Regression Testing in an Industrial Environment

Akira K. Onoma, Wei-Tek Tsai, Mustafa H. Poonawala, and Hiroshi Suganuma

Progress is attained by looking backward.

Issues such as test case revalidation, test execution, failure identification, fault identification, modification dependency, fault mitigation and test case dependency are essential for an industrial environment in applying regression testing. Most of these issues are easy to address if one is dealing with small programs, but in a large software house where multiple large programs are being developed and maintained, they suddenly become complicated and costly. Some of these issues are also related to general software testing, however, these problems become acute in regression testing because test cases are repeatedly exercised in case of regression testing.

This article discusses these problems and presents both management and technical techniques to address them. Some of these solutions are already being used by software developers, even though they may not be widely known. This article proposes a multilevel regression testing framework that can be easily adapted into a software development and maintenance process in which regression testing can play a key role in improving and ensuring consistent software quality. Our observations are based on our accumulated experience in software development and maintenance in the U.S. and Japan, including Hitachi Software Engineering (HSK).

Regression testing has been extensively studied by researchers [1, 3, 7, 10, 11] from a theoretical point of view. However, there has been little discussion on applying regression testing in industrial environments. While researchers are mostly concerned with reducing the number of test cases for retesting, there are other important issues in using regression testing in an industrial environment. Some of our observations include:

• Regression testing is used extensively. In fact,

other than functional testing (or black-box testing) and software inspection, regression testing is probably the most commonly used software testing technique.
- The frequent and extensive use of regression testing has led several companies to develop in-house regression testing tools to automate the process.
- In some companies, all existing test cases are rerun in regression testing. In other words, minimizing test cases for rerun has not been a critical issue for these companies. This is true for sites that develop real-time software (such as safety-critical applications) as well as sites that develop other kinds of software.

Even though regression testing is generally useful for software development and maintenance, it is especially useful for companies with one or more of the following characteristics:

- Companies developing a family of similar products by reusing products or test cases they had developed before;
- Companies developing mission-critical, safety-critical, or real-time systems because they need to test and retest their software frequently;
- Companies maintaining large programs over extended periods of time, such as 20 years, because regression testing can be used as a sanity check;
- Companies developing software that is under constant evolution as the market or technology changes;
- Companies developing software not using formal or semiformal processes used by those companies that develop code only;
- Companies that do not use software inspection as one of their quality assurance techniques.

Regression Testing Process in Practice

Different companies use different processes to develop and maintain software, such as the waterfall model and the spiral model. However, most of the companies use similar processes for software testing. Specifically, they use the following steps in regression testing:

Modification request: The software can be changed if a bug has been found or there is a change in specification or design.

Software artifact modification: The software artifacts—such as requirement documents, design documents and the code—must be changed to meet the new requirements or to remove bugs. Often the source code is the central focus [5].

Test case selection: Test cases must be selected to run regression testing. Sometimes test cases are revalidated at this time to ensure that the test cases are valid with respect to the changes. The goal is to obtain the right test cases instead of minimizing the number of test cases. Sometimes testers reuse all the existing test cases without any revalidation.

Test execution: After test cases are selected, they will be scheduled to run. Since the number of test cases is often large, this step is usually automated. Sometimes, test execution history, such as paths traversed and procedures called, is also recorded for each test case for future reference.

Failure identification by examining test results: Test results must be examined to see if the modified software behaves as expected. This is often done by comparing the test output with the expected output. If they are consistent, the requisite existing functionality may not be changed. If not, it is necessary to examine whether the test case, the code, or both are faulty. If test cases have not been revalidated earlier, they will be revalidated at this time, especially the ones that failed.

Fault identification: If the source code is suspected to be faulty, the programmer needs to examine the component of the software that caused the test case to fail. Fault identification can be a difficult problem if there are many modules with multiple versions and many modifications submitted for regression testing. It is necessary to identify precisely which components, versions, and modifications caused the failure.

Fault mitigation: Once the components that caused the failure are identified, programmers must mitigate the fault.

