社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

A Systematic Review of Source Code Coverage Metrics: Preliminary Results

Masayuki TANIGUCHI†, Shinsuke MATSUMOTO†, and Shinji KUSUMOTO†

† Graduate School of Information Science and Technology, Osaka University
E-mail: †{m-tanigt,shinsuke,kusumoto}@ist.osaka-u.ac.jp

Abstract Software testing plays an essential role in software quality assurance. It helps developers to reveal and remove bugs in software. Developers often use test coverage to measure the sufficiency of tests, find non-tested statements, and localize a faulty statement. Traditional coverages, such as statement and branch coverage, are widely known and used. On the other hand, researchers have proposed various metrics for measuring test coverage of source code. Because such novel coverage metrics are not organized, it is impossible to understand and compare the benefits and limitations of each metric. This paper organizes the characteristics of each coverage metric by surveying a body of 43 papers that propose coverage metrics. The survey results showed that the novel metrics could be divided into two main groups: (1) metrics that improve or complement traditional coverage and (2) metrics that are effective in specific domains, such as concurrent programming. We performed a comparative analysis to identify the characteristics of each metric, such as benefits of use, effective domains, and information needed to measure coverage. Furthermore, we provide a catalog of coverage metrics to help developers and researchers select the best metrics for their context.
Key words Software testing, Test coverage, Coverage metrics, Systematic review

1. Introduction

Software testing is an essential activity in software quality assurance. Though software testing is a broad concept that includes various verification activities such as review, walkthrough and inspection, this paper focuses a validation activity especially in programmed test. The programmed test means to confirm whether the given program behaves as expected by the program execution. This paper refers it as simply *test* or *testing*.

Developers usually evaluate the quality of test with many criterion. One of the most well known criteria is test coverage that measures the comprehensiveness of tests against source code based on execution path. Test coverage can be used for measuring the sufficiency of tests, finding non-tested statements, and localization of a faulty statement [1].

Traditional coverages, such as statement and branch coverage, are widely known and being used. The limitations of the coverage are also well known. For instance, 100% traditional coverage does not guarantee that the source code has no bugs [2] [3]. To address their drawbacks, researchers have proposed various test coverage metrics. However, these novel metrics are not structured and organized yet. So, it is impossible to understand and compare the benefits and limitations of each metric. This lack of organization also prevents consideration of the use of novel coverage metrics.

Our goal is to provide developers and researchers with a catalog of novel test coverage metrics and to allow them to select suitable metrics in their context. Hence, our research questions are, *what kind of novel coverage metrics exist?* and *what are their characteristics?*. In this paper, we conduct a systematic review for 43 papers that proposed novel coverage metrics to organize the characteristics of each metric. In our analysis, we found that the proposal of coverage metrics was primarily due to two reasons: (1) to improve or complement traditional coverage and (2) to effectively measure coverage in specific domains. Based on this finding, we analyzed the characteristics of the novel coverage metrics for each proposal reason and domain.

2. Paper Selection

In this section, we describe the procedure to select the papers for our survey. First of all, we collected the papers for our study by using a specific set of keywords in some popular digital libraries. We performed the paper collection at the beginning of May 2022.

We used the following ten keywords: *test coverage*, *coverage metrics*, *code coverage*, *testing strategies*, *software testing strategies*, *oracle quality*, *test oracle quality*, *test suite quality*, *test suite effectiveness*, *insufficiently tested code*. Since we intended to collect all papers related to our survey as much as possible, the set of keywords includes not only those directly related to coverage metrics but also those related to testing strategy and test quality.

Using the ten keywords, we searched in the three digital libraries (ACM Digital Library, IEEE Xplore and Google Scholar). For each query, we collected the top 200 publications in order of relevance. As a result, we obtained 5,818 publications. (Note that a search for *insufficiently tested code* in IEEE Xplore returned 18 publications.) After removing duplicate publications, we ended up with 4,459 publications.

Following our paper collection, we filtered the publications we obtained. We first quickly eliminated papers that were obviously irrelevant to our study by manually checking the titles and abstracts of all the collected publications. The first two authors performed this filtering to reduce the number of false negatives. This process took two months and resulted in 237 papers. After quick filtering, we conducted a full-text analysis of each selected paper. We reviewed whether each paper proposed test coverage metrics. At the end of this process, we obtained 43 relevant papers.

