Automatic Test Software

Yashwant K. Malaiya
Computer Science Department
Colorado State University
Fort Collins CO 80525

Even when software is developed using a rigorous discipline, it will contain a significant number of bugs. At one time, it was believed that the use of formal methods would eventually allow probable correct programs to be written, thus completely eliminating the need for testing. Today, we know that while the number of defects in a program written will be lower under certain development environments, they will still add up to a large number in a reasonably large program. There has to be testing and debugging, which can consume up to 60% of the total effort.

A study by McGibbon presents a perspective [mcg97]. He compares traditional development approach with two formal methods, VDM and Z. For a program with 30,000 source lines of code (SLOC), the traditional methods will be able to deliver software with 34 defects with an estimated life-cycle cost of $2.5 million. Using Z, the total cost would be reduced by $2.2 million, but still about 8 defects would be left. Additional cost savings can be achieved by using VDM; however, it will result in 24 defects in the delivered product. In all three cases, a substantial part of the cost is due to testing and debugging.

Finding most of the defects is a formidable task. Much of testing is still being done manually, and the process is intuitively guided. In this respect, software development lags behind hardware design and test, where use of tools is now regarded to be mandatory. The main deterrents are the learning curve of testers and the reluctance of management to commit to new tools in the middle of a product. However, as we discuss below, in a carefully implemented program, the cost/benefit ratio strongly favors the use of tools. Manual software testing is very repetitive, and thus very unattractive to software engineers. Use of appropriate tools will allow testers to use their skills at a much higher level. The software market today is extremely competitive. Many cutting-edge organizations have been able to achieve high process maturity levels, and they are delivering software with significantly lower defect defects than in the past. The average acceptable defect density declined from about 8 per KLOC in the late 70s to about 5 in 1990 [bw91]. It is now believed to be below 2 per KLOC in leading edge organizations. In the near future, the reliability expectations in the market will particularly dictate all developers to greatly rely on automation in testing.

Software testing tools began appearing in the 1970s. LINT, a code checker, was part of the old Unix systems. The name was appropriately chosen, because as Poston [pos98] explains, it "goes through the code and picks out all the fuzz that makes programs messy and error-prone. One of the first code instrumentors JAVS was developed by Edward Miller (who founded Software Research in 1977) in 1974 for evaluating structural coverage. In the mid-80s computing became cheap and plentiful. Powerful software technologies like programming environments, structured

relational database systems became available, making it possible to develop tools that can capture and analyze a lot of information. Today, we see integration of capabilities like capture-replay, comparison code coverage etc. in the newer tools.

Recently, we have seen emergence of a new class of tools that addresses the new programming paradigms. Memory lead defectors emerged in 1992 and have already become indispensable in many development organizations. We now see many new tools for testing web servers and documents and for handling the Y2K problem. We can expect to see new types of tolls becoming available that will automate some of the gaps in the program design/test life cycle.

Many organizations are still resisting the introduction of testing tools. Surveys suggest much of it is due to the steep learning curve faced by testers, who are very reluctant to move to new approaches when they are facing deadlines [qtc97]. Some tools have not delivered what they seem to promise. This will change when tools are better understood. Today, no hardware engineer would think of doing a design without using SPICE or VHDL level simulation. No one today thinks of doing manual test generation for hardware. The same will be true for software in only a few years.

The Quality Assurance Institute conducted a benchmark study comparing manual and automated testing in 1995 involving 1,750 test cases and 700 defects [qaq95]. The results are shown below in Table 1. It shows that while initially, the tools require some investment, they eventually result in an impressive saving in the time spent in testing.

Table1 : Manual vs. automated testing [qaq95]

Test step	Manual testing	Automated testing	Percent Improvement
Test plan development	32	40	-25%
Test case development	262	117	55%
Test execution	466	23	95%
Test result analyses	117	58	50%
Defect tracking	117	23	80%
Report creation	96	16	83%
Total hours	1090	277	75%

Terminology

The following are the important terms used in software testing literature.

Failure: a departure of the system behavior from user requirements during execution.

