Master Thesis

Title

A Literature Review of Test Coverage Metrics

Supervisor
Prof. Shinji Kusumoto

by
Masayuki Taniguchi

February 1, 2023

Departmant of Computer Science
Graduate School of Information Science and Technology
Osaka University

Master Thesis

A Literature Review of Test Coverage Metrics

Masayuki Taniguchi

Abstract

Software testing plays an essential role in software quality assurance. It helps developers to reveal and remove bugs in software. Test coverage, such as statement and branch coverage, is widely known and used in software testing. Developers often use test coverage to measure the sufficiency of tests, to find untested statements, and to localize a faulty statement. In recent years, many researchers have proposed novel metrics for measuring test coverage of source code. However, 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 80 papers that propose coverage metrics. The survey results showed that the proposed 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 also identified the characteristics of each metric, such as effective domains, information needed to measure coverage, and granularity of measurement. Furthermore, we provide a catalog of coverage metrics to help developers and researchers select the best metrics for their context.

Keywords

Software testing, Test coverage, Coverage metrics, Literature review

Contents

List of Figures

List of Tables

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 on a validation activity, especially in a programmed test. The programmed test means to confirm whether the given program behaves as expected by the program execution. This paper refers to it as simply *test* or *testing*.

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

Well-known coverages, such as statement and branch coverage, are known to have limitations. For instance, 100% statement coverage does not guarantee that the source code has no bugs[2, 3]. To address their drawbacks, many researchers have proposed 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 such coverage metrics.

In this paper, we conduct a literature review of 80 papers on test coverage metrics. We aim to organize the studies that propose novel coverage metrics and to examine the characteristics of each proposed metric. Hence, our research questions are, *what are the objectives of the test coverage metrics proposals?* and *what kind of novel test coverage metrics are proposed?*. 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 coverage metrics for each proposal reason and domain. Through this study, we provide developers and researchers with a catalog of test coverage metrics and allow them to select suitable metrics in their context.

The rest of the paper is organized as follows. In Section 2, we present our research questions, describe the scope of our survey, and explain our methodology for collecting relevant studies. In Section 3, we show the backgrounds of proposals for test coverage metrics and the characteristics of each metric through the analysis of our research questions. In Section 4, we discuss future research directions on test coverage. In Section 5, we provide final remarks and future works.

2 Research Objective and Methodology

In this section, we present the two research questions that we aim to address. Following this, we describe the scope of our survey and show our methodology to select the relevant publications for analysis.

2.1 Research Questions

Our research questions (RQs) are as follows:

- **RQ1: What are the objectives of the test coverage metrics proposals?** This RQ explores the backgrounds of coverage metrics proposals. We answer this RQ by investigating the problems that each coverage metric aims to address.
- **RQ2: What kind of novel test coverage metrics are proposed?** In this RQ, we catalogue the proposed coverage metrics in the last few decades. This RQ provides details of each coverage metric.

2.2 Survey Scope

2.2.1 Test Coverage

Test coverage is defined as a metric in software testing that measures the amount of testing performed by a set of tests. The term *test coverage* is often used to refer to *code coverage* for white box testing (e.g., statement and branch coverage), but this is inaccurate. In addition to code coverage, there are various types of test coverage for different test types and objects, such as *specification coverage*[4] and *mutation coverage*[5].

We focus on tests that verify the implementation of programs. Therefore, we treat the term *test coverage* as the coverage of tests that check the correctness of source code. Note that we are not talking about the code coverage itself. For example, we consider both white box testing and black box testing as methods for testing source code, although they are different testing approaches.

2.2.2 Target Period

In 1963, Miller and Maloney[6] first mentioned the concept of test coverage. The authors explained that if a portion of a program is not executed by at least one test, the developer lacks the means to determine if that portion of code is executing correctly. Subsequently, many studies on test coverage were carried out from the 1970s to the 1980s [7, 8, 9, 10, 11].

We are interested in the newly proposed test coverage. Therefore, we target studies that propose test coverage metrics in the last three decades (1992-2022).

2.3 Paper Selection

We collected the papers for our study by using a specific set of keywords in some popular digital libraries. This paper collection was performed 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*, and *insufficiently tested code*. Since we intended to collect all papers related to our survey as much as possible,

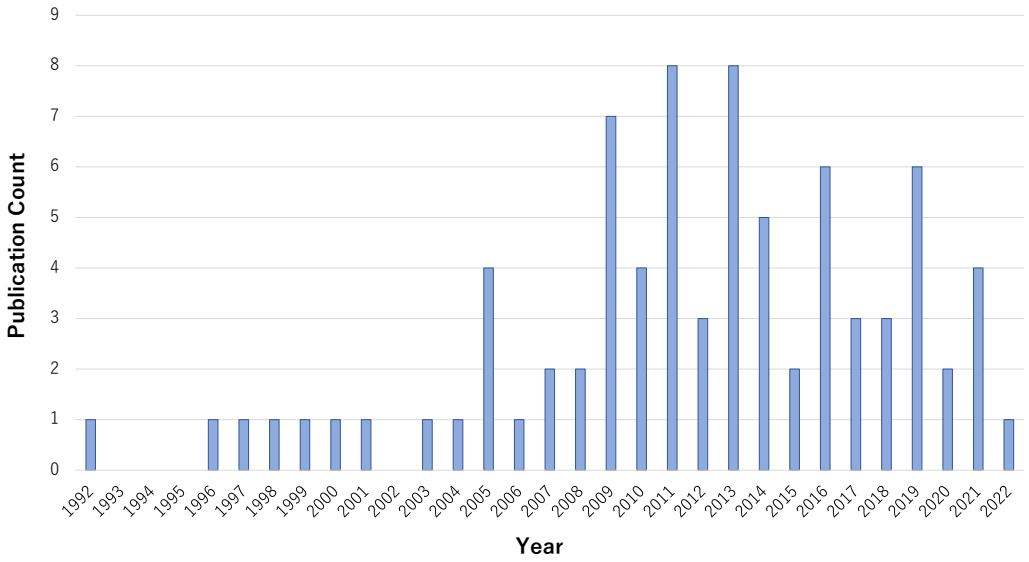

Figure 1: Publications per year from 1992 to 2022

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. Two people 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. Finally, we performed the forward and backward snowballing, i.e., analyzing publications citing or cited by the 43 papers. This activity took a month and produced 37 additional papers. Therefore, we ended up with a final set of 80 papers.

We analyzed the publication years and venues of selected papers. Figure 1 shows the number of published papers that propose coverage metrics per year. Until 2004, the number of papers was one or less per year; however, the number of published papers has increased since 2005.

Figure 2 shows the number of papers per venue. We explicitly reported venues where at least two publications were published, while we grouped all venues with only one publication together in the *Other* column. We found that many venues are related to software testing, software reliability, and software engineering. We also noticed that 13 publications (16.25%) appeared in journals, and 67 (83.75%) appeared in conferences/workshops/symposiums.