3. Novel Test Coverage Metrics

By analyzing the 43 studies, we found that novel coverage metrics can be divided into two groups: (1) those that improve or complement traditional coverage metrics and (2) those that are specific to particular domains. This section describes the overview and benefits of the 43 metrics according to this classification. Note that some metrics have multiple levels of coverage measurement granularity. Table 1 summarizes the characteristics of each metric, including effective domain, feature, information required to measure coverage and granularity of measurement.

3.1 Metrics that improve or complement traditional coverage

Belli et al. [4] proposed Test Segment Coverage as a coverage that bridges the gap between branch coverage and path coverage. Test segment coverage is the path coverage of each test segment (program fragment composed of one statement or a sequence of statements). By adjusting the size of the test segments, the thoroughness of the test coverage can be adapted to the needs of the tester.

Chen et al. [5] proposed test coverage about variables. By calculating the program slice for a variable, we can measure the test coverage for the code associated with that variable. In other words, the tester can focus the test quality evaluation on important variables.

Koster et al. [6] proposed State Coverage for test oracle assessment. This coverage measures whether variables defined at code runtime are validated by assertions, using control flow graphs and program slicing.

In the subsequent study by Vanoverberghe et al. [7], a general definition of State Coverage was proposed. The authors' definition does not require a specific structure for testing and allows more dynamic state update identifiers (e.g., object identifiers) than nodes in the control flow graph.

Schuler et al. [2] [8] proposed Checked Coverage. The concept of checked coverage is similar to state coverage. This coverage metric requires testers to verify that statements that read or write variables or that can affect the control flow of the program are checked by assertions. The authors consider statements on a dynamic backward slice from an assertion as checked statements.

Zaraket et al. [9] proposed Property Based Coverage. This coverage derives from the hypothesis that it is more effective to evaluate test suites based on their coverage of system properties than that of structural program elements. The authors view a *property* as a logical expression in an assertion and annotation. By using property based coverage criterion, we can measure the test coverage for all the possible values that variables in properties can take.

Whalen et al. [10] proposed Observable MC/DC (OMC/DC). OMC/DC is a version of MC/DC that incorporates the concept of observability. The authors state that an expression in a program is *observable* in a test case if we can modify its value, leaving the rest of the program intact, and observe changes in the output of the system. This coverage metric helps ensure that a fault encountered when executing the decision propagates to a monitored variable.

Hassan et al. [11] proposed MultiPoint Stride Coverage. This coverage is equivalent to branch coverage that incorporates the concept of dataflow coverage by taking into account the execution order of each branch. By using this coverage, we can more accurately predict the quality of a test suite than control flow based coverage such as branch coverage. We can also more easily measure it than dataflow based coverage such as def-use coverage.

Huo et al. [12] proposed Direct/Indirect Coverage. The authors argue that it is useful in the management of testing resources to consider whether entities (e.g., functions, statements and branches) were covered directly or indirectly by tests. This is because indirectly covered entities are only peripherally considered and are insufficiently tested [12].

McMinn et al. [13] proposed fault coverage for software testing. Fault coverage is a concept in electronic engineering that refers to the percentage of faults detected by tests out of a pre-defined list of faults. The authors discusses the way to automatically generate fault coverage for software engineering by using a fault database such as Defects4J [14].

Byun et al. [15] proposed Flag-Use Object Branch Coverage. Object Branch Coverage (OBC), branch coverage at the object code level, has the advantage of being programming language independent and is amenable to non-intrusive coverage measurement techniques. However, OBC strongly depends on differences in object code structure due to compilers and their optimizations. While OBC is a coverage metric based only on jump instructions, Flag-Use OBC extends OBC to include many other instructions involved in conditional behavior.

Someoliayi et al. [16] proposed Program State Coverage. This coverage metric improves the ability of line coverage to validate the effectiveness of the test suite. The authors considers the number of distinct program states in which each line is executed. Program state coverage is calculated by the ratio of program states executed in a line of tests to the maximum number of program states.

Subsequently, Aghamohammadi et al. [17] proposed Statement Frequency Coverage. Program state coverage has some limitations, such as the need to set a maximum number of states because we cannot predict the number of possible states, and the possibility of statements with infinite states during test execution. Statement frequency coverage solves these problems by incorporating the frequency of executed statements into the statement coverage.

