions are tested, where these correspond to particular combinations of data flow related statements.

One of the difficulties of using coverage metrics is the practical necessity of partial coverage. Normally, we require that only 80 to 90% of all statements or branches in a program be covered. The problem is that it seems that different programs should have different levels of coverage. In addition, what does it mean when we have tested 85% of a program's statements or branches, as opposed to 100%? A new approach will be described, called statistical coverage. In statistical coverage, we use statistical testing methods to confirm that when we have less than full coverage, that we can be C confident that the probability of the untested components being used in some program execution is less than bound

B. Acceptable numbers of tests can be used to achieve the kinds of partial coverage measures that are normally used. Under certain circumstances, it is possible to attain more stringent partial statistical coverage levels without the large numbers of tests that would be needed to achieve similar levels in pure statistical testing. Both the basic idea, and some possible areas of application will be described.

3. Auditing of Software Analysis

One approach to encouraging the documentation of analysis and code reasoning activities is to require a high density of program comments. Normally comments are associated with the kind of analysis carried out during program construction, but they may also be used to document program understanding during post construction analysis. Studies of commenting reveal two general classes: operation and state

oriented. Operation comments indicate when certain kinds of operations are performed, and state comments when changes are made to object properties. In the QDA approach, distinctions are also made between events and assumptions. Events use abstractions to describe what is happening at some point in a program. Often such imperative statements depend on assumptions about previously occurring events. Comments are also used to document these assumptions. Simple assumptions indicate that some event is expected to have previously occurred along all paths leading to an assumption. More complex assumptions, particularly operation oriented, describe previous sequences of events. Assumptions may also be made about future events, events which programmers assume will occur along outgoing paths from some location, and which justify the occurrence of events at that location. If a stylized language is used, it is possible to analyze the consistency of events and assumptions. QDA analyzers have been built for a variety of languages.

More complex issues in comments analysis concern inclusion of necessary event comments, and internal and external assumptions. It is possible to incorrectly conclude that some assumption is correct when an event is missing along a path leading to its location which allows false conclusions about the effects of previous events that would have been nullified by the missing event. This possibility can be guarded against using a fail-safe approach in which it is necessary to document all changes to concrete code objects that appear in event comments. Discussion of this and other problems associated with the use of comments for auditing the adequacy of informal program analysis will be described.

Internal assumptions reference effects inside the module in which they occur. External assumptions reference properties of other modules. Principles of modular programming would normally restrict external assumptions to input and output object properties at the interfaces of other modules. In the case of cooperating processes, where task interaction is involved, the interfaces are more complex, and assumptions may have to be made about internal sequencing of interaction operations, such as Ada semaphore operations, in another module. One approach is to use a documentation language that allows a user to summarize the interaction structure of a module,

which is then used to confirm or deny the correctness of another module's external assumptions.

More sophisticated analysis is facilitated through the use of rules which describe relationships between object properties and/or

operations. These function as rules of inference which (partially) define the meaning of the informal abstractions that are used in an informal analysis. In the QDA approach, rules can be used during an attempt to validate an assumption, along with relevant event comments.