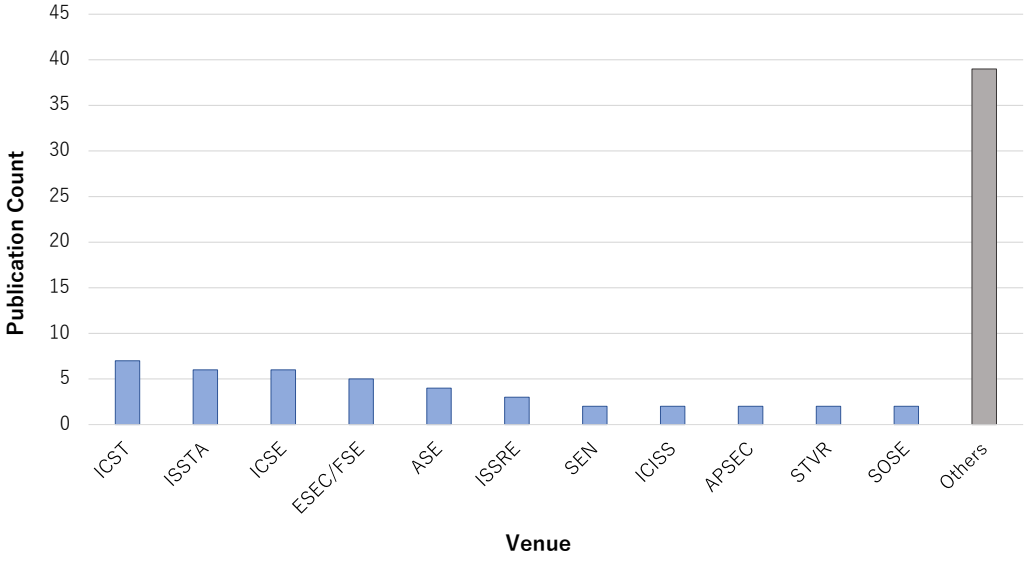

Figure 2: Publications per venue

3 Research Results

To answer RQ1, we examined the background of the proposed test coverage metrics for 80 studies. In this section, we describe the results of surveying 80 studies proposing coverage metrics. It is arranged into subsections according to the defined RQs.

3.1 RQ1: What are the objectives of the test coverage metrics proposals?

3.1.1 Backgrounds of Proposals for Coverage Metrics

To answer RQ1, we examined the backgrounds of 80 studies. The results show that a common reason for proposing coverage metrics is to address the limitations of existing coverages. We also found that each coverage can be divided into two groups: (1) general-purpose coverages that improve or complement traditional coverages and (2) domain-specific coverages that are effective in particular domains. General-purpose metrics were proposed in 20 studies, and 60 studies proposed domain-specific metrics. There are 12 domains, and the breakdown of the domains is summarized in Table 1.

3.1.2 Aims of General-purpose Coverages

This section explains the aims of the proposals for general-purpose coverage metrics.

Bug Detection. One of the aims of general-purpose coverages is to enhance the ability of tests to detect bugs. For example, Aghamohammadi et al.[12] argued that incorporating the frequency of executed statements into statement coverage leads to detecting faults more effectively. McMinn et al.[13] discussed the test coverage criteria that are well correlated with real fault revelation.

Oracle Quality. It is perfectly possible to achieve 100% coverage and still not have any result checked by even a single oracle. In the extreme case, a test code without any assertions at all that executes every statement in the source code achieves 100% statement coverage. Some researchers tackled this problem. For example, Schuler and Zeller[2, 14] proposed measuring the exhaustiveness of the verification of statements by assertions.

Table 1: Domains

Category	Domain	# papers
Programming Paradigm	Concurrent Programming	16
	Object-Oriented Programming	10
	Multi-Staged Programming	1
	Quantum Programming	1
Application type	Web Application	16
	Service-Oriented Architecture	7
	Database Application	3
	GUI application	2
	Android Application	1
	Machine Learning Program	1
Others	Fuzz Testing	1
	Agile Development	1

Test Effort. Writing additional tests can be difficult and costly. According to a survey of practicing software developers, a significant portion of this cost is due to the difficulty of identifying which parts of the code should be tested[15]. Several coverage metrics have been proposed to effectively identify entities to be tested. For example, Huo and Clause[16] proposed direct and indirect coverage that identify how entities are covered by tests. The authors demonstrated that faults in code indirectly tested are significantly less likely to be detected than those in code directly tested.

3.1.3 Aims of Domain-specific Coverages

In this section, we discuss the issues addressed by domain-specific coverage metrics.

Covering domain-specific entities. Most domain-specific coverage focuses on determining whether the test covers domain-specific elements. Examples of domain-specific elements include thread synchronization[17] and interleaving[18] in concurrent programs, inheritance[19] and polymorphism[20] in object-oriented programs, and SQL statements[21] and DOM elements[22] in web applications.

More detailed information on the elements that each domain-specific coverage metric targets is provided in Section 3.2.

Difficulty in Using Existing Coverages. In a few domains, it is difficult to take advantage of existing coverage. For example, since multi-staged programs generate and run new program code at runtime, Kim et al.[23] proposed to measure test coverage including generated code fragments detected by static analysis. Ali et al.[24] focused on quantum programs. Testing quantum programs is challenging for several reasons, such as the probabilistic nature and difficulties in estimating superposition states. The authors defined test oracles to determine the passing and failing of test suites and proposed test coverage metrics based on the inputs and outputs of a program.

3.2 RQ2: What kind of novel test coverage metrics are proposed?

This section describes the test coverage metrics proposed in the 80 studies we selected. We show the characteristics of each metric according to the classification of the proposal backgrounds from our analysis for RQ1. Note that some studies presented multiple levels of coverage measurement granularity. Table 2 summarizes the characteristics of each test coverage, including effective domain, overview, information required to measure coverage, and granularity of measurement.

3.2.1 General-purpose Coverage Metrics

Belli et al.[25] proposed a Test Segment Coverage as 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.[26] proposed a 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.[27] proposed a 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.[28], 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, 14] proposed a 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.[29] proposed a 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 the property based coverage criterion, we can measure the test coverage for all the possible values that variables in properties can take.

Whalen et al.[30] proposed an Observable Modified Condition/Decision Coverage (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.[31] proposed a Multi-Point Stride Coverage (MPSC). 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 a MPSC, 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[32].

Huo et al.[16] proposed Direct and 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[16].

McMinn et al.[13] proposed a 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 discuss the way to automatically generate fault coverage for software engineering by using a fault database such as Defects4J[33].

Byun et al.[34] proposed a Flag-Use Object Branch Coverage (Flag-Use OBC). 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.[35] proposed a Program State Coverage. This coverage metric improves the ability of line coverage to validate the effectiveness of the test suite. The authors consider 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.[12] proposed a 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.

Bollina and Gay[36] proposed a Bytecode-MCC, a new variant of Multiple Condition Coverage (MCC) for bytecode. The authors focused on masking, which occurs when one condition prevents another from influencing the output of a boolean expression. MCC is able to overcome masking within a single expression but cannot do so when conditions are defined in multiple expressions. Bytecode-MCC groups related Boolean expressions from the bytecode, reformulates the grouping into a single complex expression, and calculates all possible combinations of conditions within the constructed expression.

Miranda et al.[37, 38] proposed a 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[37], Social Coverage[37, 39], Relevant Coverage[37, 40], and Reachability Coverage[38]. 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 3.2.2.

Cox[41] proposed a 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.

Kolchin and Potiyenko[42] proposed a Required k-Use Chains Coverage. A k-use chain represents a sequence of k-length def-ues pairs (pairs of variable definitions and uses). Existing dataflow coverages do not directly require subpaths that pass a possible sequence of variable uses in conditions from a def-point to a use-point directly associated with the def-use pair. The proposed metrics measure the test coverage of sequences of def-use pairs.

3.2.2 Domain-specific Coverage Metrics

Concurrent Programming. Taylor et al.[43] pioneered a hierarchy of concurrency coverage criteria. The authors extended the notion of structural testing coverage criteria to concurrent programs. The proposed coverage metrics require tests to cover the paths of the concurrency state graph, in which the nodes represent states of concurrent programs.

Cheer-Sun et al.[44] proposed All-du-path Coverage for parallel programs. This metric considers du-pairs, triples (var, d, u) where var is a shared variable, d is a node in the parallel program flow graph (PPFG) where the value of var is defined, and u is a node in the PPFG where the value is read. In other words, this metric requires that a set of paths in which no node redefines the value of var is covered by tests.

Bron et al.[17] proposed a 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.

