International Journal of Computing and Corporate Research
ISSN (Online) : 2249-054X
Volume 5 Issue 3 May 2015
International Manuscript ID : 2249054XV5I3052015-05

THE PRAGMATIC REVIEW ON CODE COVERAGE AND ANALYSIS TECHNIQUES IN SOFTWARE TESTING

Savneet Kaur Virk
M.Tech. Research Scholar
Department of Computer Science and Engineering
Guru Nanak Institute of Technology
Mullana, Haryana, India

Varun Jasuja
Assistant Professor
Department of Computer Science and Engineering
Guru Nanak Institute of Technology
Mullana, Haryana, India

ABSTRACT

Source code analysis refers to the deep investigation of source code as well as the compiled version of code in order to help find the flaws in terms of security, readability, understanding and related parameters. Ideally, such techniques automatically find the flaws with such a high degree of confidence that what's found is indeed a flaw. However, this is beyond the state of the art for many types of application security flaws. Thus, such tools frequently serve as aids for an analyst to help them zero in on security relevant portions of code so they can find flaws more efficiently, rather than a tool that just automatically finds flaws. Code Coverage is a measure used to describe the degree to which the source code of a program is tested by a particular test suite. A program with high code coverage has been more thoroughly tested and has a lower chance of containing software bugs than a program with low code coverage. Many different metrics can be used to calculate code coverage; some of the most basic are the percent of program subroutines and the percent of program statements called during execution of the test suite. This research work focus on the quality of source code using code coverage and analysis techniques. In the proposed research work, an effective model based approach shall be developed and implemented to improve the performance of code in terms of overall execution time, code complexity and related metrics.

Keywords - Code Coverage, Software Testing, Test Case Generation, Comments Density,

INTRODUCTION

Code coverage is a way of ensuring that your tests are actually testing your code. When we run your tests you are presumably checking that you are getting the expected results. Code coverage tell how much of your code you exercised by running the test. There are a number of criteria that can be used to determine how well your tests exercise your code. The most simple is statement coverage, which simply tells you whether you exercised the statements in your code. We will examine statement coverage along with some other coverage criteria.

When working with code coverage, and when testing in general, it is wise to remember the following quote from Dijkstra, who said:Testing never proves the absence of faults, it only shows their presence.

Code coverage is a term that is used to describe how much application code is exercised when an application is running. Test coverage is often used to describe test cases that are written against the requirements document. Both are analytics which may be useful for quality assurance personnel to get an indication of how thoroughly an application has been tested.

Using code coverage is a way to try to cover more of the testing problem space so that we come closer to proving the absence of faults, or at least the absence of a certain class of faults. In particular, code coverage is just one weapon in the software engineer's testing arsenal.

Code coverage is a white box testing methodology that is it requires knowledge of and access to the code itself rather than simply using the interface provided. Code coverage is probably most useful during the module testing phase, though it also has benefit during integration testing and probably at other times, depending on how and what you are testing. Regression tests are usually black box tests and as such may be unsuitable for use with code coverage. But often, especially in the Perl world, module, integration, regression and any other tests you might perform all use the same test code, just at different times.

CODE COVERAGE METRICS

A number of different metrics are used determine how well exercised the code is. I'll describe some of the most common metrics here. Most of the metrics have slight variations and synonyms which can make things a little more confusing than they need to be. While I'm describing each metric I'll also show what class of errors it can be used to detect.

STATEMENT COVERAGE

Statement coverage is the most basic form of code coverage. A statement is covered if it is executed. Note that a statement does not necessarily correspond to a line of code. Multiple statements on a single line can confuse issues - the reporting if nothing elsewhere there are sequences of statements without branches it is not necessary to count the execution of every statement, just one will suffice, but people often like the count of every line to be reported anyway, especially in summary statistics.

It can be quite difficult to achieve 100% statement coverage. There may be sections of code designed to deal with error conditions, or rarely occurring events such as a signal received during a certain section of code. There may also be code that should never be executed:

```
if ($param > 20)
{
   die "This should never happen!";
```

}

It can be useful to mark such code in some way and flag an error if it is executed.
Statement coverage, or something very similar, can also be called statement execution, line, block, basic block or segment coverage.

