Todo list

PUTTING SOFTWARE TESTING TERMINOLOGY TO THE TEST

PUTTING SOFTWARE TESTING TERMINOLOGY TO THE TEST

By SAMUEL CRAWFORD, B.Eng.

A Thesis
Submitted to the Department of Computing and Software
and the School of Graduate Studies
of McMaster University
in Partial Fulfillment of the Requirements
for the Degree of
Master of Applied Science

Master of Applied Science (2025)	McMaster University
(Department of Computing and Software)	Hamilton, Ontario

TITLE:	Putting Software Testing Terminology to the Test
AUTHOR:	Samuel Crawford, B.Eng.
SUPERVISOR:	Dr. Carette and Dr. Smith
PAGES:	xiii, 131

Lay Abstract

It is important to test software to ensure that it achieves its goals and works as intended. However, the documents that describe how to perform testing have many flaws, making this process more difficult and less consistent. By looking through the literature, we found 567 different approaches to testing. We record how these approaches are defined and the relations between them. After collecting and analyzing all these data, we found 341 flaws, such as missing definitions or relations that contradict each other. This shows that despite the amount of information available about software testing, there are significant improvements to be made that would make testing software less confusing and more consistent.

Abstract

Despite the prevalence and importance of software testing, it lacks a standardized and consistent taxonomy, instead relying on a large body of literature with many flaws—even within individual documents! This hinders precise communication, contributing to misunderstandings when researching, planning, and performing testing. In this thesis, we explore the current state of software testing terminology by:

1. identifying established standards and prominent testing resources,
2. capturing relevant testing terms from these sources, along with their definitions and relationships (both explicit and implicit), and
3. constructing visualizations to analyze these data.

This process uncovered 567 test approaches and four in-scope methods for deriving test approaches, such as those related to 75 software qualities. We also manually record flaws as they arise and build tools to detect more flaws automatically, analyze all of our recorded flaw data, and visualize the relations between test approaches. This revealed 341 flaws, including 13 terms used as synonyms to two (or more) disjoint test approaches and 17 pairs of test approaches that may either be synonyms or have a parent-child relationship. We also found notable confusion surrounding functional testing, operational acceptance testing, recovery testing, and scalability testing. Our findings make clear the urgent need for improved testing terminology so that the discussion, analysis, and implementation of various test approaches can be more coherent.

Acknowledgements

This thesis was based on Jason Balaci's McMaster thesis template which provided many helper LaTeX functions. Christopher William Schankula also helped me with LaTeX and multiple friends discussed software testing with me, providing many of the approaches in Section 3.5. A special "thank you" to Dr. Stuart Reid for providing me with a paper he wrote that I could not find online, for his work with ISO, *and* for founding the International Software Testing Qualifications Board (ISTQB)! My family has also supported me in more ways than I can count, and I cannot thank them enough. Finally, Dr. Spencer Smith and Dr. Jacques Carette have been great supervisors and valuable sources of guidance and feedback.

ChatGPT was used to help generate supplementary Python code for constructing visualizations and generating LaTeX code, including regex. ChatGPT and GitHub Copilot were both used for assistance with LaTeX formatting. ChatGPT and ProWritingAid were both used for proofreading. ChatGPT also provided pointers to the potential existence of some of the approaches in Section 3.5.

Contents

List of Figures

List of Tables

List of Abbreviations and Symbols

API	Application Programming Interface
c-use/C-use	Computation data use
CGI	Common Gateway Interface
CLI	Command-Line Interface
CT	Continuous Testing
DOM	Document Object Model
du-path/DU-path	Definition-Use path
EMSEC	EManations SECurity
IEC	International Electrotechnical Commission
ISTQB	International Software Testing Qualifications Board
LCSAJ	Linear Code Sequence and Jump
MBT	Model-Based Testing
ML	Machine Learning
NIST	National Institute of Standards and Technology
OAT	Operational Acceptance/Orthogonal Array Testing
OT	Operational Testing
p-use/P-use	Predicate data use
PAR	Product Anomaly Report
PIR	Product Incident Report
RQ	Research Question
SoS	System of Systems
SQL	Structured Query Language
SUT	System Under Test
SWEBOK Guide	Guide to the SoftWare Engineering Body Of Knowledge
TOAT	Taguchi's Orthogonal Array Testing
UML	Unified Modeling Language
V&V	Verification and Validation

Declaration of Academic Achievement

This research and analysis was performed by Samuel Crawford under the guidance, supervision, and recommendations of Dr. Spencer Smith and Dr. Jacques Carette. The resulting contributions are three glossaries—one for each of test approaches, software qualities, and supplementary terms (see Section 3.3)—as well as the tools for data visualization and automated analysis outlined in Chapter 4. These are all available on an open-source repo for independent analysis and, ideally, extension as more test approaches are discovered and documented.

Chapter 1

Introduction

As with all fields of science and technology, software development should be approached systematically and rigorously. Peters and Pedrycz claim that “to be successful, development of software systems requires an engineering approach” that is “characterized by a practical, orderly, and measured development of software” (2000, p. 3). When a NATO study group decided to hold a conference to discuss “the problems of software” in 1968, they chose the phrase “software engineering” to “imply[] the need for software manufacture to be based on the types of theoretical foundations and practical disciplines, [sic] that are traditional in the established branches of engineering” (Naur and Randell, 1969, p. 13). “The term was not in general use at that time”, but conferences such as this “played a major role in gaining general acceptance … for the term” (McClure, 2001). While one of the goals of the conference was to “discuss possible techniques, methods and developments which might lead to the[] solution” to these problems (Naur and Randell, 1969, p. 14), the format of the conference itself was difficult to document. Two competing classifications of the report emerged: “one following from the normal sequence of steps in the development of a software product” and “the other related to aspects like communication, documentation, management, [etc.]” (p. 10). Furthermore, “to retain the spirit and liveliness of the conference, … points of major disagreement have been left wide open, and … no attempt … [was] made to arrive at a consensus or majority view” (p. 11)!

Perhaps unsurprisingly, there are still concepts in software engineering without consensus, and many of them can be found in the subdomain of software testing. Kaner et al. (2011, p. 7) give the example of complete testing, which may require the tester to discover “every bug in the product”, exhaust the time allocated to the testing phase, or simply implement every test previously agreed upon. Having a clear definition of “complete testing” would reduce the chance for miscommunication and, ultimately, the tester getting “blamed for not doing … [their] job” (p. 7). Because software testing uses “a subtantial percentage of a software development budget (in the range of 30 to 50%)”, which is increasingly true “with the growing complexity of software systems” (Peters and Pedrycz, 2000, p. 438), this is crucial to the efficiency of software development. Even more foundationally, if software engineering holds code to high standards of clarity, consistency, and robustness, the same should apply to its supporting literature!

Unfortunately, a search for a systematic, rigorous, and complete taxonomy for software testing revealed that the existing ones are inadequate and mostly focus on the high-level testing process rather than the test approaches themselves:

- Tebes et al. (2020) focus on *parts* of the testing process (e.g., test goal, test plan, testing role, testable entity) and how they relate to one another,
- Souza et al. (2017) prioritize organizing test approaches over defining them,
- Firesmith (2015) similarly defines relations between test approaches but not the approaches themselves, and
- Unterkalmsteiner et al. (2014) focus on the "information linkage or transfer" (p. A:6) between requirements engineering and software testing and "do[] not aim at providing a systematic and exhaustive state-of-the-art survey of [either domain]" (p. A:2).

In addition to these taxonomies, many standards documents (see Section 2.5.1) and terminology collections (see Section 2.5.2) define testing terminology, albeit with their own issues.

For example, a common point of discussion in the field of software is the distinction between terms for when software does not work correctly. We find the following four to be most prevalent:

- **Error:** a "human action that produces an incorrect result" (ISO/IEC and IEEE, 2017, p. 165; 2010, p. 128; van Vliet, 2000, p. 399; similar in IEEE, 2024, p. 36).
- **Fault:** "an incorrect step, process, or data definition in a computer program" (ISO/IEC and IEEE, 2010, p. 140; similar in IEEE, 2024, p. 36) inserted when a developer makes an error (p. 36; ISO/IEC and IEEE, 2010, pp. 128, 140; van Vliet, 2000, pp. 399–400).
- **Failure:** the inability of a system "to perform a required function or … within previously specified limits" that is "externally visible" (ISO/IEC and IEEE, 2019a, p. 7; similar in IEEE, 2024, pp. 15, 37; Washizaki, 2025a, p. 5-3; Lyu, 1996, p. 12; van Vliet, 2000, p. 400) and caused by a fault (Washizaki, 2025a, p. 12-3; Lyu, 1996, p. 13; van Vliet, 2000, p. 400).
- **Defect:** "an imperfection or deficiency in a project component where that component does not meet its requirements or specifications and needs to be either repaired or replaced" (ISO/IEC and IEEE, 2010, p. 96; Project Management Institute, 2013, p. 536).

These distinctions are often important, but the term "defect" "may refer to any or all of error, fault, or failure" (IEEE, 2024, p. 36), making the term "overloaded with too many meanings, as engineers and others use the word to refer to all different types of anomalies" (Washizaki, 2025a, p. 12-3). Software testers may even choose to ignore these nuances completely! Patton (2006, pp. 13–14) "just call[s] it what it is and get[s] on with it", abandoning these four terms, "problem", "incident", "anomaly", "variance", "inconsistency", "feature" (!), and "a list of unmentionable terms" in favour of "bug"; after all, "there's no reason to dice words"!

These decisions are not inherently wrong, since they may be useful in certain contexts or for certain teams (see Section 2.1.2 for more detailed discussion). Problems start to arise when teams need to make these decisions in the first place. Patton (2006, p. 14) notes that "a well-known computer company spent weeks in discussion with its engineers before deciding to rename Product Anomaly Reports (PARs) to Product Incident Reports (PIRs)", a process that required "countless dollars" and updating "all the paperwork, software, forms, and so on". While consistency and clear terminology may have been valuable to the company, "it's unknown if [this decision] made any difference to the programmer's or tester's productivity" (p. 14). A potential way to avoid similar resource sinks would be to prescribe a standard terminology. Perhaps multiple sets of terms could be designed with varying levels of specificity so a company would only have to determine which one best suits their needs.

But why are minor differences between terms like these even important? The previously defined terms "error", "fault", "failure", and "defect" are used to describe many test approaches, including:

1. Defect-based testing
2. Error forcing
3. Error guessing
4. Error tolerance testing
5. Error-based testing
6. Error-oriented testing
7. Failure tolerance testing
8. Fault injection testing
9. Fault seeding
10. Fault sensitivity testing
11. Fault tolerance testing
12. Fault tree analysis
13. Fault-based testing

When considering which approaches to use or when actually using them, the meanings of these four terms inform what their related approaches accomplish and how to they are performed. For example, the tester needs to know what a "fault" is to perform fault injection testing; otherwise, what would they inject? Information such as this is critical to the testing team, and should therefore be standardized.

These kinds of inconsistencies can lead to miscommunications—such as that previously mentioned by Kaner et al. (2011, p. 7)—and are prominent in the literature. ISO/IEC and IEEE (2022, Fig. 2) categorize experience-based testing as both a test design technique and a test practice in the same figure! The structure of tours can be defined as either quite general (p. 34) or "organized around a special focus" (Hamburg and Mogyorodi, 2024). Load testing may be performed with loads "between anticipated conditions of low, typical, and peak usage" (ISO/IEC and IEEE, 2022, p. 5) or with loads that are as large as possible (Patton, 2006, p. 86). Alpha testing can be performed by "users within the organization developing the software" (ISO/IEC and IEEE, 2017, p. 17), "a small, selected group of potential users" (Washizaki, 2025a, p. 5-8), or "roles outside the development organization" conducted "in the developer's test environment" (Hamburg and Mogyorodi, 2024). It is clear that there is a notable gap in the literature, one which we attempt to describe. While the creation of a complete taxonomy is unreasonable, especially considering the pace at which the field of software changes, we can make progress towards this goal that others can extend and update as new test approaches emerge. The main way we accomplish this is by identifying "flaws" or "inconsistencies" in the literature, or areas where there is room for improvement. We track these flaws according to both *what* information is wrong and *how* (described in more detail in Section 2.2), which allows us to analyze them more thoroughly and reproducibly.

Based on this observed gap in software testing terminology and our original motivation for this research, we only consider the component of Verification and Validation (V&V) that tests code itself. However, some test approaches are only used to testing *other* artifacts, while others can be used for both! In these cases, we only consider the subsections that focus on code. For example, reliability testing and maintainability testing can start *without* code by "measur[ing] structural attributes of representations of the software" (Fenton and Pfleeger, 1997, p. 18), but only reliability and maintainability testing performed on code *itself* is in scope of this research. This is a high-level overview of what is in scope; see Appendix A for more detailed discussion on what we include and exclude.

This document describes our process, as well as our results, in more detail. We start by documenting the 567 test approaches mentioned by 85 sources (described in Section 2.5), recording their names, categories[1], definitions, synonyms[2], parents[3], and flaws[4] (see Section 3.3.2) as applicable. We also record any other relevant notes, such as prerequisites, uncertainties, and other sources. We follow the procedure laid out in Chapter 3 and use these Research Questions (RQs) as a guide:

1. What test approaches do the literature describe?
2. How consistent are these descriptions?

An excerpt of this recorded information (excluding other notes for brevity), is given in Table 1.1. We then create tools to support our analysis of our findings (Chapter 4). Despite the amount of well-understood and organized knowledge, we find the literature to be quite flawed (Chapter 5), reinforcing the need for a proper taxonomy! We then describe the threats to the validity of our work (Chapter 6), outline potential next steps for future researchers (Chapter 7), and summarize our findings (Chapter 8).

[1] Defined in Section 2.1.1.
[2] Defined in Section 2.1.2.
[3] Defined in Section 2.1.3.
[4] Defined in Section 2.2.

Table 1.1: Selected entries from our test approach glossary with "Notes" column excluded for brevity.

Name	Approach Category	Definition	Parent(s)	Synonym(s)
A/B Testing	Practice (ISO/IEC and IEEE, 2022, Fig. 2), Type (inferred from usability testing)	Testing "that allows testers to determine which of two systems or components performs better" (ISO/IEC and IEEE, 2022, pp. 1, 36)	Statistical Testing (ISO/IEC and IEEE, 2022, pp. 1, 36), Usability Testing (Firesmith, 2015, p. 58)	Split-Run Testing (ISO/IEC and IEEE, 2022, pp. 1, 36)
Back-to-Back Testing	Practice (ISO/IEC and IEEE, 2022, p. 22)	Testing "whereby an alternative version of the system is used to generate expected results for comparison from the same test inputs" (ISO/IEC and IEEE, 2022, p. 2) …	Non-functional Testing (Washizaki, 2025a, p. 5-9)	Differential Testing (ISO/IEC and IEEE, 2022, p. 2)
Retesting	Type (Hamburg and Mogyorodi, 2024)	Testing "performed to check that modifications made to correct a fault have successfully removed the fault" (ISO/IEC and IEEE, 2022, p. 8; 2021a, p. 3; similar in 2017, p. 386; Hamburg and Mogyorodi, 2024), …	Change-Related Testing (Hamburg and Mogyorodi, 2024)	Confirmation Testing (ISO/IEC and IEEE, 2022, pp. 8, 35; 2021a, p. 3; 2017, p. 386; Hamburg and Mogyorodi, 2024)

Chapter 2

Terminology

Our research aims to describe the current state of software testing literature, including its flaws. Since we critique the lack of clarity, consistency, and robustness in the literature, we need to hold ourselves to a high standard in these areas when defining and using terms. For example, since we focus on how the literature describes “test approaches”, we first define this term (Section 2.1). Likewise, before we can constructively describe the flaws in the literature, we need to define what we mean by “flaw” (Section 2.2). To further prevent bias, we only use classifications and relations already implicitly present in the literature instead of inventing our own; for example, test approaches can have categories (Section 2.1.1), synonyms (Section 2.1.2), and parent-child relations (Section 2.1.3). We also observe flaws having both manifestations (Section 2.2.1) and domains (Section 2.2.2) and use these terms to refer to these implicit concepts in the literature. All of these classifications and relations follow logically from the literature and as such are technically “results” of our research, but we define them here for clarity since we use them throughout this thesis.

Since the literature is flawed, we need to be careful with what information we take at face value. We do this by tracking the nuance, or “explicitness”, of information (Section 2.3) found in sources and the “credibility” of these sources themselves (Section 2.4). We then use this heuristic of credibility to group our identified sources into “tiers” (Section 2.5). Defining these terms helps reduce the effect of our preconceptions on our analysis (or at least makes it more obvious to future researchers), as there may be other equally valid ways to analyze the literature and its flaws. To be clear: we do *not* prescribe what terminology software testers *should* use, we simply observe the terminology that the literature uses and try to use it as consistently and logically as possible.

2.1 Test Approaches

Software testing is an "activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component" (ISO/IEC and IEEE, 2022, p. 10; 2021c, p. 6; 2017, p. 473), usually "with the intent of finding errors" (Myers, 1976, as cited in Peters and Pedrycz, 2000, p. 438[1]). For each test, the main steps are to:

1. identify the goal(s) of the test,
2. decide on an approach,
3. develop the tests,
4. determine the expected results,
5. run the tests, and
6. compare the expected results to the actual results (p. 443).

When this process reveals errors, "the faults causing them are what can and must be removed" (Washizaki, 2025a, p. 5-3).

Of course, the approach chosen in step 2 influences what kinds of test cases should be developed and executed in later steps, so it is important that test approaches are defined correctly, consistently, and unambiguously. A "test approach" is a "high-level test implementation choice" (ISO/IEC and IEEE, 2022, p. 10) used to "pick the particular test case values" (2017, p. 465) used in step 5. The only approach that can "fully" test a system (exhaustive testing) is infeasible in most non-trivial situations (2022, p. 4; Washizaki, 2025a, p. 5-5; Peters and Pedrycz, 2000, pp. 439, 461; van Vliet, 2000, p. 421), so multiple approaches are needed (ISO/IEC and IEEE, 2022, p. 18) to "suitably cover any system" (p. 33). This is why this process should be repeated and there are so many test approaches described in the literature.

2.1.1 Approach Categories

Since there are so many test approaches, it is helpful to categorize them. The literature provides many ways to do so, but we use the one given by ISO/IEC and IEEE (2022) because of its wide usage. This schema divides test approaches into levels, types, techniques, and practices (2022, Fig. 2; see Table 2.1). These categories seem to be pervasive throughout the literature, particularly "level" and "type". For example, six non-IEEE sources also give unit testing, integration testing, system testing, and acceptance testing as examples of test levels (Washizaki, 2025a, pp. 5-6 to 5-7; Hamburg and Mogyorodi, 2024; Black, 2009, pp. 12, 14, 18, 24, 62, 178; Perry, 2006, pp. 807–808; Peters and Pedrycz, 2000, pp. 443–445; Gerrard, 2000a, pp. 9, 13). These categories seem to be orthogonal based on their

[1] See Mistake 20.

definitions and usage. For example, "a test type can be performed at a single test level or across several test levels" (ISO/IEC and IEEE, 2022, p. 15; 2021a, p. 8; 2021c, p. 7), and test practices can be "defined … for a specific level or type of testing" (2021b, p. 9). We may assess this assumption more rigorously in the future, but for now, it implies that one can derive a specific test approach by combining multiple test approaches from different categories; see Appendix A.6 for more detailed discussion. We loosely describe these categories based on what they specify as follows:

- **Level:** When in the software life cycle the test is performed
- **Practice:** How the test is structured and executed
- **Technique:** How inputs and/or outputs are derived
- **Type:** Which software quality is evaluated

While ISO/IEC and IEEE's (2022) schema includes "static testing" as a test approach category, we omit it as a category since it seems non-orthogonal to the others and thus less helpful for grouping test approaches. These authors even categorize static testing as a test level in (2021b, p. 43; see Contradiction 6)! Other schemas (see Section 6.2) consider static testing to be an orthogonal category or out of scope entirely (see Appendix A.3). We also introduce a "supplemental" category of "artifact"[2] since some terms can refer both to the application of a test approach *and* to the resulting document(s). Some sources explicitly distinguish between the two, such as between "conformity evaluation" and "conformity evaluation report" (2017, p. 93), but most do not. Therefore, we do *not* consider approaches categorized as an artifact *and* another category as flaws in Section 5.2.1. Finally, we also record the test category of "process" for completeness, although this seems to be a higher-level classification[3].

[2] Discussed in #44, #119, and #39.

[3] Discussed in #52.

Table 2.1: Categories of test approaches given by ISO/IEC and IEEE.

Term	Definition	Examples
Test Level[a]	A stage of testing "typically associated with the achievement of particular objectives and used to treat particular risks", each performed in sequence (ISO/IEC and IEEE, 2022, p. 12; 2021a, p. 6; 2021c, p. 6) with their "own documentation and resources" (2017, p. 469)	unit/component testing, integration testing, system testing, acceptance testing (2022, p. 12; 2021a, p. 6; 2021c, p. 6; 2017, p. 467)
Test Practice	A "conceptual framework that can be applied to … [a] test process to facilitate testing" (2022, p. 14; 2017, p. 471)	scripted testing, exploratory testing, automated testing (2022, p. 20)
Test Technique[b]	A "procedure used to create or select a test model …, identify test coverage items …, and derive corresponding test cases" (2022, p. 11; 2021a, p. 5; similar in 2017, p. 467) that "generate evidence that test item requirements have been met or that defects are present in a test item" (2021c, p. vii) "typically used to achieve a required level of coverage" (2021a, p. 5)	equivalence partitioning, boundary value analysis, branch testing (2022, p. 11; 2021a, p. 5)
Test Type	"Testing that is focused on specific quality characteristics" (2022, p. 15; 2021c, p. 7; 2017, p. 473)	security testing, usability testing, performance testing (2022, p. 15; 2021a, p. 8; 2017, p. 473)

[a] Also called "test phase" (see Overlap 1) or "test stage" (see Contradiction 21).

[b] Also called "test design technique" (ISO/IEC and IEEE, 2022, p. 11; 2021a, p. 5; Hamburg and Mogyorodi, 2024).

2.1.2 Synonym Relations

The same approach often has many names. For example, "specification-based testing" is also called "black-box testing" (ISO/IEC and IEEE, 2022, p. 9; 2021c, p. 8; 2017, p. 431; Washizaki, 2025a, p. 5-10; Hamburg and Mogyorodi, 2024; Firesmith, 2015, pp. 46–47[4]; van Vliet, 2000, p. 399; Sakamoto et al., 2013, p. 344). Throughout our work, we use the terms "specification-based testing" and "structure-based testing" to articulate the source of the information for designing test cases, but a team or project also using grey-box testing may prefer the terms "black-box" and "white-box testing" for consistency.

We can formally define the synonym relation S on the set T of terms used by the literature to describe test approaches based on how synonyms are used in natural language. S is symmetric and transitive, and although pairs of synonyms in natural language are implied to be distinct, a relation that is symmetric and transitive is provably reflexive; this implies that all terms are trivially synonyms of themselves. Since S is symmetric, transitive, *and* reflexive, it is an equivalence relation, reflecting the role of synonyms in natural language where they can be used interchangeably. While synonyms may emphasize different aspects or express mild variations, their core meaning is nevertheless the same.

2.1.3 Parent-Child Relations

Many test approaches are multi-faceted and can be "specialized" into others; for example, load testing and stress testing are some subtypes of performance-related testing. We refer to these "specializations" as "children" or "subapproaches" of their multi-faceted "parent(s)". This nomenclature also extends to approach categories (such as "subtype"; see Section 2.1.1 and Table 2.1) and software qualities ("subquality"; see Section 3.3).

We can formally define the parent-child relation P on the set T of terms used by the literature to describe test approaches based on directed relations between approach pairs. This relation should be irreflexive, asymmetric, and transitive, making it a strict partial order. A consequence of this is that there should be no directed cycles, although undirected cycles may exist (for example, if a child and its parent share the same parent).

Parent-child relations often manifest when a "well-understood" test approach p is decomposed into smaller, independently performable approaches $c_1, \ldots, c_n$, each with its own focus or nuance. This is frequently the case for hierarchies of approaches given in the literature (ISO/IEC and IEEE, 2022, Fig. 2; 2021c, Fig. 2; Firesmith, 2015). Another way for these relations to occur is when the completion of p indicates that "sufficient testing has been done" in regards to c (van Vliet, 2000, p. 402). While this only "compares the thoroughness of test techniques, not their ability to detect faults" (p. 434), it is sufficient to justify a parent-child relation between the two approaches. These relations may also be represented as hierarchies (ISO/IEC and IEEE, 2021c, Fig. F.1; van Vliet, 2000, Fig. 13.17).

[4] Firesmith (2015) excludes the hyphen, calling it "black box testing".

2.2 Flaws

Ideally, software testing literature would describe test approaches correctly, completely, consistently, and modularly, but this is not the case in reality. We use the term "flaw" to refer to any instance of the literature violating these ideals, *not* to instances of *software* doing the same (see Section 6.2 for further discussion). We classify flaws by both their "manifestations" (*how* information is wrong; see Section 2.2.1) and their "domains" (*what* information is wrong; see Section 2.2.2). These are orthogonal classifications, since each flaw *manifests* in a particular *domain*, which we track by assigning each flaw one "key" for each classification (listed keys in Tables 2.2 and 2.3, respectively). We also introduce terms we use when discussing flaws based on how many sources contribute to them (Section 2.2.3).

2.2.1 Flaw Manifestations

Perhaps the most obvious example of something being "wrong" with the literature is that a piece of information it presents is incorrect—"wrong" in the literal sense. However, if our standards for correctness require clarity, consistency, and robustness, then there are many ways for a flaw to manifest. This is one view we take when observing, recording, and analyzing flaws: *how* information is "wrong". We observe the "manifestations" described in Table 2.2 throughout the literature, and give each a unique key for later analysis and discussion. We also use this manifestation view when referencing a specific flaw; for instance, Contradiction 19 presents contradictory definitions of "alpha testing". Note that some flaws involve information from multiple sources (contradictions and overlaps in particular). We do not categorize these flaws as mistakes if finding the ground truth requires analysis that we have not yet performed.

Table 2.2: Observed flaw manifestations.

Manifestation	**Description**	**Key**
Mistake	Information is incorrect	`WRONG`[a]
Omission	Information that should be included is not	`MISS`[b]
Contradiction	Information from multiple places conflicts	`CONTRA`
Ambiguity	Information is unclear	`AMBI`
Overlap	Information is nonatomic or used in multiple contexts	`OVER`
Redundancy	Information is redundant	`REDUN`

[a] We use `WRONG` here to avoid clashing with `MISS`.

[b] We use `MISS` here to be more meaningful in isolation, as it implies the synonym of "missing"; `OMISS` is less intuitive and `OMIT` would be inconsistent with the keys being adjective-based.

2.2.2 Flaw Domains

Another way to categorize flaws is by *what* information is wrong, which we call the flaw's "domain". We describe those we observe in Table 2.3, and tracking these uncovers which knowledge domains are less standardized (and should therefore be approached with more rigour) than others. We explicitly define some of these domains in previous sections and thus present them in that same order. These are the domains in which we automatically detect and present flaws as described in Sections 4.2.1 and 5.2, respectively, so these are the only ones that are hyperlinked. We automatically detect the following classes of flaws:

- Test approaches with more than one category that violate our assumption of orthogonality (see Section 2.1.1).
- Synonyms that violate transitivity (see Section 2.1.2); if two distinct approaches share a synonym but are not synonyms themselves, at least one relation is incorrect or missing.
- Synonyms between independently defined approaches; if two separate approaches have their own definitions, nuances, etc. but are also labelled as synonyms, this indicates that:
    1. the terms are interchangeable and this relation is trivially reflexive (see Section 2.1.2),
    2. at least one of these terms is defined incorrectly, and/or
    3. this synonym relation is incorrect.
- Parent-child relations that violate irreflexivity as outlined in Section 2.1.3 (i.e., cases where a child is given as a parent of itself).
- Pairs of synonyms where one is a subapproach of the other; these relations cannot coexist since synonym relations are symmetric while parent-child relations are asymmetric (as outlined in Sections 2.1.2 and 2.1.3, respectively).

Table 2.3: Observed flaw domains.

Domain	**Description**	**Key**
Categories	Approach categories, defined in Section 2.1.1	CATS
Synonyms	Synonym relations, defined in Section 2.1.2	SYNS
Parents	Parent-child relations, defined in Section 2.1.3	PARS
Definitions	Definitions given to terms	DEFS
Labels	Labels or names given to terms	LABELS
Scope	Scope of the information	SCOPE
Traceability	Records of the source(s) of information	TRACE

Despite their nuance, the remaining domains are relatively straightforward, so we define them briefly as follows instead of defining them more rigorously in their own sections. Terms can be thought of as definition-label pairs, but there is a meaningful distinction between definition flaws and label flaws. Definition flaws are quite self-explanatory, but label flaws are harder to detect, despite occurring independently. Examples of label flaws include terms that share the same acronym or contain typos or redundant information. Sometimes, an author may use one term when they mean another. One could argue that their "internal" definition of the term is the cause of this mistake, but we consider this a label flaw where the wrong label is used as we would change the *label* to fix it. Additionally, some information is presented with an incorrect scope and sometimes should not have been included at all! Finally, some traceability information is flawed, such as how one document cites another or even what information is included *within* a document.

2.2.3 Additional Flaw Terminology

Some flaws involve information from more than one source, but referring to this as a "flaw between two sources" is awkward. We instead refer to this kind of flaw as an "inconsistency" between the sources. This clearly indicates that there is disagreement between the sources, but also does not imply that either one is correct—the inconsistency could be with some ground truth if *neither* source is correct!

Other flaws only involve one source, but we make a distinction between "self-contained" flaws and "internal" flaws[5]. Self-contained flaws are those that manifest by comparing a document to an assertion of ground truth. These may appear once in a document or consistently throughout it. Sometimes, these do not require an explicit comparison to ground truth; these often include omissions as the lack of information is contained within a single source and does not need to be cross-checked against an assertion of ground truth. On the other hand, internal flaws arise when a document disagrees with itself by containing two conflicting pieces of information; this includes many contradictions and overlaps. Internal flaws can even occur on the same page, such as when a source gives the same acronym to two distinct terms (see Overlap 6 and Overlap 7)!

2.3 Explicitness

When information is written in natural language, a considerable degree of nuance can get lost when interpreting or using it. We call this nuance "explicitness", or how explicit a piece of information is (or is *not*). For example, a source may provide data from which the reader can logically draw a conclusion, but may not state or prove this conclusion explicitly. In the cases where information is *not* explicit, we record it (see Section 3.3.1 for more detailed discussion) and present it

[5] Discussed in #137 and #138.

Later: Ensure this is up to date

using (at least) one of the following keywords: "implied", "can be", "sometimes", "should be", "ideally", "usually", "most", "likely", "often", "if", and "although". Most information provided by sources we investigate is given explicitly; all sources cited throughout this thesis support their respective claims explicitly unless specified otherwise, usually via one of these keywords. It is important to note that we use the term "implicit" (as well as "implied by" when describing sources of information) to refer to any instance of "not explicit" information for brevity. Any kind of information can be implicit, including the names, definitions, categories (see Section 2.1.1), synonyms (see Section 2.1.2), and parents (see Section 2.1.3) of identified test approaches.

As our research focuses on the flaws present in the literature, the explicitness of information affects how seriously we take it. We call flaws based on explicit information "objective", since they are self-evident in the literature. On the other hand, we call flaws based on implicit information "subjective", since some level of judgement is required to assess whether these flaws are *actually* problematic. By looking for the indicators of uncertainty mentioned above, we can automatically detect subjective flaws when generating graphs and performing analysis (see Sections 4.1 and 4.2, respectively).

Throughout our research, we also infer some information through "surface-level" analysis that follows straightforwardly but is not stated, explicitly or otherwise, by a source. Although these data originate from our judgement, we document them for completeness, using the phrase "inferred from" when relevant. All data in our glossaries without a citation are inferred, such as algebraic testing (Peters and Pedrycz, 2000, Fig. 12.2) being a child of mathematical-based testing. Additionally, some inferences are based on information given by a source, which we cite alongside these inferences. For example, Gerrard describes large scale integration testing and legacy system integration testing in (2000b, p. 30) and (2000a, Tab. 2; 2000b, Tab. 1), respectively. While he never explicitly says so, we infer that these approaches are children of integration testing and system integration testing, respectively. Similarly, some test approaches appear to be combinations of other (seemingly orthogonal) approaches (described in more detail in Appendix A.6), from which we can extrapolate other test approaches. For example, Moghadam (2019) uses the phrase "machine learning-assisted performance testing"; since performance testing is a known test approach, we infer the existence of the test approach "machine learning-assisted testing" and include it in our test approach glossary as such. We also infer that child approaches inherit their parents' categories (see Section 2.1.1).

2.4 Credibility

In the same way we distinguish between the explicitness of information from different sources, we also wish to distinguish between the "explicitness" of the sources themselves! Of course, we do not want to overload terms, so we define a source as more "credible" if it:

- has gone through a peer-review process,
- is written by numerous, well-respected authors,
- cites a (comparatively) large number of sources, and/or
- is accepted and used in the field of software.

Sources may meet only some of these criteria, so we use our judgement (along with the format of the sources themselves) when comparing them.

2.5 Source Tiers

For ease of discussion and analysis, we group the complete set of sources into "tiers" based on their format, method of publication, and our heuristic of credibility. In order of descending credibility, we define the following tiers:

1. established standards (Section 2.5.1),
2. terminology collections (Section 2.5.2),
3. textbooks (Section 2.5.3), and
4. papers and other documents (Section 2.5.4).