Test Case Revalidation. Test case revalidation is a largely manual activity aimed at identifying test cases that are no longer valid for the modified software. A test case consists of a test input and its expected output, and both must be examined during revalidation. In case the input is no longer valid, the test case may be discarded or considered as a negative test case [2]. It is possible that the test input is valid, but its expected output is no longer valid. In this case, the tester must redevelop the corresponding expected output. Even if both the input and expected output are valid for all the test cases, new test cases may need to be developed. This is true, for example, when a specific testing strategy is used.

Test case revalidation requires the tester to examine specifications, testing strategies and existing test cases. These are usually manual activities and can be quite expensive. Fortunately, test case revalidation can be efficient for functional testing if *traceability* between functional requirements and test cases is maintained.

Test case revalidation for white-box testing can be problematic because once the software is changed, its design is changed, and depending on the test techniques used to generate test cases, new test cases need to be developed and/or existing test cases eed to be changed to meet the coverage criteria.

Failure Identification by Examining Test Results. After the test results have been accumulated, they will be compared with their corresponding expected outputs. Automated tools are usually used to compare the results and log this information which is later inspected by the tester.

If the test cases used have been validated earlier, any significant deviation from the expected output indicates a potential software fault. If the test cases have not been validated, any test case failure may mean either the test case or the program is incorrect, or both.[1] As revalidation requires human intervention, this may take considerable time. Revalidation at this time usually requires examination of only the failed test cases. This is not a safe procedure, but it may reduce the time required for revalidation.

Fault Identification. After examining the failed test cases, the tester may declare the software has a bug. Then it is necessary to identify the software components that failed. Some development sites annotate test cases with requirement statements and the modules that the test cases are supposed to test. In this case, if a test case fails, the modules that are listed in the annotation are the potential targets. While this may help identify the faulty parts, this is not a fail-safe procedure. A group of modules may work perfectly in isolation, but when integrated with other components of the software, the integrated group may fail the same test cases that passed when the modules were tested in isolation.

It is possible to take advantage of different software versions that are available in identifying the faulty parts. Assume that a program has five components[2] A through E (see Figure 1), and through earlier testing there is a trusted version for each component. Assume that components A, B and C get changed while D and E remain unchanged. Further assume that the new software fails several test cases and the faulty component(s) need to be found. We may run the failed test cases again using the new component A with old, trusted components B, C, D and E to see if this configuration fails the test case. We can repeat the same experiments for component B (with old, trusted components A, C, D and E) and so on. In this way, we can determine whether any of the new components A, B or C is faulty.

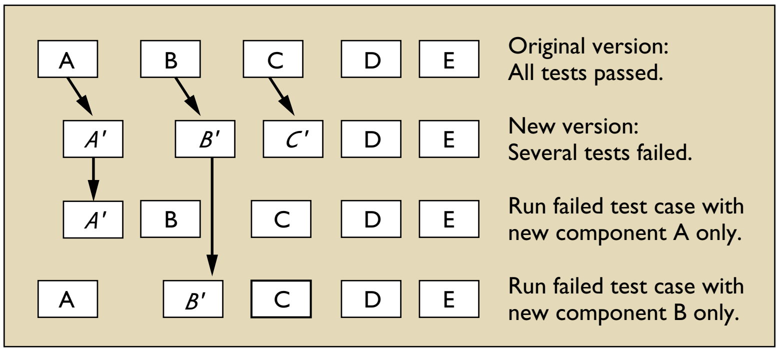

Figure 1. Fault identification

It is possible to repeat the test run to pinpoint the specific modules that failed the test cases. Although this process is not safe, it can be automated and has been found to be useful [8]. In fact, this is an application of group testing or combinatorial group testing [14].

Dependency. In practice, usually several people will be working on the same software components at the same time. It is even possible that multiple programmers will be working on the same part but with different versions, and each version depends on its earlier version. This is rather common in industry and often the parties involved may not know that they are working on the same programs or on programs that depend on each other. This phenomena can require special care in regression testing. Specifically, all the parties involved may submit their programs or modification requests[3] for evaluation, which is usually done by regression testing. If a fault has been found in a particular version of a module, it is necessary to remove all the modules and program modification cards (PMCs)[4] that depend on the particular module.