Miranda et al. [18] [19] proposed Relative Coverage. This is a coverage measurement technique that focuses on the test scope of testers. By focusing coverage measurement only on in-scope entities, we can expect to improve the cost-effectiveness of testing. The authors also proposed four instances of relative coverage: Operational Coverage [18], Social Coverage [18] [20], Relevant Coverage [18] [21], Reachability Coverage [19]. Operational coverage focuses on the operations performed by a specific user group. Relevant coverage measures test coverage in the scope of testing reused code. Reachability coverage targets the input domain that a specific user is expected to exercise. Social coverage is a coverage metric for Service-Oriented Architecture (SOA) and will be described in Section 3. 2.

Cox [22] proposed Differential Coverage. This is a concept of classifying coverage information into 12 categories (newly added code is not tested, previously unused code is covered now, etc.) by comparing the current version of the code with a baseline. Especially in large-scale development, the analysis of coverage information is very costly. We can reduce the cost of coverage analysis by automatically classifying coverage information using differential coverage.

3. 2 Domain Specific Metrics

Concurrent Program. Bron et al. [23] proposed Synchronization Coverage. This is a practical coverage based on the idea that coverage tasks should be well understood by users and be coverable by tests. This coverage is accepted by IBM. Synchronization coverage has seven synchronous processes as coverage tasks.

Sherman et al. [24] proposed coverage metrics inspired by synchronization coverage. These metric are designed for saturation-based testing in concurrent programs, hence there is no need to estimate the executable domain of each metric. The authors use a combination of three basic concurrency metrics and six contexts as coverage tasks.

Křena et al. [25] proposed coverage metrics for saturation-based and search-based testing to reflect concurrency behavior accurately. In previous work [24], the identification of elements is too rough because Java types were used to identify threads. The proposed metrics more accurately distinguish the behavior of objects and threads based on object identifier and thread identifier. The authors derived 11 coverage metrics from dynamic analyses designed for discovering bugs in concurrent programs.

Terragni et al. [26] proposed Sequential Coverage. This coverage metric has a sequence of events (write/read object fields, acquire/release locks and enter/exit methods) as a coverage task. We can measure this coverage by a single thread execution of a call sequence.

Wang et al. [3] proposed MAP-coverage. This coverage is based on memory-access patterns (MAP), which are patterns of how shared variables are accessed by multiple threads [27]. MAP have often been shown to be associated with the nature of multi-threaded bugs [27]. Thus, comprehensive testing of all MAP is effective in finding bugs.

Object Oriented (OO) Program. Hsia et al. [28] proposed coverage metrics based on Enumerate Data Member (EDM). A class is said to satisfy EDM property if its state-related data members are of enumerate type. The authors argued that each set of values assigned to the object should be covered by at least one test for classes satisfying EDM property. They provided three levels of coverage metrics.

Fisher et al. [29] proposed change-based coverage metrics. The authors focused on the impact of code changes based on the assumption that a disproportionate number of faults are likely to be present in recently modified codes. This study defines four test coverage metrics for changed and added entities (e.g., methods and statements) in the context of OO.

Biswas [30] proposed Control Dependence Inheritance Coverage metrics based on JSysDG (Java System Dependency Graph). Each time a tested class is reused through inheritance, we must retest it under new usage context [31]. Therefore, the cost of testing OO software can significantly exceed that of testing procedural programs. By using these metrics, we can measure effectively the test coverage of control dependencies associated with inheritance.

Mukherjee et al. [32] proposed coverage metrics in response to the fact that structural coverage metrics for integration testing of OO programs has been scarcely reported. These coverage metrics are based on data and control dependencies in the classes being integrated defined on JSysDG.

Mukherjee [33] also focused on testing safety-critical software, such as nuclear power plant, in the OO paradigm. Safety-critical software requires thorough testing; however, traditional coverage metrics suffers from several shortcomings. The authors proposed test coverage metrics that cover program dependencies more robustly and can detect faults at inter-object data dependencies.

Web Application. Alalfi et al. [34] proposed coverage metrics for dynamic web applications. Faults in web applications often caused by insufficient test coverage of complex interactions between components. These coverage metrics are based on the client and database interactions and require testing server pages, SQL statements and server environment variables.