We provide a summary of how many sources comprise each tier in Figure 2.1 and list all sources in each tier in Appendix B. The "papers" tier is quite large since we often "snowball" on terminology itself when a term requires more investigation (e.g., its definition is missing or unclear). This includes performing a miniature literature review on this subset to "fill in" missing information (see Section 3.4) and potentially fully investigating these additional sources, as opposed to just the original subset of interest, based on their credibility and how much extra information they provide. We use standards the second most frequently due to their high credibility and broad scope; for example, the glossary portion of ISO/IEC and IEEE (2017) has 514 pages! Using these standards allows us to record many test approaches in a similar context from a source that is widely used and well-respected.

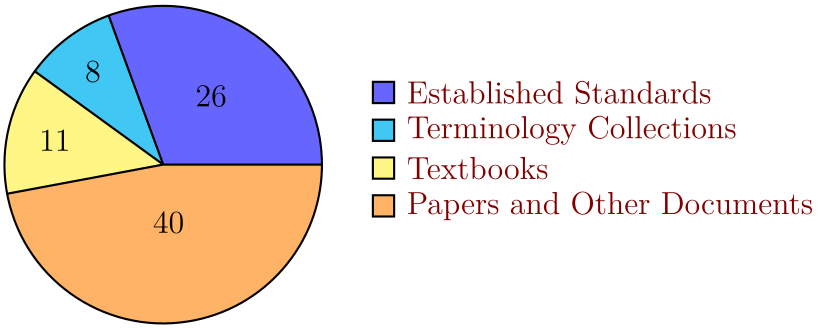


Figure 2.1: Summary of how many sources comprise each source tier.

2.5.1 Established Standards

These are documents written for the field of software engineering by reputable standards bodies, namely ISO, the International Electrotechnical Commission (IEC), and IEEE, so we consider them to be the most credible sources. Their purpose is to "encourage the use of systems and software engineering standards" and "collect and standardize terminology" by "provid[ing] definitions that are rigorous, uncomplicated, and understandable by all concerned" (ISO/IEC and IEEE, 2017, p. viii) "that can be used by any organization when performing any form of software testing" (2022, p. vii; similar in 2016, p. ix). Only standards for software development and testing are in scope for this research (see Chapter 1 for a high-level overview of what is in scope; see Appendix A for more detailed discussion on what we include and exclude).

2.5.2 Terminology Collections

These are collections of software testing terminology built up from multiple sources, such as the established standards outlined in Section 2.5.1. For example, the SWEBOK Guide is "proposed as a suitable foundation for government licensing, for the regulation of software engineers, and for the development of university curricula in software engineering" (Kaner et al., 2011, p. xix). Even though it is "published by the IEEE Computer Society", it "reflects the current state of generally accepted, consensus-driven knowledge derived from the interaction between software engineering theory and practice" (Washizaki, 2025b). Due to this combination of IEEE standards and state-of-the-practice observations, we designate it as a collection of terminology as opposed to an established standard. Collections such as this are often written by a large organization, such as the International Software Testing Qualifications Board (ISTQB), but not always. Firesmith's (2015) taxonomy presents relations between many test approaches and Doğan et al.'s (2014) literature review cites many sources from which we can "snowball" if desired (see Section 3.1), so we include them in this tier as well.

2.5.3 Textbooks

We consider textbooks to be more credible than papers (see Section 2.5.4) because they are widely used as resources for teaching software engineering and industry frequently uses them as guides. Although textbooks have smaller sets of authors, they follow a formal review process before publication. Textbooks used at McMaster University (Patton, 2006; Peters and Pedrycz, 2000; van Vliet, 2000) served as the original (albeit ad hoc and arbitrary) starting point of this research, and we investigate other books as they arise. For example, Hamburg and Mogyorodi (2024) cite Gerrard and Thompson (2002) as the original source for their definition of "scalability" which we verify by looking at this original source.

2.5.4 Papers and Other Documents

The remaining documents all have much smaller sets of authors and are much less widespread than those in higher source tiers. While most of these are journal articles and conference papers, we also include the following document types. Some of these are not peer-reviewed works but are still useful for observing how terms are used in practice:

- Report (Kam, 2008; Gerrard, 2000a;b)
- Thesis (Bas, 2024)
- Website (LambdaTest, 2024; Pandey, 2023)
- Booklet (Knüvener Mackert GmbH, 2022)
- ChatGPT (GPT-4o) (2024) with its claims supported by Rus et al. (2008)[6]

[6] Patton (2006, p. 88) says that if a specific defect is found, it is wise to look for other defects in the same location and for similar defects in other locations, but does not provide a name for this approach. After researching in vain, we ask ChatGPT (GPT-4o) (2024) to name this test approach but do *not* take its output to be true at face value. Rus et al. (2008) support calling this approach "defect-based testing" based on the principle of "defect clustering".

Chapter 3

Methodology

We collect data from a wide variety of documents related to software testing, focusing on test approaches and supporting information. This results in a large glossary of software approaches, some glossaries of supplementary terms, and a list of flaws. To ensure this data can be analyzed and expanded thoroughly and consistently, we need a process that can be repeated for future developments in the field of software testing or by independent researchers seeking to verify our work. Our methodology is as follows:

1. Identify authoritative sources on software testing and "snowball" from them (Section 3.1)
2. Identify all test approaches[1] and testing-related terms (Section 3.2) described in these authoritative sources
3. Record all relevant data (Section 3.3), including implicit data (Section 3.3.1), for each term identified in step 2; test approach data are comprised of:
   (a) Names
   (b) Categories[2]
   (c) Definitions
   (d) Synonyms[3]
   (e) Parents[4]
   (f) Flaws[5] (Section 3.3.2)
   (g) Notes (prerequisites, uncertainties, other sources, etc.)
4. Repeat steps 1 to 3 for any missing or unclear terms (Section 3.4) until the stopping criteria (Section 3.5) is reached

[1] Defined in Section 2.1.
[2] Defined in Section 2.1.1.
[3] Defined in Section 2.1.2.
[4] Defined in Section 2.1.3.
[5] Defined in Section 2.2.

3.1 Identifying Sources

As there is no single authoritative source on software testing terminology, we need to look at many sources to observe how this terminology is used in practice. We start from the vocabulary document for systems and software engineering (ISO/IEC and IEEE, 2017) and three versions of the Guide to the SoftWare Engineering Body Of Knowledge (SWEBOK Guide) (Bourque and Fairley, 2014; Washizaki, 2024; 2025a; see Chapter 5). To gather further sources, we then use a version of "snowball sampling", which "is commonly used to locate hidden populations … [via] referrals from initially sampled respondents to other persons" (Johnson, 2014). We apply this concept to "referrals" between sources. For example, Hamburg and Mogyorodi (2024) cite Gerrard and Thompson (2002) as the original source for their definition of "scalability" which we verify by looking at this original source. We group all sources into the source tiers we define in Section 2.5 and list all sources in each tier in Appendix B.

3.2 Identifying Relevant Terms

Before we can consistently track software testing terminology used in the literature, we must first determine what to record. We use heuristics to guide this process to increase confidence that we identify all relevant terms, paying special attention to the following when investigating a new source:

- glossaries, taxonomies, hierarchies, and lists of terms,
- testing-related terms (e.g., terms containing "test(ing)", "review(s)", "audit(s)", "attack(s)"[6], "validation", or "verification"),
- terms that had emerged as part of already-discovered test approaches, *especially* those that were ambiguous or prompted further discussion (e.g., terms containing "performance", "recovery", "component", "bottom-up", "boundary", or "configuration"), and
- terms that imply test approaches, including:
  - software qualities that may imply related test types[7],
  - coverage metrics that may imply related test techniques[8], and
  - software requirements that may imply related test approaches.

[6] Discussed in #55.

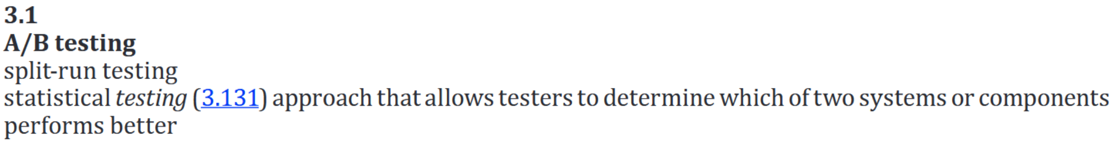
[7] See Section 6.3 for more detailed discussion.

[8] See Section 6.3 for more detailed discussion.

3.3 Recording Relevant Information

Once we have identified which terms from the literature are relevant, we can then track them consistently by building glossaries. We give each test approach its own row in our test approach glossary, recording its name and any given definitions, categories, synonyms, and parents (along with any other notes, such as questions, prerequisites, and other resources to investigate) following the procedure in Figure 3.2. Note that only the name and category fields are required; all other fields may be left blank, although a lack of definition indicates that the approach should be investigated further to see if its inclusion is meaningful (see Section 3.4). Flawed data may be documented here as dubious information (see Section 3.3.1) and/or as described in Section 3.3.2. We also include the source(s) of this information in a consistent format described in Appendix C.1 to allow for more detailed analysis of these data.

For example, when we first encounter "A/B Testing" in ISO/IEC and IEEE (2022, p. 1) as shown in Figure 3.1, we apply our procedure as follows:



3.1
A/B testing
split-run testing
statistical *testing* (3.131) approach that allows testers to determine which of two systems or components performs better

Figure 3.1: ISO/IEC and IEEE's (2022, p. 1) glossary entry for "A/B testing".

1. Create a new row with the name "A/B Testing" and the category "Approach".

2. Record the synonym "Split-Run Testing".

3. Record the parent "Statistical Testing".

4. Record the definition "Testing 'that allows testers to determine which of two systems or components performs better' "; note that we abstract away information that we have previously captured (i.e., its synonym and parent).

In addition to repeating this information on (p. 36), this source also provides the following information, which we capture as follows:

1. Record the note "It 'can be time-consuming, although tools can be used to support it', 'is a means of solving the test oracle problem by using the existing system as a partial oracle', and is 'not a test case generation technique as test inputs are not generated' " (p. 36).

2. Replace the category of "Approach" with the more specific "Practice" (Fig. 2); note that this is consistent with the exclusion of "Technique" as a possible category for this approach (p. 36).

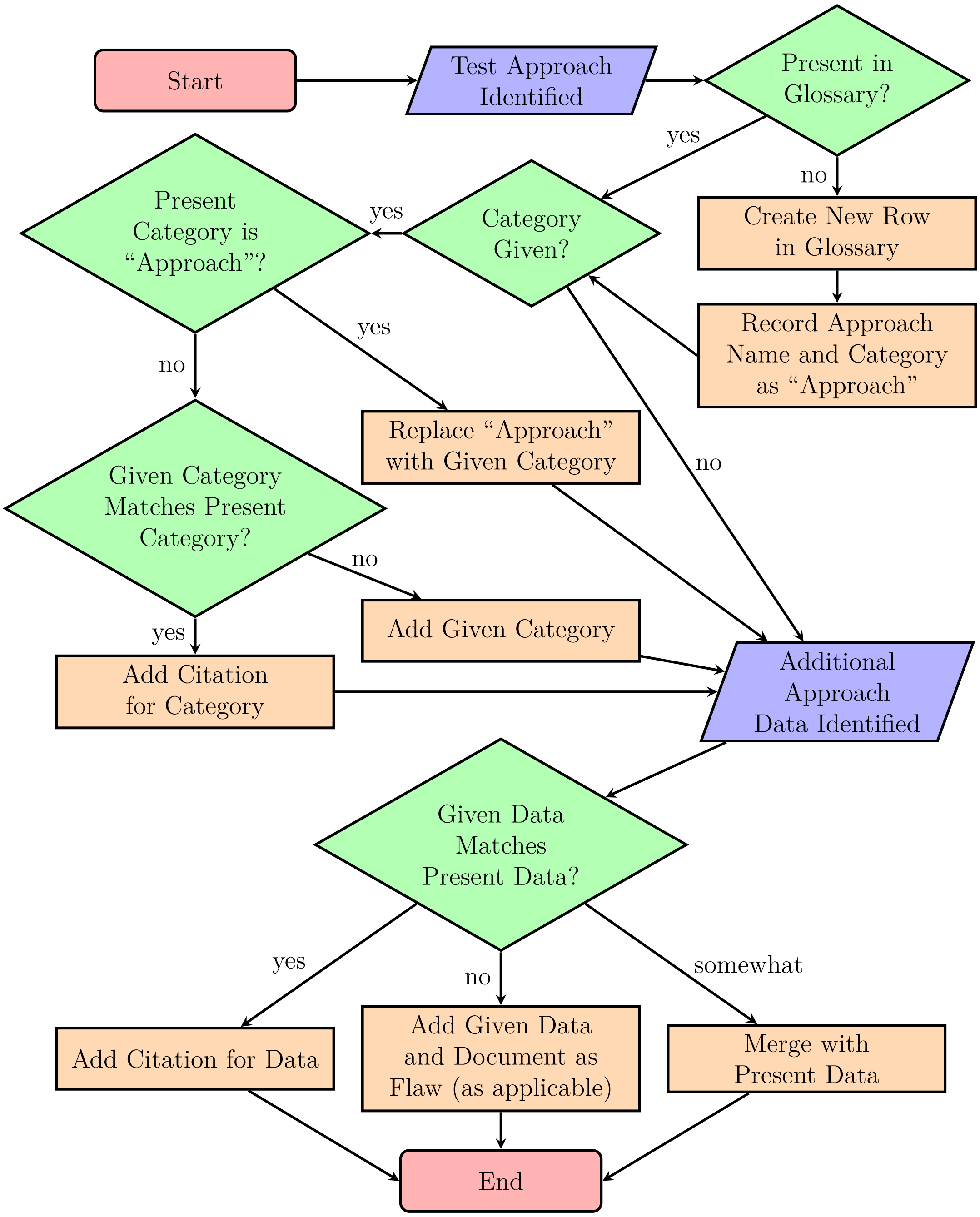

Start
Test Approach Identified
Present in Glossary?
yes
no
Create New Row in Glossary
Record Approach Name and Category as "Approach"
Category Given?
yes
no
Present Category is "Approach"?
yes
no
Replace "Approach" with Given Category
Given Category Matches Present Category?
no
yes
Add Given Category
Add Citation for Category
Additional Approach Data Identified
Given Data Matches Present Data?
yes
no
somewhat
Add Citation for Data
Add Given Data and Document as Flaw (as applicable)
Merge with Present Data
End

Figure 3.2: Procedure for recording test approaches in our glossary; "Present" refers to data already in our glossary, while "Given" refers to data that appears in the source being investigated.

As we investigate other sources, we learn more about this approach. Firesmith (2015, p. 58) includes it in his taxonomy as shown in Figure 3.3. We add to our entry for "A/B Testing" as follows:

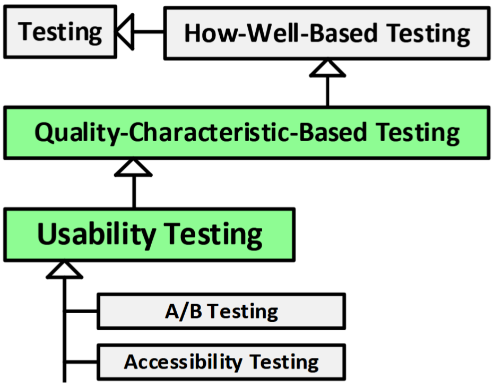


Figure 3.3: A/B testing's inclusion in Firesmith's (2015, p. 58) taxonomy.

1. Add the parent "Usability Testing".
2. Since usability testing is a test type (ISO/IEC and IEEE, 2022, pp. 22, 26-27; 2021c, pp. 7, 40, Tab. A.1; implied by its quality; Firesmith, 2015, p. 53), add the category "Type" with the citation "(inferred from usability testing)".

This second change introduces an inference (defined in Section 2.3) that violates our assumption from Section 2.1.1 that categories are orthogonal (i.e., that A/B testing *cannot* both be a test practice *and* a test type), so we consider this to be an inferred flaw that we automatically detect and document (see Section 4.2.1 and Table D.3, respectively). This results in the corresponding row in Table 1.1, although we exclude the "Notes" column for brevity.

We use this same procedure to track software qualities and supplementary terminology that is either shared by multiple approaches or too complicated to explain inline. We create a separate glossary for both qualities and supplementary terms, each with a similar format to our test approach glossary. Since these terms do not have categories, the process of recording them is much simpler, only requiring us to record the name, definition, and synonym(s) of these terms, along with any additional notes. The only new information we capture is the "precedence" for a software quality to have an associated test type, since each test type measures a particular software quality (see Table 2.1). These precedences are instances where a given software quality is related to, is covered by, or is a child, parent, or prerequisite of another quality with an associated test type, as given by the literature.

Tracking information about software qualities helps us investigate the literature more thoroughly, since these data may become relevant based on information from other yet-uninvestigated sources. When the literature mentions (or implies) a test approach that corresponds to a software quality we have recorded, we first follow the procedure given in Figure 3.2 with the information provided in the source that mentions it. We then remove the relevant data from our quality glossary and repeat our procedure with it to upgrade the quality to a test type in our test approach glossary.

3.3.1 Recording Implicit Information

Later: Ensure this is up to date

As described in Section 2.3, the use of natural language introduces significant nuance that we need to document. Keywords such as "implied", "can be", "sometimes", "should be", "ideally", "usually", "most", "likely", "often", "if", and "although" indicate that information from the literature is *not* explicit. These keywords often appear directly within the literature, but even when they do not, we use them to track explicitness in our test approach glossary to provide a more complete summary of the state of software testing literature without getting distracted by less relevant details. We find the following non-mutually exclusive cases of implicit information from the literature:

1. **The information follows logically** from the source and information from others, but is not explicitly stated.
2. **The information is not universal** but still applies in certain cases.
3. **The information is conditional**, requiring certain prerequisites to be satisfied (a more specific case of information not being universal).
4. **The information is dubious**; while it is present in the literature, there is reason to doubt its accuracy.

When we encounter information that meets one of these criteria, we use an appropriate keyword to capture this nuance in our test approach glossary (see Table 3.1). This also helps us identify implicit information when performing later analysis. Despite "implicit" only describing the first of these cases, we use it (as well as "implied by" when describing sources of information) as a shorthand for all "not explicit" information throughout this thesis for clarity.

Regarding the last entry in Table 3.1, if a test approach in our test approach glossary has a name ending in " (Testing)" (space included), then the word "Testing" might not be part of its name *or* it might not be a test approach at all! For example, the term "legacy system integration" is used in Gerrard (2000a, pp. 12–13, Tab. 2; 2000b, Tab. 1), but the more accurate "legacy system integration testing" is used in (2000b, pp. 30–31). In other cases where a term is *not* explicitly labelled as "testing", we add the suffix " (Testing)" (when it makes sense to do so) and consider the test approach to be implied.

3.3.2 Recording Flaws

While we can detect some subsets of flaws automatically by analyzing our test approach glossary (see Section 4.2.1), most are too complex and need to be tracked manually. We record these more detailed flaws along with extra information such as the flaw's manifestation (defined in Section 2.2.1), domain (defined in Section 2.2.2), and source(s) responsible, following the format in Appendix C.2. This helps us analyze these flaws later as described in Section 4.2.2.

It is important to note that when a flaw can be viewed in multiple ways, we record multiple pairs of manifestations and domains. For example, Kam (2008,

Table 3.1: Breakdown of keywords used for recording and analyzing implicitness.

Keyword	**Follows Logically**	**Not Universal**	**Conditional**	**Dubious**
“implied”	X			
“can be”		X	X	
“sometimes”		X	X	
“should be”		X		X
“ideally”		X		X
“usually”		X		
“most”		X		
“likely”	X	X		X
“often”		X		
“if”		X	X	X
“although”				X
“incorrectly”				X
“ (Testing)”	X			

p. 42) says “See *boundary value analysis*,” for the glossary entry of “boundary value testing” but does not include “boundary value analysis” in the glossary. This is trivially an example of a missing definition, but is also an example of incorrect traceability information. Therefore, we record both of these views[9], although we display this flaw as an example of incorrect traceability information as Mistake 27 since we determine this to be more meaningful.

[9] Discussed in #157.

3.4 Undefined Terms

The literature mentions many software testing terms without defining them. While this includes test approaches, software qualities, and more general software terms, we focus on the former as the main focus of our research. In particular, ISO/IEC and IEEE (2022) and Firesmith (2015) name many undefined test approaches. Once we exhaust the standards in Section 2.5.1, we perform miniature literature reviews on these subsets to "fill in" the missing definitions (along with any relations), essentially "snowballing" on these terms as described in Section 3.1. This process uncovers even more approaches, on which we can then repeat this process.

3.5 Stopping Criteria

Unfortunately, continuing to look for test approaches indefinitely is infeasible. We therefore need a "stopping criteria" to let us know when we are "finished" looking for test approaches in the literature. A reasonable heuristic is to repeat step 4 until it yields diminishing returns; i.e., investigating new sources does not reveal new approaches, relations between them, or information about them. This implies that something close to a complete taxonomy has been achieved!

Chapter 4

Tools

To better understand our findings, we build tools to visualize relations between test approaches (Section 4.1) and automatically analyze their flaws (Section 4.2). (We support this by using consistent syntax for recording data, which we outline in Appendix C to balance completeness and brevity.) Doing this manually would be daunting and error-prone because of the amount of data involved (for example, we identify 567 test approaches). There are also many situations where the underlying data would change, such as adding to it, further analyzing it, or correcting it. We also define LaTeX macros (Section 4.3) to help achieve our goals of maintainability, traceability, and reproducibility.

4.1 Approach Relation Visualization

We develop a tool to visualize the relations between test approaches so we can better understand them. This is possible because of our systematic tracking of synonym and parent-child relations (defined in Sections 2.1.2 and 2.1.3, respectively) in our test approach glossary (see Appendix C.1). For example, if the entries in Table 4.1 appear, then their parent-child relations are visualized as shown in Figure 4.1a. Overall, the parent-child relations between test approaches *should* result in something resembling a hierarchy (or multiple discrete hierarchies), although this is not the case because of flaws in the literature (see Section 2.2.2). We therefore visualize all parent-child relations as significant.

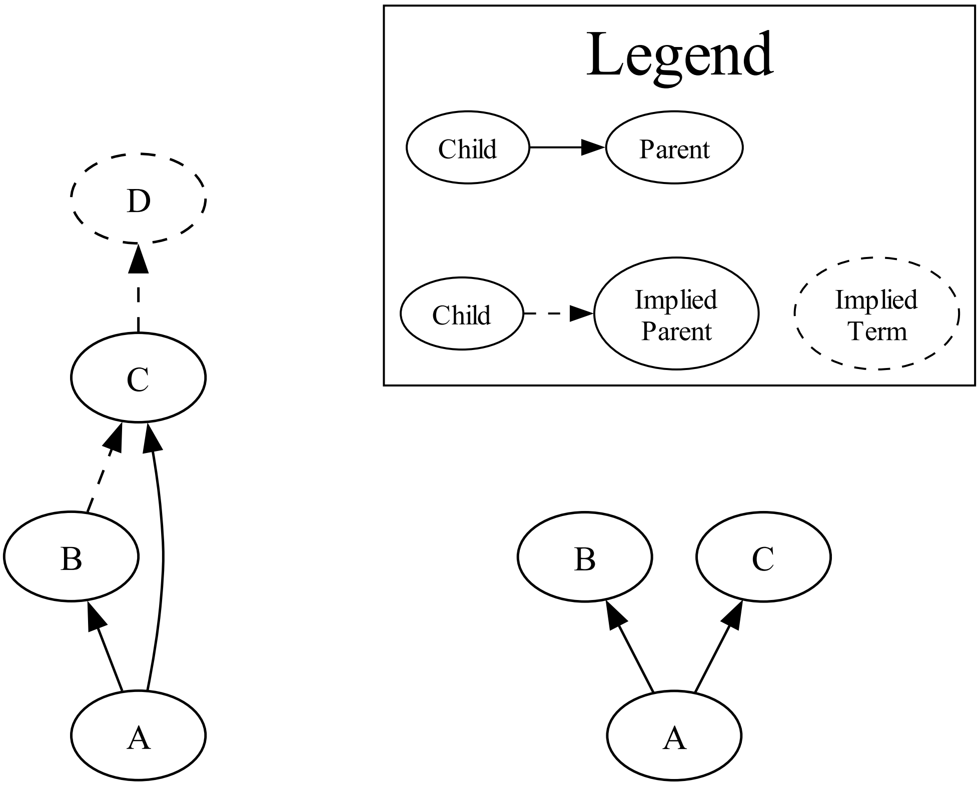
However, since each term is trivially a synonym of itself and there are many non-problematic synonyms that do not imply flaws (see Section 2.1.2), we only visualize the synonym relations that may indicate flaws given in Section 2.2.2; i.e., intransitive synonyms and synonyms between independently defined approaches. We deduce these conditions from the information in our glossary; for example, if the entries in Table 4.2 appear, then we visualize them as shown in Figure 4.2 (note that approach “X” does not appear since it does not have its own definition or violate transitivity). If a test approach does not have one of these relations *or* a parent-child relation, we call it an “orphan” approach (in contrast to the “parent” and “child” approaches defined in Section 2.1.3) and exclude it from any visualizations in which it would otherwise appear.

Q #1: Is it OK to “define” *orphan* here? We use the term infrequently and it requires us to define our “significant” synonym relations first

Table 4.1: Example glossary entries demonstrating how we track parent-child relations.

Name[a]	**Parent(s)**
A	B (Author, 2022; 2021), C (2022)
B	C (implied by Author, 2022)
C	D (implied by Author, 2017)
D (implied by Author, 2017)	

[a] "Name" can refer to the name of a test approach, software quality, or other testing-related term, but we only visualize relations between test approaches.



(a) Visualization from Table 4.1.

(b) Explicit visualization from Table 4.1.

Figure 4.1: Example generated visualizations of parent-child relations.

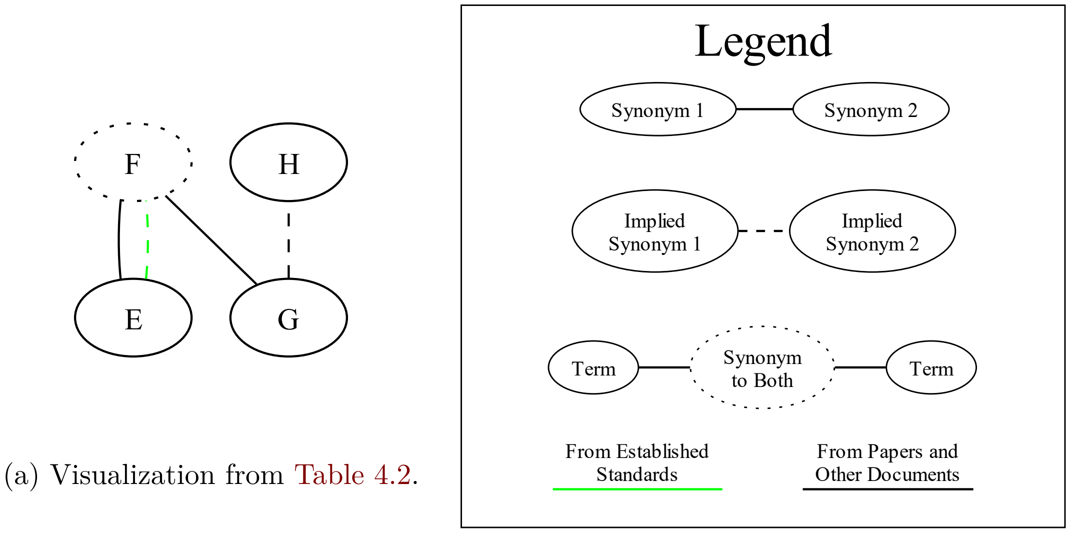
We also visualize the "explicitness" of information (defined in Section 2.3) by representing implicit approaches and relations with dashed lines (see Figures 4.1a and 4.2). If a relation is both explicit *and* implicit, we only display the latter if its source tier is more credible than the former's (see Sections 2.4 and 2.5). For example, if "StdAuthor" from Table 4.2 is the author of a standard, then we display the implicit relation from their document alongside the explicit one from "Author" as shown in Figure 4.2. Explicit approaches *always* have solid lines, even if they are also implicit. We can also omit implicit approaches and relations from visualizations; for example, Figure 4.1b is the explicit version of Figure 4.1a.

Since we cite all recorded relations as described in Appendix C.1, we can also colour each relation according to its source tier. Each source tier gets its own colour, which we label for each relevant source tier in a given visualization's legend (such as Figure 4.2), although we omit this colouring from Figures 4.1 and 4.4 for clarity. We also only display the relation with the most credible source tier (except if there is a more credible implicit relation as we previously describe). Finally, we also colour inferences (see Section 2.3) grey and proposals (see Appendix F) orange, such as in Figures 7.1, F.1, and F.2.

Table 4.2: Example glossary entries demonstrating how we track synonym relations.

Name[a]	**Synonym(s)**
E	F (Author, 2022; implied by StdAuthor, 2021)
G	F (Author, 2017), H (implied by 2022)
H	X (StdAuthor, 2021)

[a] "Name" can refer to the name of a test approach, software quality, or other testing-related term, but we only visualize relations between test approaches.



(a) Visualization from Table 4.2.

Figure 4.2: Example generated visualizations of synonym relations.

These visualizations tend to be large, so it is often useful to focus on specific subsets of them. For each approach category (defined in Section 2.1.1), we generate a visualization restricted to its approaches and the relations between them. We also generate a visualization of all static approaches along with the relations between them *and* between a static approach and a dynamic approach. This static-focused visualization is notable because static testing is sometimes considered to be a separate approach category (see Contradiction 5). Since dynamic approaches are our primary focus (see Appendix A.3), we include them in this static visualization, colouring their nodes grey to distinguish them. We can also generate more focused visualizations from a given subset of approaches, such as those pertaining to recovery testing. We use these visualizations to better understand the relations within these subsets of approaches, but we can also update them based on our recommendations in Appendix F by specifying sets of approaches and relations to add or remove.

4.2 Flaw Analysis

In addition to manually recording flaws (described in Section 3.3.2), we also automatically detect certain classes of flaws (Section 4.2.1). We can then analyze all of these flaws using automated tools (Section 4.2.2), giving us an overview of:

- how many flaws (defined in Section 2.2) there are,
- how these flaws present themselves (see Section 2.2.1),
- in which knowledge domains these flaws occur (see Section 2.2.2),
- how explicit (see Section 2.3) these flaws are, and
- how responsible each source tier (defined in Section 2.5) is for these flaws.

To understand where flaws exist in the literature, we group them based on the source tier(s) responsible for them. We then count each flaw *once* per source tier if it appears within it *and/or* between it and a more credible tier[1] (see Sections 2.4 and 2.5). This avoids counting the same flaw more than once for a given source tier, which would give the number of *occurrences* of all flaws instead of the more useful number of flaws *themselves*. When taking a more detailed look at the *sources* of flaws (as opposed to just the responsible source *tiers*) as we do in Figure 5.1, we also count the following sources of flaws separately:

1. self-contained flaws (defined in Section 2.2.3),
2. internal flaws (defined in Section 2.2.3),
3. those between documents with the same set of authors, which includes:

[1] If an inconsistency occurs between two source tiers and the more credible one is *incorrect*, we instead count it as an inconsistency between it and the asserted truth from the less credible source, as described in Appendix C.2.

(a) the various combinations of authors of established standards (defined in Section 2.5.1)—ISO, the International Electrotechnical Commission (IEC), and IEEE—as shown in Figure 4.3 and

(b) the different versions of the Guides to the SoftWare Engineering Body Of Knowledge (SWEBOK Guides) (Washizaki, 2025a; 2024; Bourque and Fairley, 2014)[2], and

4. those within a single source tier.

As before, we do not double count these sources of flaws, meaning that the maximum number of counted flaws possible within a *single* source tier in this more detailed view is four (one for each type). This only occurs if there is an example of each flaw source that is *not* ignored to avoid double counting; for example, while a single flaw within a single document would technically and trivially fulfill all four criteria, we would only count it once.