Lu et al.[18] proposed five concurrent program interleaving coverage metrics. A fundamental problem of concurrent program bug detection and testing is that the interleaving space is too large to be thoroughly explored. These interleaving coverage metrics are practical and effective for systematically exploring interleaving spaces and effectively finding concurrency bugs.

Trainin et al.[45] proposed coverage metrics for the detection of concurrent bugs. The proposed metrics focus on the use of synchronization primitives (for example, synchronization processes defined using the

Java keyword `synchronized`) and do not directly consider thread interleavings. The authors referred to two test coverage of synchronization block and synchronization pair.

Sherman et al.[46] proposed coverage metrics inspired by Synchronization Coverage[17]. These metrics 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.[47] proposed coverage metrics for saturation-based and search-based testing to reflect concurrency behavior accurately. In previous work[46], the identification of elements was 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 identifiers and thread identifiers. The authors derived 11 coverage metrics from dynamic analysis designed for discovering bugs in concurrent programs.

Yu et al.[48] proposed a coverage metric based on a set of interleaving idioms. An interleaving idiom is a pattern of inter-thread dependencies and the associated memory operations. In addition, interleaving idioms can represent more general interleaving patterns.

Tasiran et al.[49] proposed a Location-Pairs Coverage for shared-memory concurrent programs. A location-pair is a pair of two control locations that can access the same shared variable and can be executed by two different threads. The test coverage of location-pairs directly corresponds to the atomicity and refinement violations.

Steenbuck and Fraser[50] proposed a concurrency coverage metric. The authors focused on concurrent executions of combinations of shared memory access points with different schedules. Therefore, this metric requires covering all possible schedules for sets of threads, variables, and synchronization points.

Tasharofi et al.[51] proposed coverage metrics for concurrent programs based on the actor model[52]. In actor programs, concurrent entities communicate asynchronously by exchanging messages. The proposed metrics require covering all the sequence of receive events in the program execution at three levels.

Terragni et al.[53] proposed a 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.

Choudhary et al.[54] proposed a coverage metric based on the set of pairs of methods that execute concurrently. Enumerating all possible interleavings (i.e., total orders of memory accesses) is practically infeasible. Concurrent method pairs yield an effective approximation of possible and covered interleavings. Therefore, this test coverage metric leads to efficient and practical interleaving coverage

Wang et al.[3] proposed a MAP-coverage. This coverage is based on memory-access patterns (MAP), which are patterns of how shared variables are accessed by multiple threads[55]. MAP has often been shown to be associated with the nature of multi-threaded bugs[55]. Thus, comprehensive testing of all MAP is effective in finding bugs.

Guo et al.[56] proposed a All Synchronization Pairs Coverage. A synchronization pair is a pair of synchronization statements, such as `synchronize block`, `wait()`, and `notify()` of an object instance. The proposed metric measures the test coverage of synchronization pairs. Although there are too many interleavings in a multi-thread program with a high number of threads, this coverage allows interleavings to be checked efficiently.

Taheri and Gopalakrishnan et al.[57] proposed coverage metrics for concurrency in GO language. The existing concurrency coverage metrics are primarily in the context of Java and C/Pthreads. They are not necessarily applicable to languages like Go, as such languages have different concurrency primitives

and semantics. The authors proposed five coverage metrics that characterize the dynamic behavior of concurrency primitives.

Object Oriented (OO) Programming. Hsia et al.[58] 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 (e.g., type `enum`). 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.

Chen et al.[59] proposed Object-Flow Coverage criteria. The proposed metrics are based on binding of the possible classes with the program variables and object def-use pairs (pairs of object definitions and uses). We can identify object bindings and def-use pairs by using an object control flow graph whose nodes are statements that define or use an object.

Alexander et al.[60] proposed Coupling-Based Coverage metrics at three levels. These metrics focus on coupling relationships among procedures and measure the test coverage of coupling paths (paths between definition and use of state variables) of objects having dynamic binding.

Fisher et al.[61] 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.

Najumudheen et al.[20] proposed criteria for method call, inheritance, and polymorphic coverage of OO programs. These metrics are defined on the Call-based Object-Oriented System Dependence Graph (COSDG), a synthesis of the dependence graph, call graph, and flow graph. COSDG incorporates details such as class, dependence, flow, call, inheritance, and polymorphism.

Baldini et al.[62] proposed coverage metrics based on the program dependencies of methods. The authors introduced a Def/Use table that represents define-use pairs of methods based on the data members of every class. The proposed metrics measure the test coverages of all elements in the Def/Use table at three levels.

Biswas[19] 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 a new usage context[63]. Therefore, the cost of testing OO software can significantly exceed that of testing procedural programs. By using these metrics, we can effectively measure the test coverage of control dependencies associated with inheritance.

Mukherjee et al.[64,65] proposed coverage metrics in response to the fact that structural coverage metrics for integration testing of OO programs have been scarcely reported. These coverage metrics are based on data and control dependencies in the classes being integrated defined on JSysDG.

Mukherjee[66] also focused on testing safety-critical software, such as nuclear power plants, in the OO paradigm. Safety-critical software requires thorough testing; however, traditional coverage metrics suffer from several shortcomings. The authors proposed test coverage metrics that cover program dependencies more robustly and can detect faults at inter-object data dependencies.

Multi-Staged Programming. Kim et al.[23] proposed a New Decision Coverage for multi-staged language. Multi-staged language is a programming language that 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.

Quantum Programming. Ali et al.[24] proposed input/output coverage metrics for the 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 the inputs and outputs of the quantum program and measure the comprehensiveness of tests without destroying the superpositions of the quantum program.

Web Application. Sampath et al.[67] proposed coverage criteria intended for testing web applications at a page level. The authors defined URL coverage at nine levels. A URL consists of a base address and possible pairs of input field names and their values as per the RFC definition <protocol>://<host> [:<port>] [<path> [? <query>]][68]. The coverages report the static pages (URLs) visited during test execution for a given web application by using a user session.

Bai et al.[69] proposed test metrics based on Web Service Description Language (WSDL)[70] elements. The authors considered that testing of web services should cover four elements: parameters, messages (input and output), operations, and operation flows.

Smith et al.[21] proposed a 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.

Jokhio et al.[71] proposed two specification based coverage metrics: boundary coverage and transition rules path coverage. Boundary coverage refers to boundary conditions such as minimum and maximum values for parameters. Transition rules path coverage refers to the different execution paths that the program may follow when receiving a given request. These are based on the goal specification that specifies the user objectives.

Dao et al.[72] proposed coverage metrics for security testing of web applications. In many real web applications, security sensitive sinks (e.g., functions that write data) are wrapped by wrapper functions and called indirectly through these wrappers. The authors proposed three levels of coverage metrics for security sinks.

The authors also proposed a Security Sensitive Data Flow Coverage[73]. They considered that test cases should cover as many security sensitive data flows as possible to get high vulnerability detection effectiveness. This metric measures the test coverage of security sensitive branches that contain calls to security sensitive sinks or propagation of the values of security sensitive variables through assignments.

Alalfi et al.[74] proposed coverage metrics for dynamic web applications. Faults in web applications are 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.

Alshahwan and Harman[75] proposed coverage metrics based on HTML output uniqueness. The authors considered that raising the diversity of the output could lead to test suites that are more effective at exposing faults. The proposed metrics measure the test coverages of unique outputs at four levels. In a subsequent study[76], the authors extended their previous work[75] and proposed three additional coverage metrics.

Sakamoto et al.[77] proposed a Template Variable Coverage for a web application that is generated using template engines. Template variable coverage focuses on the variables and expressions for embedding in

HTML templates, which are important for testing a web application's functionality related to the dynamic content of an HTML document. This metric measures the test coverage of template variables.

Zou et al.[78] proposed a Hybrid Coverage criteria, which combines HTML element coverage with statement coverage of the code. The authors focused on richer iterations between client-side and server-side in dynamic web applications. Thus, the key idea of hybrid coverage is to combine the runtime client-side and server-side features of the web application. This coverage metric requires that tests cover all HTML elements and statements.

