

Mutating Automated Tests

STAR East 2000

Douglas Hoffman

Copyright © Software Quality Methods, LLC., 2000.

All rights reserved.

Keywords: Automated Testing, Mutating Tests, Non-deterministic Tests, Pseudo Random Numbers, Test Oracles

Key points attendees take away:

- Benefits and shortcomings of automated tests
- Design approaches for creating more powerful automated tests
- How organizations can evolve to support these more powerful tests
- Types of automated tests that are easy or hard to vary
- Some methods to improve the value of some automated tests
- Examples of non-deterministic automated tests

Summary

Most automated tests are used as regression tests – doing the same exercises each time the test is run. This paper and talk describe a powerful type of automated test – one that does something different each time it runs. These tests can augment traditional manual and automated regression tests to expose unexpected software under test (SUT) behaviors. The paper goes into the organizational issues and typical organizational evolution that are precursors for these tests and presents the idea of mutating tests. The approach doesn't apply to all situations of automated tests, but the author presents some pros and cons for mutating automated tests and provides several examples based on experience.

Background

One of the limitations of most automated tests is that they are generally less likely to uncover latent defects than equivalent manual tests. This stems from two factors: a test is most likely to find a software defect the first time it is run and a person running a test is able to perceive unexpected behaviors for which an automated verification doesn't check. The first factor occurs because software doesn't wear out, so a program should do the same thing each time it is given the same inputs, especially when the inputs are provided by an automaton.

Mutating Automated Tests

STAR East '00

Douglas Hoffman
Software Quality Methods, LLC.
24646 Heather Heights Place
Saratoga, California 95070-9710
Phone 408-741-4830
Fax 408-867-4550
doug.hoffman@acm.org

Copyright © 2000, Software Quality Methods, LLC. No part of these graphic overhead slides may be reproduced, or used in any form by any electronic or mechanical duplication, or stored in a computer system, without written permission of the author.

Douglas Hoffman Copyright © 2000, SQM, LLC. 1

The first factor is amplified because a person provides the input and evaluates results for manual tests, while automated tests use programs to do the work. A person will not do exactly the same thing the same way even when they try, while an automaton will tend to do exactly the same thing every time. A person running manual tests can easily vary the test exercise and evaluate the responses of the SUT. One has almost no expectation of finding a defect with the typical automated test unless a new defect has been put in since its last running. Manually rerunning tests introduces new variations and exercises, improving the likelihood of finding new problems even with an old test.

The second factor is usually a powerful plus for manual tests; a person may notice a flicker on the screen, an overly long pause before a program continues, a change in the pattern of clicks in a disk drive, or any of dozens of other clues that an automated results check would miss. The author has seen automated tests “pass” and then crash the system, device, or SUT immediately afterwards. Although not every person might notice these things and any one person might miss them sometimes, an automated test only verifies those things it was originally programmed to check. If an automated test isn’t written to check timing, it can never report a time delay.

Automated tests typically come from manual exercises, so the first time an automated test is run is not the first time SUT performs the test exercise. Even when an automated test is built and run without having been first run manually, the first automated run is more likely to find defects than subsequent runs. Because of the automation, the exercise very closely repeats itself every time.

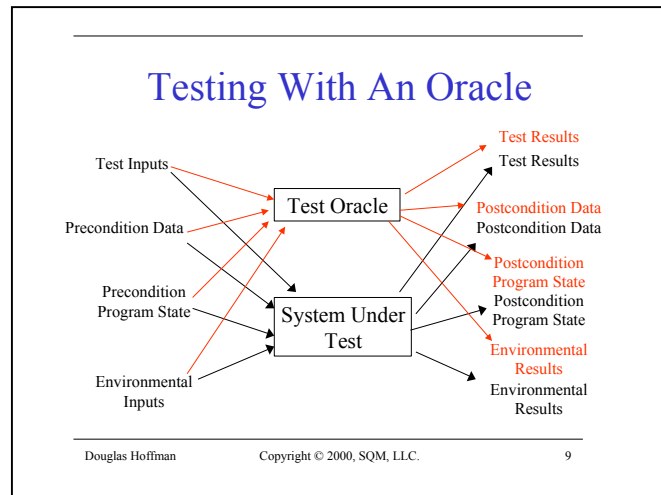
This is not to say that automated tests aren’t useful or powerful. An automated test regains some power to find problems when the software changes. Manually rerunning the same tests each time anything changes is time consuming and boring for people and, so far, the machines haven’t complained about doing it. People are also easily trained about what to expect from a test and can cognitively miss seeing errors after only a few repetitions. Thus, the first place usually considered of for automation is regression testing.