Defect (or fault): an error in system implementation that can cause a failure during execution. A defect will cause a failure only when the erroneous code is executed, and the effect is propagated to the output.

Defect density: Usually measured in terms of the number of defects per 1000 source lines of code (KSLOC).

Failure intensity: The expected number of failures per unit time.

Mean-time-to-failure (MTTF): The expected duration between two successive failures. It is inverse of failure intensity.

Operational profile: To be able to estimate operational reliability, testing must be done in accordance with the operational profile. A profile is the set of disjoint actions, operations that a program may perform, and their probabilities of occurrence. The probabilities that occur in actual operation, specify the operational profile. Obtaining an operational profile requires dividing the input space into sufficiently small leaf partitions, and then estimating the probabilities associated with each leaf partition. A subspace with high probability may need to be further divided into smaller subspaces.

Software Development/Test Phases

A competitive and mature software development organization targets a high reliability objective from the very beginning of software development. Generally, the software life cycle is divided into the following phases. As we will see later, different testing related tools may be required for different phases.

A. Requirements and definition: In this phase the developing organization interacts with the customer organization to specify the software system to be built. Ideally, the requirements should define the system completely and unambiguously. In actual practice, there is often a need to do corrective revisions during software development. A review or inspection during this phase is generally done by the design team to identify conflicting or missing requirements. A significant number of errors can be detected by this process. A change in the requirements in the later phases can cause increased defect density.

B. Design: In this phase, the system is specified as an interconnection of units, such that each unit is well defined and can be developed and tested independently. The design is reviewed to recognize errors.

C. Coding: In this phase, the actual program for each unit is written, generally in a higher level language such as C or C++. Occasionally, assembly level implementation may be required for high performance or for implementing input/output operations. The code is inspected by analyzing the code (or specification) in a team meeting to identify errors.

D. Testing: This phase is a critical part of the quest for high reliability and can take 30 to 60% of the entire development time. It is generally divided into these separate phases.

Unit test: In this phase, each unit is separately tested, and changes are done to remove the defects found. Since each unit is relatively small and can be tested independently, they can be exercised much more thoroughly than a large program.

Integration testing: During integration, the units are gradually assembled and partially assembled subsystems are tested. Testing subsystems allows the interface among modules to be tested. By incrementally adding units to a subsystem, the unit responsible for a failure can be identified more easily.

System testing: The system as a whole is exercised during system testing. Debugging is continued until some exit criterion is satisfied. The objective of this phase is to find defects as fast as possible. In general the input mix may not represent what would be encountered during actual operation.

Acceptance testing: The purpose of this test phase is to assess the system reliability and performance in the operational environment. This requires collecting (or estimating) information about how the actual users would use the system. This is also called alpha-testing. This is often followed by beta-testing, which involves actual use by the users.

Regression testing: When significant additions or modifications are made to an existing version, regression testing is done on the new or "build" version to ensure that it still works and has not "regressed" to lower reliability.

E. Operational use: Once the software developer has determined that an appropriate reliability criterion is satisfied, the software is released. Any bugs reported by the users are recorded but are not fixed until the next release.

Software Test Methodology

To test a program, a number of inputs are applied and the program response is observed. If the response is different from expected, the program has at least one defect. Testing can have one of two separate objectives. During debugging, the aim is to increase the reliability as fast as possible, by finding faults as quickly as possible. On the other hand during certification, the objective is to assess the reliability, thus the fault finding rate should be representative of actual operation. The test generation approaches can be divided into the classes.

A. Black-box (or functional) testing: When test generation is done by only considering the input/output description of the software, nothing about the implementation of the software is assumed to be known. This is the most common form of testing.

B. White-box (or structural) testing: In this approach the actual implementation is used to generate the tests. While test generation using the white-box approach is not common, evaluation of test effectiveness often requires use of structural information.

Coverage Measures

The extent to which a program has been exercised can be evaluated by measuring software *test coverage* [mal95]. Test coverage in software is measured in terms of structural or data-flow units that have been exercised. These units can be statements (or blocks), branches, etc. as defined below: Some popular coverage measures are often referred to by using a compact notation, these are given the parenthesis.