Zou et al.[79] also proposed a Virtual DOM (V-DOM) Coverage. This coverage metric is based on a V-DOM tree produced by a server script, a logical aggregation of all the possible DOM trees that represent pages on the client side. By measuring the test coverage of the V-DOM element in the V-DOM tree, we can consider executions on both server-side and client-side.

Mirzaaghaei et al.[22] proposed DOM Coverage metrics. Web application tests generally interact with the DOM. The authors argued 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 proposed six coverage metrics related to the DOM state.

Ed-douibi et al.[80] proposed coverage metrics for REST API. REST APIs are commonly described using languages such as the OpenAPI[1] Specification (OAS). The proposed metrics measure the test coverages of API elements based on the OAS at four levels.

Martin-Lopez et al.[81] proposed coverage metrics for RESTful API because there are no standardized coverage criteria for black-box testing of RESTful API. These metrics measure the 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.

Nguyen et al.[82] proposed coverage metrics for output-oriented testing of a dynamic web application. These coverage metrics measures test coverage of string literals output and decisions that affect the output. Using these metrics helps to identify presentation faults such as HTML validation errors and spelling errors.

Service-Oriented Architecture (SOA). Mei et al.[83] proposed coverage metrics for WS-BPEL[84] applications (a kind of service-oriented application). WS-BPEL applications use XPath extensively to integrate loosely-coupled workflow steps. The authors presented test coverage of the XPath query at three levels. These metrics aim to identify defects in service compositions that are caused by faulty (or ambiguous) XPath expressions selecting a different XML element at runtime than the one that the composition developer intended to be selected.

Bartolini et al.[85, 86] developed the notion of Relative Coverage that takes into consideration how the service is used by the client. Relative coverage adapts to a tester's context and measures the ratio between the covered entities and those that are considered relevant in the given situation and environment. The authors also proposed WSDL[70] based coverage metrics at three levels[87]. An interface of SOA is generally specified in WSDL. These metrics measure the test coverages of service interfaces.

Hummer et al.[88] proposed a k-Node Dataflow Coverage to significantly reduce the search space of service combinations in the integration test of dynamic composite Service-Based Systems (SBSs). This metric is based on the 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.[37, 38, 39] proposed a Social Coverage. This is the instance of relative cov-

[1]https://www.openapis.org/

erage[37, 38]. 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.[89] 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 the source code of the service under test.

Database Application. Cabal et al.[90] proposed test coverage of SQL SELECT queries. This coverage metric requires a coverage tree built from each SELECT statement encoding the conditions specified by the where and join clauses of the query. This means that 100% coverage is achieved when the entire coverage tree is covered.

Willmor and Embury[91] proposed eight test coverage metrics for database applications. These coverage metrics focus on the structural and data-oriented elements of database systems. By using the proposed metrics, we can check the test coverage of the structural aspects of the database application (e.g., operations, transaction statements, and entities represented in the database) and the possible def-use pairs of database system operations.

Halfond and Orso[92] proposed a Command-Form Coverage that is focused on the application-database interactions. Most database applications dynamically generate commands in the database language (usually SQL), pass these commands to the database for execution and process the results returned by the database. Thus, this coverage metric measures the test coverage of all the possible SQL command forms that can be issued at each database interaction point.

GUI Application. Memon et al.[93] 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.

Zhao and Cai[94] proposed GUI coverage metrics based on the event handlers in the source code of GUI applications. The authors focused on data interaction relationships between event handlers and defined two coverage metrics. These metrics measure the test coverages of event handlers and definition-use handler pairs (pairs of a handler that sets a variable and one that uses its variable.).

Android Application Jabbarvand et al.[95] proposed an 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 measure test coverage of energy-greedy segments that highly contribute to the energy consumption of the application.

Machine Learning Program. Nakajima et al.[96] proposed a Dataset Coverage for Machine Learning (ML) programs. The control structure of the 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.

Fuzz Testing. Tsankov et al.[97] proposed a 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.

Agile Development. Rott et al.[98] proposed a 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.

Table 2: Proposed Test Coverages

Coverage	Domain	Overview	Required information	Granularity
Test segment coverage[25]	General-purpose	Adjust thoroughness of test coverage	Control flow graph	Path
About variables[26]	General-purpose	Focus on important variables	Program slices	Path
State coverage[27]	General-purpose	Assess test oracle quality	Control flow graph and program slices	Variable validation by assertions
State coverage[28]	General-purpose	General definition of state coverage[27]	Control flow graph and program slices	Variable validation by assertions
Checked coverage[2, 14]	General-purpose	Assess test oracle quality	Dynamic slices	Statement validation by assertions
Property based coverage[29]	General-purpose	Test coverage of properties in assertions and annotation	All possible values of variables in properties	Value of variable in property
Observable MC/DC[30]	General-purpose	More rigorous than MC/DC by considering observability	Observability of boolean expressions	Condition and decision
Multi-point stride coverage[31]	General-purpose	Branch coverage considering execution order of branches	Execution order of branches	Sequence of branches
Direct/Indirect coverage[16]	General-purpose	Effectively identify insufficiently tested methods	Mapping entities to tests and methods	Any*
Fault coverage [13]	General-purpose	Coverage metrics in line with actual bugs	Fault database	Depends on auto-generated metric
Flag-Use Object Branch coverage[34]	General-purpose	At object code level with low dependence on compiler structure	Object code	Instruction
Program state coverage[35]	General-purpose	More effectively than line coverage with low execution cost	Number of execution of each line	Program state
Statement frequency coverage[12]	General-purpose	Overcome several shortcomings of program state coverage[35]	Number of execution of each statement	Frequency of execution of each statement
Bytecode-MCC[36]	General-purpose	Overcome the masking problem than MCC	Boolean expressions in bytecode	Condition
Relative coverage[37, 38]	General-purpose	Focus on in-scope entities	Usage scope	Any*
Operational coverage[37]	General-purpose	Focus on operations performed by a specific user group	Operational profile	Any*
Relevant coverage[37, 40]	General-purpose	Focus on entities of reused code in new (reuse) context	Input domain constraints	Any*
Reachability coverage[38]	General-purpose	Focus on input domain exercised by a specific user	Input domain expected to be exercised	Any*
Differential coverage[41]	General-purpose	Classify code coverage information into 12 categories	Version history	Any*
Required k-use chains coverage[42]	General-purpose	Cover subpaths not considered in existing data flow coverage	Set of sequences of def-use pairs	Sequence of def-use pairs
Concurrency coverage[43]	Concurrent prog.	Extend the notion of test coverage for procedural programs to concurrent programs	Concurrency state graph	State