The term “automated software tests” has many different meanings, depending upon the speaker and context. For the purposes of this paper, an automated software test is a test with the six characteristics shown in **Slide 2**. The test consists of performing some exercise of the SUT, observing some results, comparing them with expected result values, and reporting the outcome.

Automated Software Tests

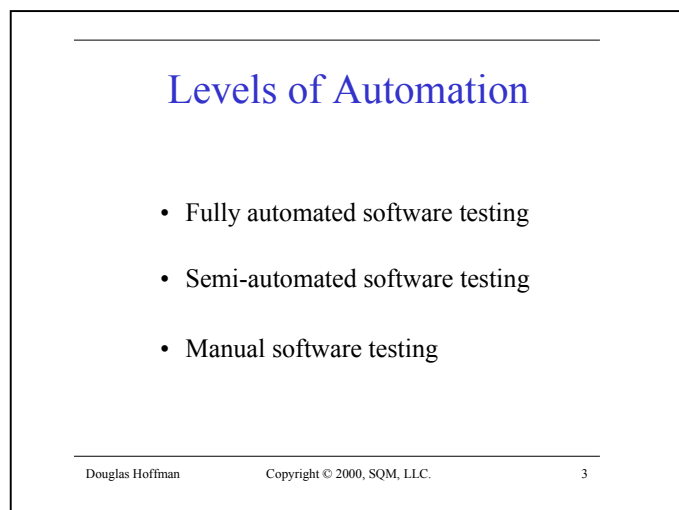
- No intervention needed after launching tests
- Automatically sets-up and/or records relevant test environment
- Runs test exercise
- Captures relevant results
- Evaluates actual against expected results
- Reports analysis of pass/fail

Software testing is difficult and automated software testing is much more difficult. In both cases one must choose or develop good test exercises for the SUT. The tester is an extremely powerful and flexible computational engine for manual testing. In an automated test the exercise is built into the automaton, including predicting and comparing the results. This leads to a situation in software regression testing that James Bach describes as “playing twenty questions with all the questions written down in advance.” Some of the biggest difficulties in software test automation are in knowing what results are expected from the SUT. **Slide 9** illustrates the scope of actual inputs and results in a software test. There are many issues with the huge number of potentially relevant results and how to record them. Often, it is extremely difficult to predict what the SUT should do and what outcomes are expected. Although the test designer typically is only conscious of the values directly given to the SUT, the SUT behavior is influenced by its data, program state, and the configuration of the system environment it runs in.



In order to check the results, all inputs to the SUT must be tracked and some means of generating a prediction of the resultant behaviors provided for some or all of the same dimensions. In a manual test, the tester usually controls or checks preconditions and inputs, or can quickly adjust the system when they encounter unexpected results. An automated test must rely upon the test design and system setup to control the important inputs. It must also include some mechanism for knowing or getting the expected results (typically from an oracle). Regardless of the exercise of the SUT in a test, an automated test will be poor with poorly selected relevant inputs or results, with poor results oracles, or if there is limited visibility into the relevant values.

There is a broad spectrum of levels of software test automation. At one extreme, there may be no automation involved. A person will perform the test exercise, observe the results, determine what should have happened, and draw the conclusions about the SUT. At the other extreme there are tests that run and self verify completely automatically (assuming no anomalies are detected in the SUT). In between, a manual tester can get many automated assists. The optimum level of test automation is dependent on the specific situation.