Branch coverage
The goal of branch coverage is to ensure that whenever a program can jump, it jumps to all possible destinations. The most simple example is a complete if statement:

```
if ($x)
{
    print "a";
}
else
{
    print "b";
}
```

Full coverage is only achieved here only if $x is true on one occasion and false on another.
Achieving full branch coverage will protect against errors in which some requirements are not met in a certain branch. For example:

```
if ($x)
{
    $h = { a => 1 }
}
else
{
    $h = 0;
}
print $h->{a};
```

Table 1 - COMPARISON OF CODE COVERAGE TOOLS

Feature	**Atlassian Clover**	**Cobertura**	**JaCoCo**	**Code Cover**	**PITest**
Source files	✓	✗	✗	✗	✗
Class files	✗	✓ off-line instrumentation	✓ off-line and on-the-flyinstrumentation	✓	✓
Coverage metrics					
Method	✓	✗	✓	✓	✗
Statement	✓	✗	✗	✓	✗
Line	✗	✓	✓	✗	✓
Branch	✓	✓	✓	✓	✗

MC/DC	✗	✗	✗	✓	✗
Instruction	✗	✗	✓ explanation	✗	✗
Global coverage	✓	✓	✓	✓	✓
Per-test coverage	✓	✓ sonar	✓ JMX / sonar	✓	✓ lists tests per file
Mutation coverage	✗	✗	✗	✗	✓
Source code metrics					
Available metrics	20+ metrics, also custom ones	cyclomatic complexity	cyclomatic complexity	✗	✗
Report types					
HTML	✓ more details	✓	✓	✓	✓
PDF	✓	✗	✗	✗	✗
XML	✓	✓	✓	✗	✓
JSON	✓	✗	✗	✗	✗
Text	✓	✓	✗	✗	✗
CSV	✗	✗	✓	✓	✗
Data management and report filtering					
Merging of coverage databases	✓ clover2:merge	✓	✓ via <jacoco:merge>	✓	✗
Historical reporting	✓	✓ via sonar	✓ via sonar	✗	✗
Selecting scope of code coverage	file patterns, class patterns, method pattern, code block type, statement's regular expression,	file patterns, code annotations	class patterns	file patterns	package patterns

	code complexity, CLOVER:OFF/ON code comments				
Cross-report linking	✓	✗	✓ via <structure> element	✗	✗
Supported languages					
Java	✓	✓	✓	✓	✓
Groovy	✓ more...	✓	✓	✗	✗
Other	✗	✓ instrumentation is class-based so theoretically any JVM language is supported, but it may lack good reporting (esp. for language-specific constructs); it may also have problems with synthetic methods etc.			
Supported JDK					
	1.6-1.8 (JRE/JDK) 1.3-1.8 (for "-source" level setting)	1.5-1.7	1.5-1.8	1.5-1.7	1.5-1.8
Supported test frameworks					
JUnit	✓	✗	✓	✓	✓
TestNG	✓	✗	✓	✗	✓
Spock	✓ more...	✗	✗	✗	✓
Other	✓ more...	✗	✗	✓ more...	✗
IDE integrations					
IntelliJ IDEA	✓	✗	✓	✗	✓
Eclipse	✓	✓ eCobertura	✓	✓	✓
NetBeans	✗	✗	✓	✗	✗
Build tools integrations					

command line	✓	✓	✓	✓	✓
Ant	✓	✓	✓	✓	✓
Maven	✓	✓	✓	✗	✓
Grails	✓	✓ code-coverage	✗	✗	✗
Gradle	✓	✓ gradle-cobertura-plugin	✓	✗	✓
SBT	✗	✓ cobertura4sbt	✓	✗	✗
CI servers integrations					
Bamboo	✓	✗	✗	✗	✗
Hudson	✓	✓	✗	✗	✗
Jenkins	✓	✓	✓	✗	✗
TeamCity	✗	✗	✓	✗	✗
Other integrations					
Sonar	✓	✓	✓	✗	✓
JIRA	✓	✗	✗	✗	✗
Development activity					
Last release	actively developed, about 7 releases / year	minor activity, last release - 2013	actively developed, few releases / year	minor activity, last release - 2011	actively developed, few releases / year
Technical support					
	Atlassian Support, 24h response	open source community	open source community	open source community	open source community
Subjective summary					
Advantages	Clover has great and highly configurable HTML	Easy to use thanks to off-line byte code instrumentation. You can	Very easy to integrate thanks to the on-the-fly byte	It has the most detailed code coverage metric	PITest is a tool for mutation