Mirzaaghaei et al. [35] proposed DOM Coverages. Web application test generally interact with the DOM. The authors argue that the DOM itself should be considered as an important structure of the system that needs to be adequately covered by tests. Based on this idea, this paper propose six coverage metrics related to the DOM state.

Nguyen et al. [36] proposed coverage metrics for output-oriented testing of dynamic web application. These coverage metrics measures test coverage of string literals output and decisions that affect the output. Using these metrics help to identify presentation faults such as HTML validation errors and spelling errors.

Service-Oriented Architecture (SOA). Hummer et al. [37] proposed k-Node Dataflow Coverage to significantly reduce the search space of service combinations in integration test of dynamic composite Service-Based Systems (SBSs). This metric is based on dataflow of service composition.By restricting the paths for coverage measurement to all k-length paths in the dependency tree, where a service composition is considered as a node, we can reduce the number of dataflows to be covered by tests.

In 2014, Miranda et al. [18] [19] [20] proposed Social Coverage. This is the instance of relative coverage [18] [19]. Social coverage was conceived for black-box environments having some notion of testing community (i.e., several users/programs using/testing the service under test). This metric measures test coverage for the in-scope entities identified by information about the entities invoked by similar users in the same test community. The authors assume that the service provider will measure this coverage and provide it to the customers.

Sneed et al. [38] proposed coverage metrics based on the structure and content of the service interface. These are test coverage of input/output parameters or combinations of parameters in input/output messages. Using these metrics, testers can evaluate test quality without considering the source code in SOA where they cannot access to the source code of the service under test.

Others. Memon et al. [39] proposed coverage metrics for GUI testing. The input to a GUI consists of a sequence of events. The proposed metrics thus focus on events in the structure of the GUI and measure the comprehensiveness of testing for events and event sequences.

Smith et al. [40] proposed SQL statement coverage for SQL injection input validation testing. Traditional coverage metrics cannot highlight how well the system protects itself through validation. SQL statement coverage metrics measure the test coverage of SQL statements or input variables of SQL statements. Coverage data based on these metrics can provide specific information about insufficient or missing input validation.

Kim et al. [41] proposed New Decision Coverage for multi-staged language. Multi-staged language is a programming language which can generate and execute new program codes in execution time. Because it is hard to estimate what code fragments would be generated and executed in multi-staged language, traditional coverage is not suitable for multi-stage languages. New decision coverage metric measures the test coverage of both branches that already exist in the program and those generated at runtime. In the study, this metric is designed for a two-staged language.

Tsankov et al. [42] proposed Semi-Valid Input Coverage for fuzz testing. Traditional coverage metrics do not measure what fuzz testing is all about, namely executing the system with semi-valid inputs. Semi-valid input coverage metric measures to what extent the tests cover the domain of semi-valid inputs, where an input is semi-valid if and only if it satisfies all the constraints but one.

Jabbarvand et al. [43] proposed eCoverage. This study aims to reduce the number of tests in energy testing of Android applications. eCoverage takes into account the energy consumption of segments (methods or system APIs). By using this metric, we can measures test coverage of energy-greedy segments that highly contribute to the energy consumption of the application.

Nakajima et al. [44] proposed Dataset Coverage for Machine Learning (ML) programs. The control structure of ML program is so simple that any execution of the program takes all control paths if the input training dataset is not trivial. Dataset coverage focuses on the characteristics of the population distribution in the training dataset in metamorphic testing.

Rott et al. [45] proposed Ticket Coverage for agile development. This coverage unveils which of the changes made in the course of a ticket are left untested. This metric measure test coverage of the methods that were added and changed during the implementation of a given ticket and helps to systematically focus testing efforts on changed code.

Martin-Lopez et al. [46] proposed coverage metrics for RESTful API because there is no standardized coverage criteria for black-box testing of RESTful API. These metrics measure test coverage of elements related to API requests/responses. The paper provides four levels of coverage metrics for each of the API requests and responses.

Ali et al. [47] proposed coverage metrics for quantum program. Testing quantum programs is difficult due to the inherent characteristics of quantum computing, such as the probabilistic nature and computations in superposition. However, automatic and systematic testing is necessary to guarantee the correct operation of quantum programs. These proposed metrics are based on inputs and outputs of the quantum program and measure the comprehensiveness of tests without destroying superpositions of the quantum program.

