Consiglio Nazionale delle Ricerche

ISTITUTO DI ELABORAZIONE DELLA INFORMAZIONE

PISA

An Introduction to the Practice and the Tools of Software Dynamic Testing

Antonia Bertolino

Nota Interna B4 - 03
Gennaio 1990

An Introduction to the Practice and the Tools of Software Dynamic Testing

Antonia Bertolino

Istituto di Elaborazione della Informazione del CNR, Pisa, Italy.

ABSTRACT

In this paper we provide a brief, yet comprehensive, introduction to the practice of dynamic testing of computer software and to currently available automated testing tools.

We first expose the basic concepts of testing, pointing out its properties and its limitations.

Then we propose a control flow diagram for the testing process. Following this diagram step by step, we describe all the activities involved and the relative tools.

Quite obviously, testing must be prepared through an accurate and well-documented specification phase.

1. INTRODUCTION

In the last decade, a relevant part of investigations within the area of software engineering has dealt with software testing, and, particularly, a great effort has been devoted to the automatization of the testing process. As a result, today many commercial or prototypal automated testing tools exist.

Properly exploited, automated testing tools can highly improve not only testing productivity, but also its efficacy.

In fact, they take upon themselves the mass of clerical, repetitive and error-prone activities which are required for the systematic testing of software systems.

Yet, of course, they can never fully replace human operators; more than that, we can affirm that human competence and ingenuity remain the *sine qua non* condition to the success of the testing process.

In this paper, we give a general overview of automated software testing.

Our goal is not to provide a survey of existing tools; for this, valuable works have already been published, e.g. [1]. Also, it must be considered that the field is quickly growing and therefore it is hard to obtain an up-to-date snapshot.

Neither our goal is to debate in principle the extent to which the testing process can be automatized: this is indeed a

stimulating argument, but it is of little use (at least in the short term) in order to operate testing.

Instead, we want to describe here the support offered by available automated testing tools or, from the opposite point of view, we want to point out which expectations can (and which cannot) be fulfilled by automatic testing today.

Preceding the discussion on automated tools, we also present an overview of the basic concepts of software testing, so that the paper is accessible also (or, we should say is devoted mainly) to testing novices.

Our central aim is, in fact, to provide a comprehensive introduction to the argument treated.

In this perspective, we do not give here a mere, unproductive tool classification. Instead, we conduct a walkthrough of the testing process, progressively introducing concepts and tools, as they are applied.

So, the paper is organized in the following way: the basic concepts behind software testing are presented in Section 2 and a control-flow diagram for the testing process is derived in Section 3. In this diagram, we identify 5 fundamental steps; for each step we discuss separately, from Section 4 to Section 8, the basic concepts and the support given by available tools. Some conclusive remarks are given in Section 9.

2. TESTING CONCEPTS

Testing is essentially an a-posteriori activity; in fact, it is conducted on a finished product to check its validity or, in a word, to *validate* it.

The notion of validity must be intended, of course, as a relative concept: in fact, to decide about the validity of a product, we must evaluate it against a model representing what we intended to realize.

In the field of software engineering, the test item is a program P, which transforms a set of input data, i.e. the Input Domain (ID), to a set of corresponding output data, i.e. the Output Domain (OD).

The program P is developed according to a function F, modelling its behaviour. In practice, F can be expressed in different ways: as a formal model, or as natural language specifications; in the worst case, F is merely an idea in the mind of the programmer. Whatever F is, in order to validate the program P, we must compare the latter with F.

In software testing, this comparison is performed by *experiments*, i.e. by executing P and, for each run, comparing its

output with the expected output, i.e. that obtained according to F. When the output of P diverges from the expected output, we say that we have discovered an error.