Coverage	Domain	Overview	Required information	Granularity
All-du-path coverage[44]	Concurrent prog.	Extend the notion of test coverage for procedural programs to parallell programs	Parallel program flow graph	Path
Synchronization coverage[17]	Concurrent prog.	Practical coverage for concurrent programs	Runtime behavior of threads	Synchronization behavior
Interleaving coverage[18]	Concurrent prog.	Practical and effective interleaving coverage	Runtime behavior of threads	Interleaving
Synchronisation primitive coverage[45]	Concurrent prog.	Focuses on the use of synchronisation primitives	Runtime behavior of threads	Synchronization primitive
Saturation-based coverage[46]	Concurrent prog.	No need to estimate the executable domain of each metric	Runtime behavior of threads	Synchronization behavior
Concurrency behavior coverage[47]	Concurrent prog.	Accurately identify behavior of threads than previous work[46]	Runtime behavior of threads and objects	Synchronization behavior
Interleaving coverage[48]	Concurrent prog.	Corresponds to a more general interleaving pattern	Set of thread interleaving idioms	Interleaving idiom
Location-pairs coverage[49]	Concurrent prog.	Directly corresponds to the atomicity and refinement violations	Set of location-pairs	Location-pair
Concurrency coverage[50]	Concurrent prog.	Focus on concurrent execution schedules	Set of thread interleavings	Thread interleaving
For actor program[51]	Concurrent prog.	Focus on schedules of receiving message events	All possible sequences of receive events	Sequence of receive events
Sequential coverage[53]	Concurrent prog.	Measure coverage by a single thread execution	Possible method call sequences	Sequence of events
Concurrent method pairs coverage[54]	Concurrent prog.	Leads to efficient and practical interleaving coverage	Set of of concurrent method pairs	Concurrent method pair
MAP-coverage[3]	Concurrent prog.	Help find multi-threaded bugs	Possible memory-access patterns[55]	Memory-access pattern[55]
All synchronization pairs coverage[56]	Concurrent prog.	Check thread interleavings efficiently	Synchronization pair thread graph	Synchronization pair
For GO lang.[57]	Concurrent prog.	Focus on concurrency in GO language	Set of concurrency primitives	Concurrency primitive
Based on enumerate data member[58]	OO prog.	Reveal faults related to object states	Possible object assignments	Object assignment
Object-flow coverage[59]	OO prog.	Focus on inheritance, polymorphism and dynamic behaviour of objects	Object control flow graph	Binding or def-use pair
Coupling-based coverage[60]	OO prog.	Focus on coupling relationships among procedures	Control flow graph	Coupling path
Change-based coverage[61]	OO prog.	Expected to be effective in revealing regression faults	Version history	Any*
Method call coverage[20]	OO prog.	Consider the dependencies that are unique to OO programs	COSDG	Method call
Def-use pair coverage[62]	OO prog.	Focuse on define and use of data members in classes	Def/Use table	Def-use pair
Inheritance coverage[19]	OO prog.	Consider dependencies through inheritance	JSysDG	Control dependency

Coverage	Domain	Overview	Required information	Granularity
For integration testing[64, 65]	OO prog.	Detect bugs missed in scenario-based integration testing	JSysDG	Data and control dependency
For safety-critical software[66]	OO prog.	Cover program dependencies robustly	JSysDG	Def-use pair
New decision coverage[23]	Two-staged lang.	Handle branches included in code generated at runtime	Code fragments generated in execution	Decision
Input/Output coverage[24]	Quantum prog.	Test coverage without destroying superpositions of quantum programs	Set of valid input/output values	Input/Output value
Page coverage[67]	Web app.	Focus on dynamic characteristics of web applications	User session	Web page (URL)
WSDL-based coverage[69]	Web app.	Focus on WSDL elements	Set of WSDL elements	WSDL element
SQL statement coverage[21]	Web app.	Provide specific information about insufficient or missing input validation	Set of sql statements or input variables	Sql statement or input variable
Specification-based coverage[71]	Web app.	Based on the goal specification that specifies the user objectives	Goal specification	Boundary condition or transition rule path
Sequrity sink coverage[72]	Web app.	Evaluate the quality of a test suite in revealing security vulnerabilities	Set of security sinks	Security sink
Interaction coverage[74]	Web app.	Cover complex interactions between client and database	Instrumentation transformation	Entity related to interactions
Security sensitive data flow coverage[73]	Web app.	Cover as many security sensitive data flows as possible	Set of security sensitive branches	Security sensitive branch
Output uniquness coverage[75, 76]	Web app.	Effectively expose faults by raising the diversity of the output	Possible values of outputs	Output value
Template variable coverage[77]	Web app.	Validate the dynamic contents corresponding to the execution results	Set of template variables	Template variable
Hybrid Coverage[78]	Web app.	Focuse on richer iterations between client-side and server-side	Set of HTML elements and statements	HTML element and statement
Virtual DOM coverage[79]	Web app.	Consider executions on both server-side and client-side	V-DOM tree	V-DOM
DOM coverage[22]	Web app.	Provide information about DOM in web application testing	DOM state flow graph	DOM state or element
API element coverage[80]	Web app.	Focus on REST API testing relying on API specifications	Set of API element	API element
API request/response coverage[81]	Web app.	Focus on elements related to API requests and responses	Set of entities related to API requests and responses	Entities related to API requests and responses
Output coverage[82]	Web app.	Help identify presentation faults such as HTML validation errors	Possible output produced from string literals	String literal or decision

Coverage	Domain	Overview	Required information	Granularity
XPath query coverage[83]	SOA	Identifying service composition faults caused by XPath expressions	XPath rewriting graph	XPath query
Relative coverage[85, 86]	SOA	Focus on entities in test scope	Test scope	Any*
WSDL-based coverage[87]	SOA	Focus on service interfaces	WSDL specification	Service interface
k-Node dataflow coverage[88]	SOA	Reduce the search space of service combinations in integration test	Trees of data dependencies between services	Dataflow
Social coverage[37, 38, 39]	SOA	Focus on operations of interest to testers of SOA-based systems	Test community	Operation
Input/Output message coverage[89]	SOA	Evaluate test quality without considering the source code in SOA	Complete input and output domains of service	Parameter in messages
SQL coverage[90]	Database app.	Help to detect faults at the level of SELECT statements	Coverage tree	SQL SELECT query
For database app.	Database app.	Focus on the structural and data-oriented elements	Control flow graph	Structural element or def-use pair
Command-form coverage[92]	Database app.	Focus on the application-database interactions	Character-level NFAs	Database command
GUI event coverage[93]	GUI app.	Handle GUI event sequences that are much more abstract than code	Event flow graph	Event or event sequence
GUI event handler coverage[94]	GUI app.	Focuse on data interaction relationships between event handlers	Set of event handlers	Event handler
eCoverage[95]	Android app.	Consider the energy consumption of segments (methods or system APIs)	Call graph of segments	Energy-greedy segment
Dataset coverage[96]	ML programs	Evaluate test quality in terms of dataset variety	Linearly separable dataset	Variety of datasets
Semi-valid input coverage[97]	Fuzz Testing	Measure how well tests covers the domain of semi-valid inputs	Input constraints	Input
Ticket coverage[98]	Agile develop.	Help focus testing efforts on changed code in a ticket	Commits related to a ticket	Method

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

prog. = programming, app. = application, ML = Machine Learning, develop. = development

4 Discussion

This section takes a high-level view of the studies covered in our survey. Through this activity, we discuss future directions for research on test coverage metrics.

4.1 Domains of Studies

As shown in Section 2.3, the number of studies that propose test coverage metrics tends to increase after 2005. We consider that the cause of this is the recent increase in the scale and complexity of software development and the diversity of software (e.g., the emergence of new programming paradigms).

We identified the domains of coverage proposal studies in our analysis for RQ1 and summarized them in Table 1. The results show that the research field is well-established in concurrent programming, object-oriented programming, web applications, and service-oriented architecture. On the other hand, there are a few studies in some domains, such as quantum programming and android applications. This analysis suggests that there are many domains in need of effective coverage metrics.

4.2 Available Tools

Many tools for test coverage measurement were developed in selected papers. However, most of them were created for the evaluation of proposed coverage metrics and are not available. As of January 2023, only nine tools are available. These tools are listed in Table 3. This makes it difficult for developers to actually use the proposed coverage metrics or for researchers to evaluate the coverage. Therefore, it is desirable to build available tools for coverage measurement.

4.3 Evaluation of Coverage

The 13 studies did not evaluate their proposed coverage metrics [13,19,23,29,38,41,42,58,64,66,69,71, 91]. Although the theoretical definitions of coverage metrics were discussed, it is challenging to determine their actual effectiveness without evaluation. From a practical standpoint, it is necessary to evaluate each metric to accurately assess their effectiveness, including their ability to measure test effectiveness and the cost of using them.