Statement coverage (C0): the fraction of the total number of statements that have been executed by the test data.

Branch coverage (C1): the fraction of the total number of branches that have been executed by the test data.

C-use coverage: the fraction of the total number of computation uses (c-uses) that have been covered during testing. A c-use pair includes two points in the program, a point where the value of a variable is defined or modified followed by a point where it is used for computation (without the variable being modified along the path).

P-use coverage: the fraction of the total number of predicate uses (p-uses) that have been covered during testing. A p-use pair includes two points in the program, a point where the value of a variable is defined or modified followed by a point which is a destination of a branching statement where it is used as a predicate (without modifications to the variable along
the path).

It has been shown that if all paths in the program have been exercised, then all p-uses must have been covered. This means that all-paths coverage requirement is stronger than all-p-uses. Similarly all-p-use coverage implies all-branches coverage and all-branches coverage implies all-instructions coverage. This is termed the *subsumption hierarchy*.

Module coverage (S0): the fraction of the total number of modules that have been called during testing. A module is a separately invocable element of a software system, sometimes also called procedure, function, or program.

Call-pair coverage (S1): the fraction of the total number of Call-pairs that have been used during testing that have been used during testing. A call-pair is connection between two functions in which one function "calls" (references) the other function.

Path coverage: the fraction of the total number of paths that have been used during testing. A path is any sequence of branches taken from the beginning to the end of a program. To achieve 100% path coverage, all permutations of paths must be executed.

Tools for different phases are examined below. Some tools are applicable to multiple phases. Some of the types of tools are now widely used, others have just started to emerge.

Requirements phase Tools

Requirement Recorder/Verifier:

Requirements can be recorded informally in a natural language like English or formally using Z, LOTOS, etc. Use of formal methods results in a more thorough recording of requirements. The requirement information needs to be unambiguous, consistent and complete. A term or an expression is unambiguous if it has one and only one definition. A requirements specification is consistent if each term is used in the same manner for each occurrence. Completeness implies presence of all needed statements, and all required components for each statement. The requirement verifiers can automatically check for ambiguity, inconsistency and completeness of statements. However, they cannot determine that the set of requirement statements is complete. This would require review by human testers. A requirements recorder may also assist in specification-based test case generation.

Test Case Generation:

Automatic test case generation can be an extremely important part of achieving high reliability software. Manual test care generation is a slow and labor intensive process and may be insufficient if not done carefully. Arbitrarily generated tests can find defects with high testability relatively easily; however, these tests can become ineffective as testing progresses. Specification-based test generation can ensure that the different test cases cover at least some different functionality by partitioning the functionality and probing the portions. Either the input-space or the state space may be partitioned. Poston [pos98] classifies the strategies used as active driven (to test for missing actions), data driven, logic driven, event driven and state driven. Both Validator (Aonix) and T-VEC (T-VEC) include specification verification and test case generation.

Orthogonal to the test generation strategy is question of test vector distribution. The distribution may be chosen to confirm with the operational profile, so that the tests replicate the normal operation. On the other hand, the strategy, at reach step, may choose to probe the functionality that has been relatively untouched by testing so far. The second approach may be implemented in the form of antirandom testing [yin97]. A combinatorial design based test generation can significantly reduce the number of combinations to be considered. This is the approach used in AETG (Bellcore) [coh97]

It is also possible to generate tests using the software implementation formation. Some tools can use this approach termed “white-box” testing. Such test generation can require enormous amounts of computation, and thus should be considered only for branches, p-uses, etc. which are very hard to test otherwise. Beizer has called such testing “kiddie testing at its worst.” Such tests cannot detect missing functions [pos98].

Programming Phase Tools

These are often called “static” tools, because they do not involve actual execution of the software.

Metrics Evaluators:

Many metrics have been used to evaluate aspects of complexity of programs. They include lines of code, number of modules, operands, operators or data/control flow measures. The belief is that if a module is more complex, it is more fault-prone and thus deserves special attention. It has been shown that many metrics are strongly correlated to the number of lines of code, and may not provide any information [ros97]. Still when the resources and time are limited, it may be a good strategy to identify the fault prone modules. Poston regards such tools to be “nice” but not essential.