In the literature, the activity of validating P by experiments is most often referred to as *dynamic testing*. For completeness, we just mention here the two other (complementary) approaches commonly used for program validation: in *program proving* [2], a program is shown correct, i.e. equivalent to its specification, through techniques of theorem proving; in *static testing*, the program is essentially inspected [3], checking pre-stated properties or structures. These two approaches are outside the scope of this paper: henceforth, our subject will be dynamic testing.

To obtain through dynamic testing the certainty of the validity of P, we should try the program on the entire ID. In fact, due to the intrinsic discontinuous nature of software, given an observation on a point of ID, we cannot infer any property for the other points in ID.

But, excluding trivial cases, ID is too large, practically infinite, for exhaustive testing. So, to contain test costs, when we test a program, we check the behaviour of P only on a subset of ID, which constitutes the Test Input Domain (TID). From this limited number of test runs, we then infer which should be the behaviour of P on the entire ID, even if there is no definitive justification to do so.

So, testing corresponds to *sampling* a number of behaviours of a program P among all its possible behaviours, by sampling a number of input data within ID.

Ultimately software development, like any other production, must obey market laws and the testing phase is generally adjusted so to reach a reasonable compromise between the need to maximize the dimensions of TID, in order to increase our confidence in the validity of the program, and the opposite need to minimize the dimensions of TID, in order to reduce testing costs.

Of course, we would like to sample ID systematically, so that we can raise our chance to discover possible errors. The individuation of a suitable testing/sampling strategy is known as the problem of *test data selection.*

A theory of test data selection has been established, starting with the influential work of Goodenough and Gerhart [4].

3. THE TESTING PROCESS

From the above-exposed basic concepts, we can identify the following 4 tasks constituting the testing process:

1) Selecting a TID (subset of ID) according to a suitable testing strategy.

2) Deriving for TID the expected output data, which form the Expected Output Domain (EOD). In testing literature, this task is known as the *oracle problem.*

3) Obtaining the effective output data by executing P on TID; the effective output data form the Test Output Domain (TOD).

4) Comparing the sets TOD and EOD, in order to validate P.

A Data Flow Diagram (DFD) depicting software testing is shown in Figure 1, in which the bubbles correspond to the above identified tasks.

In practice, we never succeed at the first attempt to identify an "adequate"[1] TID. So, the testing process becomes a trial and error activity, in which the task 1 is iterated a number of times, each time augmenting TID, until we are satisfied, according to a more or less defined criterion (unfortunately, this criterion is often the budget at disposal).

We can derive now in Figure 2 a diagram of the testing process, which reflects the DFD of Figure 1, but also takes into account the above observation. Besides, we introduce the notion of control flow between the tasks involved, according to the usual practice.

With reference to this control flow diagram, we can identify 5 fundamental steps for the testing process, corresponding to the 4 above listed tasks and, in addition, to the task of deciding to stop the test activity, according to a specified *exit criterion.*

In the following Sections 4-8, we shall discuss in the order the concepts behind each step and the automated tools provided for.

---

[1] The concept of adequacy for test input data has been formally defined in [5]. We use the term in its intuitive meaning.

4. THE SELECTION OF TID

The selection of a suitable TID is the central problem of testing. The tools devoted to this task are typically the *test data generators*.

Obviously, given ID, we could easily conceive an automatic tool which extracts a subset from it *anyhow*. But, of course, this is not, or it is not simply, our problem. In fact, for testing to be effective, the selection of TID must be done systematically, according to a testing strategy.

Therefore, before seeing the typical functions performed by test data generators, let us briefly introduce the fundamental strategies followed to select the test input data.

As we have said in Section 2, testing a program P consists of observing a representative sample of all its possible executions, by trying P on a subset TID of ID. More or less explicitly, we perform the selection of TID by first partitioning ID into a number of classes, such that P behaves equivalently for all data within a class. At this point, we can check only a few executions for each class, for example just an internal datum and a border one [2].

Of course, there is not a unique ID partition we can do. In fact, the equivalence of P executions within one input class is always relative to our point of observation, or, in other words, to what we want to validate.