Table 3: Available Coverage Measurement Tools

Tool	Coverage	URL
no name[34]	Flag-use OBC	`https://github.com/tj-byun/object-coverage-criteria`
gendiffcov[41]	Differential coverage	`https://github.com/henry2cox/lcov/tree/diffcov_initial`
Maple[48]	Interleaving coverage	`https://github.com/jieyu/maple`
CovCon[54]	Concurrent method pairs coverage	`https://github.com/michaelpradel/ConTeGe/tree/CovCon`
MAPTest[3]	MAP-coverage	`https://github.com/sail-repos/Map-Coverage`
GOAT[57]	For GO language	`https://github.com/staheri/goat`
DomCovery[22]	DOM coverage	`https://github.com/saltlab/DomCovery`
no name[80]	API element coverage	`https://github.com/opendata-for-all/api-tester`
WebTest[82]	Output coverage	`https://github.com/git1997/VarAnalysis`

5 Conclusion and Future Work

In this paper, we conducted a survey of papers proposing test coverage metrics in the last three decades. We analyzed 80 papers and answered two research questions to organize proposed coverage metrics. Our first research question considered the backgrounds of proposals for each metric. We investigated the problems that each coverage aims to address. As a result, we found that proposed coverage metrics can be classified into two categories: (1) general-purpose metrics that improve or complement traditional coverage and (2) domain-specific metrics that are effective in particular domains. Both metrics aim to address the limitations of existing coverage. Generic coverage aims to improve or complement traditional coverage in a domain-independent context, while domain-specific coverage focuses on testing domain-specific elements. Our second research question set out to identify and organize the characteristics of proposed test coverages. To that end, we examined and summarized the overview, the domain, the necessary information for measurement, and the granularity of measurement for each coverage metric. A catalog of novel coverage metrics would help developers and researchers to select suitable metrics in their context.

Our future work includes the following:

Development of coverage measurement tools. We found that only nine of the 80 studies disclosed the tools to the public. We consider that it is important to develop and publish the coverage measurement tool. Creating tools and explaining their design and implementation will help developers and researchers. Furthermore, making the tools available will assist in the actual development and facilitates the comparison of coverage metrics.

Evaluation of coverage metrics. Thirteen studies did not perform the evaluation of the proposed coverage metric. It is difficult to consider the use of coverage metrics without assessing their ability to measure test effectiveness and the costs of using them. Therefore, we plan to evaluate the proposed coverages from a practical standpoint. We also intend to identify the advantages and disadvantages of each metric by conducting a comparative evaluation. This analysis will result in the determination of the optimal coverage from multiple perspectives.

Acknowledgements

I am deeply grateful to Shinsuke Matsumoto, Assistant Professor, for his careful support and guidance throughout the entire process of this study. He provided enthusiastic and attentive guidance from the consultation stage to the writing of my thesis. In addition, I was also provided with valuable advice that will be beneficial to my future career outside of my research.

I would like to extend my sincere thanks to Professor Shinji Kusumoto. During my undergraduate years, he served as the head of my major. As a result, I was taken care of by him throughout my entire university life.

I would also like to express my gratitude to Professor Yoshiki Higo for his invaluable insights and guidance. His expertise and experience have been instrumental in shaping the study.

I am also thankful to all the members at Kusumoto Laboratory for their support and encouragement. I had many enlightening conversations and discussions with them that deepened my understanding of information technology and many other areas. I learned so much from them and am grateful for the opportunities to grow together.

I am grateful to the professors at the Graduate School of Information Science and Technology, Osaka University, for their help in lectures and exercises leading up to this research.

Finally, I would like to express my deepest gratitude to my family for their support and encouragement over the past 25 years.

References

[1] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.

[2] David Schuler and Andreas Zeller. Checked coverage: an indicator for oracle quality. *Software: Testing, Verification and Reliability*, 23(7):531–551, 2013.

[3] Zan Wang, Yingquan Zhao, Shuang Liu, Jun Sun, Xiang Chen, and Huarui Lin. MAP-Coverage: A Novel Coverage Criterion for Testing Thread-Safe Classes. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 722–734, 2019.

[4] Michael Harder, Benjamin Morse, and Michael D Ernst. Specification coverage as a measure of test suite quality. *MIT Lab for Computer Science*, 2001.

[5] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. *Mutation Testing Advances: An Analysis and Survey*. 2018.

[6] Joan C. Miller and Clifford J. Maloney. Systematic Mistake Analysis of Digital Computer Programs. *Communications of the ACM*, 6(2):58–63, 1963.

[7] J. R. Brown and R. H. Hoffman. Evaluating the Effectiveness of Software Verification: Pratical Experience with an Automated Tool. In *Proceedings of the Fall Joint Computer Conference, Part I*, pages 181–190, 1972.

[8] John B. Goodenough and Susan L. Gerhart. Toward a Theory of Test Data Selection. *SIGPLAN Notices*, 10(6):493–510, 1975.

[9] Martin R. Woodward Michael A. Hennell and David Hedley. On program analysis. *Information Processing Letters*, 5(5):136–140, 1976.

[10] Martin R. Woodward, David Hedley, and Michael A. Hennell Hennell. Experience with Path Analysis and Testing of Programs. *IEEE Transactions on Software Engineering*, SE-6(3):278–286, 1980.

[11] Phyllis Frankl and Elaine J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, 1988.

[12] Alireza Aghamohammadi, Seyed-Hassan Mirian-Hosseinabadi, and Sajad Jalali. Statement frequency coverage: A code coverage criterion for assessing test suite effectiveness. *Information and Software Technology*, 129:106426, 2021.

[13] Phil McMinn, Mark Harman, Gordon Fraser, and Gregory M. Kapfhammer. Automated Search for Good Coverage Criteria: Moving from Code Coverage to Fault Coverage through Search-Based Software Engineering. In *Proceedings of the International Workshop on Search-Based Software Testing*, pages 43–44, 2016.

[14] David Schuler and Andreas Zeller. Assessing Oracle Quality with Checked Coverage. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation*, pages 90–99, 2011.

[15] Ermira Daka and Gordon Fraser. A Survey on Unit Testing Practices and Problems. In *Proceedings of the IEEE International Symposium on Software Reliability Engineering*, pages 201–211, 2014.

[16] Chen Huo and James Clause. Interpreting Coverage Information Using Direct and Indirect Coverage. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation*, pages 234–243, 2016.

[17] Arkady Bron, Eitan Farchi, Yonit Magid, Yarden Nir, and Shmuel Ur. Applications of Synchronization Coverage. In *Proceedings of the ACM SIGPLAN Symposium Principles and Practice of Parallel Programming*, pages 206–212, 2005.

[18] Shan Lu, Weihang Jiang, and Yuanyuan Zhou. A Study of Interleaving Coverage Criteria. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 533–536, 2007.

[19] Sourav Biswas. Proposal for Control Dependency White-Box Test Coverage Metrics for Inheritance. In *Proceedings of the International Conference on Data Science and Business Analytics*, pages 155–160, 2018.

[20] E. S. F. Najumudheen, Rajib Mall, and Debasis Samanta. Test Coverage Analysis Based on an Object-Oriented Program Model. *Journal of Software Maintenance and Evolution: Research and Practice*, 23(7):465–493, 2011.

[21] Ben Smith, Yonghee Shin, and Laurie Williams. Proposing SQL Statement Coverage Metrics. In *Proceedings of the International Workshop on Software Engineering for Secure Systems*, pages 49–56, 2008.

[22] Mehdi Mirzaaghaei and Ali Mesbah. DOM-Based Test Adequacy Criteria for Web Applications. In *Proceedings of the International Symposium Software Testing and Analysis*, pages 71–81, 2014.

[23] Taeksu Kim, Chunwoo Lee, Kiljoo Lee, Soohyun Baik, Chisu Wu, and Kwangkeun Yi. Test Coverage Metric for Two-Staged Language with Abstract Interpretation. In *Proceedings of the Asia-Pacific Software Engineering Conference*, pages 301–308, 2009.