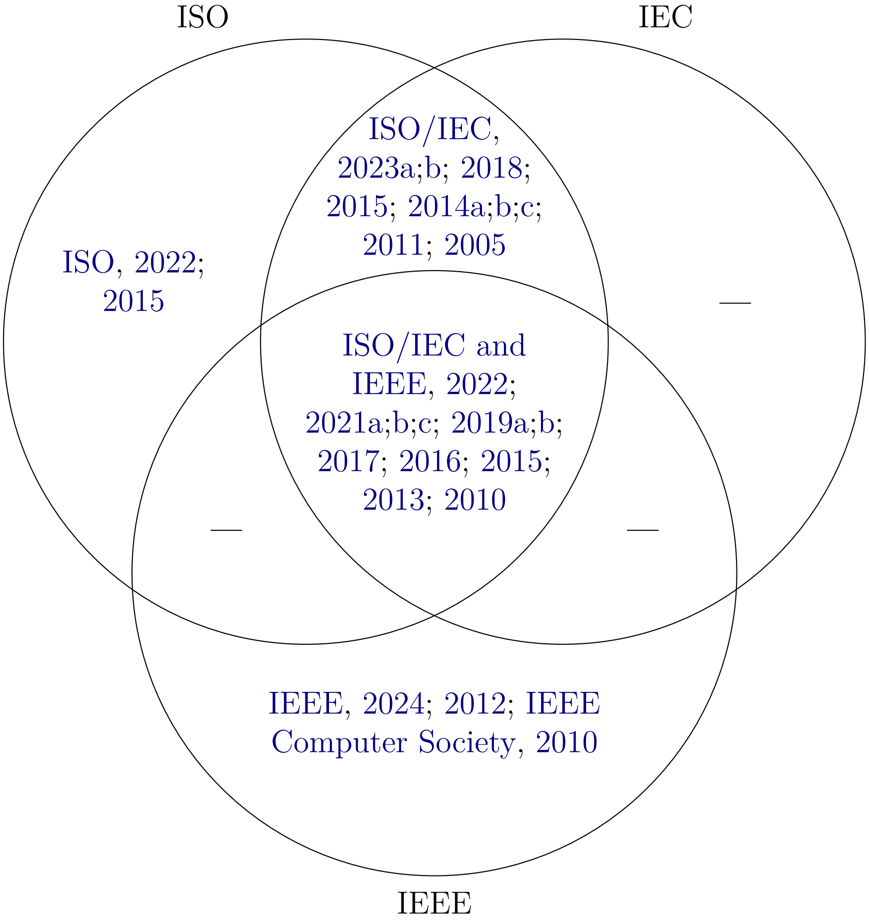

ISO
IEC
IEEE
ISO, 2022; 2015
ISO/IEC, 2023a;b; 2018; 2015; 2014a;b;c; 2011; 2005
—
ISO/IEC and IEEE, 2022; 2021a;b;c; 2019a;b; 2017; 2016; 2015; 2013; 2010
—
—
IEEE, 2024; 2012; IEEE Computer Society, 2010

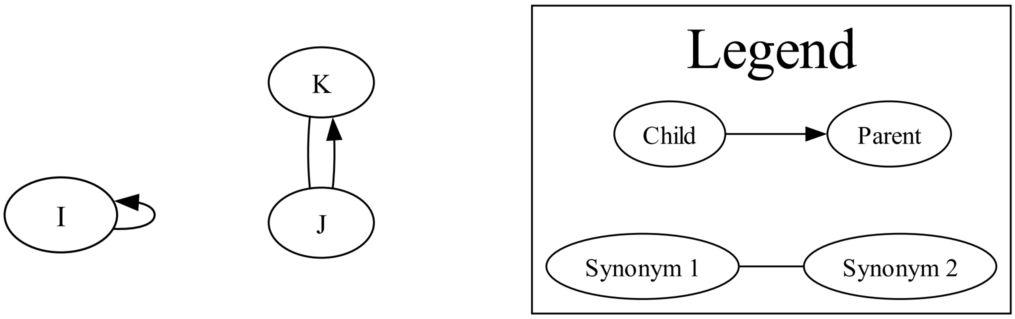
Figure 4.3: The sets of authors of established standards.

[2] Although these documents have different editors, they are published by the same organization: the IEEE Computer Society (Washizaki, 2025b; see Section 2.5.2).

4.2.1 Automated Flaw Detection

As outlined in Section 2.2.2, we automatically detect synonym relations from our test approach glossary that violate transitivity to generate our visualizations. These relations are significant because they indicate potential flaws. We automatically detect and format these flaws to present them when discussing synonym relation flaws in Section 5.2.2. For these and other kinds of flaws, we also generate the corresponding comments described in Appendix C.2 and include them in the corresponding LaTeX files to ensure that we analyze and count these flaws in addition to those we record manually. Since we already automatically detect one kind of flaw, the next logical step is then to detect more. We detect the following classes of flaws that we later discuss in Sections 5.2.1 and 5.2.3:

- Test approaches with more than one category that violate our assumption of orthogonality (see Section 2.1.1).
- Parent-child relations that violate irreflexivity as outlined in Section 2.1.3 (i.e., cases where a child is given as a parent of itself); a case of this with approach "I" would result in output similar to Figure 4.4a.
- Pairs of approaches with a synonym relation *and* a parent-child relation as described in Section 2.2.2; a case of this with approaches "J" and "K" would result in output similar to Figure 4.4b.



(a) Visualization of a reflexive parent relation.

(b) Visualization of a pair of terms with a parent-child *and* synonym relation.

Figure 4.4: Example generated visualizations containing flaws.

While just counting the total number of flaws (found automatically *or* manually) is trivial, tracking the source(s) of these flaws is more useful, albeit more involved. Since we consistently track the appropriate citations for each piece of information we record (see Tables 4.1 and 4.2 for examples of our citation format), we can use them to identify the offending source tier(s). This comes with the added benefit that we can format these citations to use with LaTeX's citation commands in this thesis, including generating the comments described in Appendix C.2.

Alongside this citation information, we include keywords so we can assess how "explicit" a piece of information is (see Section 2.3). This is useful when counting flaws, since they can be both objective and subjective but should not be double counted as both! When presenting the numbers of flaws sorted by various criteria in Chapter 5, we only count each flaw for its most "explicit" occurrence, similarly to how we visualize the relations between approaches as described in Section 4.1.

4.2.2 Flaw Comment Analysis

To perform more detailed analysis on the flaws we uncover, we use LaTeX comments to capture information about the flaws themselves as outlined in Appendix C.2. We include these flaw comments when manually recording flaws from the literature (Section 3.3) and generate them when automatically detecting flaws from our test approach glossary (Section 4.2.1).

The main way we use these comments is to determine where each flaw originates. We compare the authors and years of each source involved with a given flaw to determine if it manifests within a single document and/or between documents with the same set of authors. Then, we group these sources into their tiers (see the relevant source code). We then distill these lists of sources down to sets of tiers and compare them against each other to determine how many times a given flaw manifests between source tiers, which we use when counting flaws in Chapter 5. We also parse implicit information following the same rules given in Section 4.2.1 for automatically detecting flaws.

In cases where a flaw can be viewed in multiple ways (see Section 3.3.2), we only represent the flaw once in the subsections of Chapter 5 and the full lists in Appendix D based on the pair of keys that appears first. This allows us to decide which view is most central to understanding the flaw without affecting the results of our research. The choice of which flaw is more "meaningful" only affects its presentation, since we also count its other views as flaws (for example, in Tables 5.1 and 5.2), with the benefit of not introducing clutter by displaying it in full multiple times.

4.3 Helper Commands

To improve maintainability, traceability, and reproducibility, we define helper commands (also called "macros") for content that is prone to change or used in multiple places. For example, we use scripts to calculate values based on our glossaries and save them to files to be assigned to corresponding macros. We use these throughout our documents instead of manually updating these constantly changing values, which is prone to error. Table 4.3 lists these macros and descriptions of what they represent. Our scripts convert numbers to their textual equivalents when necessary to follow IEEE guidelines.

Table 4.3: Macros for calculated values.

Macro	**What it Counts**
`\approachCount{}`	Identified test approaches
`\undefPerc{}`[a]	Percentage of undefined test approaches
`\orphanCount{}`	Orphan approaches (described in Section 4.1)
`\uncatCount{}`	Approaches with the generic category "Approach"
`\qualityCount{}`	Identified software qualities
`\srcCount{}`[b]	Sources used in glossaries
`\flawCount{}`[c]	Identified flaws
`\TotalBefore{}`[c]	Test approaches identified before step 4[d]
`\UndefBefore{}`[c]	Undefined test approaches identified before step 4[d]
`\TotalAfter{}`[c]	Test approaches identified after step 4[d]
`\UndefAfter{}`[c]	Undefined test approaches identified after step 4[d]
`\multiCatCount{}`	Total number of approaches with multiple categories
`\multiCatMax{}`	Category with the most overlaps
`\multiCatMaxCount{}`	Number of overlaps involving the previous category
`\multiSynCount{}`	Terms given as synonyms for multiple discrete terms
`\parSynCount{}`	Pairs of test approaches with a child-parent *and* synonym relation
`\selfParCount{}`	Test approaches that are a parent of themselves

[a] Calculated in LaTeX from other macros for reuse.

[b] Calculated in LaTeX from source tier lists.

[c] These macros are defined as counters to allow them to be used in calculations within LaTeX (such as in `\undefPerc{}`, Section 3.4, and Figure 7.2).

[d] Step 4 of our methodology involves iterating over undefined terms and is described in more detail in Section 3.4.

We also generate more involved macros for flaw counts and sections. We count flaws based on their manifestation and domain, explicitness, and source tier, then store these data in corresponding files that we read in and assign to macros. For example, `\srcCount{}` is simply an alias for `\totalFlawDmnBrkdwn{15}` (which is equivalent to `\totalFlawMnfstBrkdwn{13}`). Similarly, we use scripts to generate macros for flaw manifestations, flaw domains, and source tiers as shown in Table 4.4. Macros for the latter take an integer input as follows to access either:

1. the source tier's name,
2. the list of sources in the tier, or
3. the number of sources in the tier.

Table 4.4: Macros for referencing well-defined sections.

	Flaw Manifestations[a]	**Flaw Domains**[b]	**Source Tiers**[c]
Macros (Values)	`\wrong{}` (Mistakes)	`\cats{}` (Categories)	`\stds{}` (Established Standards)
	`\miss{}` (Omissions)	`\syns{}` (Synonyms)	`\metas{}` (Terminology Collections)
	`\contra{}` (Contradictions)	`\pars{}` (Parents)	`\texts{}` (Textbooks)
	`\ambi{}` (Ambiguities)		`\papers{}` (Papers and Other Documents)
	`\over{}`[d] (Overlaps)		`\papers*{}`[e] (Papers and Others)
	`\redun{}` (Redundancies)		
Used In	Tables 2.2 and 5.1	Tables 2.3 and 5.2	Figures 2.1 and 5.1 to 5.3 Tables 5.1 and 5.2

[a] Defined in Section 2.2.1; we also define starred versions, such as `\wrong*{}` (Mistake), that use the singular noun for use in Table 2.2.

[b] Defined in Section 2.2.2; we only include domains with their own section.

[c] Defined in Section 2.5.

[d] We overwrite the primitive TEX command `\over{}` since we do not otherwise use it.

[e] Used in Tables 5.1 and 5.2 to manage line length.

Chapter 5

Observed Flaws

After gathering all these data[1], we find 341 flaws. Figure 5.1 shows where these flaws appear within each source tier (defined in Section 2.5) which reveals the following about software testing literature:

1. Established standards are not actually standardized, since:
   (a) other documents frequently disagree with them and
   (b) they are the most internally inconsistent source tier!
2. Less standardized documents, such as terminology collections and textbooks, are also not followed to the extent they should be.
3. Documents across the board have flaws within the same document, between documents with the same author(s), or even with assertions of ground truth!

To better understand and analyze these flaws, we group them by their manifestations and their domains as defined in Section 2.2. We present the total number of flaws by manifestation and by domain in Tables 5.1 and 5.2, respectively, where a given row corresponds to the number of flaws either within that source tier and/or with a more credible one (i.e., a previous row in the table). We also group these flaws by their explicitness (defined in Section 2.3) by counting (Obj)ective and (Sub)jective flaws separately, since additional context may rectify them. Since we give each flaw a manifestation *and* a domain, the totals per source and grand totals in these tables are equal. We also summarize how many flaws appear in each source tier in Figure 5.2 (note that these values align with the totals in Tables 5.1 and 5.2) and normalize these totals to the number of documents in each tier in Figure 5.3.

[1] Available in `ApproachGlossary.csv`, `QualityGlossary.csv`, and `SuppGlossary.csv` at https://github.com/samm82/TestingTesting.

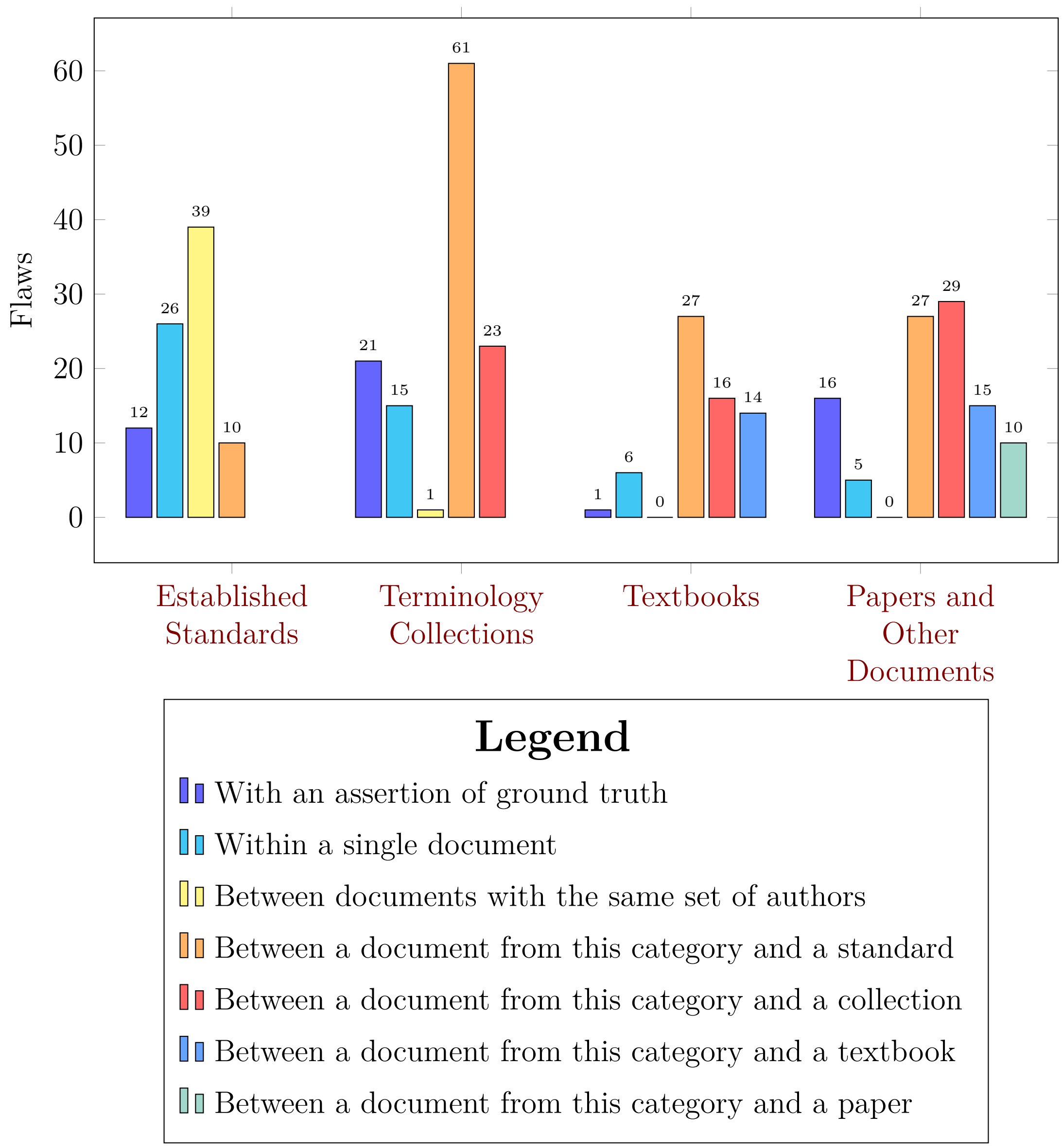


Figure 5.1: Identified flaws by the source tier responsible. Some bars are omitted as they correspond to comparisons we do not make; see Section 4.2.

Table 5.1: Breakdown of identified flaws by manifestation and source tier.

	Mistakes		Omissions		Contradictions		Ambiguities		Overlaps		Redundancies		
Source Tier	**Obj**	**Sub**	**Obj**	**Sub**	**Obj**	**Sub**	**Obj**	**Sub**	**Obj**	**Sub**	**Obj**	**Sub**	**Total**
Established Standards	9	3	2	0	23	8	6	1	7	0	3	0	62
Terminology Collections	20	0	3	0	49	18	16	1	6	0	2	0	115
Textbooks	14	2	2	0	33	9	2	0	1	0	0	0	63
Papers and Others	20	3	7	0	30	28	9	0	0	1	3	0	101
Total	63	8	14	0	135	63	33	2	14	1	8	0	341

Table 5.2: Breakdown of identified flaws by domain and source tier.

	Categories		Synonyms		Parents		Definitions		Labels		Scope		Trace.		
Source Tier	**Obj**	**Sub**	**Obj**	**Sub**	**Obj**	**Sub**	**Obj**	**Sub**	**Obj**	**Sub**	**Obj**	**Sub**	**Obj**	**Sub**	**Total**
Established Standards	12	2	4	5	10	3	14	2	10	0	0	0	0	0	62
Terminology Collections	16	10	15	3	14	4	23	0	18	2	5	0	5	0	115
Textbooks	2	0	14	3	9	7	17	1	8	0	0	0	2	0	63
Papers and Others	16	13	19	12	11	6	13	0	7	1	0	0	3	0	101
Total	46	25	52	23	44	20	67	3	43	3	5	0	10	0	341

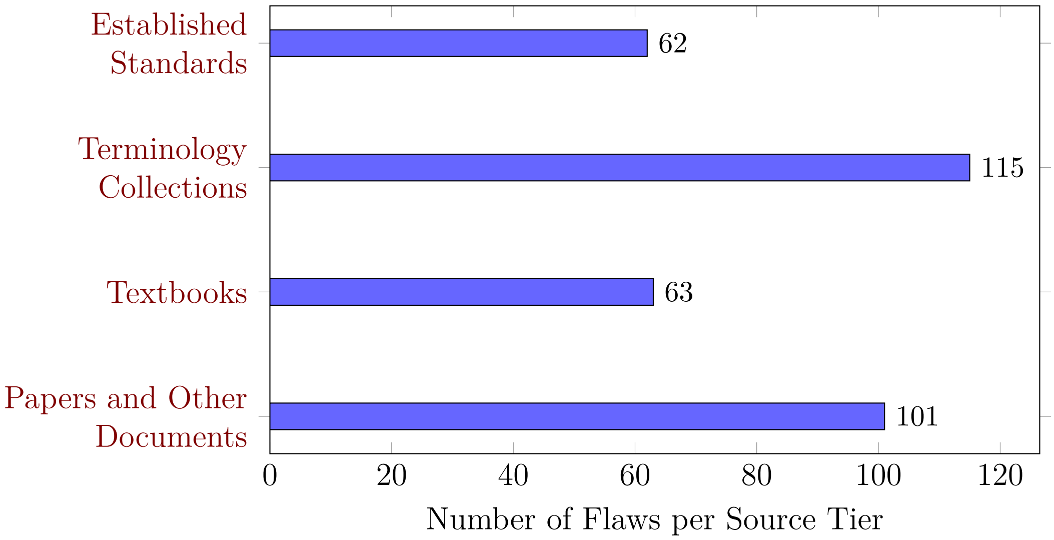


Figure 5.2: Identified flaws by the source tier responsible.

As shown in Figure 5.2, each source tier contains a comparable number of flaws, although this changes drastically when normalizing these totals by the number of documents in each source tier in Figure 5.3. Since some terminology collections are glossaries or taxonomies of terms, they contain a much larger proportion of relevant information that we critique. Conversely, standards and papers contain lots of content we do *not* investigate for scope reasons and time constraints, likely causing the relatively low numbers of flaws per document in these tiers. Textbooks lie somewhere in the middle, since we investigate them to varying degrees of thoroughness: the whole book, a specific chapter, and so on.

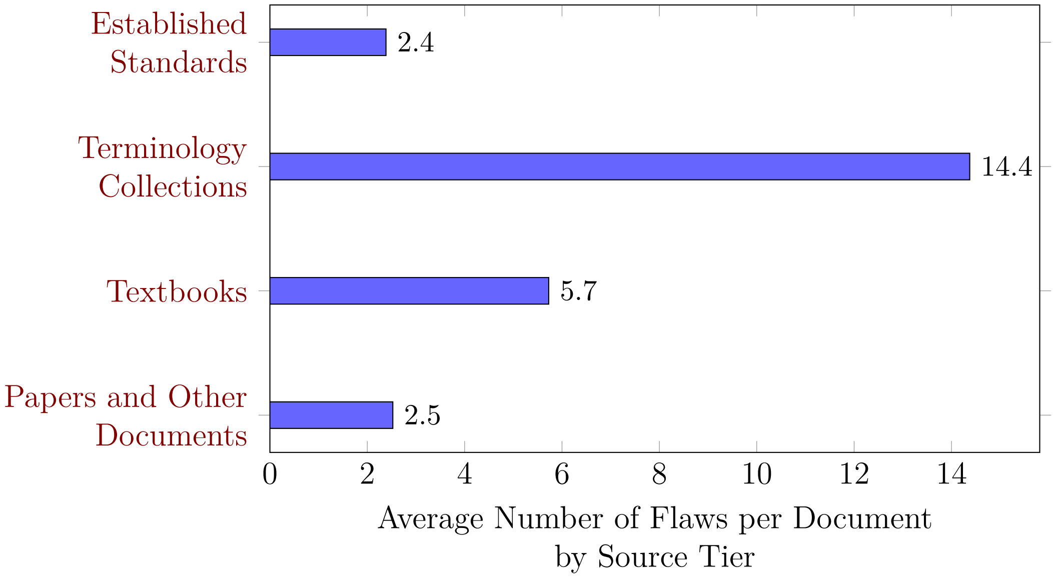


Figure 5.3: Normalized summary of identified flaws by the source tier responsible.

From these tables and figures, we can draw some conclusions about *how* the literature is flawed:

1. Contradictions are by *far* the most common manifestation. This is likely because these are the most obvious flaws to detect automatically (see Sections 4.2.1 and 5.2) and because two (sets of) authors using different resources and not communicating have a high chance of disagreeing.

2. Approach categorizations are the most subjective and one of the most common flaw domains, likely due to the lack of standardization about what categories to use (see Section 6.2 for more detailed discussion).

3. In general, semantic flaws are more common than syntactic ones. The number of category flaws is comparable to the numbers of flaws with the relations and definitions of test approaches.

We summarize the flaws that we discover manually in Section 5.1 based on their manifestation. This lets us separately summarize the flaws we automatically detect (see Section 4.2.1) based on their domain in Section 5.2. We list *all* these flaws in Appendix D to balance completeness and brevity and denote implicit relations with the phrase "implied by" in Tables D.1 to D.3 as described in Section 2.3. Moreover, certain "subsets" of testing contain many interconnected flaws which we present in Section 5.3 as a "third view" to keep related information together. The counts of flaws given in Tables 5.1 and 5.2 are essentially the sums of the flaws we describe in the following subsections. Finally, we infer some flaws as described in Section 2.3, which do not contribute to any counts because they are subjective; we list these in Appendix D.3 for completeness.

Due to time constraints, this collection of flaws is still not comprehensive! While we apply our heuristics for identifying relevant terms (see Section 3.2) to the entirety of most investigated sources, especially established standards (see Section 2.5.1), we are only able to investigate some sources in part. These mainly comprise of sources chosen for a specific area of interest or based on a test approach that was later determined to be out-of-scope. These include the following sources as described in Section 7.1: ISO (2022; 2015); Dominguez-Pumar et al. (2020); Pierre et al. (2017); Trudnowski et al. (2017); Yu et al. (2011); Tsui (2007); Goralski (1999). Since our research began, the draft version of the SWEBOK Guide v4.0 (Washizaki, 2024) has been updated and published (2025a); we revisit this version to see which flaws in the draft were resolved (we find two notable cases of this) following our heuristics based on these subsets, but lack the time to investigate this new version in full. Finally, some heuristics only arose as research progressed, particularly those for deriving test approaches (see Section 3.2); while reiterating over investigated sources would be ideal, this is infeasible due to time constraints.

From this partial implementation of our methodology, we identify 567 test approaches, 35% of which are undefined. With more time, we would continue to snowball on these undefined terms following the procedure in Section 3.4 (see Section 7.1 for more detailed discussion on what we accomplished). Likewise, we are unable to reach our stopping criteria outlined in Section 3.5. We consider the discovery of property-based testing as an alternate stopping point for our research[2] since we are surprised that it is not mentioned in any sources we investigated. Even so, we have to stop our snowballing approach before we discover property-based testing in the literature! With more time, we would uncover this along with other test approaches that did not arise (as described in Section 7.2), but unfortunately, we impose our stopping point artificially.

5.1 Flaws by Manifestation

The following sections list observed flaws grouped by *how* they manifest as presented in Section 2.2.1. These include mistakes (Section 5.1.1), omissions (Section 5.1.2), contradictions (Section 5.1.3), ambiguities (Section 5.1.4), overlaps (Section 5.1.5), and redundancies (Section 5.1.6).

5.1.1 Mistakes

There are many ways that information can be incorrect, which we identify in Table 5.3. We provide an example below for those that are less straightforward; see Appendix D.1.1 for the full list of mistakes.

Table 5.3: Different kinds of mistakes found in the literature.

Description	**Count**
Information is incorrect based on an assertion from another source	10
Information is provided with an incorrect scope	7
Information is not present where it is claimed to be	6
Information contains a minor mistake	4[a]
Incorrect information makes other information incorrect	2

[a] Comprises three typos and one duplication.

Information is incorrect based on an assertion from another source

Washizaki (2025a, p. 5-4) says that quality improvement, along with quality assurance, is an aspect of testing that involves "defining methods, tools, skills, and practices to achieve the specific quality level and objectives"; while testing that a system possesses certain qualities is in scope, actively improving the system in response to these results is *not* itself part of testing (ISO/IEC and IEEE, 2022, p. 10; 2021c, p. 6; 2017, p. 473).

[2] Discussed in #57, #81, #88, and #125.

Information is provided with an incorrect scope

Hamburg and Mogyorodi (2024) define "par sheet testing" as "testing to determine that the game returns the correct mathematical results to the screen, to the players' accounts, and to the casino account". This seems to refer to the specific example from Mistake 11 and Mistake 12 and could be a valid domain-specific test approach, but this definition does not even seem specific to PAR sheets—"list[s] of all the symbols on each reel of a slot machine" (Bluejay, 2024)—themselves!

Incorrect information makes other information incorrect

The incorrect claim that "white-box testing", "grey-box testing", and "black-box testing" are synonyms for "module testing", "integration testing", and "system testing", respectively, (see Mistake 31) casts doubt on the claim that "red-box testing" is a synonym for "acceptance testing" (Sneed and Göschl, 2000, p. 3) (see Mistake 32).

5.1.2 Omissions

We find four cases where a definition is omitted, one where a category is omitted, and one where a term (along with its relations) is omitted; we list these in Appendix D.1.2.

5.1.3 Contradictions

There are many cases where multiple sources of information (sometimes within the same document!) disagree. We find this happen with six categories, six synonym relations, seven parent-child relations, 17 definitions, and three labels. These can be found in Appendix D.1.3.

5.1.4 Ambiguities

Some information given in the literature is unclear; there is definitely something "wrong", but we cannot deduce the intent of the original author(s). We identify the kinds of ambiguous information given in Table 5.4; see Appendix D.1.4 for the full list of ambiguities.

Table 5.4: Different kinds of ambiguities found in the literature.

Description	Count
A term is defined ambiguously	7
A term is used inconsistently	3
The distinction between two terms is unclear	2

5.1.5 Overlaps

While information given in the literature should be atomic, this is not always the case. We find three definitions that overlap, two terms with multiple definitions, and three terms that share acronyms. We list these in Appendix D.1.5; note that we track two of the terms with multiple definitions as the same flaw since they are related.

5.1.6 Redundancies

We find redundancies in two parent-child relations, two definitions, and two labels as listed in Appendix D.1.6.

5.2 Flaws by Domain

The following sections present flaws that we detect automatically (see Section 4.2.1) grouped by *what* information is flawed as presented in Section 2.2.2. We also provide more detailed information for areas that may benefit from further analysis. The domains we focus on here are test approach categories (Section 5.2.1), synonym relations (Section 5.2.2), and parent-child relations (Section 5.2.3).

5.2.1 Approach Category Flaws

While the IEEE categorization of test approaches described in Table 2.1 is useful, it is not without its faults. One issue, which is not inherent to the categorization itself, is the fact that it is not used consistently (see Table 6.1). The most blatant example of this is that ISO/IEC and IEEE (2017, p. 286) describe mutation testing as a methodology, even though this is not one of the categories *they* created! These categories are not consistently defined so some approaches are categorized differently by different sources; we track these differences so we can analyze them more systematically.

In particular, we automatically detect test approaches with more than one category that violate our assumption of orthogonality (see Section 2.1.1). We identify 29 such cases that we summarize in Table 5.5 and list along with their sources in Table D.1 for completeness. Most of these flaws (24) involve the category of “test technique”, which may simply be because authors use this term more generally to mean “test approach”. A more specific reason for this is how the line between “test technique” and “test practice” can blur when these categories are *not* transitive. For example, “some test practices, such as exploratory testing or model-based testing are sometimes [incorrectly] referred to as ‘test techniques’ ... as they are not themselves providing a way to create test cases, but

Table 5.5: Summary of pairs of categories assigned to a test approach.

Categories	Count
Technique/Level	3
Technique/Practice	8
Technique/Type	13
Level/Type	5
Total	29

instead use test design techniques to achieve that" (ISO/IEC and IEEE, 2022, p. 11; 2021a, p. 5). This seems to be the case for the following approaches:

- Exploratory testing (2022, p. 33; 2021c, p. viii; 2013, p. 13)
- Experience-based testing (2022, p. 4; 2021c, pp. viii, 4; 2013, p. 33)
- Scripted testing (2022, p. 33)
- Ad hoc testing (Washizaki, 2025a, p. 5-14; Hamburg and Mogyorodi, 2024; Kam, 2008, p. 42)
- Model-based testing (Engström and Petersen, 2015, pp. 1–2; Kam, 2008, p. 4)

As described in Section 2.3, we infer that child approaches inherit their parents' categories. However, there seem to be exceptions to this, which may indicate that categories are *not* transitive or that they are *except* in certain cases. For example, the practice of experience-based testing has many subtechniques, as described above, but it *also* has subpractices, including tours (ISO/IEC and IEEE, 2022, p. 34) and exploratory testing (although the latter is categorized inconsistently; see Table D.1 and above discussion). Similarly, the conflicting categorizations of beta testing in Table D.1 may propagate to its children closed beta testing and open beta testing. When we infer these flaws, we exclude them from Tables 5.5 and D.1 and instead include them in Table D.3 for completeness.

5.2.2 Synonym Relation Flaws

While synonyms do not inherently signify a flaw (as we discuss in Section 2.1.2), the software testing literature is full of incorrect and ambiguous synonyms that do. As described in Section 2.2.2, we pay special attention to synonyms between independently defined approaches (which may be flaws) and to intransitive synonyms (which definitely *are* flaws). We present explicit (see Section 2.3) synonym relations that fit either of these criteria in Figure 5.4, which we automatically generate from our test approach glossary and manually modify for legibility. These relations are given as described by the literature and are therefore flawed. We provide the full list of synonyms that violate transitivity (along with their sources) in Appendix D.2.2 and discuss other kinds of flawed synonym relations in Sections 5.2.3, 5.3.3, and 5.3.4 and Appendices D.2.3 and D.3.3.

Later: Ensure this is up to date

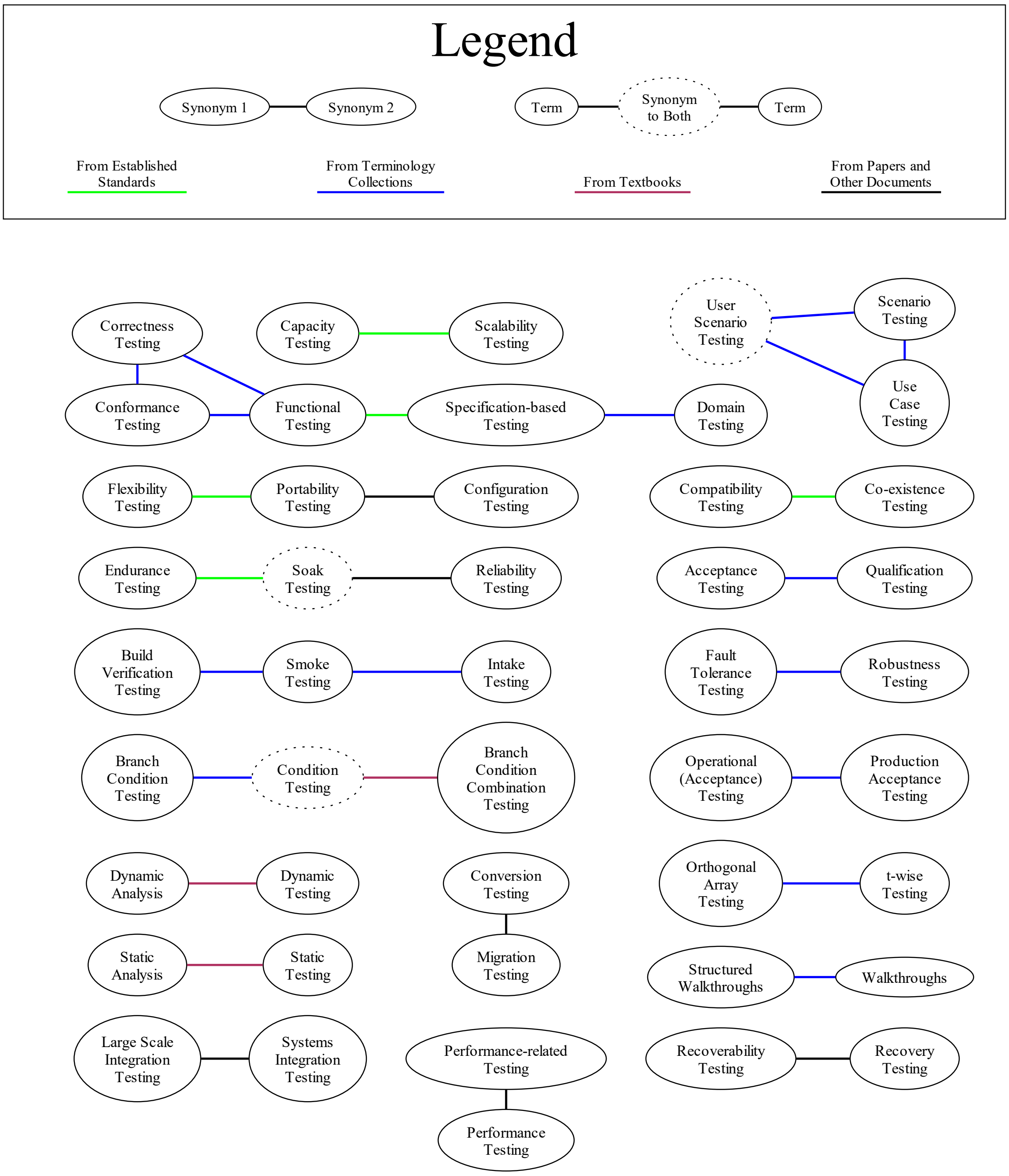


Figure 5.4: Significant synonym relations given explicitly by the literature.