There are at least the following two kinds of dependency: in *module dependency* a software module depends on other modules because it uses other modules to

[1]A list of possible scenarios is available at www.cs.umn.edu/~tsai/regression.
[2]A component as used in the text can represent anything from a combination of modules to a single function. We use component to represent any of these, to illustrate the fact that this process can be applied at different levels of granularity.

[3]Modification requests are known by different acronyms in the industry—some of the more common are MOD, MR, and PMC. HSK uses the acronym PMC to represent modification requests, which we will use in this article.
[4]PMC is the acronym used by HSK to represent modification requests.

complete its task; *modification dependency* occurs when the software organization keeps track of PMCs.

Module dependency requires a reverse engineering tool to identify the dependence relationships. It is not only necessary to keep track of direct dependencies but also indirect dependencies.

A typical PMC contains the PMC ID, the module number, versions number, the line numbers of software that need to be changed, and the statements that will be incorporated in the change. Thus the PMC ID can be used to track modification dependency. We have seen some software groups that keep track of PMC dependency by asking the programming team to write down the dependency. However, this solution is not safe.

We have seen some software development groups use line numbers to track dependency. For example, suppose PMC 5 changed lines 2, 3, and 5, and PMC 7 changed lines 5 and 8, then PMC 7 depends on PMC 5. However this approach does not take care of direct and indirect data or control dependency. We have also seen some groups that keep track of dependency by time-stamping. For example, if PMC 5 is submitted before PMC 7, then PMC 7 depends on PMC 5. This is a conservative approach and may create unnecessary dependencies. This is not a safe approach either.

However, many practical approaches are proposed that are not safe but are still used in the industry as a safe approach may be theoretically impossible or its cost may be high.

Fault Mitigation. When a fault is detected in a software component, various actions may be taken to correct that fault. The most commonly used mitigation actions are:

- Submitting a new PMC to correct the fault. If we submit a new PMC to correct the fault, essentially we create a new PMC to correct an existing PMC. This new PMC should be annotated to indicate the fault that it is meant to correct.
- Removing the PMC that caused the fault. Instead of correcting the faulty PMC, we may just remove the PMC from the software. This approach is difficult in case of large number of dependencies on this PMC. As in this case, if the PMC is removed, all the dependent PMCs must be removed.
- Ignoring the fault. Ignoring a fault is not common but has been observed [8]. When the deadline is close, an engineer tends to say that some bugs are either not important or will rarely happen, thus there is no need to fix it. This is a dangerous practice, but it does happen. HSK depends on program managers to solve this problem. The program managers must give approval before a fault can be waived.

Regression Test Case Acquiring Strategies. Software companies use various means to acquire test cases. Most of the test cases developed in-house, and since they were acquired from third party vendors they can be subject to test case minimization. However, the following test cases are often not subject to minimization:

Application environment simulation: This is a popular and effective approach. Before shipping, a realistic application environment is built and often the potential users are invited to test the ready-to-release product for some period. These simulations are then used as test cases for subsequent releases. These test cases are often not subject to minimization as they represent the customers' view of the software.

Acceptance test or test cases supplied by users: Often, the users of a computer program provide a good set of test cases based on bugs encountered in practice. They report the bugs to the software developer and the bugs then become test cases. As customer satisfaction is of utmost importance, most software organizations will not skip any test cases supplied by the customers.

A large software house usually has a good history of testing—the tester knows which test cases are most likely to detect bugs and which other test cases rarely catch any bug. Those that frequently catch bugs should always be run in regression testing, while those that rarely catch any bugs may be subject to minimization.

Test Case Dependency and Sequencing. In practice, test cases are often not completely or properly defined. A set of test cases is considered properly defined if the input and output environment specification is complete. Sometimes it is not practical to specify the inputs and outputs of the test cases completely. Theoretically, it is always possible, but to do so may require excessive effort. This is because of the extremely large size of the input/output environments. This incompleteness may result in test case dependency and may cause the testing process to fail. In addition, incomplete output specification results in test cases being declared as passed when they may have affected the output environment in an undesirable manner.