4. Conclusion and Future Work

In this paper, we conducted a survey of papers proposing novel test coverage metrics. After analyzing 43 papers, we found that coverage metrics can be classified into two categories: (1) those that aim to improve or complement traditional coverage metrics and (2) those that are specific to a particular domain. We also identified and organized the characteristics of each metric. A catalog of novel coverage metrics would help developers and researchers to select suitable metrics in their context.

In future work, we plan to perform backward and forward snowballing to make the survey as much comprehensive as possible. We will examine whether coverage metrics are proposed in each publication cited by or citing any of the papers we have analyzed.

Acknowledgments This research was partially supported by JSPS KAKENHI Japan (Grant Number: JP21H04877, JP20H04166, JP21K18302, and JP21K11829)

References

[1] H. Zhu, P.A.V. Hall, and J.H.R. May, "Software Unit Test Coverage and Adequacy," ACM Comput. Surv., vol.29, no.4, pp.366–427, 1997.

[2] D. Schuler and A. Zeller, "Checked coverage: an indicator for oracle quality," Softw.: Testing, Verif. and Reliab., vol.23, no.7, pp.531–551, 2013.

[3] Z. Wang, Y. Zhao, S. Liu, J. Sun, X. Chen, and H. Lin, "MAP-Coverage: A Novel Coverage Criterion for Testing Thread-Safe Classes," Proc. IEEE/ACM Int'l Conf. Autom. Softw. Eng., pp.722–734, 2019.

[4] F. Belli and J. Dreyer, "Program Segmentation for Controlling Test Coverage," Proc. Int'l Symp. Softw. Reliab. Eng., pp.72–83, 1997.