[24] Shaukat Ali, Paolo Arcaini, Xinyi Wang, and Tao Yue. Assessing the Effectiveness of Input and Output Coverage Criteria for Testing Quantum Programs. In *Proceedings of the IEEE Conference on Software Testing, Verification and Validation*, pages 13–23, 2021.

[25] Fevzi Belli and Javier Dreyer. Program Segmentation for Controlling Test Coverage. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 72–83, 1997.

[26] Zhenqiang Chen, Baowen Xu, Hongji Yang, and Huowang Chen. Test Coverage Analysis Based on Program Slicing. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, pages 559–565, 2003.

[27] Kenneth Koster and David C. Kao. State coverage: a structural test adequacy criterion for behavior checking. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium Foundations of Software Engineering*, pages 541–544, 2007.

[28] Dries Vanoverberghe, Jonathan de Halleux, Nikolai Tillmann, and Frank Piessens. State Coverage: Software Validation Metrics beyond Code Coverage. In *SOFSEM 2012: Theory and Practice of Computer Science*, pages 542–553, 2012.

[29] Fadi Zaraket and Wes Masri. Property Based Coverage Criterion. In *Proceedings of the International Workshop on Defects in Large Software Systems: Held in Conjunction with the ACM SIGSOFT International Symposium Software Testing and Analysis*, pages 27–28, 2009.

[30] Michael Whalen, Gregory Gay, Dongjiang You, Mats P. E. Heimdahl, and Matt Staats. Observable Modified Condition/Decision Coverage. In *Proceedings of the International Conference on Software Engineering*, pages 102–111, 2013.

[31] Mohammad Mahdi Hassan and James H. Andrews. Comparing Multi-Point Stride Coverage and Dataflow Coverage. In *Proceedings of the International Conference on Software Engineering*, pages 172–181, 2013.

[32] S. Rapps and E.J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, 1985.

[33] René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the International Symposium Software Testing and Analysis*, pages 437–440, 2014.

[34] Taejoon Byun, Vaibhav Sharma, Sanjai Rayadurgam, Stephen McCamant, and Mats P. E. Heimdahl. Toward Rigorous Object-Code Coverage Criteria. In *Proceedings of the International Symposium Software Reliability Engineering*, pages 328–338, 2017.

[35] Khashayar Etemadi Someoliayi, Sajad Jalali, Mostafa Mahdieh, and Seyed-Hassan Mirian-Hosseinabadi. Program State Coverage: A Test Coverage Metric Based on Executed Program States. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering*, pages 584–588, 2019.

[36] Srujana Bollina and Gregory Gay. Bytecode-Based Multiple Condition Coverage: An Initial Investigation. In *Search-Based Software Engineering*, pages 220–236, 2020.

[37] Breno Miranda and Antonia Bertolino. Testing Relative to Usage Scope: Revisiting Software Coverage Criteria. *ACM Transactions on Software Engineering Methodology*, 29(3):1–24, 2020.

[38] Breno Miranda. A Proposal for Revisiting Coverage Testing Metrics. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 899–902, 2014.

[39] Breno Miranda and Antonia Bertolino. Social Coverage for Customized Test Adequacy and Selection Criteria. In *Proceedings of the International Workshop on Automation of Software Test*, pages 22–28, 2014.

[40] Breno Miranda and Antonia Bertolino. Improving Test Coverage Measurement for Reused Software. In *Proceedings of the Euromicro Conference on Software Engineering and Advanced Applications*, pages 27–34, 2015.

[41] Henry Cox. Differential coverage: automating coverage analysis. In *Proceedings of the IEEE Conference on Software Testing, Verification and Validation*, pages 424–429, 2021.

[42] Alexander Kolchin and Stepan Potiyenko. Extending Data Flow Coverage to Test Constraint Refinements. In *Integrated Formal Methods*, pages 313–321, 2022.

[43] Richard N. Taylor, David L. Levine, and Cheryl D. Kelly. Structural testing of concurrent programs. *IEEE Transactions on Software Engineering*, 18(3):206–215, 1992.

[44] Cheer-Sun D. Yang, Amie L. Souter, and Lori L. Pollock. All-Du-Path Coverage for Parallel Programs. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 153–162, 1998.

[45] Ehud Trainin, Yarden Nir-Buchbinder, Rachel Tzoref-Brill, Aviad Zlotnick, Shmuel Ur, and Eitan Farchi. Forcing Small Models of Conditions on Program Interleaving for Detection of Concurrent Bugs. In *Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, pages 1–6, 2009.

[46] Elena Sherman, Matthew B. Dwyer, and Sebastian Elbaum. Saturation-Based Testing of Concurrent Programs. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium Foundations of Software Engineering*, pages 53–62, 2009.

[47] Bohuslav Křena, Zdeněk Letko, and Tomáš Vojnar. Coverage Metrics for Saturation-Based and Search-Based Testing of Concurrent Software. In *Runtime Verification*, pages 177–192, 2012.

[48] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. Maple: A Coverage-Driven Testing Tool for Multithreaded Programs. *ACM SIGPLAN Notices*, 47(10):485–502, 2012.

[49] Serdar Tasiran, M. Erkan Keremoğlu, and Kivanç Muşlu. Location Pairs: A Test Coverage Metric for Shared-Memory Concurrent Programs. *Empirical Software Engineering*, 17(3):129–165, 2012.

[50] Sebastian Steenbuck and Gordon Fraser. Generating Unit Tests for Concurrent Classes. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, pages 144–153, 2013.

[51] Samira Tasharofi, Michael Pradel, Yu Lin, and Ralph Johnson. Bita: Coverage-guided, automatic testing of actor programs. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 114–124, 2013.

[52] Gul Agha. ACTORS: A Model of Concurrent Computation in Distributed Systems. *USA: MIT Press*, 2004.

[53] Valerio Terragni and Shing-Chi Cheung. Coverage-Driven Test Code Generation for Concurrent Classes. In *Proceedings of the International Conference on Software Engineering*, pages 1121–1132, 2016.

[54] Ankit Choudhary, Shan Lu, and Michael Pradel. Efficient Detection of Thread Safety Violations via Coverage-Guided Generation of Concurrent Tests. In *Proceedings of the IEEE/ACM International Conference on Software Engineering*, pages 266–277, 2017.

[55] Sangmin Park, Richard Vuduc, and Mary Jean Harrold. UNICORN: A Unified Approach for Localizing Non-Deadlock Concurrency Bugs. *Software Testing, Verification and Reliability*, 25(3):167–190, 2015.

[56] JunXia Guo, Zheng Li, CunFeng Shi, and RuiLian Zhao. Thread Scheduling Sequence Generation Based on All Synchronization Pair Coverage Criteria. *International Journal of Software Engineering and Knowledge Engineering*, 30(01):97–118, 2020.

[57] Saeed Taheri and Ganesh Gopalakrishnan. GoAT: Automated Concurrency Analysis and Debugging Tool for Go. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 138–150, 2021.

[58] Pei Hsia, Xiaolin Li, and David C. Kung. Class Testing and Code-Based Criteria. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, page 14, 1996.

[59] Mei-Hwa Chen and H.M. Kao. Testing Object-Oriented Programs - An Integrated Approach. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 73–82, 1999.

[60] Roger Alexander, Jeff Offutt, and Andreas Stefik. Testing Coupling Relationships in Object-Oriented Programs. *Software Testing, Verification and Reliability*, 20:291–327, 2010.

[61] Marc Fisher, Jan Wloka, Frank Tip, Barbara G. Ryder, and Alexander Luchansky. An Evaluation of Change-Based Coverage Criteria. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, pages 21–28, 2011.