A tester usually executes several test cases consecutively from one test driver. If the expected output of the first test case and the input of the subsequent test cases are not properly defined, the behavior of the subsequent test cases may be affected by the execution of the previous test case. This is test case dependency and will occur if the software is a state machine such as an object where values are stored as states. It is possible to systematically test a state machine but sometimes in large programs, it may be difficult to capture all states.

In regression testing, this test case dependency can introduce additional problems especially if we try to reduce the number of test cases for retest.[5] Many development and maintenance sites do not determine test case dependency due to the complexity and cost. In this case, it is important that all test cases should be rerun during regression testing.

Note that the presence of these dependencies among the test cases lead to the necessity that certain test cases be executed in a particular sequence. Many times the output from a test case sequence will not be proper if the individual test cases in the sequence did not execute in the required order.[6] The presence of test sequences also impacts the selection of test cases by selective retest. If any test case in a sequence is selected, then all the test cases in that sequence must be selected.

Regression Testing Process Cost Analysis

Here we emphasize the cost of testers rather than the cost of machine time. The cost of machine time may still be high and sometimes must be accounted for so that it can be properly charged.

- Time spent *developing* test cases to test the new functionality. Typically from about 10 minutes for each test case to a couple of hours or even weeks.
- Time spent *revalidating* the original test suite. Typically from few minutes to a few hours or days. However, due to large number of test cases, the total time is considerable. In addition, the necessity of revalidation makes revalidating a critical activity in the testing process. In case of functional tests, this revalidation is necessary only when the functional requirements change.
- Time spent in the *execution* of the test suite. The amount of machine time spent in executing a test suite could be large, especially if the suite requires a dedicated mode. However, as the executions are usually done during non-peak hours, the time spent in this phase is less critical than the time spent in the other phases. Thus, if this step is automated, the cost will not be too high.
- Time spent *comparing* the results obtained from executing all the test cases in the test suite. Typically a few minutes for each test case. However, again due to the immense number of test cases, the total time may be large. Often this process is automated, in this case the cost is low but manual inspection may be required depending on the extent of the revalidation performed.
- Time spent in *tracking* a failure to the appropriate module or modification (fault identification). Typically ranges from 10 minutes to a few hours. The amount of time spent in this phase also depends on the person performing the failure tracking.

If we have automated regression testing, the bottleneck will come from the inspection of the test results for the test cases that have failed and test case revalidation either at the time of selection or at the time of result comparison because they are predominantly manual activities.

Multi-Level Regression Testing

Large programs are usually developed in stages by teams of developers, testers and managers using a development model such as waterfall, or prototyping. Each large program is decomposed into components, and each component can be further decomposed. During the process, the software is being tested or inspected at various stages, such as at requirement stage, design stage. Testing is divided into unit testing, multiple levels of integration testing, functional testing, reliability testing, usage testing, stress testing, acceptance testing and field testing.

Regression testing should be used whenever there is any change in the software, and *it should be embedded in the software development and maintenance process.* It should not be an independent stage of a software development and maintenance process, instead regression testing should be performed at each stage whenever there is a change. For example, if a module has been changed, it must be submitted to unit regression testing before it is submitted for integration with other modules. This is a simply an application of divide-and-conquer strategy commonly used in software engineering. We call this approach Multi-Level Regression Testing (MLRT).

In MLRT, test cases may be run multiple times during the process because a test case designed for unit testing may be rerun again at an integration level. This is so because at the time the concerned module is linked with other modules its faults may be detected using exactly the same test cases for unit

[5],[6] An illustration of this is available at www.cs.umn.edu/~tsai/.

testing. Thus, some test cases may be rerun as a quality assurance procedure.

MLRT has many advantages. First, test suites can attached to each software component at different level of granularity. At the module level, test cases for unit testing will be attached. At an integration level, test cases for integration testing will be attached. This helps in configuration control. Also, multiple components can be tested concurrently at the same time by different groups of programmers, reducing the time required to perform regression testing.