So, for example, if we want to validate P against its specified I/O relations, we shall partition ID so that one I/O relation corresponds to each class (this falls into functional testing, see below). Instead, if we want to check P execution on each segment of the source code, we shall choose a different ID partition, such that each class corresponds to executing one segment in the code (this falls into structural testing, see below).

There are 3 basic testing strategies:

1- *functional testing*: essentially, TID is selected (i.e., ID is partitioned) according to the reference model F.

2- *random testing*: essentially, TID is selected (i.e., ID is partitioned) according to the way in which P is operated (in fact, ID is generally partitioned according to the foreseen use of P).

3- *structural testing*: essentially, TID is selected (i.e., ID is partitioned) according to the structure of P.

---

[2] The weak point in this procedure (i.e the reason why testing can only give confidence in the validity of P, but never guarantee it) is, of course, the assumption that P behaves equivalently on all the data of a class, given the inherently discontinuous nature of digital elaboration.

In the literature, the first two strategies are both referred also as *black-box testing*, and the third as *white-box testing*. Besides, a variety of testing methodologies [6] fall within each of the above basic strategies. For example, structural testing can be conducted on the data structure or on the control structure.

Test data generators, of course, provide different assistance, depending on the strategy they have been designed for.

Test data generators for random testing function in the simplest way: they peek random data from ID, according to the chosen distribution. Most often, the operational distribution is chosen: in very simple words, most used input data are tried more frequently.

The opposite approach is stress testing, a methodology of functional testing in which the special conditions, i.e. the most strange cases, are insisted upon.

In functional testing, it is the operator's task to analyze the reference model F and to select TID according to it. However, some tools have been developed which assist the operator in the clerical activities pertaining to this task.

There are some test data generators which provide the operator with a language to specify the test data and then they derive the test instances automatically. These tools are typically used, for example, to generate test programs for compiler validation.

Most commonly, it is not simple to specify an algorithm to derive TID, for example, when F consists of the functional specification document. In these cases, the assistance provided is a support to organize the test data, already derived by the operator, into a suitable test plan (for example TID can be structured as a tree). Then, during test execution, the tool automatically peeks the data, as specified, and feeds them to P. This support is usually included into the *test-driver*, which is an automated testing tool which launches P and controls its execution: test-drivers are further treated in Section 6.

Finally, many different support tools to generate test data for structural testing exist: they can be classified in *data flow analyzers* and *coverage analyzers*, according to whether the data structure or the control structure of a program is referred.

These tools essentially provide the operator with information about the structure of P and these information are then exploited by him to select TID. Precisely, the structure of P is analyzed in order to individuate the program *paths* and ID is then partitioned so that each class corresponds to a different path: in fact, these tools are called more specifically *pathwise test data generators*.

In data flow analysis, the paths are typically associated to the way in which program variables are used [7]. A criterion for

selecting TID could be, for example, to exercise P so that all definitions, i.e. all assignments, are referenced, i.e. used, at least once.

In coverage analysis [8], instead, the paths are associated to the track followed by control flow during execution: a criterion for selecting TID could be, for example, to exercise P so that all segments are executed at least once.

In both cases, path selection is previously performed during a phase of static analysis of the code. Particularly, for the analysis phase, *symbolic evaluators* [9] are used, which simulate P execution by using as input symbolic values, in order to derive the conditions on which a path is executed.

Successively, during test execution, the effective level of path coverage is measured through dynamic analysis, i.e. monitoring P execution. The tools which allow to perform this measure are called *program instrumenters*. They insert appropriately some *probes* (essentially, some procedure calls) into the source code during a preprocessing phase and then, during program execution, they monitor the passage of control-flow to these probes, in order to measure test coverage according to the specified criterion. For example, to measure the coverage of segments, a probe is inserted where the control flow passes to each branch.