Code Checkers:

These are also static tools like metric evaluators. These tools look for violations of good programming practices to generate warnings. They can identify misplaced pointers, uninitialized variables and non-conformance to coding standards Testing Phase Tools. STW/Advisor (Software Research) includes both code checking and metric evaluation.

Inspection Based Error Estimation:

A design document or code can be inspected. Many defects can often be detected simply by inspection. If separate teams or individuals do inspection independently, it amounts to sampling the defects present. Statistical methods are available that can be used to obtain a preliminary estimate of the remaining number of bugs remaining [ebr97].

Testing Phase Tools

These tools were the earliest to appear and are now widely used. They are often termed “dynamic” because they involve actual execution of the software using the test cases selected and evaluation of test thoroughness.

Capture-Playback Tool:

This is somewhat like a VCR or perhaps more closely like recording and running spreadsheet macros. Older capture-playback tools are worked at the bit-map level. Modern tools can capture and replay information at bit-map, widget, object or control levels. Information captured can be edited to replace hard coded values and path name to make them more general by passing setting environment variables and passing parameter values. One can build a library of small test scripts which can be combined to obtain different test sequences. A test sequence can be implemented by using a state table as a driver.

An alternative is to have data driven scripts that input data as well as parameters and environment variables. Using appropriate data values, the same test scripts can be made to cover different functionalities of the program. The data files can also contain the expected results for

specific test cases, e.g. success or failure. Most capture-replay tools today incorporate a comparator, which compares the expected and actual outputs. QA Partner (Seague) and WinRunner/Xrunner (Mercury Interactive) are examples of this class of tools.

Memory Leak Detectors:

Modern programming practices use dynamic memory allocation. If a program fails to deallocate memory that is no longer being used, it keeps on reserving more and more of the memory to itself, until eventually it runs out of memory. Such memory leaks can be detected by tools like BoundsChecker (Relational Software) or Purify (Purify).

Test Harness:

A software under test needs to interface with a capture-replay tool as well as a data-base system and perhaps with other systems also. These interfaces also need to be tested. Such a test execution environment is termed a test harness. They may include "stubs" to stimulate missing parts. In the past, test harnesses have been custom built. Some test harness generators like Cantata (Information Processing) have recently become available

Coverage Analyzers:

It is impossible to test a program exhaustively. The testers must decide if a program has been exercised sufficiently thoroughly. One way is to use a coverage analyzer which will keep track of the fraction of all structural or data-flow units that have been exercised. It has been shown that coverage measures are approximately linearly correlated with the defect coverage [mal94].

Most analyzers are intrusive. They "instrument" the code by inserting test probes in the software, before it is compiled. Instrumenting affects the performance slightly. Non-intrusive analyzers are a much more expensive alternative; they collect information using a separate hardware processor.

Statement coverage is not a rigorous measure even with 100% coverage, the residual defect density ca still be high. Branch coverage is a popular measure. Sometimes a threshold value, say 85% branch coverage is used. Pure coverage is stricter than branch coverage and is suitable for high reliability programs. Module coverage and call-pair coverage are common system level coverage measures. At the present time use path coverage is feasible only if its definition is revised to reduce the total path count. GCT (Testing Foundations) and
ATAC (Bellcore) are coverage analyzers.

Load/performance tester:

These tools allow stress testing of client/server applications, which are often expected to work correctly under high loads. SQA LoadTest (SQA) allows stress testing of Windows client/server applications, Final Exam Internet Load Test (Platinum) is specifically for web applications.

Bug-tracker:

A bug-tracker records the status of each bug found. Depending on the strategy used, a bug may or may not be fixed immediately after it is found. A bug-taker is basically a data-base tool.

Reliability Growth Modeling tools:

As defects are found and removed, the reliability of the program increases. This is manifested by a gradual decline in the defect finding rate. A wealth of methods is available that use software reliability growth models (SRGMs). Several SRGM tools are available that have these features [lyu96]:

1. Pre-processing or smoothing of data
2. Estimating parameters of a selected SRGM
3. Answering queries like how much larger the software needs to be tested

SMERFS (NSWRC) is a popular SRGM tool. ROBUST (CSU) allows parameters of SRGMs to be estimated even before testing begins, which can be useful for preliminary resource planning.