	reports(showing not only code coverage but also top risks etc), per-test code coverage and test optimization,distributed per-test coverageand many tool integrations; it is being actively developed and supported.	measure coverage without having the source code. It has very nice and easy to navigate HTML report (example).	code instrumentation. You can measure coverage without having the source code. It has nice HTML report (example).	(MC/DC), which may be useful for critical systems (medical, aeronautical etc). The Eclipse plug-in comes also with a cool Boolean Expression Analyzer view and a Test Correlation matrix. It has also an interesting feature to start/stop test case via JMX, which can be useful for manual testing.	coverage, which means it will not only measure line coverage of your code but will also perform mutations in application logic in order to check how well written your tests are.
Disadvantages	Due to a fact that Clover is based on source code instrumentation, integration requires a build - it's necessary to recompile code with Clover. Most Clover's integrations have an automatic integration feature, but in some cases you may need to add Clover JAR to a class path or set some Clover options.	Classes must be compiled with debug option.	Classes must be compiled with debug option.	Last release has been performed 3 years ago. The HTML report generated is quite fragmented - source code is shown separately for every method.	

Path coverage

There are classes of errors which branch coverage cannot detect, such as:

```
$h = 0;
if ($x)
```

```
{
   $h = { a => 1 };
}
if ($y)
{
   print $h->{a};
}
```

Condition coverage

When a boolean expression is evaluated it can be useful to ensure that all the terms in the expression are exercised. For example:

```
a if $x || $y;
```

To achieve full condition coverage, this expression should be evaluated with $x and $y set to each of the four combinations of values they can take.

REAL TIME APPLICATIONS OF CODE COVERAGE

- Bugs Analysis and Avoidance
- Reduction of Code Complexity
- Cross Platform Compatibility

OPEN SOURCE TOOLS - CODE ANALYSIS

- Google CodeSearchDiggity
- FindBugs
- FxCop (Microsoft)
- PMD
- PreFast (Microsoft) RATS (Fortify)
- OWASP SWAAT Project
- Flawfinder Flawfinder
- RIPS
- Brakeman
- Codesake Dawn
- VCG

COMMERCIAL TOOLS

- BugScout (Buguroo Offensive Security)
- Contrast from Contrast Security
- IBM Security AppScan Source Edition (formerly Ounce)
- Insight (KlocWork)
- Parasoft Test (Parasoft)
- Pitbull Source Code Control (Pitbull SCC)
- Seeker (Quotium)
- Source Patrol (Pentest)
- Static Source Code Analysis with CodeSecure™ (Armorize Technologies)
- Kiuwan - SaaS Software Quality & Security Analysis (Optimyth)
- Static Code Analysis (Checkmarx)
- Security Advisor (Coverity)
- Source Code Analysis (HP/Fortify)
- Veracode (Veracode)
- Sentinel Source solution (Whitehat)

REVIEW OF LITERATURE

To propose and defend the research work, a number of research papers are analyzed. Following are the excerpts from the different research work performed by number of academicians and researchers.

[1] In this paper, the authors provide the details of an efficient method to compute an observability-based code coverage metric that can be used while simulating complex hardware description language (HDL) designs. This method offers a more accurate assessment of design verification coverage than line coverage and is significantly more computationally efficient than prior efforts to assess observability information because it breaks up the computation into two phases: functional

simulation of a modified HDL model followed by analysis of a flow graph extracted from the HDL model.

[2] Penetration testing is the most commonly applied mechanism used to gauge software security, but it's also the most commonly misapplied mechanism as well. By applying penetration testing at the unit and system level, driving test creation from risk analysis, and incorporating the results back into an organization's SDLC, an organization can avoid many common pitfalls. As a measurement tool, penetration testing is most powerful when fully integrated into the development process in such a way that finding scan help improve design, implementation, and deployment practices

[3] In this paper the authors present a new approach to dynamically insert and remove instrumentation code to reduce the runtime overhead of code coverage. The work also explores the use of dominator tree information to reduce the number of instrumentation points needed. Our experiments show that the approach reduces runtime overhead by 38-90% compared with purecov, a commercial code coverage tool.