Before closing this section on test data selection strategies, we want to emphasize that there is not a strategy that is the best one. The adoption of a strategy could be based on tester's background, on program development practice and on available resources. Of course, integrating diverse strategies always enhances testing effectiveness; some experimental studies have been conducted [10], which confirm this intuitive concept.

5. THE DERIVATION OF EOD

This step does not present substantial obstacle to automation: in fact, the same program P under test, when it is valid, can be seen itself as an automated tool which transforms the TID to the EOD. Of course, this is only a paradox, since we need to specify EOD just to check P validity.

The derivation of EOD can be made through an automated tool, called *expected results generator,* by specifying F in an executable language.

In practice, developing a running version of F is often considered too expensive and EOD is derived manually.

Sometimes, EOD is not derived at all and the effective test outputs are inspected by the operator: this may result a

dangerous testing practice, since testing evaluation becomes dependent on the competence and the fairness of the operator.

6. THE DERIVATION OF TOD

The task of deriving TOD is obtained fully automatically by running the program under the control of a testing tool, known as *test driver* (also *testbed* or *test harness*).

Test drivers conduct all the clerical activities required during test: as said in Section 4, they feed the selected test input data to the program under test.

They provide a test environment, also supplying drivers and stub, i.e. simulators for the calling procedure and the called ones.

They launch automatically the tests, iterating on all data in TID.

They register the test outputs.

They evaluate the test outcome, when an automatic comparison with EOD is feasible (see below).

Furthermore, they often supply statistics on test performance.

Test drivers are essential for economic *regression testing*. Regression testing consists of retrying previously passed test cases, after having modified P, to check that changes have not introduced also unwanted effects.

7. THE COMPARISON BETWEEN EOD AND TOD

When EOD has been exactly specified, the comparison between TOD and EOD can be performed by an automated tool called *comparator*.

A comparator is a simple utility that compares the contents of 2 files, like the command DIFF provided by the UNIX system.

Some comparators also provide the capability to mask a part of EOD, so allowing to tolerate some unimportant differences.

Comparators are often included in test drivers.

If EOD has not been specified in advance, as hinted in Section 5, the comparison must be performed by the operator through the inspection of TOD.

8. THE EXIT CRITERION

Unfortunately, in the practice, when to stop test is often decided arbitrarily on management constraints, e.g. the budget at disposal or the time schedule. However, the right approach is that of establishing a criterion on which to evaluate test extensiveness and eventually to stop testing.

This exit criterion can be based on: i) the achieved coverage value on a path selection criterion; ii) a reliability measure [10]; iii) the rate of errors discovered among a set of intentional errors.

The achieved path coverage value is provided by means of program instrumenters already sketched in Section 4. When the coverage measure reaches a desired (high) threshold, testing is stopped. Consider that 100% coverage for a given criterion is not always feasible, since portions of dead code may be present; besides, when a consistent path coverage has already been reached, raising it further becomes more and more expensive.

The exit criterion can be based on a reliability measure when TID has been selected according to the operational distribution. Very briefly, discovered errors are related to the execution time after which they have been observed. When the *failure intensity*, i.e. the rate of failure detection, reaches a desired (low) threshold, testing is stopped.

For this test evaluation methodology, the operator has to define the operational distribution for P, according to which a random test data generator can select TID; also, he establishes the desired failure intensity value. A tool for *reliability measurement* can be used to record time intervals between failures and to calculate failure intensity.

Finally, a methodology has been proposed to introduce intentional errors in the program and then to evaluate testing effectiveness on the rate of detection of these known errors. When a desired (high) threshold in the rate of discovered errors against the total amount of errors introduced is reached, testing is stopped. The errors can be put altogether into the program under test (error seeding) or they can be distributed among many similar versions of the program, called *mutants* (mutation testing). *Mutation tools* are essentially test drivers, which also compute the mutation rate, in brief the percentage of erroneous versions detected. They also provide the capability to specify the mutation algorithm, and then derive the mutant programs automatically.