Table 1: Novel Source Code Coverage Metrics

Metric	Domain	Feature	Required information	Granularity
Test segment coverage [4]	Non-domain specific	Adjust thoroughness of test coverage	Control flow graph	Path
About variables [5]	Non-domain specific	Focus on important variables	Program slices	Path
State coverage [6]	Non-domain specific	Assess test oracle quality	Control flow graph and program slices	Variable validation by assertions
State coverage [7]	Non-domain specific	General definition of state coverage [6]	Control flow graph and program slices	Variable validation by assertions
Checked coverage [2] [8]	Non-domain specific	Assess test oracle quality	Dynamic slices	Statement validation by assertions
Property based coverage [9]	Non-domain specific	Test coverage of properties in assertions and annotation	All the possible values of variables in properties	Values of variables in property
Observable MC/DC [10]	Non-domain specific	More rigorous than MC/DC	Observability of boolean expressions	Condition and decision
MultiPoint stride coverage [11]	Non-domain specific	Branch coverage considering execution order of branches	Execution order of branches	Sequence of branches
Direct/Indirect coverage [12]	Non-domain specific	Effectively identify insufficiently tested methods	Mapping entities to tests and methods	Any*
Fault coverage for Software Testing [13]	Non-domain specific	Create coverage metrics in line with actual bugs	Fault database	Depends on auto-generated metric
Flag-Use Object Branch coverage [15]	Non-domain specific	At object code level with low dependence on compiler structure	Object code	Instruction
Program state coverage [16]	Non-domain specific	More effectively than line coverage with low execution cost	Number of execution of each line	Program state
Statement frequency coverage [17]	Non-domain specific	Overcome several shortcomings of program state coverage [16]	Number of execution of each statement	Frequency of execution of each statement
Relative coverage [18] [19]	Non-domain specific	Focus on in-scope entities	Usage scope	Any*
Operational coverage [18]	Non-domain specific	Focus on operations performed by a specific user group	Operational profile	Any*
Relevant coverage [18] [21]	Non-domain specific	Focus on entities of reused code in new (reuse) context	Input domain constraints	Any*
Reachability coverage [19]	Non-domain specific	Focus on input domain that is expected to be exercised by a specific user	Input domain expected to be exercised	Any*
Differential coverage [22]	Non-domain specific	Classify code coverage information into 12 categories	Version history	Any*
Synchronization coverage [23]	Concurrent programs	Practical coverage for concurrent programs	Runtime behavior of threads	Synchronization behavior
For saturation-based testing [24]	Concurrent programs	No need to estimate the executable domain of each metric	Runtime behavior of threads	Synchronization behavior
From dynamic analysis [25]	Concurrent programs	Accurately identify behavior of threads than previous work [24]	Runtime behavior of threads and objects	Synchronization behavior
Sequential coverage [26]	Concurrent programs	Measure coverage by a single thread execution	Possible method call sequences	Sequence of events
MAP-coverage [3]	Concurrent programs	Help find multi-threaded bugs	Possible memory-access patterns [27]	Memory-access pattern [27]
Based on enumerate data member [28]	OO programs	Reveal faults related to object states	Possible object assignments	Object assignment
Change-based coverage [29]	OO programs	Expected to be effective in revealing regression faults	Version history	Any*
Control dependence inheritance coverage [30]	OO programs	Consider dependencies through inheritance	JSysDG	Control dependency
For integration testing [32]	OO programs	Detect bugs missed in scenario-based integration testing	JSysDG	Data and control dependency
For safety-critical software [33]	OO programs	Cover program dependencies robustly	JSysDG	Def-use pair
For dynamic web application [34]	Web application	Cover complex interactions between client and database	Instrumentation transformation	Entity related to interactions
DOM coverage [35]	Web application	Provide information about DOM in web application testing	DOM state flow graph	DOM state or element
For output string literals [36]	Web application	Help identify presentation faults such as HTML validation errors	Possible output produced from string literals	String literal or decision
k-Node dataflow coverage [37]	SOA	Reduce the search space of service combinations in integration test	Trees of data dependencies between services	Dataflow
Social coverage [18] [19] [20]	SOA	Focus on operations of interest to testers of SOA-based systems	Test community	Operation
For parameters in messages [38]	SOA	Evaluate test quality without considering the source code in SOA	Complete input and output domains of service	Parameter in messages
GUI event coverage [39]	GUI	Handle GUI event sequences that are much more abstract than code	Event flow graph	Event or event sequence
SQL statement coverage [40]	SQLi input valid.	Provide specific information about insufficient or missing input validation	Set of sql statements or input variables	Sql statement or input variable
New decision coverage [41]	Two-staged languages	Handle branches included in code generated at runtime	Code fragments generated in execution	Decision
Semi-valid input coverage [42]	Fuzzing	Measure how well tests covers the domain of semi-valid inputs	Input constraints	Input
eCoverage [43]	Android application	Consider the energy consumption of segments (methods or system APIs)	Call graph of segments	Energy-greedy segment
Dataset coverage [44]	Machine learning	Evaluate test quality in terms of dataset variety	Linearly separable dataset	Variety of datasets
Ticket coverage [45]	Agile development	Help focus testing efforts on changed code in a ticket	Commits related to a ticket	Method
For RESTful API [46]	RESTful API	Test coverage of elements related to API requests/responses	Set of entities related to API requests/responses	Entities related to API requests/responses
For quantum programs [47]	Quantum programs	Test coverage without destroying superpositions of quantum programs	Set of valid input/output values	Input/Output value

* We can select any granularity (statement, branch, method, etc.)