5.2.3 Parent-Child Relation Flaws

Parent-child relations are also not immune to flaws. For example, some approaches are given as parents of themselves, which violates the irreflexivity of this relation as defined in Section 2.1.3. We identify the following three examples through automatic analysis of our generated graphs (see Section 4.2.1):

1. Performance Testing (Gerrard, 2000a, Tab. 2; 2000b, Tab. 1)
2. System Testing (Firesmith, 2015, p. 23)
3. Usability Testing (Gerrard, 2000a, Tab. 2; 2000b, Tab. 1)

Interestingly, Gerrard (2000a;b) does *not* give performance testing as a subapproach of usability testing, which would have been more meaningful information to include.

There are also pairs of synonyms where one is a subapproach of the other; these relations cannot coexist since synonym relations are symmetric while parent-child relations are asymmetric (as outlined in Sections 2.1.2 and 2.1.3, respectively). We identify 17 of these pairs through automatic analysis of our generated visualizations as described in Section 4.2.1. We visualize the pairs where both relations are explicit in Figure 5.5 and list all identified pairs in Table D.2, as well as pairs where we infer a flaw in Appendix D.3.3 for completeness.

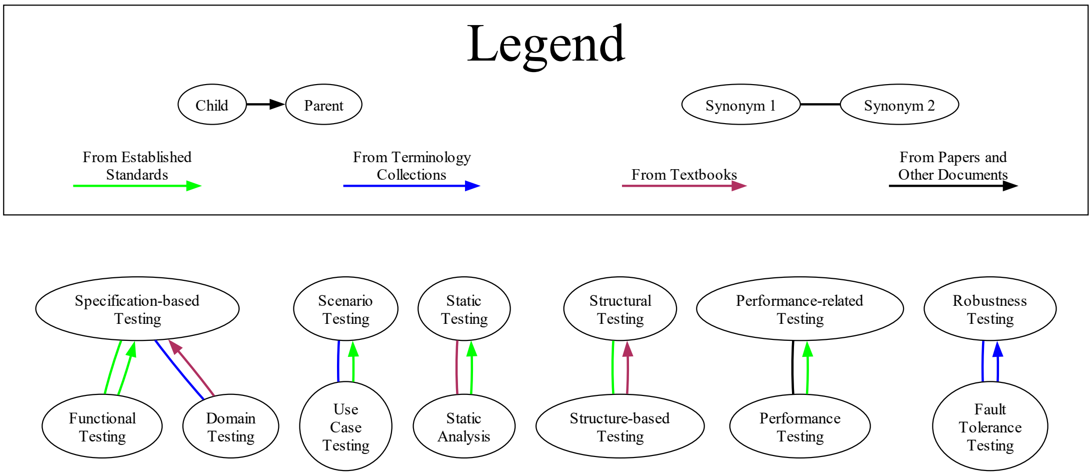


Figure 5.5: Pairs of test approaches with a parent-child *and* synonym relation given explicitly by the literature.

Of particular note is the relation between path testing and exhaustive testing. While van Vliet (2000, p. 421) claims that path testing done completely “is equivalent to exhaustively testing the program”[3], this overlooks the effects of input data (ISO/IEC and IEEE, 2021c, pp. 129–130; Patton, 2006, p. 121; Peters and

[3] The contradictory definitions of path testing given in Contradiction 17 add another layer of complexity to this claim.

Pedrycz, 2000, p. 467) and implementation issues (p. 476) on the code's behaviour. Exhaustive testing requires "all combinations of input values *and* preconditions … [to be] tested" (ISO/IEC and IEEE, 2022, p. 4, emphasis added; similar in Hamburg and Mogyorodi, 2024; Patton, 2006, p. 121).

5.3 Flaws by Subset

Some "subsets" of the software testing literature contain interconnected flaws. We describe them here, along with relevant data (e.g., definitions) that are not necessarily flawed. We use the key-pairs for each flaw (as described in Section 2.2) to denote the actual flaws in these subsets; for example, `(CONTRA, SYNS)` denotes a contradictory synonym. We highlight the subsets of operational (acceptance) testing (Section 5.3.1), recovery testing (Section 5.3.2), scalability testing (Section 5.3.3), and compatibility testing (Section 5.3.4).

5.3.1 Operational (Acceptance) Testing

(CONTRA, LABELS)

There are two names that the literature gives to this test approach:

- *Operational Acceptance Testing (OAT)* (ISO/IEC and IEEE, 2022, p. 22; Hamburg and Mogyorodi, 2024) and
- *Operational Testing (OT)* (ISO/IEC, 2018; ISO/IEC and IEEE, 2017, p. 303; Washizaki, 2025a, p. 6-9, in the context of software engineering operations; Bourque and Fairley, 2014, pp. 4-6, 4-9).

(CONTRA, SYNS)

Firesmith (2015, p. 30) lists the above terms separately, but they are considered synonyms elsewhere (LambdaTest, 2024; Bocchino and Hamilton, 1996); since Firesmith does not define these terms, it is hard to evaluate his distinction.

find more academic sources

5.3.2 Recovery Testing

"Recovery testing" is "testing … aimed at verifying software restart capabilities after a system crash or other disaster" (Washizaki, 2025a, p. 5-9) including "recover[ing] the data directly affected and re-establish[ing] the desired state of the system" (ISO/IEC, 2023a) so that the system "can perform required functions" (ISO/IEC and IEEE, 2017, p. 370). However, the literature also describes similar test approaches with vague or non-existent distinctions between them. We describe these approaches and their flaws here and present the relations between them in Figure F.1a.

- *Recoverability testing* evaluates "how well a system or software can recover data during an interruption or failure" (Washizaki, 2025a, p. 7-10; similar in

ISO/IEC, 2023a) and "re-establish the desired state of the system" (2023a). Kam (2008, p. 47) gives this as a synonym for "recovery testing".

- *Disaster/recovery testing* evaluates if a system can "return to normal operation after a hardware or software failure" (ISO/IEC and IEEE, 2017, p. 140) or if "operation of the test item can be transferred to a different operating site and … be transferred back again once the failure has been resolved" (2021c, p. 37).
  - `(OVER, DEFS)` These two definitions seem to describe different aspects of the system, where the first is intrinsic to the hardware/software and the second might not be, making this term nonatomic.
- *Backup and recovery testing* "measures the degree to which system state can be restored from backup within specified parameters of time, cost, completeness, and accuracy in the event of failure" (ISO/IEC and IEEE, 2013, p. 2). This may be what is meant by "recovery testing" in the context of performance-related testing (2022, Fig. 2).
- *Backup/recovery testing* determines the ability of a system "to restor[e] from back-up memory in the event of failure, without transfer[ing] to a different operating site or back-up system" (ISO/IEC and IEEE, 2021c, p. 37).
  - `(CONTRA, PARS)` This given as a subtype of "disaster/recovery testing" which tests if "operation of the test item can be transferred to a different operating site" (2021c, p. 37), even though this is *explicitly* excluded from its definition on the same page!
  - `(OVER, LABELS)` Its name is also quite similar to "backup and recovery testing", adding further confusion.
- *Failover/recovery testing* determines the ability "to mov[e] to a back-up system in the event of failure, without transfer[ing] to a different operating site" (ISO/IEC and IEEE, 2021c, p. 37).
  - `(CONTRA, PARS)` This is also given as a subtype of "disaster/recovery testing" which tests if "operation of the test item can be transferred to a different operating site" (p. 37), even though this is *explicitly* excluded from its definition on the same page!
  - `(AMBI, PARS)` While not explicitly related to recovery, *failover testing* "validates the SUT's ability to manage heavy loads or unexpected failure to continue typical operations … by allocating extra resources" (Washizaki, 2025a, p. 5-9) or entering a "backup operational mode in which [these responsibilities] … are assumed by a secondary system" (Hamburg and Mogyorodi, 2024). Its name implies that it is a child of "failover/recovery testing" but its definition makes it more broad (as it includes handling "heavy loads" where failover/recovery testing does not) which may reverse the direction of this relation.

  - `(AMBI, SYNS)` Firesmith (2015, p. 56) uses the term “failover and recovery testing” which may be a synonym of “failover/recovery testing”.

- *Restart & recovery (testing)* is listed as a test approach by Gerrard (2000a, Fig. 5) but is not defined `(MISS, DEFS)` and may simply be a synonym to “recovery testing” `(AMBI, SYNS)`.

5.3.3 Scalability Testing

`(CONTRA, SYNS)`

ISO/IEC and IEEE (2021c, p. 39) give “scalability testing” as a synonym of “capacity testing” while other sources differentiate between the two (Firesmith, 2015, p. 53; Bas, 2024, pp. 22–23).

`(CONTRA, DEFS)`

ISO/IEC and IEEE (2021c, p. 39) also include the external modification of the system as part of “scalability” but ISO/IEC (2023a) describe it as testing the “capability of a product to handle growing or shrinking workloads or to adapt its capacity to handle variability”, implying that this is done by the system itself.

5.3.4 Compatibility Testing

`(OVER, DEFS)`

“Compatibility testing” is defined as “testing that measures the degree to which a test item can function satisfactorily alongside other independent products in a shared environment (co-existence), and where necessary, exchanges information with other systems or components (interoperability)” (ISO/IEC and IEEE, 2022, p. 3). This definition is nonatomic as it combines the ideas of “co-existence” and “interoperability”.

`(WRONG, SYNS)`

The “interoperability” element of “compatibility testing” is explicitly excluded by ISO/IEC and IEEE (2021c, p. 37), (incorrectly) implying that “compatibility testing” and “co-existence testing” are synonyms.

`(AMBI, SYNS)`

Furthermore, the definition of “compatibility testing” in Kam (2008, p. 43) unhelpfully says “see *interoperability testing*”, adding another layer of confusion to the direction of their relationship.

`(WRONG, LABELS)`

ISO/IEC and IEEE (2022, pp. 22, 43) say “interoperability testing helps confirm that applications can work on multiple operating systems and devices”, but this seems to instead describe “portability testing”, which evaluates the “capability of a product to be adapted to changes in its requirements, contexts of use, or system environment” (ISO/IEC, 2023a; similar in ISO/IEC and IEEE, 2022, p. 7; 2017, pp. 184, 329; Hamburg and Mogyorodi, 2024), such as being “transferred from one hardware … environment to another” (ISO/IEC and IEEE, 2021c, p. 39).

Chapter 6

Threats to Validity

A case study is valid if its “results are true and not biased by the researchers’ subjective point of view” (Runeson and Höst, 2009, p. 153), which we do our best to achieve. The benefit of our work being open source means that others can make their own decisions, modify our data/code appropriately, and observe how this changes the results. However, to present and explain our findings, we make some decisions with which others may disagree, resulting in some threats to validity. We use Runeson and Höst’s (2009, pp. 153–154) definitions of these threats and list them in their respective subsections (ordered approximately by complexity) as follows:

1. **Reliability:** If another researcher were to conduct the case study later on and get the same result, then the case study is reliable. (Section 6.1)

2. **Construct Validity:** This refers to the extent to which “the operational measures that are studied really represent what the researcher[s] have in mind and what is investigated”, or how well the studied data aligns with the data that the researchers *intended* to study. (Section 6.2)

3. **Internal Validity:** A study is internally valid if the discovered causal relations are trustworthy and cannot be explained by other factors. (Section 6.3)

4. **External Validity:** For the results of a study to be externally valid, they should be generalizable and “of interest to other people outside the investigated case”. (Section 6.4)

6.1 Reliability

Natural language is nuanced and software testing has a wide scope, so we have to make judgement calls throughout the research process. Other researchers may make different decisions, which would threaten the reliability of our results. We mitigate this at a high level by outlining what we exclude from our scope in Appendix A. This means that even if other researchers decide on a different scope, the data and results within the intersection of these scopes should match. By following the methodology outlined in Chapter 3, we further mitigate the following threats to reliability:

- **Single Researcher:** Since only one researcher collected our data, it was solely up to them to determine what data and relations were important to record and investigate. To reduce the negative impact this might have on our research, we follow our methodology as rigorously as possible, including heuristics for what data is important (see Section 3.2). When ambiguity arises or more involved judgement is required, we discuss these details as a team to include more perspectives, reducing the amount of bias and oversight that would propagate throughout our research; these discussions can be found on our repo.

- **Implicit Information:** While natural language can be ambiguous, only recording explicit information would omit a lot of data provided by the software testing literature. We therefore document implicit information from the literature for completeness, as described in Section 3.3.1. While other researchers may disagree on what information "follows logically" or "is dubious", these data are innately subjective, so excluding them should result in our *explicit* data (and the results based on them) matching those of other researchers.

- **Drift Over Time:** Since we began recording data systematically in January 2024 (see the first version of our test approach glossary), we have iterated on what we consider to be in scope and what methodology to use. Some sources we investigate have even been updated (see Chapter 5)! Future researchers following our methodology would therefore end up with different results. While we could partially mitigate this by re-iterating over all sources with our "finalized" methodology, this is not possible because of time constraints. Regarding sources that were updated during our research, future researchers could replicate our results by reviewing a "snapshot" of them from when we reviewed them. However, if future researchers were to repeat this research, they *should* use the most up-to-date sources to update our existing data, methodology, and tools. This is one benefit of our research being open source: it can exist as a "living document" that can (ideally) keep up with innovations to software testing!

Q #2: Present tense? And does this make sense?

6.2 Construct Validity

For clarity and consistency throughout this thesis, we define the terminology we use in Chapter 2. The following threats to validity come from our use of ISO/IEC and IEEE's (2022) categorization scheme; while the testing literature supports it, we make the final decision to use it as described in Section 2.1.1:

- **Similar Approach Categories:** Some sources (such as Washizaki, 2025a; Barbosa et al., 2006) propose similar yet distinct categories that clash or overlap with our categories given in Table 2.1. We give these alternate categorizations, which seem to map to their "IEEE Equivalent"s, in Table 6.1. While these categories could provide new perspectives and be useful in some contexts, either in place of or in tandem with ours, their existence suggests that ISO/IEC and IEEE's (2022) categorization scheme is not universal.

- **Alternate Approach Categories:** Some of these categories can be divided further into "classes" or "families" such as the classes of combinatorial (ISO/IEC and IEEE, 2021c, p. 15) and data flow testing (p. 3) and the family of performance-related testing (Moghadam, 2019, p. 1187)[1].

  Similarly, we find many other criteria for categorizing test approaches in the literature. These have less systematic definitions but are more fine-grained, seeming to "specialize" our categories from Table 2.1. The existence of these categorizations is not inherently wrong, as they may be useful for specific teams or in certain contexts. For example, functional testing and structural testing "use different sources of information and have been shown to highlight different problems", and deterministic testing and random testing have "conditions that make one approach more effective than the other" (Washizaki, 2025a, p. 5-16). Unfortunately, even these alternate categories are not used consistently (see Mistake 7)! While these categories suggest that ours are not complete or minimal, they seem to be supplementary and rarely conflict with them.

[1] The original source describes "performance testing … as a family of performance-related testing techniques", but it makes more sense to consider "performance-related testing" as the "family" with "performance testing" being one of the variabilities (see Appendix F.4).

Table 6.1: Categories of testing given by other sources.

Term	Definition	Examples	IEEE Equivalent
Level (objective-based)[a]	Test levels based on the purpose of testing (Washizaki, 2025a, p. 5-6) that "determine how the test suite is identified … regarding its consistency … and its composition" (p. 5-2)	conformance testing, installation testing, regression testing, performance testing, security testing (Washizaki, 2025a, pp. 5-7 to 5-9)	Type
Phase	none given by Perry (2006) or Barbosa et al. (2006)	unit testing, integration testing, system testing, regression testing (Perry, 2006, p. 221; Barbosa et al., 2006, p. 3)	Level
Procedure	The basis for how testing is performed that guides the process; "categorized in [to] testing methods, testing guidances[b] and testing techniques" (Barbosa et al., 2006, p. 3)	none given generally by Barbosa et al. (2006); see "Technique"	Approach
Process	"A sequence of testing steps" (Barbosa et al., 2006, p. 2) "based on a development technology and … paradigm, as well as on a testing procedure" (p. 3)	none given by Barbosa et al. (2006)	Practice
Stage	An alternative to the "traditional … test stages" based on "clear technical groupings" (Gerrard, 2000a, p. 13)	desktop development testing, infrastructure testing, post-deployment monitoring (Gerrard, 2000a, p. 13)	Level
Technique	"Systematic procedures and approaches" based on "key aspects" such as the amount of information known about the SUT (Washizaki, 2025a, p. 5-10)	specification-based testing, structure-based testing, fault-based testing[c] (Washizaki, 2025a, pp. 5-10, 5-12, 5-14)	Technique

[a] See Contradiction 21.

[b] Testing methods and guidances are omitted from this table since Barbosa et al. (2006) do not define or give examples of them.

[c] Synonyms for these examples are used by Souza et al. (2017, p. 3; OG Mathur, 2012) and Barbosa et al. (2006, p. 3).

More threats exist because of other terms we define, such as the following:

- **Definition of Flaw:** We define a "flaw" as an instance where the software testing literature describes a testing-related term (especially a test approach) in a way that is incorrect, incomplete, inconsistent, and/or improperly coupled (see Section 2.2). When picking a word to describe one of these instances, we wanted to avoid words "overloaded with too many meanings" like "error" and "fault" (Washizaki, 2025a, p. 12-3; see Chapter 1 for more detailed discussion). A small literature review revealed that established standards (see Section 2.5.1) primarily use the term "flaw" to refer software artifacts that are *not* code: requirements (ISO/IEC and IEEE, 2022, p. 38), design (p. 43), and "system security procedures … and internal controls" (IEEE, 2012, p. 194). However, this term sometimes refers to problems with software itself (p. 92; Washizaki, 2025a, p. 7-9), which introduces some confusion (Dr. R. Paige, private communication, Oct. 14, 2025). We attempt to mitigate this by being precise with our use of the words "flaw", "error", "fault", "defect", and "failure".

- **Manifestation and Domain:** Similarly, we define the terms "manifestation" and "domain" (see Sections 2.2.1 and 2.2.2, respectively) specifically in the context of how flaws appear in the literature. However, these terms can also be used in the context of software itself: a fault is a manifestation of a human error (ISO/IEC and IEEE, 2017, p. 278) and a domain is a "distinct scope" (p. 145) or "problem space" (2010, p. 114). We likewise mitigate these threats by using these terms precisely throughout this thesis.

OG ISO/IEC, 2012

OG IEEE Std 1517-2010

- **Notion of Credibility:** We also define a metric for ranking the impact a document has on testing literature as a whole. We call this metric "credibility" and provide some properties that a credible source should have in Section 2.4. While these influence how we sort sources into tiers in Section 2.5, the format of a given source is a larger factor. Therefore, other researchers may have different ideas about what kinds of sources are more credible than others or how they will group sources to facilitate comparing them. We use credibility as a heuristic, describe each source tier thoroughly, and justify the use of our sources, mitigating (or at least minimizing the impact of) this threat.

6.3 Internal Validity

Internal threats to our research result from relations that we observe based on our constructs. The following are the most prominent of these threats:

- **Overloaded Terms:** The ambiguity of natural language means that terms are often overloaded. In Table 2.1, we describe the categories we use, but the literature may not universally use these terms in the same way. For example, Kam (2008, p. 45, emphasis added) defines interface testing as "an integration *test type* that is concerned with testing … interfaces", but since he does not define "test type", this may not have special significance.

  Additionally, Firesmith (2015, p. 23) uses the same acronym ("HIL") for "hardware-in-the-loop testing" and "human-in-the-loop testing". We track this as a flaw (see Overlap 7), but these terms "might be disambiguated in practice as they often come at very different stages/phases of testing" (Dr. R. Paige, private communication, Oct. 14, 2025).

- **Qualities Implying Test Types:** Since test types are "focused on specific quality characteristics" (ISO/IEC and IEEE, 2022, p. 15; 2021c, p. 7; 2017, p. 473; similar in Koomen et al., 2006, p. 50), we posit that they can be derived from software qualities: "capabilit[ies] of software product[s] to satisfy stated and implied needs when used under specified conditions" (ISO/IEC, 2014b, p. 6). We track 75 software qualities[2] following our procedure in Section 3.3, "upgrading" them to test types when a source mentions (or implies) them. Examples of this include conformance testing (Washizaki, 2025a, p. 5-7; Jard et al., 1999, p. 25; implied by ISO/IEC and IEEE, 2017, p. 93), efficiency testing (Kam, 2008, p. 44), and survivability testing (Ghosh and Voas, 1999, p. 40). While other researchers may disagree with this relation between software qualities and test types, the literature seems to support it as described above.

- **Coverage Metrics Implying Test Techniques:** Test techniques can "identify test coverage items … and derive corresponding test cases" (ISO/IEC and IEEE, 2022, p. 11; 2021a, p. 5; similar in 2017, p. 467), which allows for "the coverage achieved by a specific test design technique" to be calculated as a percentage (2021c, p. 30). Therefore, we posit that a given coverage metric implies a test technique with the goal of maximizing it. For example, path testing "aims to execute all entry-to-exit control flow paths in a SUT's control flow graph" (Washizaki, 2025a, p. 5-13), thus maximizing the path coverage. Again, while other researchers may disagree with this relation between coverage metrics and test techniques, the literature seems to support it as described above and by Doğan et al. (2014, pp. 183–185), Sharma et al. (2021, Fig. 1), and Reid (1996, pp. 2–3).

[2] Discussed in #21, #23, and #27.

6.4 External Validity

While we document issues with "standardized" software testing terminology so that they can be addressed in the future, some dismiss the importance of standardized terminology for various reasons, including the following:

- **Limitations of Standardized Terminology:** Schoots (2014) holds that "common terminology is dangerous" and "to be able to truly understand each other, we need to ask questions and discuss in depth". However, these in-depth discussions are *not* mutually exclusive with common terminology! Having a shared understanding of how terms are defined allows for common ground during these sorts of discussions with the chance to adapt them to serve the context of a given team or project (which we give examples of in Sections 2.1.2 and 6.2).

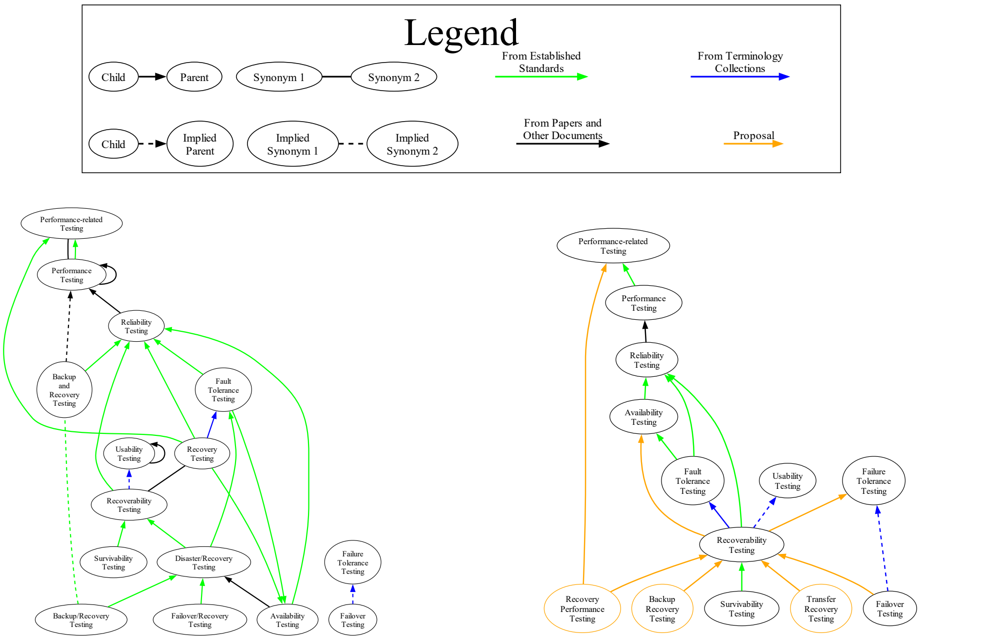
- **Standards Being Mandated:** Schoots (2014) also states that while he wishes "standards would be guidelines, … reality shows standards become mandatory often". He supports this with examples from Soundararajan (2015) where "contracts and bids from large companies" often "reference[] ISO linking to industry best practices". This claim, however, overlooks:

  1. the possibility of renegotiating contracts and
  2. the notion of "tailored conformance" to standards, which is mentioned throughout the family of standards these authors critique (ISO/IEC and IEEE, 2021a, pp. 9-10; 2021b, pp. 5, 17, 37; 2021c, p. 7) and was perhaps introduced later to address concerns such as these.

Chapter 7

Future Work

With more time, we would continue iterating over undefined terms (Section 7.1) and investigate terms we *expected* to find but never did (Section 7.2). We could also look for other approach data that never arose from our methodology (Section 7.3) and further analyze our collected data by improving our visualizations (Section 7.4) and detecting (and identifying!) more classes of flaws (Section 7.5). "Can we systematically resolve any of these inconsistencies?" was one of our original Research Questions (RQs). We start this process in Appendix F and give an excerpt in Figure 7.1, but doing this more systematically once the previous tasks are finished would be more effective.



(a) Visualization of current relations.

(b) Visualization of proposed relations.

Figure 7.1: Visualizations of relations in the subset of recovery testing after applying our recommendations.

7.1 Iterating Over Undefined Approaches

As we explain in Section 3.4, our methodology includes performing miniature literature reviews on undefined test approaches to record their missing definitions (and any relations). We were able to do this for the following approaches, although some are out of scope, such as EManations SECurity (EMSEC) testing, aspects of Orthogonal Array Testing (OAT) and loop testing (see Appendix A.1), and HTML testing (see Appendix A.5). We investigate the following terms (and their respective related terms) in the sources given:

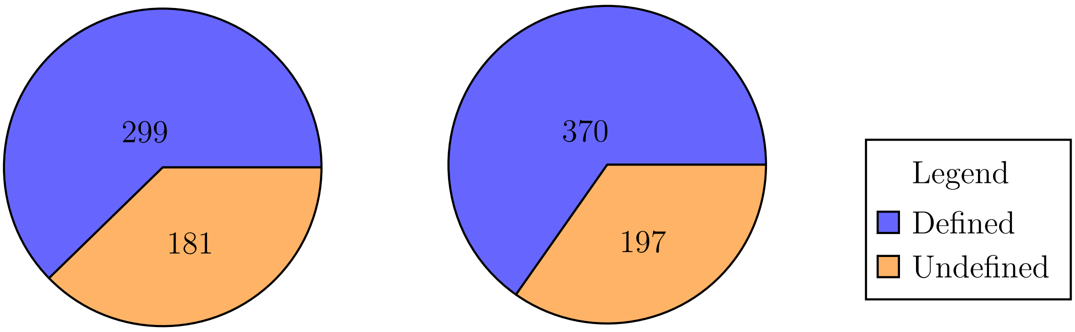
- **Assertion Checking:** Lahiri et al. (2013); Chalin et al. (2006); Berdine et al. (2006)
- **Loop Testing[1]:** Dhok and Ramanathan (2016); Godefroid and Luchaup (2011); Preuße et al. (2012); Forsyth et al. (2004)
- **EMSEC Testing:** Zhou et al. (2012); ISO (2021)
- **Asynchronous Testing:** Jard et al. (1999)
- **Performance(-related) Testing:** Moghadam (2019)
- **Web Application Testing:** Doğan et al. (2014); Kam (2008)
  - **HTML Testing:** Choudhary et al. (2010); Sneed and Göschl (2000); Gerrard (2000b)
  - **Document Object Model (DOM) Testing:** Bajammal and Mesbah (2018)
- **Sandwich Testing:** Sharma et al. (2021); Sangwan and LaPlante (2006)
- **Orthogonal Array Testing[2]:** Mandl (1985); Valcheva (2013)
- **Backup Testing[3]:** Bas (2024)
- **System of Systems (SoS) (Integration) Testing:** ISO/IEC and IEEE (2019b)

[1] We use ISO (2022; 2015) as references for terms but do not fully investigate them and add Pierre et al. (2017); Trudnowski et al. (2017) as potentially in-scope and Dominguez-Pumar et al. (2020); Goralski (1999) as out-of-scope examples.

[2] We add Yu et al. (2011); Tsui (2007) as out-of-scope examples.

[3] See Section 5.3.2.

Applying our procedure shown in Figure 3.2 to these sources uncovers 87 new approaches and 71 new definitions. These definitions are either for existing undefined approaches or new uncovered approaches; while not every new approach is presented alongside a definition, if we assume that each of these definitions is for a new approach, we can deduce that about 82% of added test approaches are defined. This higher proportion of defined terms indicates that our procedure leads to a (62% vs. 65%), shown in Figure 7.2, which helps verify that our procedure constructively uncovers *and* defines new terminology. With repeated iterations, this ratio would approach 100%, resulting in a (plausibly) complete taxonomy. We give the full list of undefined test approaches in Appendix E.1 for completeness.



(a) The 480 approaches before investigating undefined terms.

(b) The 567 approaches after investigating undefined terms.

Figure 7.2: Breakdown of how many test approaches are undefined.

7.2 Investigating Missing Test Approaches

In addition to these undefined approaches, there are some that we do not uncover at all! We each have preexisting knowledge of what test approaches exist (a form of experience-based testing, if you will) but not all of these arise as a result of our snowballing approach. These approaches may serve as starting points for continued research if we do not find them in the literature using our iterative approach. The following terms come from previous knowledge, conversations with colleagues, research for other projects, or ad hoc cursory research to see what other test approaches exist:

1. Chaos engineering
2. Chosen-ciphertext attacks
3. Concolic testing
4. Concurrent testing[a]
5. Context-driven testing
6. Destructive testing
7. Dogfooding
8. Implementation-based testing[b]
9. Interaction-based testing
10. Lunchtime attacks[c]
11. Parallel testing
12. Property-based testing
13. Pseudo-random bit testing
14. Rubber duck testing
15. Sanity testing
16. Scream testing
17. Shadow testing
18. Situational testing

[a] This seems to be distinct from "concurrency testing".

[b] This may or may not be distinct from "implementation-oriented testing."

[c] In previous meetings, Dr. Smith mentioned that with the number of test approaches that suggest that people just like to label everything as "testing", he would not be surprised if something like "Monday morning testing" existed. While independently researching chosen-ciphertext attacks out of curiosity, this prediction of a time-based test approach came true with "lunchtime attacks".

7.3 Filling in Other Approach Data

Definitions are not the only piece of information that can be missing for some test approaches. As mentioned in Section 4.1, some test approaches are "orphans" without a parent-child relation *or* a significant synonym relation. To reduce clutter, we omit these from our visualizations, but iterating over these orphan approaches to find any relations in the literature could provide a more complete picture of the state of software testing. We give the full list of these orphans in Appendix E.2 for completeness.

Additionally, some test approaches are not given one of the categories we use in Table 2.1. Following our methodology, we assign each category the "default" category "Approach" until we uncover a more specific one as shown in Figure 3.2. Assuming these categories are complete (which they may not be; see Section 6.2), we could either decide on an appropriate category for each uncategorized approach or further investigate these terms to see if they are categorized by other sources. We give the full list of test approaches with the general category of "Approach" in Appendix E.3 for completeness.

7.4 Improving Relation Visualizations

As described in Section 4.1, we currently visualize the relations between test approaches by identifying relations that are notable to include. We then add a line for each relation to each relevant LaTeX `digraph` to render them. While sufficient for the purposes of this research, this was mainly done as a proof of concept[4] and could be done more robustly and efficiently. The use of a more elaborate tool could make these visualizations interactive, making these dense overviews more usable and accessible.

7.5 Detecting More Flaws

In addition to the classes of flaws we *do* detect automatically, we could detect many more if time permitted. We currently detect parent-child relations that violate irreflexivity (see Section 4.2.1) which can be thought of as cycles with length $n = 1$. Since parent-child relations should also be transitive (see Section 2.1.3), cycles of *any* size are flaws. Given the current way we generate visualizations of these relations (see Section 4.1), detecting cycles where $n = 2$ would be straightforward: if a parent-child relation *and* its inverse (i.e., `A -> B` and `B -> A` for test approaches with labels `A` and `B`) both exist in the generated LaTeX file for a visualization, (at least) one of these parent-child flaws is incorrect since they contradict each other and we have found a cycle. The main reason this would be time-consuming would be deciding on how to format these findings, writing code to do so automatically for use in this thesis, and resolving any issues that arise during this process. Detecting larger cycles would also be possible but would likely require the use of an additional tool to analyze the graph of parent-child relations; this would likely be done alongside improving these visualizations in general (see Section 7.4).

Unfortunately, writing code to detect flaws is only half the battle. First, we need to identify classes of flaws we even want to detect, which may never be complete as new test approaches emerge with potentially new relations. The classes of flaws we detect emerged over time based on our observations of the literature. For example, our understanding of the “standard” test approach categories described in Section 2.1.1 led to us being able to detect approaches that are categorized inconsistently in Section 5.2.1 and Appendix D.2.1. Performing more thorough analysis, both on the literature and on our collected data, will likely reveal more classes of flaws that can be tracked automatically but is infeasible for the purposes of this thesis.

[4] Discussed in #67.

Chapter 8

Conclusion

While a good starting point, the software testing literature contains many flaws that create unnecessary barriers to testing software; standardized conventions give software testers common ground to use or adapt for their needs. This thesis exposes how unstandardized software testing terminology is by outlining the flaws we find in the literature.