Advantages and Drawbacks of Automation¹

In spite of the difficulties involved with creating good automated tests, there are clear advantages in many circumstances. **Slides 4 and 5** list many of the initial advantages and disadvantages gained from automating existing tests. Additional advantages and challenges from more advanced automation approaches (second generation automated tests and beyond) are discussed below.

Most of the initial advantages from test automation are derived from the standardization and discipline required for automation. Automating software testing requires a more formal engineering approach than manual testing. This is a double edged sword, potentially improving the efficiency and effectiveness but requiring different (and usually more technical) job skills.

Some of the disadvantages stem from financial and time consumption costs, while others are related to the cultural changes an organization undergoes. The financial costs involved include direct costs for acquiring or creating and then maintaining the automation mechanisms and longer times for test design and implementation. The cultural changes occur due to the changes in the roles for test designers and testers, and the new testing paradigms.

The initial costs and benefits are generally something organizations incur in the course of introducing test automation. In the author's experience, most organizations go through several stages in creating their automated test suites. It helps when explaining the concepts to view more powerful testing as an evolution, although it is not necessary to go through such a progression.

Generations of Automation

The initial decision to automate testing usually revolves around either integration/build tests or regression testing. In the former case the automated tests have the special purpose of providing a quick check of an automated software build process. These tests are usually not very useful as a

¹ For a more complete treatment of the issues, see Cem Kaner's paper from Quality Week 1997, "Improving the Maintainability of Automated Test Suites."

Advantages To Automation

- Repeatable
- Faster running
- Reusable components
- Standardized formats
- Easy to generalize
- Better environment control

Douglas Hoffman Copyright © 2000, SQM, LLC. 4

Disadvantages To Automation

- Requires test tools
 - Expensive
 - Confine the paradigm
- Time consuming test creation
- Does the same thing each time
- Limit possible observations
- Tests and tools require real maintenance

Douglas Hoffman Copyright © 2000, SQM, LLC. 5

foundation for more general automated tests, although they may provide an excellent example of a simple automation architecture. Automated regression tests are more likely to provide the foundation for more advanced automation. Regardless, the initial tests are usually automated versions of manual test exercises.

The author views this initial automation as the first generation. **Slide 6** describes some of the characteristics of this level of software test automation. This is where the main investments in the infrastructure to support test automation take place. Unfortunately, many organizations fail to manage a coherent strategy at this level and invest heavily to make a marginal architecture work or they abandon automation altogether.


Although there is nothing wrong with first generation automated tests, the organization has to gain enough experience to get past the view that automated tests are like manual tests in order to evolve beyond this level of automation. **Slide 7** shows some of the ways that these automated tests can be improved.

When a tester manually runs a test, the tester makes continuous observations about correct SUT behaviors. Even when result comparisons are done after a test is run, the tester can see immediately when the behavior of the SUT deviates radically from expected. For automated tests, this is only true in if explicit mechanisms to check behaviors are designed in, and even then, the checking is limited to those factors actually checked.

An automated test can often be expanded beyond what a manual test can do by generalizing the exercise. Where a person finds trying all of the combinations of configuration values to be tedious and slow, an automated test can walk through the combinations at machine speeds. Likewise, trying all the fields on a screen or all the screen transitions may be straightforward for an automated test, but difficult or impossible for a human tester.

Handling negative test cases is much more difficult for automated tests because the SUT may react in unusual or unspecified ways when given invalid or conflicting inputs. Where a human tester can fairly easily analyze the acceptability of the response and formulate a recovery strategy, these things are very difficult to build into automated tests.

First Generation Automation

- Automate existing tests by creating equivalent exercises
- Small improvements 
- Test scripting
- Reuse test components
- Hard coded oracles

Douglas Hoffman

Copyright © 2000, SQM, LLC.

6

The Power Of Tests

- Self verifying tests
 - Check results ASAP in the test
 - Dump data only on errors
- More general exercises
 - All alternatives in turn
 - “Walk the tree” approach
- Include positive and negative cases

Douglas Hoffman

Copyright © 2000, SQM, LLC.