[62] Fabrizio Baldini, Giacomo Bucci, Leonardo Grassi, and Enrico Vicario. Test Coverage Analysis for Object Oriented Programs - Structural Testing Through Aspect Oriented Instrumentation. In *Proceedings of the International Conference on Software and Data Technologies*, 2016.

[63] Dewayne E. Perry and Gail E. Kaiser. Adequate Testing and Object-Oriented Programming. *Journal of Object-Oriented Programming*, 2(5):13–19, 1990.

[64] Debashis Mukherjee, Dibyanshu Shekhar, and Rajib Mall. Proposal for A Structural Integration Test Coverage Metric for Object-Oriented Programs. *SIGSOFT Software Engineering Notes*, 43(1):1–4, 2018.

[65] Debashis Mukherjee and Rajib Mall. An integration test coverage metric for Java programs. *International Journal of System Assurance Engineering and Management*, 10(4):576–601, 2019.

[66] Debashis Mukherjee. A Novel Test Coverage Metric for Safety-Critical Software. In *Proceedings of the TENCON - IEEE Region 10 Conference*, pages 486–491, 2019.

[67] Sreedevi Sampath, Emily Hill, Sara Sprenkle, and Lori Pollock. Coverage Criteria for Testing Web Applications. *Computer and Information Sciences, University of Delaware*, pages Technical Report 2005–17, 2005.

[68] *RFC 1738: Uniform resource locators (url).*

[69] Xiaoying Bai, Wenli Dong, Wei-Tek Tsai, and Yinong Chen. WSDL-based automatic test case generation for Web services testing. In *Proceedings of the IEEE International Workshop on Service-Oriented System Engineering*, pages 207–212, 2005.

[70] WWW Consortium. *WSDL version 2.0.*, 2007.

[71] M. Shaban Jokhio, Gillian Dobbie, and Jing Sun. Towards Specification Based Testing for Semantic Web Services. In *Australian Software Engineering Conference*, pages 54–63, 2009.

[72] Thanh Binh Dao and Etsuya Shibayama. Coverage Criteria for Automatic Security Testing of Web Applications. In *Information Systems Security*, pages 111–124, 2010.

[73] Thanh Binh Dao and Etsuya Shibayama. Security Sensitive Data Flow Coverage Criterion for Automatic Security Testing of Web Applications. In *Engineering Secure Software and Systems*, pages 101–113, 2011.

[74] Manar H. Alalfi, James R. Cordy, and Thomas R. Dean. Automating Coverage Metrics for Dynamic Web Applications. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 51–60, 2010.

[75] Nadia Alshahwan and Mark Harman. Augmenting test suites effectiveness by increasing output diversity. In *Proceedings of the International Conference on Software Engineering*, pages 1345–1348, 2012.

[76] Nadia Alshahwan and Mark Harman. Coverage and Fault Detection of the Output-Uniqueness Test Selection Criteria. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 181–192, 2014.

[77] Kazunori Sakamoto, Kaizu Tomohiro, Daigo Hamura, Hironori Washizaki, and Yoshiaki Fukazawa. POGen: A Test Code Generator Based on Template Variable Coverage in Gray-Box Integration Testing for Web Applications. In *Fundamental Approaches to Software Engineering*, pages 343–358, 2013.

[78] Yunxiao Zou, Chunrong Fang, Zhenyu Chen, Xiaofang Zhang, and Zhihong Zhao. A Hybrid Coverage Criterion for DynamicWeb Testing (S). In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering*, 2013.

[79] Yunxiao Zou, Zhenyu Chen, Yunhui Zheng, Xiangyu Zhang, and Zebao Gao. Virtual DOM Coverage for Effective Testing of Dynamic Web Applications. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 60–70, 2014.

[80] Hamza Ed-douibi, Javier Luis Cánovas Izquierdo, and Jordi Cabot. Automatic Generation of Test Cases for REST APIs: A Specification-Based Approach. In *Proceedings of the International Enterprise Distributed Object Computing Conference*, pages 181–190, 2018.

[81] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. Test Coverage Criteria for RESTful Web APIs. In *Proceedings of the ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, pages 15–21, 2019.

[82] Hung Viet Nguyen, Hung Dang Phan, Christian Kästner, and Tien N. Nguyen. Exploring output-based coverage for testing PHP web applications. *Automated Software Engineering*, 26(1):59–85, 2019.

[83] Lijun Mei, W.K. Chan, and T.H. Tse. Data Flow Testing of Service-Oriented Workflow Applications. In *Proceedings of the International Conference on Software Engineering*, pages 371–380, 2008.

[84] Charlton Barreto, Vaughn Bullard, Thomas Erl, John Evdemon, Diane Jordan, Khanderao Kand, Dieter König, Simon Moser, Ralph Stout, Ron Ten-Hove, Ivana Trickovic, Danny V. D. Rijn, and Alex Yiu. Web services business process execution language version 2.0: Primer. In *OASIS*, 2007.

[85] Cesare Bartolini, Antonia Bertolino, Sebastian Elbaum, and Eda Marchetti. Whitening SOA Testing. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 161–170, 2009.

[86] Marcelo Medeiros Eler, Antonia Bertolino, and Paulo Cesar Masiero. More testable service compositions by test metadata. In *Proceedings of the IEEE International Symposium on Service Oriented System*, pages 204–213, 2011.

[87] Cesare Bartolini, Antonia Bertolino, Eda Marchetti, and Andrea Polini. WS-TAXI: A WSDL-based Testing Tool for Web Services. In *Proceedings of the International Conference on Software Testing Verification and Validation*, pages 326–335, 2009.

[88] Waldemar Hummer, Orna Raz, Onn Shehory, Philipp Leitner, and Schahram Dustdar. Test Coverage of Data-Centric Dynamic Compositions in Service-Based Systems. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation*, pages 40–49, 2011.

[89] Harry M. Sneed and Chris Verhoef. Measuring test coverage of SoA services. In *Proceedings of the International Symposium the Maintenance and Evolution of Service-Oriented and Cloud-Based Environments*, pages 59–66, 2015.

[90] María José Suárez-Cabal and Javier Tuya. Using an SQL Coverage Measurement for Testing Database Applications. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 253–262, 2004.

[91] David Willmor and Suzanne Embury. Exploring test adequacy for database systems. *UK Software Testing Research Workshop*, pages 123–133, 2005.

[92] William G.J. Halfond and Alessandro Orso. Command-Form Coverage for Testing Database Applications. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 69–80, 2006.

[93] Atif M. Memon, Mary Lou Soffa, and Martha E. Pollack. Coverage Criteria for GUI Testing. In *Proceedings of the European Software Engineering Conference Held Jointly with ACM SIGSOFT International Symposium Foundations of Software Engineering*, pages 256–267, 2001.

[94] Lei Zhao and Kai-Yuan Cai. Event Handler-Based Coverage for GUI Testing. In *Proceedings of the International Conference on Quality Software*, pages 326–331, 2010.

[95] Reyhaneh Jabbarvand, Alireza Sadeghi, Hamid Bagheri, and Sam Malek. Energy-Aware Test-Suite Minimization for Android Apps. In *Proceedings of the International Symposium Software Testing and Analysis*, pages 425–436, 2016.

[96] Shin Nakajima and Hai Ngoc Bui. Dataset Coverage for Testing Machine Learning Computer Programs. In *Proceedings of the Asia-Pacific Software Engineering Conference*, pages 297–304, 2016.

[97] Petar Tsankov, Mohammad Torabi Dashti, and David Basin. Semi-Valid Input Coverage for Fuzz Testing. In *Proceedings of the International Symposium Software Testing and Analysis*, pages 56–66, 2013.

[98] Jakob Rott, Rainer Niedermayr, Elmar Juergens, and Dennis Pagano. Ticket Coverage: Putting Test Coverage into Context. In *Proceedings of the Workshop on Emerging Trends in Software Metrics*, pages 2–8, 2017.