We do this by documenting the 567 test approaches mentioned by 85 sources (described in Section 2.5), recording their names, categories, definitions, synonyms, parents, and flaws (see Section 3.3.2) as applicable. In addition to these manually recorded flaws, we identify classes of flaws we can detect automatically in Section 2.2.2. These include:

- 29 test approaches with more than one category,
- 13 terms used as synonyms to two (or more) disjoint test approaches,
- three test approaches that are parents of themselves, and
- 17 pairs of test approaches that may either be synonyms or have a parent-child relationship.

We also identify synonyms between independently defined approaches (see Figure 5.4 for a visualization of those that are explicit) but only count these as flaws when appropriate, since this type of synonym relation may just mean that the terms are interchangeable and the relation is trivially reflexive. In total, we found 341 flaws in the software testing literature, from which we observe the following:

1. Contradictions are by *far* the most common manifestation. This is likely because these are the most obvious flaws to detect automatically (see Sections 4.2.1 and 5.2) and because two (sets of) authors using different resources and not communicating have a high chance of disagreeing.

2. Approach categorizations are the most subjective and one of the most common flaw domains, likely due to the lack of standardization about what categories to use (see Section 6.2 for more detailed discussion).

3. In general, semantic flaws are more common than syntactic ones. The number of category flaws is comparable to the numbers of flaws with the relations and definitions of test approaches.

Our work provides a solid baseline for addressing flaws such as these in the software testing literature: a centralized location for recording data that the software community can update as standards change or as new test approaches emerge, along with tools for analyzing these data and checking for flaws. However, there is more work to be done in this area, such as addressing the threats to validity described in Chapter 6 and continuing our research as described in Chapter 7.

Bibliography

Mominul Ahsan, Stoyan Stoyanov, Chris Bailey, and Alhussein Albarbar. Developing Computational Intelligence for Smart Qualification Testing of Electronic Products. *IEEE Access*, 8:16922–16933, January 2020. ISSN 2169-3536. doi: 10.1109/ACCESS.2020.2967858. URL https://www.webofscience.com/api/gateway?GWVersion=2&SrcAuth=DynamicDOIArticle&SrcApp=WOS&KeyAID=10.1109%2FACCESS.2020.2967858&DestApp=DOI&SrcAppSID=USW2EC0CB9ABcVz5BcZ70BCfIlmtJ&SrcJTitle=IEEE+ACCESS&DestDOIRegistrantName=Institute+of+Electrical+and+Electronics+Engineers. Place: Piscataway.

Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, Cambridge, United Kingdom, 2nd edition, 2017. ISBN 978-1-107-17201-2. URL https://eopcw.com/find/downloadFiles/11.

Mohammad Bajammal and Ali Mesbah. Web Canvas Testing Through Visual Inference. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 193–203, Västerås, Sweden, 2018. IEEE. ISBN 978-1-5386-5012-7. doi: 10.1109/ICST.2018.00028. URL https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8367048.

Ellen Francine Barbosa, Elisa Yumi Nakagawa, and José Carlos Maldonado. Towards the Establishment of an Ontology of Software Testing. volume 6, pages 522–525, San Francisco, CA, USA, January 2006.

Luciano Baresi and Mauro Pezzè. An Introduction to Software Testing. *Electronic Notes in Theoretical Computer Science*, 148(1):89–111, February 2006. ISSN 1571-0661. doi: 10.1016/j.entcs.2005.12.014. URL https://www.sciencedirect.com/science/article/pii/S1571066106000442.

Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015. doi: 10.1109/TSE.2014.2372785.

Mykola Bas. *Data Backup and Archiving*. Bachelor thesis, Czech University of Life Sciences Prague, Praha-Suchdol, Czechia, March 2024. URL https://theses.cz/id/60licg/zaverecna_prace_Archive.pdf.

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, pages 115–137, Berlin, Heidelberg, 2006. Springer. ISBN 978-3-540-36750-5. doi: 10.1007/11804192_6.

Antonia Bertolino, Guglielmo De Angelis, Micael Gallego, Boni García, Francisco Gortázar, Francesca Lonetti, and Eda Marchetti. A Systematic Review on Cloud Testing. *ACM Computing Surveys*, 52(5), September 2019. ISSN 0360-0300. doi: 10.1145/3331447. URL https://doi.org/10.1145/3331447. Place: New York, NY, USA Publisher: Association for Computing Machinery.

Rex Black. *Advanced Software Testing*, volume 1. Rocky Nook Inc., Santa Barbara, CA, USA, 1st edition, December 2009. ISBN 1-933952-19-9. URL https://tisten.ir/blog/wp-content/uploads/2019/03/Advanced-Software-Testing-Vol-1Test-Analyst.pdf.

Michael Bluejay. Slot Machine PAR Sheets, May 2024. URL https://easy.vegas/games/slots/par-sheets.

Chris Bocchino and William Hamilton. Eastern Range Titan IV/Centaur-TDRSS Operational Compatibility Testing. In *International Telemetering Conference Proceedings*, San Diego, CA, USA, October 1996. International Foundation for Telemetering. ISBN 978-0-608-04247-3. URL https://repository.arizona.edu/bitstream/handle/10150/607608/ITC_1996_96-01-4.pdf?sequence=1&isAllowed=y.

Pierre Bourque and Richard E. Fairley, editors. *Guide to the Software Engineering Body of Knowledge, Version 3.0*. IEEE Computer Society Press, Washington, DC, USA, 2014. ISBN 0-7695-5166-1. URL www.swebok.org.

Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, pages 342–363, Berlin, Heidelberg, 2006. Springer. ISBN 978-3-540-36750-5. doi: 10.1007/11804192_16.

ChatGPT (GPT-4o). Defect Clustering Testing, November 2024. URL https://chatgpt.com/share/67463dd1-d0a8-8012-937b-4a3db0824dcf.

Shauvik Roy Choudhary, Husayn Versee, and Alessandro Orso. A Cross-browser Web Application Testing Tool. In *2010 IEEE International Conference on Software Maintenance*, pages 1–6, Timisoara, Romania, September 2010. IEEE. ISBN 978-1-4244-8629-8. doi: 10.1109/ICSM.2010.5609728. URL https://ieeexplore.ieee.org/abstract/document/5609728. ISSN: 1063-6773.

Alan Dennis, Barbara Haley Wixom, and Roberta M. Roth. *System Analysis and Design*. John Wiley & Sons, 5th edition, 2012. ISBN 978-1-118-05762-9. URL https://www.uoitc.edu.iq/images/documents/informatics-institute/Competitive_exam/Systemanalysisanddesign.pdf.

Monika Dhok and Murali Krishna Ramanathan. Directed Test Generation to Detect Loop Inefficiencies. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 895–907, New York, NY, USA, November 2016. Association for Computing Machinery. ISBN 978-1-4503-4218-6. doi: 10.1145/2950290.2950360. URL https://dl.acm.org/doi/10.1145/2950290.2950360.

M. Dominguez-Pumar, J. M. Olm, L. Kowalski, and V. Jimenez. Open loop testing for optimizing the closed loop operation of chemical systems. *Computers & Chemical Engineering*, 135:106737, 2020. ISSN 0098-1354. doi: https://doi.org/10.1016/j.compchemeng.2020.106737. URL https://www.sciencedirect.com/science/article/pii/S0098135419312736.

Serdar Doğan, Aysu Betin-Can, and Vahid Garousi. Web application testing: A systematic literature review. *Journal of Systems and Software*, 91:174–201, 2014. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2014.01.010. URL https://www.sciencedirect.com/science/article/pii/S0164121214000223.

Emelie Engström and Kai Petersen. Mapping software testing practice with software testing research — serp-test taxonomy. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–4, 2015. doi: 10.1109/ICSTW.2015.7107470.

Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous & Practical Approach*. PWS Publishing Company, Boston, MA, USA, 2nd edition, 1997. ISBN 0-534-95425-1.

Donald G. Firesmith. A Taxonomy of Testing Types, 2015. URL https://apps.dtic.mil/sti/pdfs/AD1147163.pdf.

P. Forsyth, T. Maguire, and R. Kuffel. Real Time Digital Simulation for Control and Protection System Testing. In *2004 IEEE 35th Annual Power Electronics Specialists Conference (IEEE Cat. No.04CH37551)*, volume 1, pages 329–335, Aachen, Germany, 2004. IEEE. ISBN 0-7803-8399-0. doi: 10.1109/PESC.2004.1355765.

Paul Gerrard. Risk-based E-business Testing - Part 1: Risks and Test Strategy. Technical report, Systeme Evolutif, London, UK, 2000a. URL https://www.agileconnection.com/sites/default/files/article/file/2013/XUS129342file1_0.pdf.

Paul Gerrard. Risk-based E-business Testing - Part 2: Test Techniques and Tools. Technical report, Systeme Evolutif, London, UK, 2000b. URL wenku.uml.com.cn/document/test/EBTestingPart2.pdf.

Paul Gerrard and Neil Thompson. *Risk-based E-business Testing*. Artech House computing library. Artech House, Norwood, MA, USA, 2002. ISBN 978-1-58053-570-0. URL https://books.google.ca/books?id=54UKereAdJ4C.

Anup K Ghosh and Jeffrey M Voas. Inoculating software for survivability. *Communications of the ACM*, 42(7):38–44, July 1999. URL https://dl.acm.org/doi/pdf/10.1145/306549.306563. Publisher: ACM New York, NY, USA.

Patrice Godefroid and Daniel Luchaup. Automatic Partial Loop Summarization in Dynamic Test Generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 23–33, New York, NY, USA, July 2011. Association for Computing Machinery. ISBN 978-1-4503-0562-4. doi: 10.1145/2001420.2001424. URL https://dl.acm.org/doi/10.1145/2001420.2001424.

W. Goralski. xDSL loop qualification and testing. *IEEE Communications Magazine*, 37(5):79–83, 1999. doi: 10.1109/35.762860.

Matthias Hamburg and Gary Mogyorodi, editors. ISTQB Glossary, v4.3, 2024. URL https://glossary.istqb.org/en_US/search.

Bill Hetzel. *The Complete Guide to Software Testing*. QED Information Sciences, Inc., Wellesley, MA, USA, 2nd edition, 1988. ISBN 0-89435-242-3. URL https://archive.org/details/completeguidetos00hetz/mode/2up.

Daniel C Holley, Gary D Mele, and Sujata Naidu. NASA Rat Acoustic Tolerance Test 1994-1995: 8 kHz, 16 kHz, 32 kHz Experiments. Technical Report NASA-CR-202117, San Jose State University, San Jose, CA, USA, January 1996. URL https://ntrs.nasa.gov/api/citations/19960047530/downloads/19960047530.pdf.

R. Brian Howe and Robert Johnson. Research Protocol for the Evaluation of Medical Waiver Requirements for the Use of Lisinopril in USAF Aircrew. Interim Technical Report AL/AO-TR-1995-0116, Air Force Materiel Command, Brooks Air Force Base, TX, USA, November 1995. URL https://apps.dtic.mil/sti/tr/pdf/ADA303379.pdf.

IEEE. IEEE Standard for System and Software Verification and Validation. *IEEE Std 1012-2012 (Revision of IEEE Std 1012-2004)*, 2012. doi: 10.1109/IEEESTD.2012.6204026.

IEEE. IEEE Standard for Measures of the Software Aspects of Dependability. *IEEE Std 982-2024 (Revision of IEEE Std 982-2005)*, November 2024. doi: 10.1109/IEEESTD.2024.10745903.

IEEE Computer Society. IEEE Standard Classification for Software Anomalies. *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)*, January 2010. doi: 10.1109/IEEESTD.2010.5399061.

Adisak Intana, Monchanok Thongthep, Phatcharee Thepnimit, Phaplak Saethapan, and Tanawat Monpipat. SYNTest: Prototype of Syntax Test Case Generation Tool. In *5th International Conference on Information Technology (InCIT)*, pages 259–264. IEEE, 2020. ISBN 978-1-72819-321-2. doi: 10.1109/InCIT50588.2020.9310968.

ISO. ISO 13849-1:2015 - Safety of machinery –Safety-related parts of control systems –Part 1: General principles for design. *ISO 13849-1:2015*, December 2015. URL https://www.iso.org/obp/ui#iso:std:iso:13849:-1:ed-3:v1:en.

ISO. ISO 21384-2:2021 - Unmanned aircraft systems –Part 2: UAS components. *ISO 21384-2:2021*, December 2021. URL https://www.iso.org/obp/ui#iso:std:iso:21384:-2:ed-1:v1:en.

ISO. ISO 28881:2022 - Machine tools –Safety –Electrical discharge machines. *ISO 28881:2022*, April 2022. URL https://www.iso.org/obp/ui#iso:std:iso:28881:ed-2:v1:en.

ISO/IEC. ISO/IEC 25000:2005 - Software Engineering –Software product Quality Requirements and Evaluation (SQuaRE) –Guide to SQuaRE. *ISO/IEC 25000:2005*, August 2005. URL https://www.iso.org/obp/ui/#iso:std:iso-iec:25000:ed-1:v1:en.

ISO/IEC. ISO/IEC 25010:2011 - Systems and software engineering –Systems and software Quality Requirements and Evaluation (SQuaRE) –System and software quality models. *ISO/IEC 25010:2011*, March 2011. URL https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en.

ISO/IEC. ISO/IEC 25051:2014 - Software engineering –Systems and software Quality Requirements and Evaluation (SQuaRE) –Requirements for quality of Ready to Use Software Product (RUSP) and instructions for testing. *ISO/IEC 25051:2014*, February 2014a. URL https://www.iso.org/obp/ui/#iso:std:iso-iec:25051:ed-2:v1:en.

ISO/IEC. ISO/IEC 25000:2014 - Systems and software engineering –Systems and software Quality Requirements and Evaluation (SQuaRE) –Guide to SQuaRE. *ISO/IEC 25000:2014*, March 2014b. URL https://www.iso.org/obp/ui/#iso:std:iso-iec:25000:ed-2:v1:en.

ISO/IEC. ISO/IEC 90003:2014 - Software engineering –Guidelines for the application of ISO 9001:2008 to computer software. *ISO/IEC 90003:2014*, December 2014c. URL https://www.iso.org/obp/ui/#iso:std:iso-iec:90003:ed-2:v1:en.

ISO/IEC. ISO/IEC 2382:2015 - Information technology –Vocabulary. *ISO/IEC 2382:2015*, May 2015. URL https://www.iso.org/obp/ui/en/#iso:std:iso-iec:2382:ed-1:v2:en.

ISO/IEC. ISO/IEC TS 20540:2018 - Information technology – Security techniques –Testing cryptographic modules in their operational environment. *ISO/IEC TS 20540:2018*, May 2018. URL https://www.iso.org/obp/ui#iso:std:iso-iec:ts:20540:ed-1:v1:en.

ISO/IEC. ISO/IEC TR 29119-11:2020 - Software and systems engineering –Software testing –Part 11: Guidelines on the testing of AI-based systems. *ISO/IEC TR 29119-11:2020*, November 2020. URL https://www.iso.org/obp/ui/en/#iso:std:iso-iec:tr:29119:-11:ed-1:v1:en.

ISO/IEC. ISO/IEC 25010:2023 - Systems and software engineering –Systems and software Quality Requirements and Evaluation (SQuaRE) –Product quality model. *ISO/IEC 25010:2023*, November 2023a. URL https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-2:v1:en.

ISO/IEC. ISO/IEC 25019:2023 - Systems and software engineering –Systems and software Quality Requirements and Evaluation (SQuaRE) –Quality-in-use model. *ISO/IEC 25019:2023*, November 2023b. URL https://www.iso.org/obp/ui/en/#iso:std:iso-iec:25019:ed-1:v1:en.

ISO/IEC and IEEE. ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary. *ISO/IEC/IEEE 24765:2010(E)*, December 2010. doi: 10.1109/IEEESTD.2010.5733835.

ISO/IEC and IEEE. ISO/IEC/IEEE International Standard - Systems and software engineering –Software testing –Part 1: General concepts. *ISO/IEC/IEEE 29119-1:2013*, September 2013. doi: 10.1109/IEEESTD.2013.6588537.

ISO/IEC and IEEE. ISO/IEC/IEEE International Standard - Systems and software engineering –System life cycle processes. *ISO/IEC/IEEE 15288:2015*, May 2015. doi: 10.1109/IEEESTD.2015.7106435.

ISO/IEC and IEEE. ISO/IEC/IEEE International Standard - Software and systems engineering –Software testing –Part 5: Keyword-Driven Testing. *ISO/IEC/IEEE 29119-5:2016*, November 2016. doi: 10.1109/IEEESTD.2016.7750539.

ISO/IEC and IEEE. ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary. *ISO/IEC/IEEE 24765:2017(E)*, September 2017. doi: 10.1109/IEEESTD.2017.8016712.

ISO/IEC and IEEE. ISO/IEC/IEEE International Standard - Systems and software engineering –Systems and software assurance –Part 1: Concepts and vocabulary. *ISO/IEC/IEEE 15026-1:2019*, March 2019a. doi: 10.1109/IEEESTD.2019.8657410.

ISO/IEC and IEEE. ISO/IEC/IEEE International Standard - Systems and software engineering –Guidelines for the utilization of ISO/IEC/IEEE 15288 in the context of system of systems (SoS). *ISO/IEC/IEEE 21840:2019*, December 2019b. doi: 10.1109/IEEESTD.2019.8929110.

ISO/IEC and IEEE. ISO/IEC/IEEE International Standard - Software and systems engineering –Software testing –Part 2: Test processes. *ISO/IEC/IEEE 29119-2:2021(E)*, October 2021a. doi: 10.1109/IEEESTD.2021.9591508.

ISO/IEC and IEEE. ISO/IEC/IEEE International Standard - Software and systems engineering –Software testing –Part 3: Test documentation. *ISO/IEC/IEEE 29119-3:2021(E)*, October 2021b. doi: 10.1109/IEEESTD.2021.9591577.

ISO/IEC and IEEE. ISO/IEC/IEEE International Standard - Software and systems engineering –Software testing –Part 4: Test techniques. *ISO/IEC/IEEE 29119-4:2021(E)*, October 2021c. doi: 10.1109/IEEESTD.2021.9591574.

ISO/IEC and IEEE. ISO/IEC/IEEE International Standard - Systems and software engineering –Software testing –Part 1: General concepts. *ISO/IEC/IEEE 29119-1:2022(E)*, January 2022. doi: 10.1109/IEEESTD.2022.9698145.

Claude Jard, Thierry Jéron, Lénaïck Tanguy, and César Viho. Remote testing can be as powerful as local testing. In Jianping Wu, Samuel T. Chanson, and Qiang Gao, editors, *Formal Methods for Protocol Engineering and Distributed Systems: Forte XII / PSTV XIX'99*, volume 28 of *IFIP Advances in Information and Communication Technology*, pages 25–40, Beijing, China, October 1999. Springer. ISBN 978-0-387-35578-8. doi: 10.1007/978-0-387-35578-8_2. URL https://doi.org/10.1007/978-0-387-35578-8_2.

Timothy P. Johnson. Snowball Sampling: Introduction. In *Wiley StatsRef: Statistics Reference Online*. John Wiley & Sons, Ltd, 2014. ISBN 978-1-118-44511-2. doi: https://doi.org/10.1002/9781118445112.stat05720. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118445112.stat05720. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781118445112.stat05720.

Ben Kam. Web Applications Testing. Technical Report 2008-550, Queen's University, Kingston, ON, Canada, October 2008. URL https://research.cs.queensu.ca/TechReports/Reports/2008-550.pdf.

Cem Kaner, James Bach, and Bret Pettichord. *Lessons Learned in Software Testing: A Context-Driven Approach*. John Wiley & Sons, December 2011. ISBN 978-0-471-08112-8. URL https://www.wiley.com/en-ca/Lessons+Learned+in+Software+Testing%3A+A+Context-Driven+Approach-p-9780471081128.

Upulee Kanewala and Tsong Yueh Chen. Metamorphic testing: A simple yet effective approach for testing scientific software. *Computing in Science & Engineering*, 21(1):66–72, 2019. doi: 10.1109/MCSE.2018.2875368.

Knüvener Mackert GmbH. *Knüvener Mackert SPICE Guide*. Knüvener Mackert GmbH, Reutlingen, Germany, 7th edition, 2022. ISBN 978-3-00-061926-7. URL https://knuevenermackert.com/wp-content/uploads/2021/06/SPICE-BOOKLET-2022-05.pdf.

Tim Koomen, Leo van der Aalst, Bart Broekman, and Michiel Vroon. *TMap Next for result-driven testing*. UTN Publishers, The Netherlands, 2006. ISBN 90-72194-80-2. URL https://www.scribd.com/document/79393769/TMap-NEXT-Look-Inside-Version.

Ivans Kuļešovs, Vineta Arnicane, Guntis Arnicans, and Juris Borzovs. Inventory of Testing Ideas and Structuring of Testing Terms. 1:210–227, January 2013.

Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, and Chris Hawblitzel. Differential Assertion Checking. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 345–355, New York, NY, USA, August 2013. Association for Computing Machinery. ISBN 978-1-4503-2237-9. doi: 10.1145/2491411.2491452. URL https://dl.acm.org/doi/10.1145/2491411.2491452.

LambdaTest. What is Operational Testing: Quick Guide With Examples, 2024. URL https://www.lambdatest.com/learning-hub/operational-testing.

Danye Liu, Shaonan Tian, Yu Zhang, Chaoquan Hu, Hui Liu, Dong Chen, Lin Xu, and Jun Yang. Ultrafine SnPd nanoalloys promise high-efficiency electrocatalysis for ethanol oxidation and oxygen reduction. *ACS Applied Energy Materials*, 6 (3):1459–1466, January 2023. doi: https://doi.org/10.1021/acsaem.2c03355. URL https://pubs.acs.org/doi/pdf/10.1021/acsaem.2c03355?casa_token=ItHfKxeQNbsAAAAA:8zEdU5hi2HfHsSony3ku-lbH902jkHpA-JZw8jIeODzUvFtSdQRdbYhmVq47aX22igR52o2S22mnC88Mxw. Publisher: ACS Publications.

Michael R. Lyu, editor. *Handbook of Software Reliability Engineering*. McGraw-Hill and IEEE Computer Society Press, New York, NY, USA, April 1996. ISBN 0-07-039400-8. URL https://www.cse.cuhk.edu.hk/~lyu/book/reliability/.

Robert Mandl. Orthogonal Latin squares: an application of experiment design to compiler testing. *Communications of the ACM*, 28(10):1054–1058, October 1985. ISSN 0001-0782. doi: 10.1145/4372.4375. URL https://doi.org/10.1145/4372.4375.

Robert M. McClure. Introduction, July 2001. URL http://homepages.cs.ncl.ac.uk/brian.randell/NATO/Introduction.html.

Ali Mesbah and Arie van Deursen. Invariant-based automatic testing of AJAX user interfaces. In *2009 31st International Conference on Software Engineering*, pages 210–220, Vancouver, BC, Canada, 2009. IEEE. ISBN 978-1-4244-3452-7. doi: 10.1109/ICSE.2009.5070522. URL https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5070522.

Mahshid Helali Moghadam. Machine Learning-Assisted Performance Testing. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 1187–1189, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 978-1-4503-5572-8. doi: 10.1145/3338906.3342484. URL https://doi.org/10.1145/3338906.3342484.

V. V. Morgun, L. I. Voronin, R. R. Kaspransky, S. L. Pool, M. R. Barratt, and O. L. Novinkov. The Russian-US Experience with Development Joint Medical Support Procedures for Before and After Long-Duration Space Flights. Technical report, NASA, Houston, TX, USA, 1999. URL https://ntrs.nasa.gov/api/citations/20000085877/downloads/20000085877.pdf.

E. E. Mukhin, V. M. Nelyubov, V. A. Yukish, E. P. Smirnova, V. A. Solovei, N. K. Kalinina, V. G. Nagaitsev, M. F. Valishin, A. R. Belozerova, S. A. Enin, A. A. Borisov, N. A. Deryabina, V. I. Khripunov, D. V. Portnov, N. A. Babinov, D. V. Dokhtarenko, I. A. Khodunov, V. N. Klimov, A. G. Razdobarin, S. E. Alexandrov, D. I. Elets, A. N. Bazhenov, I. M. Bukreev, An P. Chernakov, A. M. Dmitriev, Y. G. Ibragimova, A. N. Koval, G. S. Kurskiev, A. E. Litvinov, K. O. Nikolaenko, D. S. Samsonov, V. A. Senichenkov, R. S. Smirnov, S. Yu Tolstyakov, I. B. Tereschenko, L. A. Varshavchik, N. S. Zhiltsov, A. N. Mokeev, P. V. Chernakov, P. Andrew, and M. Kempenaars. Radiation tolerance testing of piezoelectric motors for ITER (first results). *Fusion Engineering and Design*, 176(article 113017), 2022. ISSN 0920-3796. doi: https://doi.org/10.1016/j.fusengdes.2022.113017. URL https://www.sciencedirect.com/science/article/pii/S0920379622000175.

John D. Musa, Anthony Iannino, and Kazuhira Okumoto. *Software Reliability: Measurement, Prediction, Application*. McGraw-Hill, Inc., New York, NY, USA, March 1987. ISBN 0-07-044093-X.

Glenford J. Myers. *Software Reliability: Principles and Practices*, volume 7 of *Business Data Processing: A Wiley Series*. Wiley, New York, NY, USA, 1976. ISBN 978-0-471-62765-4. URL https://books.google.ca/books?id=DXoyAAAAIAAJ.

J. Paul Myers. The complexity of software testing. *Software Engineering Journal*, 7(1):13–24, January 1992. doi: 10.1049/sej.1992.0002. URL https://digitalcommons.trinity.edu/cgi/viewcontent.cgi?article=1011&context=compsci_faculty. Publisher: The Institution of Engineering and Technology.

Peter Naur and Brian, editors Randell. Software Engineering: Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968. Brussels, Belgium, January 1969. Scientific Affairs Division, NATO. URL http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF.

Pranav Pandey. Scalability vs Elasticity, February 2023. URL https://www.linkedin.com/pulse/scalability-vs-elasticity-pranav-pandey/.

Bhupesh A. Parate, K.D. Deodhar, and V.K. Dixit. Qualification Testing, Evaluation and Test Methods of Gas Generator for IEDs Applications. *Defence Science Journal*, 71(4):462–469, July 2021. doi: 10.14429/dsj.71.16601. URL https://publications.drdo.gov.in/ojs/index.php/dsj/article/view/16601.

Ron Patton. *Software Testing*. Sams Publishing, Indianapolis, IN, USA, 2nd edition, 2006. ISBN 0-672-32798-8.

Celia Paulsen and Robert Byers. Glossary of Key Information Security Terms. *NIST Internal Report (IR) NIST IR 7298r3*, July 2019. doi: https://doi.org/10.6028/NIST.IR.7298r3. URL https://nvlpubs.nist.gov/nistpubs/ir/2019/NIST.IR.7298r3.pdf.

William E. Perry. *Effective Methods for Software Testing*. Wiley Publishing, Inc., Indianapolis, IN, USA, 3rd edition, 2006. ISBN 978-0-7645-9837-1.

J.F. Peters and W. Pedrycz. *Software Engineering: An Engineering Approach*. Worldwide series in computer science. John Wiley & Sons, Ltd., 2000. ISBN 978-0-471-18964-0.

Brian J. Pierre, Felipe Wilches-Bernal, David A. Schoenwald, Ryan T. Elliott, Jason C. Neely, Raymond H. Byrne, and Daniel J. Trudnowski. Open-loop testing results for the pacific DC intertie wide area damping controller. In *2017 IEEE Manchester PowerTech*, pages 1–6, 2017. doi: 10.1109/PTC.2017.7980834.

Sebastian Preuße, Hans-Christian Lapp, and Hans-Michael Hanisch. Closed-loop System Modeling, Validation, and Verification. In *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012)*, pages 1–8, Krakow, Poland, 2012. IEEE. ISBN 978-1-4673-4736-5. doi: 10.1109/ETFA.2012.6489679. URL https://ieeexplore.ieee.org/abstract/document/6489679.

Project Management Institute. *A Guide to the Project Management Body of Knowledge: PMBOK(R) Guide*. Project Management Institute, 5th edition, 2013. ISBN 1-935589-67-9.

Stuart C. Reid. Popular Misconceptions in Module Testing. In *Proceeding of the 13 International Conference on Testing Computer Software*, Washington, DC, USA, 1996.

Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14:131–164, April 2009. doi: https://doi.org/10.1007/s10664-008-9102-8. URL https://link.springer.com/article/10.1007/s10664-008-9102-8.

Vasile Rus, Sameer Mohammed, and Sajjan G Shiva. Automatic Clustering of Defect Reports. In *Proceedings of the Twentieth International Conference on Software Engineering & Knowledge Engineering (SEKE 2008)*, pages 291–296, San Francisco, CA, USA, July 2008. Knowledge Systems Institute Graduate School. ISBN 1-891706-22-5. URL https://core.ac.uk/download/pdf/48606872.pdf.

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 Vittorio Cortellessa and Dániel Varró, editors, *Fundamental Approaches to Software Engineering*, pages 343–358, Berlin, Heidelberg, March 2013. Springer Berlin Heidelberg. ISBN 978-3-642-37057-1. URL https://link.springer.com/chapter/10.1007/978-3-642-37057-1_25.

Raghvinder S. Sangwan and Phillip A. LaPlante. Test-Driven Development in Large Projects. *IT Professional*, 8(5):25–29, October 2006. ISSN 1941-045X. doi: 10.1109/MITP.2006.122. URL https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1717338.

Huib Schoots. The ISO29119 debate, September 2014. URL https://www.huibschoots.nl/wordpress/?p=1800.

Sheetal Sharma, Kartika Panwar, and Rakesh Garg. Decision Making Approach for Ranking of Software Testing Techniques Using Euclidean Distance Based Approach. *International Journal of Advanced Research in Engineering and Technology*, 12(2):599–608, February 2021. ISSN 0976-6499. doi: 10.34218/IJARET.12.2.2021.059. URL https://iaeme.com/Home/issue/IJARET?Volume=12&Issue=2.

Harry Sneed and Siegfried Göschl. A Case Study of Testing a Distributed Internet-System. *Software Focus*, 1:15–22, September 2000. doi: 10.1002/1529-7950(200009)1:13.3.CO;2-#. URL https://www.researchgate.net/publication/220116945_Testing_software_for_Internet_application.

Pradeep Soundararajan. An open letter to the President of the International Organization for Standardization about ISO 29119, December 2015. URL https://moolya.com/blog/testing-stories/an-open-letter-to-the-president-of-the-international-organization-for-standardization-about-iso-29119/.

Erica Souza, Ricardo Falbo, and Nandamudi Vijaykumar. ROoST: Reference Ontology on Software Testing. *Applied Ontology*, 12:1–32, March 2017. doi: 10.3233/AO-170177.

Keith Stouffer, Michael Pease, CheeYee Tang, Timothy Zimmerman, Victoria Pillitteri, Suzanne Lightman, Adam Hahn, Stephanie Saravia, Aslam Sherule, and Michael Thompson. Guide to Operational Technology (OT) Security. *NIST Special Publication (SP) NIST SP 800-82r3*, September 2023. doi: https://doi.org/10.6028/NIST.SP.800-82r3. URL https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-82r3.pdf.

Ephraim Suhir, Laurent Bechou, Alain Bensoussan, and Johann Nicolics. Photovoltaic reliability engineering: quantification testing and probabilistic-design-reliability concept. In *Reliability of Photovoltaic Cells, Modules, Components, and Systems VI*, volume 8825, pages 125–138. SPIE, September 2013. doi: 10.1117/12.2030377. URL https://www.spiedigitallibrary.org/conference-proceedings-of-spie/8825/88250K/Photovoltaic-reliability-engineering--quantification-testing-and-probabilistic-design-reliability/10.1117/12.2030377.full.

Guido Tebes, Luis Olsina, Denis Peppino, and Pablo Becker. TestTDO: A Top-Domain Software Testing Ontology. pages 364–377, Curitiba, Brazil, May 2020. ISBN 978-1-71381-853-3.

Daniel Trudnowski, Brian Pierre, Felipe Wilches-Bernal, David Schoenwald, Ryan Elliott, Jason Neely, Raymond Byrne, and Dmitry Kosterev. Initial closed-loop testing results for the pacific DC intertie wide area damping controller. In *2017 IEEE Power & Energy Society General Meeting*, pages 1–5, 2017. doi: 10.1109/PESGM.2017.8274724.

Kwok-Leung Tsui. An Overview of Taguchi Method and Newly Developed Statistical Methods for Robust Design. *IIE Transactions*, 24(5):44–57, May 2007. doi: 10.1080/07408179208964244. URL https://doi.org/10.1080/07408179208964244. Publisher: Taylor & Francis.

Matheus A. Tunes, Sean M. Drewry, Jose D. Arregui-Mena, Sezer Picak, Graeme Greaves, Luigi B. Cattini, Stefan Pogatscher, James A. Valdez, Saryu Fensin, Osman El-Atwani, Stephen E. Donnelly, Tarik A. Saleh, and Philip D. Edmondson. Accelerated radiation tolerance testing of Ti-based MAX phases. *Materials Today Energy*, 30(article 101186), October 2022. ISSN 2468-6069. doi: https://doi.org/10.1016/j.mtener.2022.101186. URL https://www.sciencedirect.com/science/article/pii/S2468606922002441.

Michael Unterkalmsteiner, Robert Feldt, and Tony Gorschek. A Taxonomy for Requirements Engineering and Software Test Alignment. *ACM Transactions on Software Engineering and Methodology*, 23(2):1–38, March 2014. ISSN 1049-331X, 1557-7392. doi: 10.1145/2523088. URL http://arxiv.org/abs/2307.12477. arXiv:2307.12477 [cs].