[5] Z. Chen, B. Xu, H. Yang, and H. Chen, "Test Coverage Analysis Based on Program Slicing," Proc. IEEE Workshop on Mobile Comput. Systems and Applications, pp.559–565, 2003.
[6] K. Koster and D.C. Kao, "State coverage: a structural test adequacy criterion for behavior checking," Proc. Joint Meeting of the European Softw. Eng. Conf. and the ACM SIGSOFT Int'l Symp. Foundations of Softw. Eng., pp.541–544, 2007.
[7] D. Vanoverberghe, J. deHalleux, N. Tillmann, and F. Piessens, "State Coverage: Software Validation Metrics beyond Code Coverage," SOFSEM 2012: Theory and Practice of Computer Science, pp.542–553, 2012.
[8] D. Schuler and A. Zeller, "Assessing Oracle Quality with Checked Coverage," Proc. IEEE Int'l Conf. Softw. Testing, Verif. and Valid., pp.90–99, 2011.
[9] F. Zaraket and W. Masri, "Property Based Coverage Criterion," Proc. Int'l Workshop on Defects in Large Softw. Systems: Held in Conjunction with the ACM SIGSOFT Int'l Symp. Softw. Testing and Anal., pp.27–28, 2009.
[10] M. Whalen, G. Gay, D. You, M.P.E. Heimdahl, and M. Staats, "Observable Modified Condition/Decision Coverage," Proc. Int'l Conf. Softw. Eng., pp.102–111, 2013.
[11] M.M. Hassan and J.H. Andrews, "Comparing Multi-Point Stride Coverage and Dataflow Coverage," Proc. Int'l Conf. Softw. Eng., pp.172–181, 2013.
[12] C. Huo and J. Clause, "Interpreting Coverage Information Using Direct and Indirect Coverage," IEEE Int'l Conf. Softw. Testing, Verif. and Valid., pp.234–243, 2016.
[13] P. McMinn, M. Harman, G. Fraser, and G.M. Kapfhammer, "Automated Search for Good Coverage Criteria: Moving from Code Coverage to Fault Coverage through Search-Based Software Engineering," Proc. Int'l Workshop on Search-Based Softw. Testing, pp.43–44, 2016.
[14] R. Just, D. Jalali, and M.D. Ernst, "Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs," Proc. Int'l Symp. Softw. Testing and Anal., pp.437–440, 2014.
[15] T. Byun, V. Sharma, S. Rayadurgam, S. McCamant, and M.P.E. Heimdahl, "Toward Rigorous Object-Code Coverage Criteria," Proc. Int'l Symp. Softw. Reliab. Eng., pp.328–338, 2017.
[16] K.E. Someoliayi, S. Jalali, M. Mahdieh, and S.-H. Mirian-Hosseinabadi, "Program State Coverage: A Test Coverage Metric Based on Executed Program States," IEEE Int'l Conf. Softw. Anal., Evolution and Reengineering, pp.584–588, 2019.
[17] A. Aghamohammadi, S.-H. Mirian-Hosseinabadi, and S. Jalali, "Statement frequency coverage: A code coverage criterion for assessing test suite effectiveness," Info. and Softw. Technology, vol.129, p.106426, 2021.
[18] B. Miranda and A. Bertolino, "Testing Relative to Usage Scope: Revisiting Software Coverage Criteria," ACM Trans. Softw. Eng. Methodol., vol.29, no.3, pp.1–24, 2020.
[19] B. Miranda, "A Proposal for Revisiting Coverage Testing Metrics," Proc. IEEE/ACM Int'l Conf. Autom. Softw. Eng., pp.899–902, 2014.
[20] B. Miranda and A. Bertolino, "Social Coverage for Customized Test Adequacy and Selection Criteria," Proc. Int'l Workshop on Autom. of Softw. Test, pp.22–28, 2014.
[21] B. Miranda and A. Bertolino, "Improving Test Coverage Measurement for Reused Software," Proc. Euromicro Conf. Softw. Eng. and Advanced Applications, pp.27–34, 2015.
[22] H. Cox, "Differential coverage: automating coverage analysis," Proc. IEEE Conf. Softw. Testing, Verif. and Valid., pp.424–429, 2021.
[23] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur, "Applications of Synchronization Coverage," Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming, pp.206–212, 2005.
[24] E. Sherman, M.B. Dwyer, and S. Elbaum, "Saturation-Based Testing of Concurrent Programs," Proc. Joint Meeting of the European Softw. Eng. Conf. and the ACM SIGSOFT Symp. Foundations of Softw. Eng., pp.53–62, 2009.
[25] B. Křena, Z. Letko, and T. Vojnar, "Coverage Metrics for Saturation-Based and Search-Based Testing of Concurrent Software," Runtime Verif., pp.177–192, 2012.