9. CONCLUSIONS

We have described the testing process and the associated tools.

We have seen that:

•) the clerical activities, like launching the tests, registering the outputs, comparing files, reporting, etc, are automatically performed by test drivers.

We can further comment that, not only these activities can be automatized at low cost, but also that, given the increasing

complexity of current software products, it is not thinkable to operate today a testing activity without the support of a test driver.

•) On the contrary, some tasks always require the intervention of a competent operator, particularly the selection of test cases, even if some assistance is provided by test data generators.

In summary, testing comprises a planning part and an operative part. While the latter can take great advantage from the support of automated tools, the planning part is mostly based on human ingenuity and competence.

We cannot conclude this overview on software testing without pointing out how the effectiveness of the testing phase is strictly dependent on the preceding phases in the life cycle.

In fact, we stress again that a program is always validated against a model of the *correct function*. So, before entering the implementation and testing phases, we must specify and document precisely P expected behaviour.

References

1. R. A. DeMillo, et al., *Software Testing and Evaluation*, The Benjamin/Cummings Publishing Company, Inc., 1987.

2. S. L. Hantler and J. C. King, An Introduction to Proving the Correctness of Programs, *ACM Comp. Surveys*, 8 - 3, 331-353 (September 1976).

3. M. E. Fagan, Design and Code Inspections to Reduce Errors in Program Development, *IBM System Journal*, 15 - 3, 219-248 (1976).

4. J. B. Goodenough and S. L. Gerhart, Toward a Theory of Test Data Selection, *IEEE Tr. on Software Eng.*, SE-1 - 2, 156-173 (1975).

5. R. A. DeMillo, R. J. Lipton and F. G. Sayward, Hints on Test Data Selection: Help for the Practicing Programmer, *Computer*, 11 - 4, 34-41 (April 1978).

6. W. R. Adrion, M. A. Branstad and J. C. Cherniavsky, Validation, Verification and Testing of Computer Software, *ACM Comp. Surveys*, 14 - 2, 159-192 (June 1982).

7. J. Laski, On Data Flow Guided Program Testing, *SIGPLAN Notices*, 17 - 9, 62-71 (September 1982).

8. E. F. Miller, Software Testing Technology: An Overview, in *Handbook of Software Engineering* (C. R. Vick and C. V. Ramamoorthy, eds.), Van Nostrand Reinhold Company, 1984.

9. L. A. Clarke and D. J. Richardson, Applications of Symbolic Evaluation, *J. of Systems and Software*, 5-1, 15-35 (February 1985).

10. R. W. Selby, Combining Software Testing Strategies: An Empirical Evaluation, *Proceedings, Workshop on Software Testing*, July 1986, 82-90.

11. J. D. Musa and A. F. Ackerman, Quantifying Software Validation: When to Stop Testing?, *IEEE Software*, 6 - 3, 19-27 (May 1989).