Petya Valcheva. Orthogonal Arrays and Software Testing. In Dimiter G. Velev, editor, *3rd International Conference on Application of Information and Communication Technology and Statistics in Economy and Education*, volume 200, pages 467–473, Sofia, Bulgaria, December 2013. University of National and World Economy. ISBN 978-954-644-586-5. URL https://icaictsee-2013.unwe.bg/proceedings/ICAICTSEE-2013.pdf.

Hans van Vliet. *Software Engineering: Principles and Practice*. John Wiley & Sons, Ltd., Chichester, England, 2nd edition, 2000. ISBN 0-471-97508-7.

Hironori Washizaki, editor. *Guide to the Software Engineering Body of Knowledge, Version 4.0*. January 2024.

Hironori Washizaki, editor. *Guide to the Software Engineering Body of Knowledge, Version 4.0a*. May 2025a. URL https://ieeecs-media.computer.org/media/education/swebok/swebok-v4.pdf.

Hironori Washizaki. Software Engineering Body of Knowledge (SWEBOK), September 2025b. URL https://www.computer.org/education/bodies-of-knowledge/software-engineering/.

Wikibooks Contributors. *Haskell/Variables and functions*. Wikimedia Foundation, October 2023. URL https://en.wikibooks.org/wiki/Haskell/Variables_and_functions.

Han Yu, C. Y. Chung, and K. P. Wong. Robust Transmission Network Expansion Planning Method With Taguchi's Orthogonal Array Testing. *IEEE Transactions on Power Systems*, 26(3):1573–1580, August 2011. ISSN 0885-8950. doi: 10.1109/TPWRS.2010.2082576. URL https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5620950.

Kaiqiang Zhang, Chris Hutson, James Knighton, Guido Herrmann, and Tom Scott. Radiation Tolerance Testing Methodology of Robotic Manipulator Prior to Nuclear Waste Handling. *Frontiers in Robotics and AI*, 7(article 6), February 2020. ISSN 2296-9144. doi: 10.3389/frobt.2020.00006. URL https://www.frontiersin.org/articles/10.3389/frobt.2020.00006.

Changlin Zhou, Qun Yu, and Litao Wang. Investigation of the Risk of Electromagnetic Security on Computer Systems. *International Journal of Computer and Electrical Engineering*, 4(1):92, February 2012. URL http://ijcee.org/papers/457-JE504.pdf. Publisher: IACSIT Press.

Appendix A

Detailed Scope Analysis

As outlined in Chapter 1, the scope of our research is limited to testing applied to code itself. Throughout our research, we identify many approaches that are out of scope based on this criteria.

A.1 Hardware Testing

While testing the software run *on* or in control *of* hardware is in scope, testing performed on the hardware *itself* is out of scope. The following are some examples of hardware test approaches we exclude from our research:

- Ergonomics testing and proximity-based testing (Hamburg and Mogyorodi, 2024) are out of scope, since they are used for testing hardware.
- EManations SECurity (EMSEC) testing (ISO, 2021; Zhou et al., 2012, p. 95), which deals with the "security risk" of "information leakage via electromagnetic emanation" (p. 95), is also out of scope.
- Continuity testing is a child of non-functional testing (Washizaki, 2025a, p. 6-8) and a test type that is concerned with "backup, restore and/or failover facilities" (Koomen et al., 2006, p. 50) which have corresponding test approaches. This form of continuity testing is in scope, but the child of hardware testing given by Firesmith (2015, p. 21) is not.
- All the examples of domain-specific testing given by Firesmith (2015, p. 26) are focused on hardware, so these examples are out of scope. However, this might not be representative of *all* kinds of domain-specific testing (e.g., Machine Learning (ML) model testing seems domain-specific), so some subset of this approach may be in scope.
- Similarly, the examples of environmental tolerance testing given by Firesmith (2015, p. 56) do not seem to apply to software. For example, radiation tolerance testing seems to focus on hardware, such as motors (Mukhin et al., 2022), robots (Zhang et al., 2020), or "nanolayered carbide and nitride materials" (Tunes et al., 2022, p. 1). Acceleration tolerance testing seems to

focus on astronauts (Morgun et al., 1999, p. 11), aviators (Howe and Johnson, 1995, pp. 27, 42), or catalysts (Liu et al., 2023, p. 1463) and acoustic tolerance testing on rats (Holley et al., 1996), which are even less related! Since these all focus on environment-specific factors that would not impact the code, these examples are out of scope. As with domain-specific testing, a subset of environmental tolerance testing may be in scope, but since no candidates have been found, this approach is out of scope for now.

- Knüvener Mackert GmbH (2022) uses the terms "software qualification testing" and "system qualification testing" in the context of the automotive industry. While these may be in scope, the more general idea of "qualification testing" seems to refer to the process of making a hardware component, such as an electronic component (Ahsan et al., 2020), gas generator (Parate et al., 2021) or photovoltaic device, "into a reliable and marketable product" (Suhir et al., 2013, p. 1). Therefore, it is currently unclear if this is in scope.

Investigate further

- Orthogonal Array Testing (OAT) can be used when testing software (Mandl, 1985) (in scope) but can also be used when testing hardware (Valcheva, 2013, pp. 471–472), such as "processors … made from pre-built and pre-tested hardware components" (p. 471) (out of scope). A subset of OAT called "Taguchi's Orthogonal Array Testing (TOAT)" is used for "experimental design problems in manufacturing" (Yu et al., 2011, p. 1573) or "product and manufacturing process design" (Tsui, 2007, p. 44) and is thus also out of scope.

- Since control systems often have a software *and* hardware component (ISO, 2015; Preuße et al., 2012; Forsyth et al., 2004), only the software component is in scope. In some cases, it is unclear whether the "loops"[1] being tested are implemented by software or hardware, such as those in wide-area damping controllers (Pierre et al., 2017; Trudnowski et al., 2017).
  - A related note: "path coverage" or "path testing" seems to be able to refer to either paths through code (as a subset of control-flow testing) (Washizaki, 2025a, p. 5-13; Reid, 1996, p. 4; implied by Hamburg and Mogyorodi, 2024) or through a model, such as a finite-state machine (as a subset of model-based testing) (Doğan et al., 2014, p. 184).

- Physical testing (inferred from physical requirements (ISO/IEC and IEEE, 2017, p. 322) and requirements-based testing; see Section 3.2) tests "a physical characteristic that a system or system component must possess" (p. 322).

[1] Humorously, the testing of loops in chemical systems (Dominguez-Pumar et al., 2020) and copper loops (Goralski, 1999) are out of scope.

A.2 V&V of Other Artifacts

While many artifacts produced by the software life cycle can be tested, only Verification and Validation (V&V) performed on the code *itself* — the System Under Test (SUT) — is in scope. Therefore, we exclude the following test approaches either in full or in part:

- Design reviews and documentation reviews are out of scope, as they focus on the V&V of design (ISO/IEC and IEEE, 2017, pp. 132) and documentation (p. 144), respectively.
- Security audits can focus on "an organization's … processes and infrastructure" (Hamburg and Mogyorodi, 2024) (out of scope) or "aim to ensure that all of the products installed on a site are secure when checked against the known vulnerabilities for those products" (Gerrard, 2000b, p. 28) (in scope).
- Error seeding is the "process of intentionally adding known faults[2] to those already in a computer program", done to both "monitor[] the rate of detection and removal", which is a part of V&V of the V&V itself (out of scope), "and estimat[e] the number of faults remaining" (ISO/IEC and IEEE, 2017, p. 165), which helps verify the actual code (in scope).
- Fault injection testing, where "faults are artificially introduced[2] into the SUT", can be used to evaluate the effectiveness of a test suite (Washizaki, 2025a, p. 5-18), which is a part of V&V of the V&V itself (out of scope), or "to test the robustness of the system in the event of internal and external failures" (ISO/IEC and IEEE, 2022, p. 42), which helps verify the actual code (in scope).
- "Mutation [t]esting was originally conceived as a technique to evaluate test suites in which a mutant is a slightly modified version of the SUT" (Washizaki, 2025a, p. 5-14), which is in the realm of V&V of the V&V itself (out of scope). However, it "can also be categorized as a structure-based technique" and can be used to assist fuzz and metamorphic testing (p. 5-15) (in scope).
- Nontechnical testing (inferred from nontechnical requirements (ISO/IEC and IEEE, 2017, p. 293) and requirements-based testing; see Section 3.2) tests "product and service acquisition or development that is not a property of the product or service" (p. 293).

[2] While error seeding and fault injection testing both introduce faults as part of testing, they do so with different goals: to "estimat[e] the number of faults remaining" (ISO/IEC and IEEE, 2017, p. 165) and "test the robustness of the system" (2022, p. 42), respectively. Therefore, these approaches are not considered synonyms, and the lack of this relation in the literature is not included in Section 5.2.2 as a synonym flaw.

A.3 Static Testing

Throughout the literature, static testing is more ambiguous than dynamic testing, with more ad hoc processes and inconsistent inclusion/exclusion from the scope of software testing in general (see Contradiction 5). Furthermore, it seems less relevant to our original goal of automatic generating test cases. In particular, many techniques require human intervention, either by design (such as code inspections) or to identify and resolve false positives (such as intentional exceptions to linting rules). Nevertheless, understanding the breadth of test approaches requires a "complete" picture of how software can be tested and how the various approaches relate to one another, and parts of these static approaches may even be generated in the future! For these reasons, we keep static testing in scope for this stage of our research.

A.4 Vague Terminology

Some terms are so vague that they do not provide any new, meaningful information. For example, the "systematic determination of the extent to which an entity meets its specified criteria" (ISO/IEC and IEEE, 2017, p. 167) is certainly relevant to testing software; while this definition of "evaluation" may be meaningful when defining software testing generally, it does not define a new approach or procedure and applies much more broadly than just to testing. The following terms are too vague to merit tracking in our glossaries or analyzing further[3]:

- **Evaluation:** the "systematic determination of the extent to which an entity meets its specified criteria" (ISO/IEC and IEEE, 2017, p. 167)
- **Product Analysis:** the "process of evaluating a product by manual or automated means to determine if the product has certain characteristics" (ISO/IEC and IEEE, 2017, p. 343)
- **Quality Audit:** "a structured, independent process to determine if project activities comply with organizational and project policies, processes, and procedures" (Project Management Institute, 2013, p. 247)
- **Software Product Evaluation:** a "technical operation that consists of producing an assessment of one or more characteristics of a software product according to a specified procedure" (ISO/IEC and IEEE, 2017, p. 424)

[3] Discussed in #39, #44, and #28.

A.5 Language-specific Approaches

Specific programming languages are sometimes used to define test approaches. If the reliance on a specific programming language is intentional, then this really implies an underlying test approach that may be generalized to other languages. These are therefore considered out-of-scope, including the following non-exhaustive list of examples:

- SQL injection is just defined as "a type of code injection in the Structured Query Language (SQL)" (Hamburg and Mogyorodi, 2024), which does not provide any new information.

OG Alalfi et al., 2010

- Similarly, SQL statement coverage (Doğan et al., 2014, Tab. 13) is just statement coverage used specifically for SQL statements.
- "An approach … for JavaScript testing (referred to as Randomized)" (Doğan et al., 2014, p. 192) is really just random testing used within JavaScript.
- Testing for "faults specific to PHP" is just a subcategory of fault-based testing, since "execution failures … caused by missing an included file, wrong MySQL quer[ies] and uncaught exceptions" (Doğan et al., 2014, Tab. 27) are not exclusive to PHP.

OG Artzi et al., 2008

A.6 Orthogonally Derived Approaches

Some test approaches appear to be combinations of other (seemingly orthogonal) approaches. While the use of a combination term can sometimes make sense, such as when writing a paper or performing testing that focuses on the intersection between two test approaches, they are sometimes given the same "weight" as their atomic counterparts. For example, Hamburg and Mogyorodi (2024) include "formal reviews" and "informal reviews" in their glossary as separate terms, despite their definitions essentially boiling down to "reviews that follow (or do not follow) a formal process", which do not provide any new information. These approaches are simply the combinations of "reviews" with "formal" and "informal testing", respectively. If a source describes an orthogonally derived approach in more detail, such as security audits, we record it as a distinct approach in our test approach glossary with its related information. Otherwise, we consider it out of scope since its details are captured by its in-scope subapproaches. The following are examples of these orthogonally derived approaches, most of which are out of scope:

1. Black box conformance testing (Jard et al., 1999, p. 25)
2. Black-box integration testing (Sakamoto et al., 2013, pp. 345–346)
3. Checklist-based reviews (Hamburg and Mogyorodi, 2024)
4. Closed-loop HiL[4] verification (Preuße et al., 2012, p. 6)

[4] See Overlap 7.

5. Closed-loop protection system testing (Forsyth et al., 2004, p. 331)
6. Conformity evaluations (ISO/IEC and IEEE, 2017, p. 93)
7. Design-based systems testing (Hetzel, 1988, pp. 138, 281)
8. Elastic end-to-end testing (Bertolino et al., 2019, p. 93:30)
9. Endurance stability testing (Firesmith, 2015, p. 55)
10. End-to-end functionality testing (ISO/IEC and IEEE, 2021c, p. 20; Gerrard, 2000a, Tab. 2)
11. Formal reviews (Hamburg and Mogyorodi, 2024; Washizaki, 2025a, p. 12-14)
12. Grey-box integration testing (Sakamoto et al., 2013, p. 344)

OG [19]

13. Incremental integration testing (Sharma et al., 2021, pp. 601, 603, 605–606)
14. Informal reviews (Hamburg and Mogyorodi, 2024; Washizaki, 2025a, p. 12-14)
15. Infrastructure compatibility testing (Firesmith, 2015, p. 53)
16. Invariant-based automatic testing (Doğan et al., 2014, pp. 184–185, Tab. 21; Mesbah and van Deursen, 2009)
17. Legacy system integration (testing) (Gerrard, 2000a, Tab. 2)
18. Manual procedure testing (Firesmith, 2015, p. 47)
19. Manual security audits (Gerrard, 2000b, p. 28)
20. Model-based GUI testing (Doğan et al., 2014, Tab. 1; implied by Sakamoto et al., 2013, p. 356)
21. Model-based web application testing (implied by Sakamoto et al., 2013, p. 356)
22. Non-functional search-based testing (Doğan et al., 2014, Tab. 1)
23. Offline Model-Based Testing (MBT) (Hamburg and Mogyorodi, 2024)
24. Online MBT (Hamburg and Mogyorodi, 2024)
25. Requirements-based systems testing (Hetzel, 1988, p. 135)
26. Role-based reviews (Hamburg and Mogyorodi, 2024)
27. Scenario walkthroughs (Gerrard, 2000a, Fig. 4)
28. Scenario-based reviews (Hamburg and Mogyorodi, 2024)
29. Security attacks (Hamburg and Mogyorodi, 2024)

30. Security audits (ISO/IEC and IEEE, 2021c, p. 40; Gerrard, 2000b, p. 28)
31. Statistical web testing (Doğan et al., 2014, p. 185)
32. Usability test script(ing) (Hamburg and Mogyorodi, 2024)
33. Web application regression testing (Doğan et al., 2014, Tab. 21)
34. White-box unit testing (Sakamoto et al., 2013, pp. 345–346)

There are some cases where the subapproaches of the "compound" approaches listed previously are *not* from separate categories. However, these cases can be explained by insufficient data or by edge cases that require special care. While we assume that the categories given in Table 2.1 are orthogonal, further analysis may disprove this. For now, all of these special cases are affected by at least one of the following conditions:

1. **At least one subapproach is categorized inconsistently.** When a subapproach has more than one category (see Section 5.2.1), it is unclear which one should be used to assess orthogonality.
2. **At least one subapproach's category is inferred.** When the category of a test approach is not given by the literature but is inferred from related context (see Section 2.3), it is unclear if it can be used to assess orthogonality.
3. **At least one subapproach is only categorized as an approach.** Since "approach" is a catch-all categorization, it does not need to be orthogonal to its subcategories.
4. **A subapproach is explicitly based on another in the same category.** An example of this is stability testing, which tests a "property that an object has with respect to a given failure mode if it cannot exhibit that failure mode" (ISO/IEC and IEEE, 2017, p. 434). This notion of "property" is similar to that of "quality" that the test type category is built on, so it is acceptable that is implied to be a test type by its quality (ISO/IEC and IEEE, 2017, p. 434) and by Firesmith (2015, p. 55).

OG ISO/IEC, 2009

OG ISO/IEC, 2009

Appendix B

Sources by Tier

The following lists of sources comprise the corresponding source tier as defined in Section 2.5.

B.1 Established Standards

(ISO/IEC and IEEE, 2022; 2021a;b;c; 2019a;b; 2017; 2016; 2015; 2013; 2010; IEEE, 2024; 2012; IEEE Computer Society, 2010; ISO/IEC, 2023a;b; 2018; 2015; 2014a;b;c; 2011; 2005; ISO, 2022; 2015; Stouffer et al., 2023)

B.2 Terminology Collections

(Washizaki, 2025a; 2024; Hamburg and Mogyorodi, 2024; Firesmith, 2015; Doğan et al., 2014; Bourque and Fairley, 2014; Project Management Institute, 2013; Lyu, 1996)

B.3 Textbooks

(Ammann and Offutt, 2017; Dennis et al., 2012; Kaner et al., 2011; Black, 2009; Koomen et al., 2006; Patton, 2006; Perry, 2006; Gerrard and Thompson, 2002; Peters and Pedrycz, 2000; van Vliet, 2000; Hetzel, 1988)

B.4 Papers and Other Documents

(Bas, 2024; ChatGPT (GPT-4o), 2024; LambdaTest, 2024; Pandey, 2023; Knüvener Mackert GmbH, 2022; Sharma et al., 2021; Intana et al., 2020; Bertolino et al., 2019; Kanewala and Yueh Chen, 2019; Moghadam, 2019; Bajammal and Mesbah, 2018; Souza et al., 2017; Dhok and Ramanathan, 2016; Barr et al., 2015; Engström and Petersen, 2015; Kuļešovs et al., 2013; Lahiri et al., 2013; Sakamoto et al., 2013; Valcheva, 2013; Preuße et al., 2012; Yu et al., 2011; Godefroid and Luchaup, 2011; Choudhary et al., 2010; Mesbah and van Deursen, 2009; Rus et al., 2008; Kam, 2008; Tsui, 2007; Barbosa et al., 2006; Berdine et al., 2006; Chalin et al., 2006;

Baresi and Pezzè, 2006; Sangwan and LaPlante, 2006; Forsyth et al., 2004; Sneed and Göschl, 2000; Gerrard, 2000a;b; Jard et al., 1999; Ghosh and Voas, 1999; Reid, 1996; Mandl, 1985)

Appendix C

Tools User Guide

Since we keep the description of our tools abstract in Chapter 4, we provide a more in-depth description of our tools as follows. We first outline how we cite information in our glossaries (Appendix C.1) then explain how we annotate flaws so we can use tools to automatically analyze them (Appendix C.2).

C.1 Citation Syntax

When recording data in our glossaries, we capture relevant citation information using the author-year citation format. When citing the same authors or sources multiple times, we "reuse" information from previous citations when applicable. For example, the citation "(ISO/IEC and IEEE, 2022; 2017)" means that the relevant information appears in both ISO/IEC and IEEE (2022) *and* ISO/IEC and IEEE (2017). If the following citation was "(2022)", it would have the same *author* as the last citation with one specified; this would be equivalent to "(ISO/IEC and IEEE, 2022)". If this was then followed by "(p. 36)", this would have the same *year* as the last citation with one specified and the same *author* as the last citation before that, resulting in "(ISO/IEC and IEEE, 2022, p. 36)".

Important: Figure out better way to describe "last citation". "Most recent"?

This reduces duplication and improves maintainability when recording this information. When processing these data when visualizating relations and detecting flaws (see Sections 4.1 and 4.2.1), we do so according to this logic (see the relevant source code) so we can consistently track the source(s) of data throughout our analysis.

One minor note to make here is that in our test approach glossary, we *actually* record "(ISO/IEC and IEEE, 2022)" as "(IEEE, 2022)" for brevity, since most standards are written by ISO/IEC *and* IEEE (see Figure 4.3). To faciliate this, we choose corresponding BibTeX keys, such as "`IEEE2022`" in this example, and specify which documents are only written by IEEE (see the relevant source code).

C.2 Flaw Comment Syntax

As described in Section 3.3.2, we include comments alongside the flaws we document so we can analyze them automatically (as described in Section 4.2.2). These comments have the following format:

```
% Flaw count (MNFST, DMN): {A1} {A2} … | {B1} … | {C1} …
```

`MNFST` and `DMN` are placeholders for the "keys" given in Tables 2.2 and 2.3, respectively, that we use to track a flaw's manifestation(s) and domain(s) (defined in Section 2.2). For example, the comment line for an incorrect synonym relation would start with "`% Flaw count (WRONG, SYNS)`" and one for a redundant label would start with "`% Flaw count (REDUN, LABELS)`". We omit these keys from this chapter for simplicity. Finally, `A1`, `A2`, `B1`, and `C1` are each placeholders for a source involved in this example flaw; in general, there can be arbitrarily many. We represent each source by its BibTeX key and wrap each one in curly braces (with the exception of the ISTQB glossary due to its use of custom commands via `\citealias{}`) to mimic LaTeX's citation commands for ease of parsing. We then separate each "group" of sources with a pipe symbol (`|`) so we can compare each pair of groups; in general, a flaw can have any number of groups of sources.

As mentioned in Section 2.2.3, we make a distinction between "self-contained" flaws and "internal" flaws. We track self-contained flaws by recording the single source that the flaw is present in such as in the first line below. In constrast, we track internal flaws by recording the single source in multiple groups (as defined above). The second line is a standard example of this, while the third is more complex; in this case, source `Y` agrees with only one of the conflicting sources of information in `X`.

```
% Flaw count: {X}
% Flaw count: {X} | {X}
% Flaw count: {X} | {X} {Y}
```

We can also specify the "explicitness" (see Section 2.3) of a flaw by inserting the phrase "implied by" after the sources of explicit information and before those of implicit information, such as in the following example:

```
% Flaw count: {X} {Y} | {Z} implied by {X}
```

Later: Ensure this is up to date

Note that we only count subjective flaws if there is not an equivalent objective flaw, as we do when visualizing relations (as described in Section 4.1). The following comment line from Contradiction 10 is an example of a flaw that is both objective and subjective:

```
% Flaw count (CONTRA, DEFS): {IEEE2021c} {IEEE2017} |
↪  {vanVliet2000} implied by {IEEE2021c}
```

This indicates that the following flaws are present:

- an objective inconsistency between a textbook and a standard,
- a subjective flaw within a single document, and
- a subjective inconsistency between documents with the same set of authors (ISO/IEC and IEEE; see Figure 4.3).

This third flaw only affects our more nuanced breakdown of the sources of flaws in Figure 5.1. Note that we do not double count the first flaw. We likewise do not double count flaws that reappear when comparing between pairs of groups; for example, we would only count the inconsistency between `X` and `Z` *once* in the following flaw comment:

```
% Flaw count: {X} | {X} {Y} | {Z}
```

Occasionally, we assert that a source from a less credible tier is more correct than a source from a more credible tier[1]. If we documented these flaws as above, they would incorrectly be counted as flaws within the less credible tier! Therefore, we document these "assertion" sources separately to track them for traceability without counting them incorrectly. For example, if we assert that a textbook `W` is correct and indicates a flaw in established standards `X` and `Y`, we would track this assertion separately from its associated flaw as follows:

```
% Flaw count: {X} {Y}
% Assertion: {W}
```

[1] Discussed in #184.

Appendix D

Full Lists of Flaws

The following are the full lists of manually identified flaws grouped by their manifestation (Appendix D.1) and automatically identified flaws grouped by their domain (Appendix D.2) as defined in Section 2.2. We then present inferred flaws (Appendix D.3) as described in Section 2.3 for completeness, although these flaws do not contribute to any counts.

D.1 Full Lists of Flaws by Manifestation

We sort the following groups of flaws approximately by their source tier (defined in Section 2.5) in descending order of credibility (defined in Section 2.4). We use a flaw's manifestation and its number in its corresponding list when referring to it as described in Section 2.2.1.

D.1.1 Full List of Mistakes

1. ISO/IEC and IEEE (2022, p. 5) give fuzz testing the tag "artificial intelligence"; while fuzz testing could certainly be implemented in this way, it does not seem to be a requirement.

2. Since the differences between the terms "error", "fault", "failure", and "defect" are significant (see Chapter 1), "error guessing"—a term used by multiple sources (ISO/IEC and IEEE, 2022; 2021c; 2013; Washizaki, 2025a; Firesmith, 2015; Patton, 2006)—should either be called:

    - "defect guessing" if it is based on a "checklist of potential defects" (ISO/IEC and IEEE, 2021c, p. 29),
    - "failure guessing" if it is based on "the tester's knowledge of past failures" (2017, p. 165), or
    - "fault guessing" if it is a "fault-based technique" (Bourque and Fairley, 2014, p. 4-9) that "anticipate[s] the most plausible faults in each SUT" (Washizaki, 2025a, p. 5-13).

3. Since faults and errors are distinct (see Chapter 1), "fault seeding" is not a synonym of "error seeding" as claimed by ISO/IEC and IEEE (2017, p. 165) and van Vliet (2000, p. 427). The term "error seeding", also used by Firesmith (2015, p. 34), should be abandoned in favour of "fault seeding" if it is defined as the "process of intentionally adding known faults to those already in a computer program … [to] estimat[e] the number of faults remaining" (ISO/IEC and IEEE, 2017, p. 165) based on the ratio between the number of new faults and the number of introduced faults that were discovered (van Vliet, 2000, p. 427).

4. "Functionality" is defined as the "capabilities of the various … features provided by a product" (ISO/IEC and IEEE, 2017, p. 196) while "functional suitability" refers to the "capability of a product to provide [specified] functions" (ISO/IEC, 2023a; similar in ISO/IEC and IEEE, 2017, p. 196; Hamburg and Mogyorodi, 2024). However, Hamburg and Mogyorodi (2024) say these terms are synonyms, despite the former focusing on the *capabilities* of a product's features as opposed to simply their *provision* as outlined by the latter.

5. A typo in ISO/IEC and IEEE (2021c, Fig. 2) means that "specification-based techniques" is listed twice, when the latter should be "structure-based techniques".

6. ISO/IEC and IEEE (2017) use the same definition for "partial correctness" (p. 314) and "total correctness" (p. 480).

7. Since keyword-driven testing can be used in automated *or* manual testing (ISO/IEC and IEEE, 2022, pp. iii, vii, 6, 9, 29, 33, 35, 37; 2016, pp. 3–6), the claims that "test cases can be either manual test cases or keyword test cases" and "a keyword test case implements a manual test case" (p. 6) are incorrect. These statements could also be interpreted as implying that "keyword-driven testing" is a synonym of "automated testing", which is also incorrect.

8. Hamburg and Mogyorodi (2024) define "test level" as "a specific instantiation of a test process" which is vague and does not match ISO/IEC and IEEE's definition given in Table 2.1 and further discussed in Section 2.1.1.

9. Washizaki (2025a, p. 5-4) says that quality improvement, along with quality assurance, is an aspect of testing that involves "defining methods, tools, skills, and practices to achieve the specific quality level and objectives"; while testing that a system possesses certain qualities is in scope, actively improving the system in response to these results is *not* itself part of testing (ISO/IEC and IEEE, 2022, p. 10; 2021c, p. 6; 2017, p. 473).

10. The terms "acceleration tolerance testing" and "acoustic tolerance testing" do not seem to refer to software testing, but Firesmith (2015, p. 56) includes them regardless. Elsewhere, they seem to refer to testing the acoustic tolerance of rats (Holley et al., 1996) or the acceleration tolerance of astronauts

(Morgun et al., 1999, p. 11), aviators (Howe and Johnson, 1995, pp. 27, 42), or catalysts (Liu et al., 2023, p. 1463), which which are not relevant to software testing.

11. Mathematical-based testing is a test practice based on “the test item’s required behaviour, input space or output space” when they “can be described in sufficient detail” (ISO/IEC and IEEE, 2022, p. 36). However, Hamburg and Mogyorodi (2024) define “math testing” as “testing to determine the correctness of the pay table implementation, the random number generator results, and the return to player computations”. While these are all subsets of mathematical-based testing, this definition is likely taken from a particular case study or test item, making its scope too narrow for it to be widely useful.

12. Hamburg and Mogyorodi (2024, emphasis added) define “multiplayer testing” as “testing to determine if many players can simultaneously interact with the *casino* game world, … computer-controlled opponents, game servers, and … each other based on the game design”. This definition sheds light on the particular test item they use to define “math testing” (see Mistake 11); omitting the word “casino” would make this definition apply more widely.

13. Hamburg and Mogyorodi (2024) define “par sheet testing” as “testing to determine that the game returns the correct mathematical results to the screen, to the players’ accounts, and to the casino account”. This seems to refer to the specific example from Mistake 11 and Mistake 12 and could be a valid domain-specific test approach, but this definition does not even seem specific to PAR sheets—“list[s] of all the symbols on each reel of a slot machine” (Bluejay, 2024)—themselves!

14. Hamburg and Mogyorodi (2024) cite Paulsen and Byers (2019) for their definition of “security attack”, but this source does not provide one. Another standard by the same organization—the National Institute of Standards and Technology (NIST)—provides a similar definition for the term “attack” (Stouffer et al., 2023, p. 160) as opposed to “security attack”.

15. Hamburg and Mogyorodi (2024) cite ISO/IEC (2020) for their definition of “visual testing”, but this source does not provide one.

16. Hamburg and Mogyorodi (2024) incorrectly list “M. Koomen” as an author of Koomen et al. (2006), when the correct author name is “Tim Koomen” and should therefore be cited as “T. Koomen”.

17. Reid (1996, p. 4) says “the use of the term ‘condition’ in branch condition testing can mislead the reader into thinking all conditions are exercised”, which Hamburg and Mogyorodi (2024) seem to do by giving “condition coverage” as a synonym of “branch condition coverage”.

18. Doğan et al. (2014, p. 184) claim that Sakamoto et al. (2013) define “prime path coverage”, but they do not.

19. The publisher of Black (2009) is spelled wrong (“Rock Nook Inc.” instead of “Rocky Nook Inc.”) on page six but spelled correctly elsewhere.

20. Peters and Pedrycz (2000, pp. 438, 497) cite Myers (1976) but misspell the author’s name as “Meyers” in the References section of the relevant chapter (p. 500). This is especially confusing since they also include Myers (1992) in the bibliography, which was written by a different author.

21. The differences between the terms “error”, “fault”, “failure”, and “defect” are significant (see Chapter 1), but Patton (2006, pp. 13–14) “just call[s] it what it is and get[s] on with it”, abandoning these four terms, “problem”, “incident”, “anomaly”, “variance”, “inconsistency”, “feature” (!), and “a list of unmentionable terms” in favour of “bug”; after all, “there’s no reason to dice words”! Not only does he set aside established conventions because of their negative connotations, including severity and blame (p. 14), he also includes “feature” in this list. This is likely because to his definition of “bug”, which includes the case where “the software does something that the product specification doesn’t mention”; this means that “an ambitious programmer” who implements extra features is actually introducing bugs (p. 15). While this synonym relation holds in this specific case, not all features are bugs!

22. Peters and Pedrycz (2000, Fig. 12.31) imply that decision coverage is a child of both Computation data use (c-use or C-use)[1] coverage *and* Predicate data use (p-use or P-use)[1] coverage; this seems incorrect, since decisions are the result of p-uses and *not* c-uses (ISO/IEC and IEEE, 2021c, pp. 5, 27; 2017, p. 332; van Vliet, 2000, p. 424), and only the p-use relation is implied by ISO/IEC and IEEE (2021c, Fig. F.1).

23. Sharma et al. (2021, p. 601) seem to use the terms “grey-box testing” and “(stepwise) code reading” interchangeably, which would incorrectly imply that they are synonyms.

FIND SOURCES

24. Kam (2008, p. 46) gives “program testing” as a synonym of “component testing” but it would make more sense as a synonym of “system testing”, which is conducted on the system, or program, as a whole (ISO/IEC and IEEE, 2017, p. 456; Hamburg and Mogyorodi, 2024; Peters and Pedrycz, 2000, Tab. 12.3; van Vliet, 2000, p. 439; Sakamoto et al., 2013, pp. 343–344).

OG 2012

25. Kam (2008, p. 46) gives “mutation testing” as a synonym of “back-to-back testing”; while the two are related (ISO/IEC and IEEE, 2010, p. 30), the variants used in mutation testing are generated or designed to be detected as incorrect by the test suite (Washizaki, 2025a, p. 5-15; similar in van Vliet, 2000, pp. 428–429) which is not a requirement of back-to-back testing.

26. “Negative testing” is defined as “testing a component or system in a way for which it was not intended to be used” (Hamburg and Mogyorodi, 2024;

[1] See Contradiction 32.

OG Beizer

similar in Patton, 2006, p. 84–87), such as for functionality "not included in the specification" or "inputs … that should either be ignored … or cause … an error message" (ISO/IEC and IEEE, 2021c, p. 11; similar in Patton, 2006, p. 84–87). However, Kam (2008, p. 46) says that it is "aimed at showing that a component or system does not work" which misrepresents this approach: yes, it often involves "testing with invalid input values or exceptions", but these are used to see how a component or system handles them, not to prove that it cannot!

27. Kam (2008, p. 42) says "See *boundary value analysis*," for the glossary entry of "boundary value testing" but does not include "boundary value analysis" in the glossary.

28. Kam (2008) misspells "state-based" as "state-base" (pp. 13, 15) and "stated-base" (Tab. 1).

29. Chalin et al. (2006, p. 343) list runtime assertion checking and static verification as "two complementary forms of assertion checking"; based on how the term "static assertion checking" is used by Lahiri et al. (2013, p. 345), it seems like this should be the complement to runtime assertion checking instead.

30. Gerrard's (2000b, p. 28) definition of "manual security audits" may be too specific, only applying to "the products installed on a site" and "the known vulnerabilities for those products".