Coverage based Reliability Tools:

Recently, a model describing defect coverage and test coverage has been proposed and validated. The model tends to fit the data quite closely and can yield very stable estimates of the number of residual defects [mal98]. ROBUST (CSU) allows coverage to be used as the stopping rule criterion. It also allows stable estimation of the number of residual defects [mal98].

Identifying Tools Needed

The software testing tools can be expensive. The cost to license a tool can be just a fraction of the overall cost. The testers need to understand the tools and become familiar with them. The use of the tools needs to be incorporated in the process.

Poston [pos98] regards these to be the essential tools at most development organizations ought to have. He terms them the “Big 3” tools.

1. Requirement recorder/test case generator
2. Test execution tool
3. Test evaluation tool

He terms some of the other tools as “nice to have” and considers structure-based test generation tools to be useless.

Not all projects need sophisticated tools. Many can significantly benefit by using some of the simpler tools. One good approach to identify the tools needed is to consider the quality required in

the final project. A measure of quality called *TestWorks Quality Index* has been defined by Software Research [twq98]. It is composed of 10 additive factors as given in Table 2 below.

Table 2: TestWorks Quality Index [twq98]

F#	Evaluation Factor	50 Points	60 Points	70 Points	80 Points	90 Points	100 Points	My Score on This Factor
F1	Cumulative C1 for All Tests	>25%	>40%	>60%	>85%	>90%	>95%	
F2	Cumulative S1 for All Tests	>50%	>65%	>80%	90%	95%	95%	
F3	Functions with Cyclomatic complexity E(n) <20	<25%	>25%	>50%	>75%	>90%	>95%	
F4	Percent of Functions with “Clean” Static Analysis	<20%	>20%	>30%	>40%	>50%	>60%	
F5	Last PASS/FAIL Percentage	>25%	<25%	>50%	>75%	>90%	>95%	
F6	Total Number of Test Cases/KLOC	<10	>10	>15	>20	>30	>40	
F7	Calling Tree Aspect Ratio (Width/Height)	>1.0	<1.25	<1.5	<1.75	<2.0	>2.0	
F8	Number of OPEN Criticality 1 Defects/KLOC	>5	<5	<3	<2	<1	<0.5	
F9	Path Coverage Performed for % of Functions	<1%	>2%	>5%	>10%	>15%	>25%	
F10	Cost Impact Per Defect	>$100K	>$50K	>$25K	>$10K	>$1K	<$1K	
	TOTAL POINTS	->	->	->	->	->	->	

For example, a “quick and dirty” (but still useful) order tracker may have the Quality Index calculated as below.

Table 3: A “Quick and Dirty” Order Tracker [twq98]

Cumulative C1 (Branch Coverage) Value for All Tests	75
Cumulative S1 (Callpair Coverage) Value for All Tests	70
Percent of Functions with E(n) < 20	50
Percent of Functions with Clean Static Analysis	50
Last PASS/FAIL Percentage	80
Total Number of Test Cases/KLOC	50
Calling Tree Aspect Ratio (Width/Height)	60
Current Number of OPEN Defects/KLOC	50
Path Coverage Performed for % of Functions	50
Cost Impact/Defect	100
TOTAL POINTS SCORED	535
TestWorks Quality Index	53.5

On the other hand, a bedside cardiac monitor may be required to have a much higher Quality Index, as calculated below. It will require reliance on more powerful tools to achieve higher quality.

Table 4: Bedside Cardiac Monitor [twq98]

Cumulative C1 (Branch Coverage) Value for All Tests	100
Cumulative S1 (Callpair Coverage) Value for All Tests	100
Percent of Functions with E(n) < 20	80
Percent of Functions with Clean Static Analysis	80
Last PASS/FAIL Percentage	90
Total Number of Test Cases/KLOC	85
Calling Tree Aspect Ratio (Width/Height)	60
Current Number of OPEN Defects/KLOC	95
Path Coverage Performed for % of Functions	60
Cost Impact/Defect	50
TOTAL POINTS SCORED	800
TestWorks Quality Index	80

Sources of information

Here major sources of detailed information on software testing related tools are listed.