7

Second generation automation continues beyond these improvements with exercises that are even more powerful and analysis. More emphasis can be placed on generation of expected results by both modeling the SUT behavior and extending the characteristics and values being checked. Tests can cover more situations by using better oracles to check SUT behavior.

Once the framework is established, the first generation tests can be extended as illustrated in **Slide 8**. The test exercises can be expanded to more exhaustively cover input variation, broaden the scope of coverage, and increase the intensity of activities using facilities like looping and parallel threads. Different aspects of the SUT can be emphasized and analyzed using specialized tools to identify test conditions and generate test data.


The focus of this paper is on creating more powerful exercises that, unlike most regression tests, vary conditions each time they run. The second generation tests take advantage of automation to increase the coverage while doing more thorough verification. Verification can expand into intermediate results and internal program data and state information, or develop into diagnostics that explore and isolate errors when unexpected behaviors are detected.

The second generation may increase the frequency, intensity, or duration of automated test activities to find certain classes of errors. Speeding up the execution or increasing the number of parallel activities can increase the frequency. Generating extreme values and extreme combinations increases intensity. Duration is increased by running the SUT with typical inputs for extended periods (load or life testing).

With improved verification it becomes practical to expand tests to exercise broader areas of the SUT, while checking for deviations in the SUT behaviors. Without such verification, broad tests tend to find unreproducible defects or ones extremely difficult to identify and isolate.

The second generation improvement of significance to the concept of mutating automated tests is the use of pseudo random numbers to decide test behaviors. (Pseudo random in that the sequence of values is random, but repeatable based upon a seed value. For any given test the seed can be randomly selected and saved so the test can be repeated if need be.) This is particularly useful

Second Generation Automation

- Automated oracles 
- Exhaustive/extensive/intensive exercises
- Auto generated tests and data
- More powerful exercises
- Random selections among alternatives

Douglas Hoffman Copyright © 2000, SQM, LLC. 8

More Powerful Exercises

- Increase the number of combinations
- Self-verifying tests and diagnostics
- More frequency, intensity, duration
- Increase the variety in exercises

Douglas Hoffman Copyright © 2000, SQM, LLC. 10

when the input domain is larger than practical for exhaustive testing or when only a small number of combinations can be exercised in a single test. By randomly choosing values, the test improves the possibility of encountering new conditions in the SUT, and thus increasing the opportunity for finding undetected problems. These tests try to expose unexpected SUT behaviors that a tester might never think of.

It is important to understand the constraints for tests using random values. Noel Nyman's smart monkeys² are simple examples of such automated tests. In their earliest formation, these

are second generation mutating tests. They require some model for input behavior, whether hard coded or read into the test as data. More importantly, they require some mechanism for determining whether or not the behavior of the SUT is expected. The oracle must be able to deal with predicting SUT behaviors for any (and therefore all) inputs.

One of the most difficult constraints on these tests is designing the test recovery behavior for when the unexpected happens. The SUT and the system don't always behave well in the face of a test that doesn't. As test behavior becomes more sophisticated and more varied, the test has to become ever more capable of handling responses. Even in those rare situations when all negative case behaviors are specified, erroneous behaviors in the SUT do not necessarily follow specified rules. In order for the automated test to report it's findings it has to be able to recognize and survive almost any response. Often the test environment itself has to be made robust enough to recognize and report when the SUT or system aborts. The test has to handle arbitrary errors that may crop up at any time.

Whether or not randomness is used in second generation tests, third generation tests can be even more powerful by using knowledge and visibility into the SUT and system. These tests may look nothing like manual tests, as they take advantage of the internal characteristics of the SUT and system environment.

These automated tests take advantage of standard or special hooks to control and monitor SUT and system behaviors. A test that programmatically checks for

Random Selection Among Alternatives

- Partial domain coverage
- Small number of combinations
- Requires an oracle for verification
- Pseudo random number generation

The beginnings of mutating tests!

Douglas Hoffman

Copyright © 2000, SQM, LLC.

11

Third Generation Tests

- System instrumentation
- Multi-threaded tests
- Fall back compares
- Heuristic oracles
- Diagnostics



Douglas Hoffman

Copyright © 2000, SQM, LLC.