31. Sneed and Göschl (2000, p. 3) give "white-box testing", "grey-box testing", and "black-box testing" as synonyms for "module testing", "integration testing", and "system testing", respectively, but this mapping is incorrect; for example, ISO/IEC and IEEE (2017, p. 444) say "structure-based [(or white-box)] testing is not restricted to use at component level and can be used at all levels" and Sakamoto et al. (2013, pp. 345–346) describe "black-box integration testing". They likely take the "implicit" claims from the original source they cite (Hetzel, 1988) too far; for example, "as a practical matter, *most* system testing relies on the black-box perspective" (p. 11, emphasis added). However, this does not mean that system testing is *always* the same as black-box testing.

32. Sneed and Göschl's (2000, p. 3) incorrect claims about test level synonyms from Mistake 31 make their claim that "red-box testing" is a synonym for "acceptance testing" lose credibility.

33. Sneed and Göschl (2000, p. 3) assert that "module testing" can also be called "class testing", but this claim does not appear in the source they cite (Hetzel, 1988). They might have meant to say that "module testing" and "program testing" are synonyms, which is implied by this source (p. 73) and supported by Kam (2008, p. 46).

D.1.2 Full List of Omissions

1. ISO/IEC and IEEE (2021c) cite Reid (1996) as the source for their Fig. F.1 but they omit Linear Code Sequence and Jump (LCSAJ) testing with no explanation, both from this figure and from the document as a whole. They label the figure as a "partial ordering", which might explain its omission from Fig. F.1, but Reid (1996, p. 7) already identifies that his hierarchy is incomplete as "it relates only a subset of the available test completion criteria, so other criteria … should still be considered".

2. The acronym for System of Systems (SoS) (ISO/IEC and IEEE, 2019b) is used but not defined by Firesmith (2015, p. 23).

3. The Project Management Institute (2013, p. 476) uses the acronym "QA" which is implied to refer to "quality assurance" as "QC" refers to "quality control" (p. 535), but they do not make this explicit.

4. Van Vliet (2000, p. 425) defines many types of data flow coverage, including all-p-uses, all-p-uses/some-c-uses, and all-c-uses/some-p-uses, but excludes all-c-uses, which is implied by these definitions and defined elsewhere (ISO/IEC and IEEE, 2021c, p. 27; 2017, p. 83; Peters and Pedrycz, 2000, p. 479).

5. Bas (2024, p. 16) lists "three [backup] location categories: local, offsite and cloud based [sic]" but does not define or discuss "offsite backups" (pp. 16–17).

6. Kam (2008, p. 42) cites "Musa" as the source of his definition of "operational profile testing", but this does not appear in his references section. Based on Lyu's (1996, p. 539) discussion of similar content, this likely refers to (Musa et al., 1987).

7. Gerrard (2000a, Tab. 2) makes a distinction between "transaction verification" and "transaction testing" and uses the phrase "transaction flows" (Fig. 5) but doesn't explain them.

8. Availability testing is not assigned to a test priority (Gerrard, 2000a, Tab. 2), despite the claim that "the test types[2] have been allocated a slot against the four test priorities" (p. 13); usability testing and/or performance testing would have been good candidates.

D.1.3 Full List of Contradictions

1. Regression testing and retesting are sometimes given as two distinct approaches (ISO/IEC and IEEE, 2022, p. 8; 2021a, p. 3; Firesmith, 2015, p. 34), but sometimes regression testing is defined as a form of "selective retesting" (ISO/IEC and IEEE, 2017, p. 372; Washizaki, 2025a, pp. 5-8, 6-5,

[2] "Each type of test addresses a different risk area" (Gerrard, 2000a, p. 12), which is distinct from the notion of "test type" described in Table 2.1.

7-5; Barbosa et al., 2006, p. 3). Moreover, the two possible variations of regression testing given by van Vliet (2000, p. 411) are "retest-all" and "selective retest", which is possibly the source of the above misconception. This creates a cyclic relation between regression testing and selective retesting.

Are these separate approaches?

2. Accessibility testing is a subtype of usability testing (ISO/IEC and IEEE, 2022, p. 1; 2021c, Tab. A.1; 2017, p. 6; ISO/IEC, 2011; Firesmith, 2015, p. 58) but these two test types are listed at the same level by ISO/IEC and IEEE (2022, Fig. 2).

3. Integration testing, system testing, and system integration testing are all listed as separate test levels (ISO/IEC and IEEE, 2022, p. 12, Fig. 2; 2021b, p. 41–44, 46, 51, 58, 74; 2021c, p. 6), but system integration testing is listed as a child of both integration testing (Hamburg and Mogyorodi, 2024) and system testing (Firesmith, 2015, p. 23).

4. Hamburg and Mogyorodi (2024) define "component integration testing" as "the integration testing of components". This is consistent with the definition of "integration testing" — testing that "verifies the interactions among SUT elements (for instance, *components*, modules, or subsystems)" as well as "external interfaces" (Washizaki, 2025a, p. 5-7, emphasis added; similar in ISO/IEC and IEEE, 2022, p. 13; 2021a, p. 6; 2021c, p. 6; van Vliet, 2000, p. 438; Sakamoto et al., 2013, p. 343) — which gives the former as a child of the latter. However, Hamburg and Mogyorodi's (2024) mind map of static and dynamic testing gives "integration testing" as a child of "component integration testing", inverting this relationship.

5. "Software testing" is often defined to exclude static testing (Firesmith, 2015, p. 13; Peters and Pedrycz, 2000, p. 439; Ammann and Offutt, 2017, p. 222), restricting "testing" to mean "dynamic validation" (Washizaki, 2025a, p. 5-1) or verification "in which a system or component is executed" (ISO/IEC and IEEE, 2017, p. 427). However, "terminology is not uniform among different communities, and some use the term 'testing' to refer to static techniques[3] as well" (Washizaki, 2025a, p. 5-2). This is done by ISO/IEC and IEEE (2022, pp. 16–17; 2021b, p. 43) and Gerrard (2000a, pp. 8–9), although the former authors explicitly *exclude* static testing in another document (2017, p. 440)!

6. ISO/IEC and IEEE categorize static testing as a test level in (2021b, p. 43) but make it its own test approach category in (2022, p. 10, 23, Fig. 2).

7. ISO/IEC (2023a) and ISO/IEC and IEEE (2017, p. 196) both say that functional suitability is "concerned with whether the functions meet stated and implied needs", but the former includes "the functional specification" as part of its scope while the latter explicitly excludes it.

[3] Not formally defined, but distinct from the notion of "test technique" described in Table 2.1.

8. ISO/IEC and IEEE (2017) define both "error" *and* "mistake" as "human action[s] that produce[] an incorrect result" (pp. 165, 278, respectively) but state that "the fault tolerance discipline distinguishes between a human action (a mistake)… and the amount by which the result is incorrect (the error)" (p. 278). This makes "error" and "mistake" simultaneously synonyms *and* not synonyms!

9. A component is an "entity with discrete structure … within a system considered at a particular level of analysis" (ISO/IEC, 2023b) and "the terms module, component, and unit are often used interchangeably or defined to be subelements of one another in different ways depending upon the context" with no standardized relationship (ISO/IEC and IEEE, 2017, p. 82). For example, Hamburg and Mogyorodi (2024) define them as synonyms while Baresi and Pezzè (2006, p. 107) say "components differ from classical modules for being re-used in different contexts independently of their development". Additionally, since components are structurally, functionally, or logically discrete (ISO/IEC and IEEE, 2017, p. 419) and "can be tested in isolation" (Hamburg and Mogyorodi, 2024), "unit/component/module testing" could refer to the testing of both a module *and* a specific function in a module, introducing a further level of ambiguity.

10. While a c-use is defined as the "use of the value of a variable in *any* type of statement" (ISO/IEC and IEEE, 2021c, p. 2; 2017, p. 83, emphasis added), it is often qualified to *not* be a p-use (van Vliet, 2000, p. 424; implied by ISO/IEC and IEEE, 2021c, p. 27).

11. ISO/IEC and IEEE define an "extended entry (decision) table" both as a decision table where the "conditions consist of multiple values rather than simple Booleans" (2021c, p. 18) and one where "the conditions and actions are generally described but are incomplete" (2017, p. 175).

OG ISO1984

12. ISO/IEC and IEEE (2021c, Fig. F.1) is an adaptation of Reid (1996, Fig. 2) and one of the changes they make is replacing "branch [coverage]" with "branch/decision coverage". Reid notes that "the term decision coverage is used interchangeably with that of branch coverage" but that comparing one to the other is not a direct mapping (p. 4). ISO/IEC and IEEE agree, saying "branch and decision coverage are closely related…, although lower levels of coverage may not be the same" (2021c, p. 104) and separating these two terms (Fig. G.1, Secs. 5.3.2, 5.3.3, Annex C.2.2). However, (Fig. F.1) implies that these terms are synonyms, contradicting this separation and making decision testing's relations ambiguous.

13. ISO/IEC and IEEE's (2017, p. 83) definition of "all-c-uses testing"—testing that aims to execute all data "use[s] of the value of a variable in any type of statement"—is *much* more vague than the definition they give in (2021c, p. 27; similar in van Vliet, 2000, p. 425; Peters and Pedrycz, 2000, p. 479): testing that exercises "control flow sub-paths from each variable definition to each c-use of that definition (with no intervening definitions)".

14. ISO/IEC and IEEE (2022, p. 36) say "A/B testing is not a test case generation technique as test inputs are not generated", where "test case generation technique" may be a synonym of "test design technique". However, the inclusion of A/B testing under the heading "Test design and execution" in the same document implies that it may be considered a test technique.[4]

15. Performance testing and security testing are subtypes of reliability testing (ISO/IEC, 2023a) but Firesmith (2015, p. 53) lists all three separately.

16. Random testing is a subtechnique of specification-based testing (ISO/IEC and IEEE, 2022, pp. 7, 22; 2021c, pp. 5, 20, Fig. 2; Washizaki, 2025a, p. 5-12; Hamburg and Mogyorodi, 2024) but Firesmith (2015, p. 46) lists them separately.

17. Path testing "aims to execute all entry-to-exit control flow paths in a SUT's control flow graph" (Washizaki, 2025a, p. 5-13; similar in Patton, 2006, p. 119), but ISO/IEC and IEEE (2017, p. 316) add that it can also be "designed to execute … selected paths."

18. The structure of tours can be defined as either quite general (ISO/IEC and IEEE, 2022, p. 34) or "organized around a special focus" (Hamburg and Mogyorodi, 2024). These differences make it unclear how testers should perform tours, which could lead to miscommunication and unmet expectations.

19. Alpha testing can be performed by "users within the organization developing the software" (ISO/IEC and IEEE, 2017, p. 17), "a small, selected group of potential users" (Washizaki, 2025a, p. 5-8), or "roles outside the development organization" conducted "in the developer's test environment" (Hamburg and Mogyorodi, 2024). These differences make it unclear how testers should perform alpha testing, which could lead to miscommunication and unmet expectations.

20. "Conformance testing" is defined by Washizaki (2025a, p. 5-7) as testing that "aims to verify that the SUT conforms to standards, rules, specifications, requirements, design, processes, or practices", but this disagrees with the definition given by the Project Management Institute (2013, p. 523): testing that evaluates the degree to which "results … fall within the limits that define acceptable variation for a quality requirement". Washizaki's definition instead seems to correspond to the definition of compliance testing given by Hamburg and Mogyorodi (2024) and Firesmith (2015, p. 33), which may explain why Kam (2008, p. 43) gives them as synonyms (along with his unhelpful definition of compliance testing: "testing to determine the compliance of the component or system").

OG IREB Glossary

21. The terms "test level" and "test stage" are given as synonyms (Hamburg and Mogyorodi, 2024; implied by ISO/IEC and IEEE, 2015, p. 9; Gerrard,

[4] For simplicity, this implied categorization as "technique" is omitted from Table D.1.

2000a, p. 9), but Washizaki (2025a, p. 5-6) says "[test] levels can be distinguished based on the object of testing, the *target*, or on the purpose or *objective*" and calls the former "test stages", giving the term a child relation (see Section 2.1.3) to "test level" instead. However, the examples of "test stages" listed—unit testing, integration testing, system testing, and acceptance testing (Washizaki, 2025a, pp. 5-6 to 5-7)—are commonly categorized as "test levels" (see Section 2.1.1).

22. "Pair testing" and "buddy testing" are synonyms (Washizaki, 2025a, p. 5-14) but Firesmith (2015, pp. 36, 39) lists them separately.

23. Hamburg and Mogyorodi (2024) give "operational acceptance testing" and "production acceptance testing" as synonyms, but Firesmith (2015, p. 30) lists them separately.

24. Washizaki (2025a, p. 1-1) defines "defect" as "an observable difference between what the software is intended to do and what it does", but this seems to instead match the definition of "failure": the inability of a system "to perform a required function or … within previously specified limits" that is "externally visible" (ISO/IEC and IEEE, 2019a, p. 7; similar in IEEE, 2024, pp. 15, 37; Washizaki, 2025a, p. 5-3; Lyu, 1996, p. 12; van Vliet, 2000, p. 400).

25. Retesting and regression testing seem to be categorized separately from the rest of the test approaches (ISO/IEC and IEEE, 2022, pp. 15, 23; 2021a, p. 8; 2021b, p. 4) but this is not justified. Hamburg and Mogyorodi (2024) consider regression testing to be a test type while Lyu (1996, p. 532) and Barbosa et al. (2006, p. 3) consider it a test level; since it is not included as an example of a test level by the sources that describe them (see Section 2.1.1), this latter categorization is likely not universal at best and incorrect at worst.

26. While Patton (2006, p. 120) implies that condition testing is a subtechnique of path testing, van Vliet (2000, Fig. 13.17) says that multiple condition coverage (which seems to be a synonym of condition coverage (p. 422)) does not subsume and is not subsumed by path coverage.

27. Load testing may be performed with loads "between anticipated conditions of low, typical, and peak usage" (ISO/IEC and IEEE, 2022, p. 5) or with loads that are as large as possible (Patton, 2006, p. 86). These differences make it unclear how testers should perform load testing, which could lead to miscommunication and unmet expectations.

28. State testing requires that "all states in the state model … [are] 'visited'" (ISO/IEC and IEEE, 2021c, p. 19), but Patton (2006, pp. 82–83) lists this as only one of its possible criteria.

29. System testing is "conducted on a complete, integrated system" (ISO/IEC and IEEE, 2017, p. 456; similar in Peters and Pedrycz, 2000, Tab. 12.3; van

Vliet, 2000, p. 439), but Patton (2006, p. 109) says it can also be done on "at least a major portion" of the product.

30. "Walkthroughs" and "structured walkthroughs" are given as synonyms by Hamburg and Mogyorodi (2024) but Peters and Pedrycz (2000, p. 484) imply that they are different, saying a more structured walkthrough may have specific roles.

31. While Patton (2006, p. 92, emphasis added) says that reviews are "*the* process[es] under which static white-box testing is performed", van Vliet (2000, pp. 418–419) gives correctness proofs as another example.

32. "Computation data use" and "predicate data use" are usually abbreviated as the lowercase "c-use" and "p-use" (ISO/IEC and IEEE, 2021c, pp. 3, 27–29, 35–36, 114–155, 117–118, 129; 2017, p. 124; Peters and Pedrycz, 2000, p. 477, Tab. 12.6; Reid, 1996, Fig. 2), but van Vliet (2000, pp. 424–425) uses the uppercase "C-use" and "P-use" instead.

33. "Definition-use path" is usually abbreviated as the lowercase "du-path" (ISO/IEC and IEEE, 2021c, pp. 3, 27, 29, 35, 119–121, 129; Peters and Pedrycz, 2000, pp. 478–479; Reid, 1996, Fig. 2), but van Vliet (2000, p. 425) uses the uppercase "DU-path" instead.

34. Van Vliet (2000, pp. 424–425) specifies that every successor of a data definition use needs to be executed as part of all-uses testing, but this condition is not included elsewhere (ISO/IEC and IEEE, 2021c, pp. 28–29; 2017, p. 120; Peters and Pedrycz, 2000, pp. 478–479).

35. All-du-paths testing is usually defined as exercising all "loop-free control flow sub-paths from each variable definition to every use (both p-use and c-use) of that definition (with no intervening definitions)" (ISO/IEC and IEEE, 2021c, p. 29; similar in 2017, p. 125; Washizaki, 2025a, p. 5-13; Peters and Pedrycz, 2000, p. 479). However, van Vliet (2000, p. 425) says that paths containing simple cycles may also be required, in which case exercising all "loop-free control flow sub-paths" would be insufficient.

36. Van Vliet (2000, pp. 432–433) says that all-du-paths testing is only stronger than all-uses testing if there are infeasible paths, but Washizaki (2025a, p. 5-13) does not specify this caveat.

37. Acceptance testing is "usually performed by the purchaser … with the … vendor" (ISO/IEC and IEEE, 2017, p. 5), "may or may not involve the developers of the system" (Bourque and Fairley, 2014, p. 4-6), and/or "is often performed under supervision of the user organization" (van Vliet, 2000, p. 439); these descriptions of who the testers are contradict each other *and* all introduce some uncertainty ("usually", "may or may not", and "often", respectively).

Q #3: Does this merit counting this as an Ambiguity as well as a Contradiction?

38. Although ad hoc testing is classified as a "technique" (Washizaki, 2025a, p. 5-14), it is one in which "no recognized test design technique is used" (Kam, 2008, p. 42).

OG Beizer

39. Kam (2008, p. 46) says "negative testing is related to the testers' attitude rather than a specific test approach or test design technique"; while ISO/IEC and IEEE (2021c) seem to support this idea of negative testing being at a "higher" level than other approaches, they also imply that it is a test technique (pp. 10, 14).

D.1.4 Full List of Ambiguities

1. "Data definition" is defined as a "statement where a variable is assigned a value" (ISO/IEC and IEEE, 2021c, p. 3; 2017, p. 115; similar in 2012, p. 27; van Vliet, 2000, p. 424), but for functional programming languages such as Haskell with immutable variables (Wikibooks Contributors, 2023), this could cause confusion and/or be imprecise.

2. While Firesmith (2015) likely uses the hollow triangle to mean "subtype" (distinct from the notion of "test type" described in Table 2.1) following Unified Modeling Language (UML) notation (Dr. R. Paige, private communication, Oct. 14, 2025), he never explicitly specifies this notation.

3. The distinctions between development testing (ISO/IEC and IEEE, 2017, p. 136), developmental testing (Firesmith, 2015, p. 30), and developer testing (p. 39; Gerrard, 2000a, p. 11) are unclear and seem miniscule.

Is this a def flaw?

4. Hamburg and Mogyorodi (2024) define "Machine Learning (ML) model testing" and "ML functional performance" in terms of "ML functional performance criteria", which is defined in terms of "ML functional performance metrics", which is defined as "a set of measures that relate to the functional correctness of an ML system". The use of "performance" (or "correctness") in these definitions is at best ambiguous and at worst incorrect.

5. While "error" is defined as a "human action that produces an incorrect result" (ISO/IEC and IEEE, 2017, p. 165; 2010, p. 128; Washizaki, 2025a, p. 12-3[5]; van Vliet, 2000, p. 399), Washizaki does not use this consistently, sometimes implying that errors can be instrinsic to software itself (2025a, pp. 4-9, 6-5, 7-3, 12-4, 12-9, 12-13).

6. Washizaki (2025a, p. 4-11) says "*fault tolerance* is a collection of techniques that increase software reliability by detecting errors and then recovering from them or containing their effects if recovery is not possible". Since errors and faults are distinct (see Chapter 1), this should either be called "*error*

[5] Washizaki (2025a, p. 12-3) references the definition given in ISO/IEC and IEEE (2017, p. 165); while we would usually omit the former in favour of the original source, we include it here as an example of a flaw within a document.

tolerance" or be described as "detecting *faults*". However, the notion of "detecting errors and then *recovering from them* or containing their effects if recovery is not possible" seems to imply that Washizaki (2025a, p. 4-11, emphasis added) is really talking about *failures*: the visible effects of faults (IEEE, 2024, pp. 15, 37; Washizaki, 2025a, p. 5-3; Lyu, 1996, p. 12; van Vliet, 2000, p. 400).

7. While ergonomics testing is out of scope (as it tests hardware, not software; see Appendix A.1), its definition of "testing to determine whether a component or system and its input devices are being used properly with correct posture" (Hamburg and Mogyorodi, 2024) seems to focus on how the system is *used* as opposed to the system *itself*.

8. Hamburg and Mogyorodi (2024) define "end-to-end testing" as testing "in which business processes are tested from start to finish under production-like circumstances", but it is unclear whether this tests the business processes *themselves* or the *system* that performs them.

9. Hamburg and Mogyorodi (2024) describe the term "software in the loop" as a kind of testing, while the source they reference seems to describe "Software-in-the-Loop-Simulation" as a "simulation environment" that may support software integration testing (Knüvener Mackert GmbH, 2022, p. 153); is this a test approach or a tool that supports testing?

10. While model testing is said to test the object under test, it seems to describe testing the models themselves (Firesmith, 2015, p. 20); using the models to test the object under test seems to be called "driver-based testing" (p. 33).

11. "Tool/environment testing" could ambiguously refer to either testing the tools/environment *themselves* or *using* them to test the object under test; the wording of its subtypes (Firesmith, 2015, p. 25) seems to imply the former.

12. The Project Management Institute (2013, p. 244; similar on p. 535) says "quality assurance work will fall under the conformance work category in the cost of quality framework", but (Fig. 8-2) suggests that "conformance work" is a part of quality assurance. This introduces ambiguity at best and creates a directed cyclic parent-child relation (which violates our definition in Section 2.1.3) at worst.

13. Hamburg and Mogyorodi (2024) claim that code inspections are related to peer reviews but Patton (2006, pp. 94–95) makes them quite distinct.

14. Patton (2006, p. 119) says that branch testing is "the simplest form of path testing" which is also implied by ISO/IEC and IEEE (2021c, Fig. F.1) and van Vliet (2000, p. 433). This is true in the example Patton gives, but is not necessarily generalizable; one could test the behaviour at branches without testing even a *subset* of complete paths, which ISO/IEC and IEEE (2017, p. 316) give as a definition of "path testing" (see Contradiction 17)!

D.1.5 Full List of Overlaps

1. ISO/IEC and IEEE (2017, pp. 469, 470; 2013, p. 9) say that "test level" and "test phase" are synonyms, both meaning a "specific instantiation of [a] test sub-process", but they have other definitions as well. "Test level" can also refer to the scope of a test process; for example, "across the whole organization" or only "to specific projects" (2022, p. 24) and "test phase" can also refer to the "period of time in the software life cycle" when testing occurs (2017, p. 470), usually after the implementation phase (pp. 420, 509; Perry, 2006, p. 56).

2. ISO/IEC and IEEE (2010, p. 128) define "error" as "a human action that produces an incorrect result", but also as "an incorrect result" itself. Since faults are inserted when a developer makes an error (IEEE, 2024, p. 36; ISO/IEC and IEEE, 2010, pp. 128, 140; van Vliet, 2000, pp. 399–400; implied by ISO/IEC and IEEE, 2017, p. 179), this means that faults are *also* "incorrect results", incorrectly implying that "error" and "fault" are synonyms.

3. Besides being "a human action that produces an incorrect result" (ISO/IEC and IEEE, 2010, p. 128), "error" can also be defined as the "difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition" (2017, p. 165; 2010, p. 128; similar in Washizaki, 2025a, pp. 17-18 to 17-19, 18-7 to 18-8). While this is a widely used definition, particularly in mathematics, it makes some test approaches ambiguous. For example, back-to-back testing is "testing in which two or more variants of a program are executed with the same inputs, the outputs are compared, and errors are analyzed in case of discrepancies" (ISO/IEC and IEEE, 2010, p. 30; similar in Hamburg and Mogyorodi, 2024), which seems to refer to this definition of "error".

4. The SWEBOK Guide V4 defines "privacy testing" as testing that "assess[es] the security and privacy of users' personal data to prevent attacks" (Washizaki, 2025a, p. 5-9). This seems to overlap (both in scope and name) with the definition of "security testing" in (ISO/IEC and IEEE, 2022, p. 7): testing "conducted to evaluate the degree to which a test item, and associated data and information, are [sic] protected so that" only "authorized persons or systems" can use them as intended.

5. "Orthogonal array testing" (Washizaki, 2025a, pp. 5-1, 5-11; implied by Valcheva, 2013, pp. 467, 473; Yu et al., 2011, pp. 1573–1577, 1580) and "operational acceptance testing" (Firesmith, 2015, p. 30) have the same acronym ("OAT").

6. "Customer acceptance testing" and "contract(ual) acceptance testing" have the same acronym ("CAT") (Firesmith, 2015, p. 30).

7. "Hardware-in-the-loop testing" and "human-in-the-loop testing" have the same acronym ("HIL") (Firesmith, 2015, p. 23), although Preuße et al. (2012, p. 2) use "HiL" for the former.

D.1.6 Full List of Redundancies

1. ISO/IEC and IEEE (2021c, p. 4) define "exit point" as the "last executable statement within a test item", then later note that it "is most commonly the last executable statement within the test item".

2. ISO/IEC and IEEE (2017, p. 375) say that "dependability characteristics include availability and its inherent or external influencing factors, such as availability".

3. ISO/IEC and IEEE (2017, p. 228) provide a definition for "inspections and audits", despite also giving definitions for "inspection" (p. 227) and "audit" (p. 36); while the first term *could* be considered a superset of the latter two, this distinction doesn't seem useful.

4. Including "testing" in the term "ethical hacking test[ing]" (Washizaki, 2025a, p. 13-5) is redundant since the clearer term "ethical hacking" used by Gerrard (2000b, p. 28) already indicates that this is an activity to be performed.

5. Hamburg and Mogyorodi (2024) define "specification-based testing" circularly as "testing based on an analysis of the specification of the component or system".

6. The phrase "continuous automated testing" (Gerrard, 2000a, p. 11) is redundant since Continuous Testing (CT) is already a subapproach of automated testing (ISO/IEC and IEEE, 2022, p. 35; Hamburg and Mogyorodi, 2024).

D.2 Full Lists of Flaws by Domain

The following sections provide all of the data that we automatically detect (as described in Section 4.2.1) and summarize in Section 5.2.

D.2.1 Multiple Categorizations

As mentioned in Section 5.2.1, we automatically detect test approaches with more than one category that violate our assumption of orthogonality (see Section 2.1.1) and list them in Table D.1.

Table D.1: Test approaches with more than one category.

Approach	Category 1	Category 2
Ad Hoc Testing	Practice (ISO/IEC and IEEE, 2013, p. 33)	Technique (Washizaki, 2025a, p. 5-14)
Capacity Testing	Technique (ISO/IEC and IEEE, 2021c, p. 38–39)	Type (ISO/IEC and IEEE, 2022, p. 22; 2013, p. 2; implied by its quality (ISO/IEC, 2023a; ISO/IEC and IEEE, 2021c, Tab. A.1); Firesmith, 2015, p. 53)
Checklist-based Testing	Practice (ISO/IEC and IEEE, 2022, p. 34)	Technique (Hamburg and Mogyorodi, 2024)
Data-driven Testing	Practice (ISO/IEC and IEEE, 2022, p. 22)	Technique (Kam, 2008, p. 43; OG Fewster and Graham)
End-to-end Testing	Type (Hamburg and Mogyorodi, 2024)	Technique (Firesmith, 2015, p. 47; Sharma et al., 2021, pp. 601, 603, 605–606)

Continued on next page

Table D.1: Test approaches with more than one category. (Continued)

Approach	Category 1	Category 2
Endurance Testing	Technique (ISO/IEC and IEEE, 2021c, p. 38–39)	Type (ISO/IEC and IEEE, 2013, p. 2; implied by Firesmith, 2015, p. 55)
Error Guessing	Practice (ISO/IEC and IEEE, 2013, p. 33)	Technique (ISO/IEC and IEEE, 2022, pp. 4, 34, Fig. 2; 2021c, pp. iii–iv, 4, 11, 29, 35, 122, 125, Fig. 2, Tab. A.2; 2013, pp. 3, 33; Washizaki, 2025a, p. 5-13; Firesmith, 2015, p. 50)
Experience-based Testing	Technique (ISO/IEC and IEEE, 2022, Fig. 2; Firesmith, 2015, pp. 46, 50)	Practice (ISO/IEC and IEEE, 2022, Fig. 2; 2021c, p. viii; 2013, pp. iii, 31, 33)
Exploratory Testing	Technique (Washizaki, 2025a, pp. 5-13 to 5-14; Firesmith, 2015, p. 50)	Practice (ISO/IEC and IEEE, 2022, pp. 11, 20, 34, Fig. 2; 2021a, p. 5; 2021c, p. viii; 2013, pp. 13, 33; implied by 2022, p. 33)
Functional Testing	Technique (Barbosa et al., 2006, p. 3; inferred from specification-based testing)	Type (ISO/IEC and IEEE, 2022, pp. 15, 20, 22; 2021a, pp. 8, 11; 2021b, p. 41; 2021c, pp. 7, 38, Tab. A.1; 2017, p. 473; 2016, p. 4; Hamburg and Mogyorodi, 2024; Koomen et al., 2006, p. 50; implied by the quality of "correctness" (ISO/IEC and IEEE, 2017, p. 104; Washizaki, 2025a, p. 3-13))
(Code) Inspections	Technique (ISO/IEC and IEEE, 2017, p. 227)	Level (Washizaki, 2025a, p. 5-13)

Continued on next page

Table D.1: Test approaches with more than one category. (Continued)

Approach	**Category 1**	**Category 2**
Load Testing	Technique (ISO/IEC and IEEE, 2021c, p. 38–39)	Type (ISO/IEC and IEEE, 2022, pp. 5, 20, 22; 2017, p. 253; OG IEEE 2013; Hamburg and Mogyorodi, 2024; implied by Firesmith, 2015, p. 54)
Model-based Testing	Technique (Souza et al., 2017, p. 3; implied by ISO/IEC and IEEE, 2017, p. 469)	Practice (ISO/IEC and IEEE, 2022, p. 11, Fig. 2; 2021a, p. 5; 2021c, p. viii; 2013, pp. iii, 31)
Penetration Testing	Technique (ISO/IEC and IEEE, 2021c, p. 40; Hamburg and Mogyorodi, 2024)	Type (ISO/IEC and IEEE, 2021b, pp. 41, 43; implied by Firesmith, 2015, p. 57; inferred from security testing)
Performance Testing	Technique (ISO/IEC and IEEE, 2021c, p. 38–39)	Type (ISO/IEC and IEEE, 2022, pp. 7, 22, 26–27; 2021a, pp. 2, 8, 11; 2021b, pp. 41, 43; 2021c, p. 7; Koomen et al., 2006, pp. 9, 50; implied by Firesmith, 2015, p. 53)
Regression Testing	Level (Lyu, 1996, p. 532; Barbosa et al., 2006, p. 3)	Type (Hamburg and Mogyorodi, 2024; Koomen et al., 2006, pp. 9, 51)
Specification-based Testing	Type (Hamburg and Mogyorodi, 2024)	Technique (ISO/IEC and IEEE, 2022, p. 22; 2021b, p. 45; 2021c, p. 8; Washizaki, 2025a, p. 5-10; Hamburg and Mogyorodi, 2024; Firesmith, 2015, pp. 46–47; Souza et al., 2017, p. 3; Sakamoto et al., 2013, p. 344; implied by ISO/IEC and IEEE, 2022, pp. 2–4, 6–9)

Continued on next page

Table D.1: Test approaches with more than one category. (Continued)

Approach	Category 1	Category 2
Stress Testing	Technique (ISO/IEC and IEEE, 2021c, p. 38–39)	Type (ISO/IEC and IEEE, 2022, pp. 9, 22; 2017, p. 442; implied by Firesmith, 2015, p. 54)
Structure-based Testing	Type (Hamburg and Mogyorodi, 2024)	Technique (ISO/IEC and IEEE, 2022, p. 22; 2021b, p. 45; 2021c, p. 8; Washizaki, 2025a, pp. 5-10, 5-12; Hamburg and Mogyorodi, 2024; Firesmith, 2015, pp. 46, 49; Souza et al., 2017, p. 3; Sakamoto et al., 2013, p. 344; implied by ISO/IEC and IEEE, 2022, pp. 2, 4, 6, 9; Barbosa et al., 2006, p. 3)
Alpha Testing	Type (implied by Firesmith, 2015, p. 58)	Level (ISO/IEC and IEEE, 2022, p. 22; inferred from acceptance testing)
Attacks	Practice (ISO/IEC and IEEE, 2022, p. 34; 2013, p. 33)	Technique (implied by Hamburg and Mogyorodi, 2024)
Beta Testing	Type (implied by Firesmith, 2015, p. 58)	Level (ISO/IEC and IEEE, 2022, p. 22; inferred from acceptance testing)
Integration Testing	Technique (implied by Sharma et al., 2021, pp. 601, 603, 605–606)	Level (ISO/IEC and IEEE, 2022, pp. 12, 20–22, 26–27; 2021a, pp. 6, 11; 2021b, Fig. 2, pp. 41, 43, 51; 2021c, p. 6; Washizaki, 2025a, p. 5-7; Hamburg and Mogyorodi, 2024; Lyu, 1996, p. 532; Black, 2009, pp. 12, 14, 24, 178; Peters and Pedrycz, 2000, Tab. 12.3; van Vliet, 2000, p. 438; Hetzel, 1988, pp. 11, 122, 134, 149, 171; Souza et al., 2017, p. 3; Sakamoto et al., 2013, p. 343; Barbosa et al., 2006, p. 3)
Interface Testing	Type (Kam, 2008, p. 45)	Level (implied by ISO/IEC and IEEE, 2017, p. 235; Sakamoto et al., 2013, p. 343; inferred from integration testing)

Continued on next page

Table D.1: Test approaches with more than one category. (Continued)

Approach	**Category 1**	**Category 2**
Procedure Testing	Technique (implied by Firesmith, 2015, p. 47)	Type (ISO/IEC and IEEE, 2022, pp. 7, 22; 2021c, p. 39, Tab. A.1; 2017, p. 337; OG IEEE, 2013)
Survivability Testing	Technique (Ghosh and Voas, 1999, p. 39)	Type (implied by its quality (ISO/IEC, 2011); inferred from robustness testing and security testing)
Unit Testing	Technique (implied by Engström and Petersen, 2015, pp. 1–2)	Level (ISO/IEC and IEEE, 2022, pp. 12, 20–22, 26–27; 2021a, pp. 6, 11; 2021b, Fig. 2, pp. 41, 43, 51; 2021c, p. 6; 2017, p. 467; 2016, p. 4; Washizaki, 2025a, p. 5-6; Hamburg and Mogyorodi, 2024; Lyu, 1996, p. 532; Black, 2009, pp. 12, 14, 18, 24; Peters and Pedrycz, 2000, Tab. 12.3; van Vliet, 2000, p. 438; Hetzel, 1988, pp. 9–10, 32, 73, 134; Souza et al., 2017, p. 3; Sakamoto et al., 2013, p. 343; Barbosa et al., 2006, p. 3)
Volume Testing	Technique (ISO/IEC and IEEE, 2021c, p. 38–39)	Type (implied by Firesmith, 2015, p. 54; inferred from performance-related testing)
Infrastructure Testing	Type (implied by Firesmith, 2015, p. 57)	Level (implied by Gerrard, 2000a, p. 13; see Table 2.1)

D.2.2 Intransitive Synonyms

As described in Section 2.2.2, intransitive synonyms are examples of flaws since they violate our definition of the synonym relation in Section 2.1.2. We identify 13 such cases through automatic analysis of generated visualizations, listed below (test approaches in *italics* are synonyms with each other, but not with other terms not in italics):

Better way to handle/display this?