[4] This paper presents a technique intended to solve this Problem, using both time & code coverage measures for the prediction of software failures in operation. Coverage information collected during testing is used only to consider the effective portion of the test data. Execution time between test cases, which neither increases code coverage nor causes a failure, is reduced by a parameterized factor. Experiments the work reconducted to evaluate this technique, on a program created in a simulated environment with simulated faults, and on two industrial systems that contained tenths of ordinary faults.

[5] This work focuses on assorted code inspection techniques with multiple case generations. Using this work, the major work is done of penetration testing and its association with the software complexity issues.

[6] In this paper, the authors describe our algorithm for mapping CVS comments to the corresponding source code, present a search tool based on this technique, and discuss preliminary feedback.

[7] This paper describes in a general way the process we went through to determine the goals, principles, audience, content and style for writing comments in source code for the Java platform at the Java Software division of Sun Microsystems. This includes how the documentation comments evolved to become the home of the Java platform API specification, and the guidelines we developed to make it practical for this document to reside in the same files as the source code.

[8] A code clone is a code portion in source files that is identical or similar to another. Since code clones are believed to reduce the maintainability of software, several code clone detection techniques and tools have been proposed. This paper proposes a new clone detection technique, which consists of the transformation of input source text and a token-by-token comparison. For its implementation with several useful optimization techniques, we have developed a tool, named CCFinder (Code Clone Finder), which extracts code clones in C, C++, Java, COBOL and other source files. In addition, metrics for the code clones have been developed. In order to evaluate the usefulness of CCFinder and metrics, we conducted several case studies where we applied the new tool to the source code of JDK,

FreeBSD, NetBSD, Linux, and many other systems. As a result, CCFinder has effectively found clones and the metrics have been able to effectively identify the characteristics of the systems. In addition, we have compared the proposed technique with other clone detection techniques.

[9] Comments are valuable especially for program understanding and maintenance, but do developers comment their code? To which extent do they add comments or adapt them when they evolve the code? We examine the question whether source code and associated comments are really changed together along the evolutionary history of a software system. In this paper, we describe an approach to map code and comments to observe their co-evolution over multiple versions. We investigated three open source systems (i.e., ArgoUML, Azureus, and JDT core) and describe how comments and code co-evolved over time. Some of our findings show that: 1) newly added code - despite its growth rate - barely gets commented; 2) class and method declarations are commented most frequently but far less, for example, method calls; and 3) that 97% of comment changes are done in the same revision as the associated source code change.

[10] It is common, especially in large software systems, for developers to change code without updating its associated comments due to their unfamiliarity with the code or due to time constraints. This is a potential problem since outdated comments may confuse or mislead developers who perform future development. Using data recovered from CVS, we study the evolution of code comments in the PostgreSQL project. Our study reveals that over time the percentage of commented functions remains constant except for early fluctuation due to the commenting style of a particular active developer.

[11] An important software engineering artefact used by developers and maintainers to assist in software comprehension and maintenance is source code documentation. It provides insights that help software engineers to effectively perform their tasks, and therefore ensuring the quality of the documentation is extremely important. Inline documentation is at the forefront of explaining a programmer's original intentions for a given implementation. Since this documentation is written in natural language, ensuring its quality needs to be performed manually. In this paper, we present an effective and automated approach for assessing the quality of inline documentation using a set of heuristics, targeting both quality of language and consistency between source code and its comments. We apply our tool to the different modules of two open source applications (ArgoUML and Eclipse), and correlate the results returned by the analysis with bug defects reported for the individual modules in order to determine connections between documentation and code quality.