12

² Nyman, Noel, "Using Monkey Test Tools," Software Testing and Quality Engineering Magazine, Vol. 2, Issue 1, Jan/Feb 2000

memory leaks is one example. The instrumentation may be simple, publicly available API software calls or it could involve special hardware and software instrumentation. The important factor for these tests is that they (and probably the test management system) are relying on the instrumentation to monitor and/or control SUT behaviors. These tests are able to set and detect conditions that may be impossible to do manually.

The third generation may also include multithreaded tests; ones that have interacting portions that run simultaneously. These are often more complex than load tests, as the various threads monitor and adjust their behaviors based on the behavior and status of the SUT and other threads. Dynamic conditioning of the SUT can be monitored and adjusted by the tests at machine speeds, creating dynamic test conditions far beyond a person's capabilities.

Oracles for the third generation tend to become more numerous and more complex. The same "answer" can often appear in many ways; printer output may come in the form of Adobe PostScript, a bitmap, or something in between. It is easy to create two different PostScript files that will render the same image, so simply finding a mismatch between the expected and generated PostScript files does not mean there is an error. The test designer must construct a second, fallback comparison using some other oracle. For example, the test could generate the bitmaps and automatically compare them. (One group designed and built an automation architecture that included five levels of fallback comparisons before resorting to calling for human assistance.) The obvious solution of printing the two files has the drawback of requiring a person to interpret the results, making it a semi-automated test.

Another frequent characteristic of third generation automation is the use of fuzzy comparisons, approximations, or heuristic oracles. It is often difficult to generate oracles for SUT behaviors, and it becomes even more of a chore to model internal behaviors to compare with observations from instrumented software. The heuristic techniques allow simpler oracles to screen SUT behaviors against. By combining this with fallback comparisons, these automated tests can quickly, automatically, and (more or less) accurately verify results.

Like fallback compares, tests in this generation may go much further than just exercising or testing the SUT. Where an exercise may be defined as providing inputs as stimulus to the SUT, and testing as adding verification of expected SUT behavior in response to the exercise, a diagnostic performs specific tests in response to unexpected SUT behavior in order to further identify the nature and scope of an error. A diagnostic test looks for errors and then performs additional tests based on the specific type of error encountered. An example would be a data communications test that responds to finding a message mismatch by checking the output and input buffers and program data to identify where in transit the data was changed. Diagnostic

Tools For Third Generation

- Software instrumentation
 - Observations
 - Controls
- Oracles and comparators
- Test execution control

Douglas Hoffman




Copyright © 2000, SQM, LLC.

13

level testing becomes necessary when the test complexity grows and the human visibility into the SUT behavior shrinks.

In this third generation automation there is the possibility of combining elements to create tests that go far beyond the regression test paradigm. The tests that use random selections may be called mutating automated tests because they do different, reasonably sophisticated exercises of an instrumented SUT and employ automated oracle techniques to manage verification of actual against expected results. They do different things each time they run and still verify proper SUT behaviors in the face of it.

Mutating Automated Tests

- Closely tied to instrumentation and oracles
- Using pseudo random numbers
 - Random selection from domains 
 - Nesting and looping 
 - With and without replacement 
- Positive and negative cases possible
- Drill down on error

Douglas HoffmanCopyright © 2000, SQM, LLC.14

Some of the most powerful of these tests include random selection of valid and invalid inputs, instrumentation, fallback comparisons, and diagnostic techniques to provide substantial amounts of data about SUT behavior. Sophisticated models of the SUT, interfaces, or data may be generated from formal specifications or source code and used as input to configure or condition the tests and instrumentation. By combining these elements it is possible to automatically generate tests that explore SUT responses to weighted random inputs, with oracles based on independent interpretation of specifications or code.

Some examples of such automated tests the author has created are listed in **Slide 15** and briefly described below. Since they were developed for different products, in entirely different environments, with different risks and priorities, and at different companies, they are not made up of the same elements.