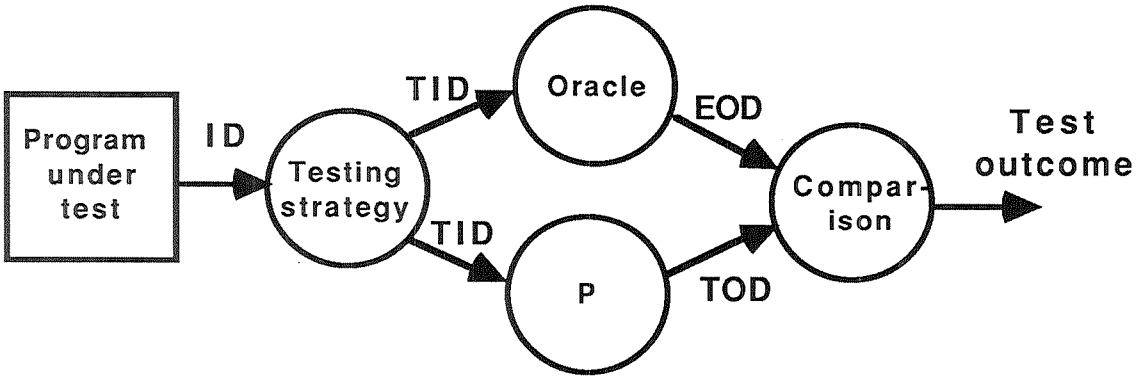

Figure 1: A DFD for Software Testing

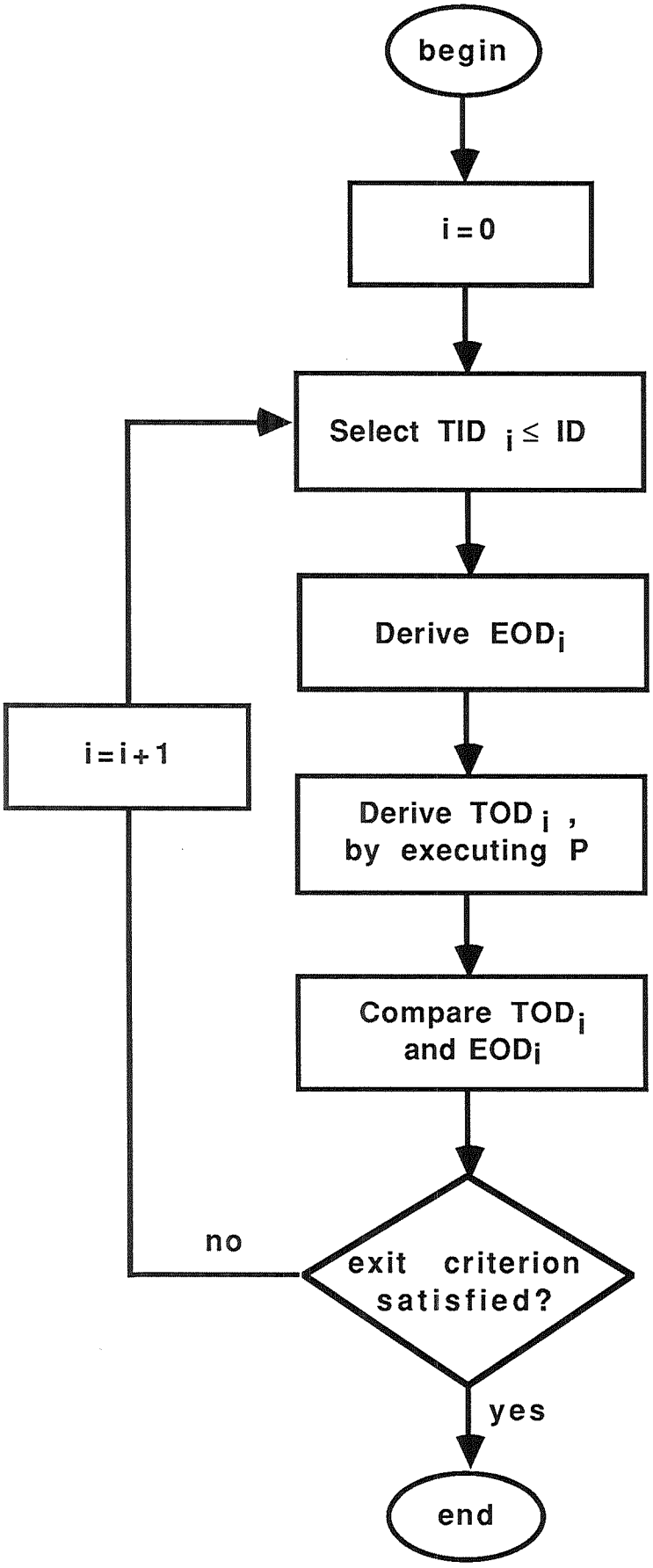

Figure 2: The Process of Software Testing