Another major reason for practicing MLRT is that the delay in detecting faults is minimized. If a software component is submitted for integration with other components without thorough testing (including new functional testing and regression testing), its bug may be detected several weeks later by integration testing. If the fault is found during integration regression testing, the effort to correct the fault may have increased tremendously.

Regression Testing Tools

If a software group is interested in using regression testing, we recommend at least the following tools:

Test execution tool: This tool is a must because the number of test cases required to be run is enormous and it will be impossible without an automated test execution tool;

Test result comparator: This tool is helpful in identifying test failures and can save significant time and effort in regression testing;

Configuration tool: This tool will prove to be useful if it can track both software modules and their associated test cases, as well as software versions and software architecture;

Test management tool: This tool should keep track of status of testing including the failures identified so far, the faults identified so far, actions taken for those identified failures and faults, test case dependency and modification dependency.

For those companies that will use regression testing frequently, we recommend a group testing tool to identify the faulty components.

Conclusion

In this article, we presented several important issues regarding regression testing in an industrial environment. We notice that although regression testing is one of the most important and widely used testing strategies in the industry today, many of its issues have not been investigated by researchers. The issues discussed have a major impact on regression testing, quality assurance, configuration management, software development and maintenance processes. We also identified a testing technique that can be used in software testing for large applications (group testing). We also recommended some tools to automate the regression testing process.

We know of companies that are well-known for producing quality software but do not use software inspection or have a formal software development process. What is their secret in delivering quality software? Regression testing. We know of some companies where the machines work hard during weekends. What are their machines doing? Regression testing. C

References

1. Agrawal, H., Horgan, J.R., Krauser, E.W., and London, S.A. Incremental regression testing. In *Proceedings of the IEEE Software Maintenance Conference* (1993), pp. 348–357.
2. Beizer, B. *Software Testing Techniques.* Van Nostrand Reinhold, New York, 2d. ed., 1990.
3. Chen, Y.F., Rosenblum, D.S., and Vo, K.P. TestTube: A system for selective regression testing. In *Proceedings of the IEEE Software Engineering Conference* (1994), pp. 211–222.
4. Du, D.Z. and Huang, F. *Combinatorial Group Testing*. World Scientific, 1994.
5. Joiner, J., Tsai, W.T., Chen, X.P., Subramanian, S., Sun, J., and Gandamaneni, H. Data-centered program understanding. In *Proceedings of the IEEE Software Maintenance Conference* (1994), pp. 272–273.
6. Kernighan, B. and Richie, D. *The C Programming Language*, 1988.
7. Leung, H.K.N. and White, L. Insights into regression testing. In *Proceedings of the IEEE Software Maintenance Conference* (1989), pp. 60–69.
8. Ness, B. and Ngo, V. Regression containment through source code isolation. In *Proceedings of the IEEE Computer Software and Applications Conference* (1997), pp. 616–621.
9. Onoma, A.K., Tsai, W.T., Tsunoda, F., Suganuma, H., and Subramanian, S. Software maintenance—Industrial experience. *J. Software Maintenance* (1995), 333–375.
10. Rothermel, G. and Harrold, M.J. A safe, efficient algorithm for regression test selection. In *Proceedings of the IEEE Software Maintenance Conference* (1993), pp. 358–367.
11. Rothermel, G. and Harrold, M.J. *A Comparison of Regression Test Selection Techniques.* Tech. Rep., Department of Computer Science, Clemson University, Clemson, SC, Oct. 1994.

Akira K. Onoma (a.k.onoma@ieee.org) is a general manager and director at Hitachi Software Engineering Company, Yokohama, Japan.
Wei-Tek Tsai (tsai@cs.umn.edu) is a professor of Computer Science and Engineering at the University of Minnesota, Minneapolis.
Mustafa Poonawala (mustafa@cs.umn.edu) is currently a Ph.D. candidate in the Department of Computer Science and Engineering at the University of Minnesota, Minneapolis.
Hiroshi Suganuma (suga@computer.org) is a software engineer at Hitachi Software Engineering, Yokohama, Japan.