- A set of rules was created for building records given a programmatically specified database layout to test a data base software engine. The test generates random add/edit/delete function calls that are performed on a test data base. Based on the test developers' knowledge of the database design, test data is constructed to describe the record link and database constructor information relative to each record and incorporated as data in fields within the records. The test covers positive and negative inputs, as the random value generation and verification includes both. When the test run is complete, each record in the test data base contains links and constructor information about itself. The test runs for a specified length of time, followed by

Mutating Tests Examples

- Data base contents
- Processor instruction sets
- Compiler language syntax
- Stacking of data objects

Douglas HoffmanCopyright © 2000, SQM, LLC.15

running a separate verification program that walks through the database records and verifies that the actual linkages correspond to the information stored in the records.

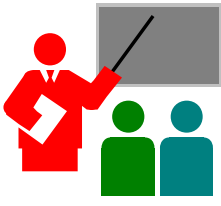
- A second example is a test for a computer processor. The test generates random, well formed instruction sequences to exercise the processor instructions and micro code. The instruction sequences are constructed to avoid circumstances such as clearing the registers or halting the machine. The test verifies the internal register values every 100 instructions. The instructions are then fed to a simulator as the oracle and to the actual processor. (The limiting factor being the simulator, since it is thousands of times slower than the processor itself.) Different test sets can be constructed with emphasis on different instruction types by changing the likelihood of generating the particular instructions.
- At another company, an automated test for a compiler uses weighted random, well formed syntax to generate programs to exercise the language syntax and semantics. The test compares the responses of the compiler under test with results from one of several other compilers, including an earlier version of the compiler and commercially available compilers for the same language. Syntax checking includes both positive and negative test cases. The semantic checks are primarily negative cases, where the various compilers report errors in the program, and for which fall back manual checking is often needed against commercial compiler responses.
- A fourth example tests nesting of objects in a desktop publishing system. By design, the system objects can contain other objects (e.g., a paragraph can contain a drawing, which in turn can have a table in it with a table within one cell and a paragraph in one cell of the inner table). The test uses syntax rules and weighted random object selection to generate and nest arbitrary objects within objects. A previously tested and released version of the product operates as an oracle, with the resulting documents being compared successively in a fallback scheme (starting from the desktop proprietary object file format, through file interchange formats and several printer output file formats) until there is a match or exhaust the oracle generated files.

Conclusion

There are techniques to make very powerful automated tests that don't just do the same thing every time. When used well, they result in better test coverage and detecting more defects. A solid foundation for such automated tests has to be formed including elements like models for input and results, results oracles, SUT instrumentation, and test execution controls. These tests use capabilities only available through automation to control and monitor SUT behavior and therefore cannot be duplicated manually.

Summary

- Automated tests can be powerful
- Static automated tests are unlikely to find defects
- More powerful automated tests cannot be duplicated manually



Douglas Hoffman Copyright © 2000, SQM, LLC. 16

Experience and qualifications

Douglas Hoffman, BACS, MSEE, MBA, CSQE
Software Quality Methods, LLC.
24646 Heather Heights Place
Saratoga, California 95070-9710
Phone 408-741-4830
Fax 408-867-4550
doug.hoffman@acm.org

Douglas Hoffman has over twenty-five years experience in software quality assurance and degrees in Computer Science, Electrical Engineering, and an MBA. He has been a participant at dozens of software quality conferences and has been Program Chairman for several international conferences on software quality. He has architected test automation environments and automated tests for several commercial systems and software companies, and has been an active participant in the Los Altos Workshops on Software Testing (LAWST).

He is an independent consultant with Software Quality Methods, LLC., where he teaches courses and consults primarily with Silicon Valley companies in strategic and tactical planning for software development and software quality. He is active as a Senior Member in the ASQ, participating in the Software Division, the Software Quality Task Group, and the ISO 9000 Task Group, and is a member of the ACM and IEEE. He was past Chairman of the Santa Clara Valley Software Quality Association (SSQA) and is currently the Chairman of the Santa Clara Valley Section of the ASQ. He was among the first to earn a Certificate from ASQ in Software Quality Engineering, and has been a registered ISO 9000 Lead Auditor.