[26] V. Terragni and S.-C. Cheung, "Coverage-Driven Test Code Generation for Concurrent Classes," Proc. Int'l Conf. Softw. Eng., pp.1121–1132, 2016.
[27] S. Park, R. Vuduc, and M.J. Harrold, "UNICORN: A Unified Approach for Localizing Non-Deadlock Concurrency Bugs," Softw. Test. Verif. Reliab., vol.25, no.3, pp.167–190, 2015.
[28] P. Hsia, X. Li, and D.C. Kung, "Class Testing and Code-Based Criteria," Proc. Conf. the Centre for Advanced Studies on Collaborative Research, p.14, 1996.
[29] M. Fisher, J. Wloka, F. Tip, B.G. Ryder, and A. Luchansky, "An Evaluation of Change-Based Coverage Criteria," Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Anal. for Softw. Tools, pp.21–28, 2011.
[30] S. Biswas, "Proposal for Control Dependency White-Box Test Coverage Metrics for Inheritance," Proc. Int'l Conf. Data Science and Business Analytics, pp.155–160, 2018.
[31] D.E. Perry and G.E. Kaiser, "Adequate Testing and Object-Oriented Programming," J. Obj. Oriented Prog., vol.2, no.5, pp.13–19, 1990.
[32] D. Mukherjee, D. Shekhar, and R. Mall, "Proposal for A Structural Integration Test Coverage Metric for Object-Oriented Programs," SIGSOFT Softw. Eng. Notes, vol.43, no.1, pp.1–4, 2018.
[33] D. Mukherjee, "A Novel Test Coverage Metric for Safety-Critical Software," IEEE Region 10 Conf., pp.486–491, 2019.
[34] M.H. Alalfi, J.R. Cordy, and T.R. Dean, "Automating Coverage Metrics for Dynamic Web Applications," Proc. European Conf. Softw. Maintenance and Reengineering, pp.51–60, 2010.
[35] M. Mirzaaghaei and A. Mesbah, "DOM-Based Test Adequacy Criteria for Web Applications," Proc. Int'l Symp. Softw. Testing and Anal., pp.71–81, 2014.
[36] H.V. Nguyen, H.D. Phan, C. Kästner, and T.N. Nguyen, "Exploring output-based coverage for testing PHP web applications," Autom. Softw. Eng., vol.26, no.1, pp.59–85, 2019.
[37] W. Hummer, O. Raz, O. Shehory, P. Leitner, and S. Dustdar, "Test Coverage of Data-Centric Dynamic Compositions in Service-Based Systems," IEEE Int'l Conf. Softw. Testing, Verif. and Valid., pp.40–49, 2011.
[38] H.M. Sneed and C. Verhoef, "Measuring test coverage of SoA services," Proc. Int'l Symp. the Maintenance and Evolution of Service-Oriented and Cloud-Based Environments, pp.59–66, 2015.
[39] A.M. Memon, M.L. Soffa, and M.E. Pollack, "Coverage Criteria for GUI Testing," Proc. European Softw. Eng. Conf. Held Jointly with ACM SIGSOFT Int'l Symp. Foundations of Softw. Eng., pp.256–267, 2001.
[40] B. Smith, Y. Shin, and L. Williams, "Proposing SQL Statement Coverage Metrics," Proc. Int'l Workshop on Softw. Eng. for Secure Systems, pp.49–56, 2008.
[41] T. Kim, C. Lee, K. Lee, S. Baik, C. Wu, and K. Yi, "Test Coverage Metric for Two-Staged Language with Abstract Interpretation," Asia-Pacific Softw. Eng. Conf., pp.301–308, 2009.
[42] P. Tsankov, M.T. Dashti, and D. Basin, "Semi-Valid Input Coverage for Fuzz Testing," Proc. Int'l Symp. Softw. Testing and Anal., pp.56–66, 2013.
[43] R. Jabbarvand, A. Sadeghi, H. Bagheri, and S. Malek, "Energy-Aware Test-Suite Minimization for Android Apps," Proc. Int'l Symp. Softw. Testing and Anal., pp.425–436, 2016.
[44] S. Nakajima and H.N. Bui, "Dataset Coverage for Testing Machine Learning Computer Programs," Asia-Pacific Softw. Eng. Conf., pp.297–304, 2016.
[45] J. Rott, R. Niedermayr, E. Juergens, and D. Pagano, "Ticket Coverage: Putting Test Coverage into Context," Proc. Workshop on Emerging Trends in Softw. Metrics, pp.2–8, 2017.
[46] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "Test Coverage Criteria for RESTful Web APIs," Proc. ACM SIGSOFT Int'l Workshop on Automating TEST Case Design, Selection, and Evaluation, pp.15–21, 2019.
[47] S. Ali, P. Arcaini, X. Wang, and T. Yue, "Assessing the Effectiveness of Input and Output Coverage Criteria for Testing Quantum Programs," Proc. IEEE Conf. Softw. Testing, Verif. and Valid., pp.13–23, 2021.