Web sites:

1. Testing Tools Supplier List, http://www.stlabs.com/MARICK/faqs/tools.htm, includes information and links to a very detailed list of tools classified into test design Tools, GUI test drivers and capture/replay tools, load and performance tools, non-GUI test drivers and test managers, other test implementation tools, test evaluation tools, static analysis tools and miscellaneous tools.

2. Testing and Test Management Tools, http://www.methods-tools.com/tools/frames_testing.html, another detailed list of tools.

3. RST Hotlist, http://www.rstcorp.com/hotlist.html

4. STORM Software Testing On-line Resources, http://www.mtsu.edu/~storm/

5. SR/Institute's Software Quality HotList, http://www.soft.com/Institute/HotList/index.html

6. Software Testing Hotlist, http://www.io.com/~wazmo/qa.html

7. Papers by Cem Kaner, http://www.kaner.com/writing.htm

8. The Testers' Network, http://www.stlabs.com/testnet.htm

Books:

1. Software Design, Automated Testing, and Maintenance: A Practical Approach by D. Hoffman and P. Strooper (1995). International Thomson Computer Press. London.

2. Ovum Evaluates: Software Testing Tools by Graham Titterington. Ovum Limited, London.

3. Automating Specification-Based Software Testing by R. M. Poston (1996). IEEE Computer Society Press, Los Alamitos.

4. The Automated Testing Handbook by Lina G. Hayes (1995). Software Testing Institute.

Conferences:

1. IEEE High-Assurance Systems Engineering Workshop
2. International Conference on Computer Safety, Reliability and Security
3. International Conference on SOFTWARE MAINTENANCE.

4. Metrics - International Symposium on Software Metrics
5. ISSRE - International Symposium on Software Reliability Engineering

References

[mcg97] Tom McGibbon, "An Analysis of Two Formal methods VDM and Z, http//www.dacs.dtic.mil, Aug. 13, 1997.

[bw91] Business Week, "The Quality Imperative", special bonus issue, Fall 1991.

[pos98] Robert Poston, "A Guided Tour of Software Testing Tools," Aonix, March 30, 1998.

[qtc97] "Results from August's Survey on Automated Testing", Quality Tree, http://www/qualitytree.com/survey/august/results.htm, 1997.

[qaq95] QA Quest, The Newsletter of the Quality Assurance Institute, Nov. 1995.

[yin97] H. Yin, Z. Lebne-Dengal and Y.K. Malaiya, "Automatic Test Generation using Checkpoint encoding and Antirandom Testing", Proc. Int. Symp. Software Reliability Engineering, 1997, pp. 84-95.

[coh97] D.M. Cohen, S.R. Dalal, M.L. Fredman and G.C. Patten, "The AETG System: An Approach to Testing based on Combinatorial Design", IEEE Trans. Software Engineering, July 1997, pp. 437-444.

[ros97] J. Rosenberg, "Some Misconceptions about Lines of Code", Proc. Int. Software Metrics Symp., Nov. 1997, pp. 137-142.

[ebr97] N.B. Ebrahimi, "On the Statistical Analysis of the number of Errors Remaining in a Software Design Document after Inspection", IEEE Trans. Software Engineering, Aug. 1997, pp. 529-532.

[mal94] Y.K. Malaiya, N. Li, J. Bieman, R. Karcich and B. Skibbe, "The Relationship between Test Coverage and Reliability", Proc. Int. Symp. Software Reliability Engineering, Nov. 1994, pp. 1867-195.

[lyu96] M.R. Lyu Ed., Software Reliability Engineering, McGraw-Hill, 1996.

[mal98] Y.K. Malaiya and J. Denton, "Estimating Defect Density using Test Coverage", Colorado State University Tech. Report CS-98-104.

[twq98] TestWorks Quality Index: Overview, Software Research Application Note, 1998.