PROPOSED WORK

As the domain of software testing is much diversified, there is lots of scope of research for the scholars and practitioners. In Code Based Software Testing, the following research areas can be worked out by the research scholars -

- Component Based Code Investigation
- Security and Privacy Issues in Code Modules

- Cross Platform Compatibility and Efficiency Issues
- Functional Aspects and Scenarios
- Analysis of Comments Density
- Analysis of Operands and relative performance on overall code
- Halstead Metrics Analysis

To improve the base work done in the existing algorithm having the classical approach with haphazard manner of operands and comments, we will calculate the execution time and complexity of the existing algorithm in the base paper. At the end, the whole research work will be concluded with some future research work. To design an effective and improved model for code coverage including comments density analysis, variables and operands used. To design and implement the effective model for code investigation using Monte Carlo Simulation Techniques, the proposed work will deliver the optimized rules and solutions so that the proportional aspects of operands, constants and comments can be used in the source code. Comparison shall be done on multiple parameters in Existing and Proposed Approach.

CONCLUSION AND SCOPE OF FUTURE WORK

Source code analysis is the automated testing of source code for the purpose of debugging a computer program or application before it is distributed or sold. Source code consists of statements created with a text editor or visual programming tool and then saved in a file. The source code is the most permanent form of a program, even though the program may later be modified, improved or upgraded. Code coverage analysis is used to measure the quality of software testing, usually using dynamic execution flow analysis. There are many different types of code coverage analysis, some very basic and others that are very rigorous and complicated to perform without advanced tool support. The proposed work shall be implemented on a simulation based scenario for proportional comments, operands and related aspects of the source code.

REFERENCES

[1] Fallah, Farzan, Srinivas Devadas, and Kurt Keutzer. "OCCOM-efficient computation of observability-based code coverage metrics for functional verification." Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 20, no. 8 (2001): 1003-1015.

[2] Burr, Kevin, and William Young. "Combinatorial test techniques: Table-based automation, test generation and code coverage." In Proc. of the Intl. Conf. on Software Testing Analysis & Review. 1998.

[3] Tikir, Mustafa M., and Jeffrey K. Hollingsworth. "Efficient instrumentation for code coverage testing." ACM SIGSOFT Software Engineering Notes 27, no. 4 (2002): 86-96.

[4] Chen, M. H., Lyu, M. R., & Wong, W. E. (2001). Effect of code coverage on software reliability measurement. Reliability, IEEE Transactions on, 50(2), 165-170.

[5] Fagan, M. (2002). Design and code inspections to reduce errors in program development. In Software pioneers (pp. 575-607). Springer Berlin Heidelberg.

[6] Kramer, D. (1999, October). API documentation from source code comments: a case study of Javadoc. In Proceedings of the 17th annual

international conference on Computer documentation (pp. 147-153). ACM.

[7] Yao, A. Y. (2001, November). CVSSearch: Searching through source code using CVS comments. In Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01) (p. 364). IEEE Computer Society.

[8] Fluri, B., Wursch, M., & Gall, H. C. (2007, October). Do code and comments co-evolve? on the relation between source code and comment changes. In Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on (pp. 70-79). IEEE.

[9] Fluri, B., Wursch, M., & Gall, H. C. (2007, October). Do code and comments co-evolve? on the relation between source code and comment changes. In Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on (pp. 70-79). IEEE.

[10] Jiang, Z. M., & Hassan, A. E. (2006, May). Examining the evolution of code comments in PostgreSQL. In Proceedings of the 2006 international workshop on Mining software repositories (pp. 179-180). ACM.

[11] Khamis, N., Witte, R., & Rilling, J. (2010). Automatic quality assessment of source code comments: the JavadocMiner. In Natural language processing and information systems (pp. 68-79). Springer Berlin Heidelberg.

[12] Kilgour, R. I., Gray, A. R., Sallis, P. J., & MacDonell, S. G. (1998). A fuzzy logic approach to computer software source code authorship analysis.

[13] Nagappan, N., & Ball, T. (2005, May). Static analysis tools as early indicators of pre-release defect density. In Proceedings of the 27th international conference on Software engineering (pp. 580-586). ACM.

[14] Kilgour, R. I., Gray, A. R., Sallis, P. J., & MacDonell, S. G. (1998). A fuzzy logic approach to computer software source code authorship analysis.

[15] Gabel, M., & Su, Z. (2010, November). A study of the uniqueness of source code. In Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering (pp. 147-156). ACM.

[16] Frantzeskou, G., MacDonell, S., Stamatatos, E., & Gritzalis, S. (2008). Examining the significance of high-level programming features in source code author classification. Journal of Systems and Software, 81(3), 447-460.