1. **Functional Testing:**
   - Specification-based Testing (ISO/IEC and IEEE, 2017, p. 196; van Vliet, 2000, p. 399; Kam, 2008, pp. 44–45, 48; implied by ISO/IEC and IEEE, 2021c, p. 129; 2017, p. 431)
   - *Conformance Testing* (Washizaki, 2025a, p. 5-7; implied by ISO/IEC and IEEE, 2017, p. 93)
   - *Correctness Testing* (Washizaki, 2025a, p. 5-7)
2. **Portability Testing:**
   - Flexibility Testing (ISO/IEC, 2023a)
   - Configuration Testing (Kam, 2008, p. 43)
3. **Smoke Testing:**
   - Build Verification Testing (Washizaki, 2025a, p. 5-14)
   - Intake Testing (Hamburg and Mogyorodi, 2024)
4. **Specification-based Testing:**
   - Functional Testing (ISO/IEC and IEEE, 2017, p. 196; van Vliet, 2000, p. 399; Kam, 2008, pp. 44–45, 48; implied by ISO/IEC and IEEE, 2021c, p. 129; 2017, p. 431)
   - Domain Testing (Washizaki, 2025a, p. 5-10)
5. **Soak Testing:**
   - Endurance Testing (ISO/IEC and IEEE, 2021c, p. 39)
   - Reliability Testing[6] (Gerrard, 2000a, Tab. 2; 2000b, Tab. 1, p. 26)
6. **Condition Testing:**
   - Branch Condition Testing (Hamburg and Mogyorodi, 2024)
   - Branch Condition Combination Testing (Patton, 2006, p. 120; Sharma et al., 2021, Fig. 1)
   - Decision Testing (implied by Washizaki, 2025a, p. 5-13)

[6] Endurance testing is given as a child of reliability testing by Firesmith (2015, p. 55), although the terms are not synonyms.

7. **Link Testing:**
   - Component Integration Testing (Kam, 2008, p. 45)
   - Branch Testing (implied by ISO/IEC and IEEE, 2021c, p. 24; Reid, 1996, p. 4)
   - Integration Testing (implied by Gerrard, 2000a, p. 13)
8. **Exhaustive Testing:**
   - Branch Condition Combination Testing (if "each subcondition is viewed as a single input") (Peters and Pedrycz, 2000, p. 464)
   - Path Testing[7] (incorrectly) (van Vliet, 2000, p. 421)
9. **Orthogonal Array Testing:**
   - t-wise Testing (Washizaki, 2025a, p. 5-11; implied by Valcheva, 2013, p. 473)
   - Pairwise Testing (implied by Valcheva, 2013, p. 473)
10. **Performance Testing:**
    - Performance-related Testing (Moghadam, 2019, p. 1187)
    - Performance Efficiency Testing (implied by Hamburg and Mogyorodi, 2024)
11. **Static Verification:**
    - Static Assertion Checking[8] (incorrectly) (implied by Chalin et al., 2006, p. 343)
    - Static Testing (implied by Chalin et al., 2006, p. 343)
12. **Testing-to-Fail:**
    - Negative Testing (Patton, 2006, pp. 67, 78, 84–87)
    - Forcing Exception Testing (implied by Patton, 2006, pp. 66–67, 78)
13. **Invalid Testing:**
    - Negative Testing (Hamburg and Mogyorodi, 2024; implied by ISO/IEC and IEEE, 2021c, p. 10)
    - Error Tolerance Testing (implied by Kam, 2008, p. 45)

[7] See Section 5.2.3 for more detail on why this synonym relation is incorrect.

[8] See Mistake 29 for more detail on why this synonym relation is incorrect.

D.2.3 Synonym and Parent-Child Overlaps

As described in Section 5.2.3, there are also pairs of synonyms where one is a subapproach of the other; these relations cannot coexist since synonym relations are symmetric while parent-child relations are asymmetric (as outlined in Sections 2.1.2 and 2.1.3, respectively). Below are all 17 of these pairs that we identify through automatic analysis of our generated visualizations as described in Section 4.2.1.

Table D.2: Pairs of test approaches with a parent-child *and* synonym relation.

“Child” → “Parent”	**Parent-Child Source(s)**	**Synonym Source(s)**
Domain Testing → Specification-based Testing	(Peters and Pedrycz, 2000, Tab. 12.1)	(Washizaki, 2025a, p. 5-10)
Fault Tolerance Testing → Robustness Testing[a]	(Firesmith, 2015, p. 56)	(Hamburg and Mogyorodi, 2024)
Functional Testing → Specification-based Testing[b]	(ISO/IEC and IEEE, 2021c, p. 38; Kam, 2008, p. 42; implied by Washizaki, 2025a, p. 5-7)	(ISO/IEC and IEEE, 2017, p. 196; van Vliet, 2000, p. 399; Kam, 2008, pp. 44–45, 48; implied by ISO/IEC and IEEE, 2021c, p. 129; 2017, p. 431)
Performance Testing → Performance-related Testing	(ISO/IEC and IEEE, 2022, p. 22; 2021c, p. 38)	(Moghadam, 2019, p. 1187)
Static Analysis → Static Testing	(ISO/IEC and IEEE, 2022, pp. 9, 17, 25, 28; 2021a, pp. 3, 21; Hamburg and Mogyorodi, 2024; Gerrard, 2000a, Fig. 4, p. 12; 2000b, p. 3)	(Peters and Pedrycz, 2000, p. 438)
Structural Testing → Structure-based Testing[d]	(Patton, 2006, pp. 105–121; Peters and Pedrycz, 2000, p. 447)	(ISO/IEC and IEEE, 2022, p. 9; 2017, pp. 443–444; Hamburg and Mogyorodi, 2024; implied by Barbosa et al., 2006, p. 3)

Continued on next page

Table D.2: Pairs of test approaches with a parent-child *and* synonym relation. (Continued)

"Child" → "Parent"	**Parent-Child Source(s)**	**Synonym Source(s)**
Use Case Testing → Scenario Testing[c]	(ISO/IEC and IEEE, 2021c, p. 20; OG Hass, 2008)	(Hamburg and Mogyorodi, 2024)
Co-existence Testing → Compatibility Testing	(ISO/IEC, 2023a; ISO/IEC and IEEE, 2022, p. 3; 2021c, Tab. A.1)	(incorrectly) (ISO/IEC and IEEE, 2021c, p. 37)
Landmark Tours → Tours	(ISO/IEC and IEEE, 2022, p. 34)	(implied by ISO/IEC and IEEE, 2022, p. 34)
Pairwise Testing → Orthogonal Array Testing	(Washizaki, 2025a, p. 5-11; Valcheva, 2013, p. 473; implied by Mandl, 1985, p. 1055)	(implied by Valcheva, 2013, p. 473)
Path Testing → Exhaustive Testing	(Peters and Pedrycz, 2000, pp. 466–467, 476; implied by Patton, 2006, pp. 120–121)	(incorrectly) (van Vliet, 2000, p. 421)
Reviews → Structural Analysis	(Patton, 2006, p. 92)	(implied by Patton, 2006, p. 92)
Smoke Testing → Build Verification Testing	(implied by Hamburg and Mogyorodi, 2024)	(Washizaki, 2025a, p. 5-14)

Continued on next page

Table D.2: Pairs of test approaches with a parent-child *and* synonym relation. (Continued)

"Child" → "Parent"	**Parent-Child Source(s)**	**Synonym Source(s)**
Exploratory Testing → Unscripted Testing	(ISO/IEC and IEEE, 2022, p. 33; 2021a, p. 2; 2017, p. 174; Firesmith, 2015, p. 45; implied by Washizaki, 2025a, p. 5-14)	(implied by Kuļešovs et al., 2013, p. 214)
Beta Testing → User Testing	(implied by Firesmith, 2015, p. 39)	(implied by Firesmith, 2015, p. 39)
Branch Condition Combination Testing → Exhaustive Testing	(implied by Patton, 2006, p. 121)	(if "each subcondition is viewed as a single input") (Peters and Pedrycz, 2000, p. 464)
Performance Efficiency Testing → Performance Testing	(implied by ISO/IEC and IEEE, 2017, p. 319)	(implied by Hamburg and Mogyorodi, 2024)

[a] Fault tolerance testing may also be a subapproach of reliability testing (ISO/IEC and IEEE, 2017, p. 375; Washizaki, 2025a, p. 7-10), which is distinct from robustness testing (Firesmith, 2015, p. 53).

[b] Hamburg and Mogyorodi (2024) cite ISO/IEC and IEEE (2017, p. 431) for their definition of "functional testing" but exclude the transitive synonym relationship (see Section 2.1.2) they give to "specification-based testing". These terms are also defined separately elsewhere (2022, Fig. 2; 2021c, pp. 8, 49, 125), further supporting that they are not synonyms.

[c] ISO/IEC and IEEE (2022, Fig. 2) also list "use case testing" and "scenario testing" separately, further supporting that these terms are not synonyms.

[d] Peters and Pedrycz (2000, p. 447) claim that "structural testing subsumes white box testing", but also say "structure tests are aimed at exercising the internal logic of a software system" and "in white box testing …, using detailed knowledge of code, one creates a battery of tests in such a way that they exercise all components of the code (say, statements, branches, paths)" on the same page!

D.3 Inferred Flaws

Throughout our research, we infer many potential flaws as described in Section 2.3. Some of these have a conflicting source while others do not. Since these are more subjective and are based on our own judgement, we exclude them from any counts of the numbers of flaws but give them here for completeness.

D.3.1 Inferred Multiple Categorizations

As mentioned in Section 5.2.1, we automatically detect test approaches with more than one category that violate our assumption of orthogonality (see Section 2.1.1). This includes those with categories that we infer based on our assumption that child approaches inherit their parents' categories (as described in Section 2.3). We list these approaches and their given and inferred categories (along with their relevant parent in parentheses) in Table D.3.

Table D.3: Test approaches inferred to have more than one category.

Approach	Given Category	Inferred Category
A/B Testing	Practice (ISO/IEC and IEEE, 2022, Fig. 2)	Type (usability testing)
Big-Bang Testing	Technique (Sharma et al., 2021, pp. 601, 603, 605–606)	Level (integration testing)
Bottom-Up Testing	Technique (Sharma et al., 2021, pp. 601, 603, 605–606)	Level (integration testing)
Fuzz Testing	Technique (Hamburg and Mogyorodi, 2024; Firesmith, 2015, p. 51)	Practice (mathematical-based testing)
Memory Management Testing	Technique (ISO/IEC and IEEE, 2021c, p. 38–39)	Type (performance-related testing)
Privacy Testing	Technique (ISO/IEC and IEEE, 2021c, p. 40)	Type (security testing)
Sandwich Testing	Technique (Sharma et al., 2021, pp. 601, 603, 605–606)	Level (integration testing)
Closed Beta Testing	Type (implied by Firesmith, 2015, p. 58)	Level (beta testing)
Open Beta Testing	Type (implied by Firesmith, 2015, p. 58)	Level (beta testing)

D.3.2 Inferred Intransitive Synonyms

In addition to the 13 cases of intransitive synonyms we list in Appendix D.2.2, we infer some synonym relations based on the terms' definitions that create the following inferred flaws (some pairs of synonyms have sources; those that do not are ones we infer):

1. **Structure-based Testing:**
    - Structural Testing (ISO/IEC and IEEE, 2022, p. 9; 2017, pp. 443–444; Hamburg and Mogyorodi, 2024; implied by Barbosa et al., 2006, p. 3)
    - Implementation-oriented Testing
2. **Production Acceptance Testing:**
    - Operational (Acceptance) Testing (Hamburg and Mogyorodi, 2024)[9]
    - Production Verification Testing[10]
3. **Operational (Acceptance) Testing:**
    - Production Acceptance Testing (Hamburg and Mogyorodi, 2024)
    - Field Testing
    - Qualification Testing
4. **Customer Product Integration Testing:**
    - Customer Acceptance Testing
    - Usability Testing (implied by Washizaki, 2025a, p. 5-10)
5. **System Testing:**
    - System Qualification Testing
    - Systems Testing (implied by Hetzel, 1988, pp. 73, 121–122, 132–134, 139, 280)
6. **Systems Integration Testing:**
    - Large Scale Integration Testing (Gerrard, 2000a, p. 13)
    - Integrated System Testing

[9] See Contradiction 23.

[10] "Production acceptance testing" (Firesmith, 2015, p. 30) seems to be the same as "production verification testing" (ISO/IEC and IEEE, 2022, p. 22) but neither is defined.

D.3.3 Inferred Parent Flaws

As discussed in Section 5.2.3, some pairs of synonyms also have a parent-child relation, abusing the meaning of "synonym" and causing confusion. While Table D.2 gives the cases where both relations are supported by the literature, some are less explicit. The following automatically generated lists contain examples where at least one of these conflicting relations is *not* explicitly supported by the literature but may, nonetheless, be correct. The relations in the first two lists are explicitly given in the literature but may be incorrect, while those in the third list are unsubtantiated by the literature and require more thought before a recommendation can be made.

Pairs given a parent-child relation

1. Programmer Testing → Developer Testing (Firesmith, 2015, p. 39)

Pairs given a synonym relation

1. Dynamic Analysis → Dynamic Testing[11] (Peters and Pedrycz, 2000, p. 438; implied by ISO/IEC and IEEE, 2017, p. 149; Hamburg and Mogyorodi, 2024)
2. Qualification Testing → Acceptance Testing (Bourque and Fairley, 2014, p. 4-6)
3. Structured Walkthroughs → Walkthroughs[12] (Hamburg and Mogyorodi, 2024)
4. Functionality Testing → Functional Suitability Testing (although this seems wrong) (implied by Hamburg and Mogyorodi, 2024)

Pairs that could have a parent/child *or* synonym relation

1. Computation Flow Testing → Computation Testing
2. Field Testing → Operational Testing
3. Organization-based Testing → Role-based Testing[13]
4. System Qualification Testing → System Testing

In addition to these flaws, Gerrard (2000a, Tab. 2) does *not* give "functionality testing" as a parent of "end-to-end functionality testing" as we infer he should (see Appendix A.6).

[11] The proposed relation is inferred from its static counterpart (ISO/IEC and IEEE, 2022, pp. 9, 17, 25, 28; 2021a, pp. 3, 21; Hamburg and Mogyorodi, 2024; Gerrard, 2000a, Fig. 4, p. 12; 2000b, p. 3).

[12] See Contradiction 30.

[13] The distinction between organization- and role-based testing in (Firesmith, 2015, pp. 17, 37, 39) seems arbitrary, but further investigation may prove it to be meaningful; we discussed this in #59.

Appendix E

Approaches to Investigate Further

As outlined in Chapter 7, there are many test approaches with missing data due to time constraints. The following are full lists of approaches of test approaches that are missing definitions (Appendix E.1), significant relations to other approaches (Appendix E.2), and/or a more specific category than "Approach" (Appendix E.3).

E.1 Full List of Undefined Test Approaches

The following is a list of all 197 undefined test approaches over which we would iterate if time allowed (as described in Section 7.1):

1. Absolute correctness testing
2. Access control testing
3. Acquisition organization testing
4. Agent-based testing
5. Algebraic testing
6. All-input-GUI testing
7. All-round-trip-paths testing
8. All-transition-k-tuples testing
9. Anomaly testing
10. Anti-spoofing testing
11. Anti-tamper testing
12. Architect testing
13. At-the-beginning testing
14. Axiomatic testing
15. Backwards compatibility testing
16. Baseline testing
17. Behaviour analysis?
18. Block testing
19. Blue team testing
20. Bug hunt testing
21. Built-in testing
22. Business acceptance testing
23. Checked statement testing
24. Closed beta testing
25. Cloud testing
26. (Stepwise) code reading
27. Comparison testing
28. Compiler testing
29. Compiler-based testing

30. Complete regression testing
31. Computation testing
32. Concrete execution
33. Conditional testing
34. Content usage testing
35. Continuity testing
36. COTS testing
37. COTS vendor testing
38. Customer acceptance testing
39. Customer product integration testing
40. Dark launches
41. Data center testing
42. Data dependence transition relation testing
43. Data generation (testing)
44. Data migration testing
45. Database admin testing
46. Database testing
47. Deterministic testing
48. Development environment testing
49. Development organization testing
50. Development tool testing
51. Domain testing
52. Domain-based testing
53. Domain-independent testing
54. Domain-specific testing
55. DT organization testing
56. Embedded tester testing
57. Encryption testing
58. Expression testing
59. External links integration (testing)
60. Extreme value testing
61. E-business testing
62. Factory acceptance testing
63. Failure tolerance testing
64. Fault sensitivity testing
65. Feature interaction testing
66. Feature-based testing
67. Features testing
68. Follow-on operational testing
69. Formal methods
70. Formal modular verification
71. Functional configuration audit
72. Functions testing
73. Galumphing
74. Group testing
75. Heartbeat
76. High frequency testing
77. Human factors engineer testing
78. Human-in-the-loop testing
79. Independent test organization testing
80. Independent tester testing
81. Individual testing
82. Industrial web application testing
83. Infection-oriented testing

84. Initial operational testing
85. Initial testing
86. Input domain testing
87. Intake testing
88. Integrated system testing
89. Interrupt-driven built-in testing
90. Layer-based testing
91. Legacy system integration (testing)
92. Legacy testing
93. Lifecycle-based testing
94. Linear code sequence and jump testing
95. Link discoverability (testing)
96. Link dependence transition relation testing
97. Load balancing testing
98. Local testing
99. Machine learning-assisted testing
100. Menu item testing
101. Method testing
102. Migration testing
103. (Flash) mob testing
104. Mobile testing
105. Model verification
106. Multi-user testing
107. Mutation analysis
108. Network admin testing
109. Network traffic testing
110. Nominal testing
111. Object-based testing
112. Object-oriented testing
113. Off-nominal testing
114. Ongoing built-in testing
115. OO web testing
116. Open beta testing
117. Open loop testing (control flow)
118. Open loop testing (control systems)
119. Open source testing
120. Operational effectiveness testing
121. Operational suitability testing
122. Operations organization testing
123. Operator testing
124. Organization-based testing
125. OT organization testing
126. Output domain testing
127. Outside-in testing
128. Partial regression testing
129. Patterns-based testing
130. Periodic built-in testing
131. Personalization testing
132. Perturbation testing
133. Power-up built-in testing
134. Prime contractor testing
135. Prime path testing
136. Probable correctness testing
137. Processor-in-the-loop testing

138. Product lines testing
139. Production acceptance testing
140. Production verification testing
141. Prognostics and health management
142. Programmer testing
143. Propagation-oriented testing
144. Protection system testing
145. Qualification operational testing
146. Red team testing
147. Relative correctness testing
148. Reliability enhancement testing
149. Reliability growth testing
150. Reliability mechanism testing
151. Remote testing
152. Request testing
153. Requirements animation
154. Requirements engineer testing
155. Reuse testing
156. ReWeb testing
157. Role-based testing
158. Safety engineer testing
159. Scenario walkthroughs
160. Scenario-based evaluations
161. Scenario-based testing
162. Search-based testing
163. Security engineer testing
164. Self-testing
165. Shift-left testing
166. Shoe testing
167. Shutdown built-in testing
168. SOA testing
169. Software interaction testing
170. State-based web browser testing
171. Stepwise abstraction
172. Structure-oriented testing
173. Structured testing
174. Stuck key testing
175. Subcontractor testing
176. Subsystem testing
177. Sys admin testing
178. Systems testing
179. Test environment testing
180. Test tool testing
181. Tester testing
182. TestWeb testing
183. Time-domain-based testing
184. Transaction flow testing
185. Translation validation
186. UI testing
187. UML model-based testing
188. Usability reviews
189. User as tester testing
190. User interface navigation testing
191. User organization testing
192. User session data testing

193. User session testing
194. User testing
195. User-initiated built-in testing
196. User-session-based testing
197. WebApp slicing

E.2 Full List of Orphan Approaches

The following is a list of all 37 orphan approaches over which we could iterate if time allowed (as described in Section 7.3):

1. Audio testing
2. Baseline testing
3. Canary testing
4. Capability testing
5. Command-Line Interface (CLI) testing
6. Consistency testing
7. Crowd testing
8. Dark launches
9. Design-driven testing
10. Desk checking
11. Deterministic testing
12. Device-based testing
13. Ergonomics testing
14. E-business testing
15. Fault seeding
16. High-level testing
17. Hypothesis testing
18. Individual testing
19. Initial testing
20. Insourced testing
21. Loopback testing
22. Low-level testing
23. Machine learning-assisted testing
24. Needs-driven testing
25. Nominal testing
26. Off-nominal testing
27. Patterns-based testing
28. Player perspective testing
29. Product lines testing
30. Search-based testing
31. SOA testing
32. Spike testing
33. Technical reviews
34. Test tool testing
35. User-agent based testing
36. Validation testing
37. Verification testing

E.3 Full List of Uncategorized Approaches

The following is a list of all 160 uncategorized approaches over which we could iterate if time allowed (as described in Section 7.3):

1. Ad hoc reviews
2. Agent-based testing
3. Algebraic testing
4. Anomaly testing
5. Application Programming Interface (API) testing
6. Application system testing
7. Architecture-driven testing
8. Audio testing
9. Axiomatic testing
10. Backup testing
11. Behaviour analysis?
12. Boundary condition testing
13. Browser page testing
14. Capability testing
15. Certification
16. Common Gateway Interface (CGI) component testing
17. Change-related testing
18. Command-Line Interface (CLI) testing
19. Cloud testing
20. Code injection
21. (Stepwise) code reading
22. Code reviews
23. Comparison testing
24. Compiler testing
25. Compiler-based testing
26. Complete regression testing
27. Computation flow testing
28. Computation testing
29. Concurrency testing
30. Conditional testing
31. Construction testing
32. Content checking
33. Control flow analysis
34. Control system testing
35. Cookie testing
36. COTS testing
37. Cross-browser compatibility testing
38. Crowd testing
39. Data flow analysis
40. Data generation (testing)
41. Data integrity testing
42. Database integrity testing
43. Denial of service
44. Design-driven testing
45. Distributed testing
46. Domain analysis
47. Domain testing
48. Dynamic analysis

49. Dynamic testing
50. Ergonomics testing
51. Error-oriented testing
52. Exhaustive testing
53. Expert usability reviews
54. Expression testing
55. E-business testing
56. Failover testing
57. Fault injection testing
58. Fault sensitivity testing
59. Feature-based testing
60. Features testing
61. Field testing
62. Formal methods
63. Formal modular verification
64. Formative evaluations
65. Group testing
66. GUI testing
67. High-level testing
68. Hypothesis testing
69. Implementation-oriented testing
70. In-container testing
71. Individual testing
72. Industrial web application testing
73. Infection-oriented testing
74. Initial testing
75. Input domain testing
76. Intake testing
77. Layer-based testing
78. Legacy testing
79. Link checking
80. Link discoverability (testing)
81. Load balancing testing
82. Low-level testing
83. Malware scanning
84. Menu item testing
85. Minimized testing
86. Model verification
87. Multiplayer testing
88. Multi-user testing
89. Mutation analysis
90. Needs-driven testing
91. Network traffic testing
92. Nominal testing
93. Object-based testing
94. Off-nominal testing
95. One-to-one testing
96. OO web testing
97. Operational profile testing
98. Output domain testing
99. Partial regression testing
100. Password cracking
101. Peer reviews
102. Perturbation testing
103. Pharming
104. Power testing

105. Probable correctness testing
106. Product lines testing
107. Propagation-oriented testing
108. Protection system testing
109. Quality assurance
110. Requirements animation
111. Requirements-driven testing
112. Reuse testing
113. Reviews
114. ReWeb testing
115. Role-based reviews
116. Safety demonstrations
117. Scenario walkthroughs
118. Scenario-based testing
119. Scenario-based reviews
120. Security attacks
121. Self-testing
122. Session-based testing
123. Sign change testing
124. Sign-sign testing
125. SOA testing
126. Software interaction testing
127. Spike testing
128. Static analysis
129. Structure-oriented testing
130. Summative evaluation(s)
131. Syntactic testing
132. Technical reviews
133. Technical testing
134. Template variable testing
135. Test browsing
136. TestUML testing
137. TestWeb testing
138. Textual testing
139. Threshold testing
140. Time-domain-based testing
141. Transaction testing
142. Transaction verification
143. UI testing
144. Unscripted testing
145. Usability reviews
146. Usability walkthroughs
147. User session data testing
148. User session testing
149. User story testing
150. User surveys
151. User-agent based testing
152. User-based evaluations
153. User-oriented testing
154. User-session-based testing
155. Validation testing
156. Verification testing
157. Visual browser validation
158. Visual testing
159. Web application testing
160. WebApp slicing

Appendix F

Preliminary Recommendations

As shown in Chapter 5, the software testing literature is quite flawed. Due to the sheer number of flaws and the size of the domain itself, it will take a lot of time, effort and expertise to organize these terms (and their relations) logically. However, the hardest step is often the first one, so we give some examples of how these flaws can be resolved. These changes arise when we notice an issue with the current state of the terminology and think about what *we* would do to make it better. We do not claim that these are correct, unbiased, or exclusive, just that they can be used as an inspiration for those wanting to pick up where we leave off.

When redefining test approaches, we seek to make them:

1. **Atomic:** Each term should only define one thing.

2. **Distinct:** Each term should be meaningfully separate from others and not overlap with them.

3. **Consistent:** Each piece of data associated with a term should be cohesive and not contradict others.

4. **Intuitive:** Each term's definition and relations should follow logically from, or at least be consistent with, the term's name.

Likewise, we seek to eliminate classes of flaws that can be detected automatically, such as test approaches that are given as synonyms to multiple distinct approaches (Appendix D.2.2) or as parents of themselves (Section 5.2.3), or pairs of approaches with both a parent-child *and* synonym relation (Section 5.2.3 and Appendix D.2.3).

We give recommendations for each "subset" of testing that we describe in Section 5.3—operational (acceptance) testing (Appendix F.1), recovery testing (Appendix F.2), scalability testing (Appendix F.3), and compatibility testing (Appendix F.5)—as well as the "superset" of performance-related testing (Appendix F.4). We provide graphical representations (see Section 4.1) of these subsets when helpful in Figures 7.1, F.1, and F.2, in which arrows representing relations between approaches are coloured based on the source tier (see Section 2.5) that defines them. We colour all proposed approaches and relations orange. We also include inferred approaches and relations in grey for completeness, although they are not explicitly given in the literature (see Section 2.3).

F.1 Operational (Acceptance) Testing

find more academic sources

Since this terminology is not standardized (see Section 5.3.1), we propose that "Operational Acceptance Testing (OAT)" and "Operational Testing (OT)" are treated as synonyms for a type of acceptance testing (ISO/IEC and IEEE, 2022, p. 22; Hamburg and Mogyorodi, 2024) that focuses on "non-functional" attributes of the system (LambdaTest, 2024). Indeed, this is how we track this approach in our test approach glossary! We define it as "test[ing] to determine the correct installation, configuration and operation of a module and that it operates securely in the operational environment" (ISO/IEC, 2018) or to "evaluate a system or component in its operational environment" (ISO/IEC and IEEE, 2017, p. 303), particularly "to determine if operations and/or systems administration staff can accept [it]" (Hamburg and Mogyorodi, 2024).

F.2 Recovery Testing

To remedy the flaws we describe in Section 5.3.2, we recommend that the literature uses these terms more consistently, resulting in the improved graph in Figure F.1b. The following proposals "recapture" information from the literature more consistently:

1. Prefer the term "recoverability testing" over "recovery testing" to indicate its focus on a system's *ability* to recover, not its *performance* of recovering (Kam, 2008, p. 47). "Recovery testing" may be an acceptable synonym, since it seems to be more prevelant in the literature.

2. Introduce the term *recovery performance testing* when evaluating performance metrics of a system's recovery as a subapproach of recoverability testing and performance-related testing[1] (ISO/IEC and IEEE, 2022, Fig. 2; 2013, p. 2).

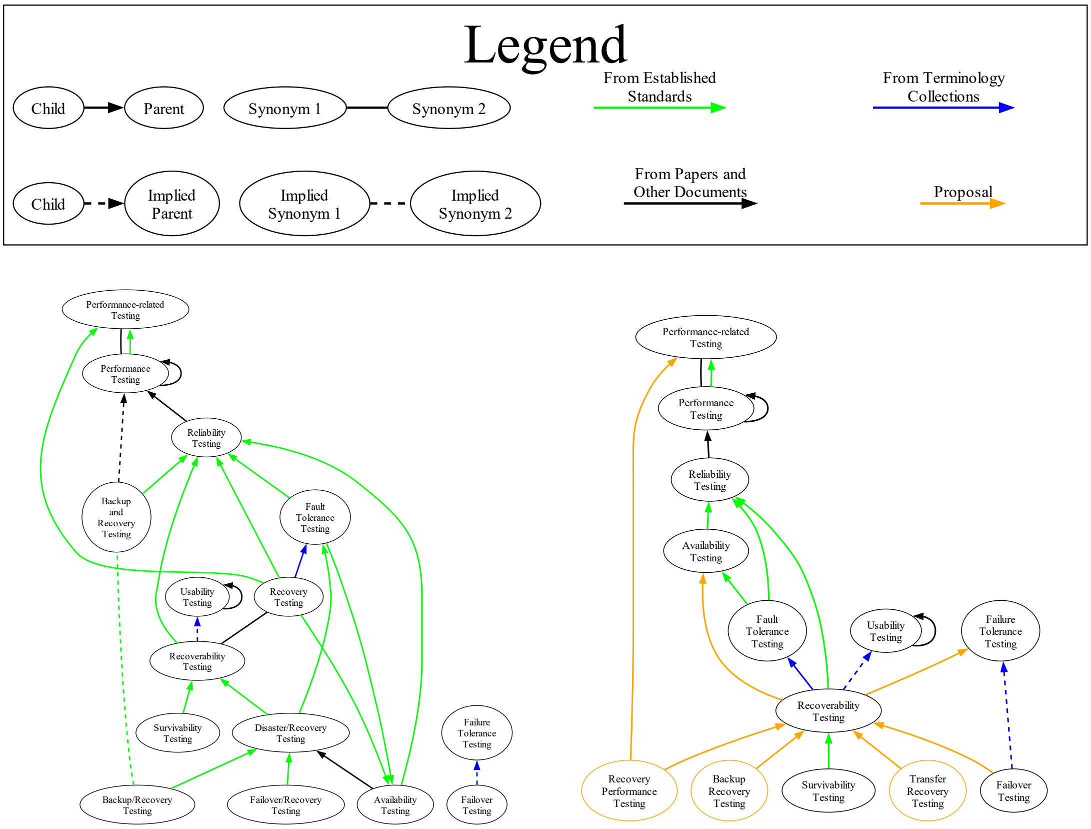
3. Introduce separate terms for the different methods of recovery which are all subapproaches of recoverability testing:

   (a) from backup memory (*backup recovery testing*) (ISO/IEC and IEEE, 2021c, p. 37; 2013, p. 2),

   (b) from a back-up system (*failover testing*) (Washizaki, 2025a, p. 5-9; Hamburg and Mogyorodi, 2024), or

   (c) by transferring operations elsewhere (*transfer recovery testing*) (ISO/IEC and IEEE, 2021c, p. 37).

[1] See Appendix F.4.



(a) Visualization of current relations.

(b) Visualization of proposed relations.

Figure F.1: Visualizations of relations in the subset of recovery testing.

F.3 Scalability Testing

The issues with scalability testing terminology we describe in Section 5.3.3 are resolved and/or explained by other sources! Taking this extra information into account provides a more accurate description of scalability testing.

(CONTRA, SYNS)

ISO/IEC and IEEE (2021c, p. 39) define "scalability testing" as the testing of a system's ability to "perform under conditions that may need to be supported in the future". This focus on "the future" is supported by Hamburg and Mogyorodi (2024), who define "scalability" as "the degree to which a component or system can be adjusted for changing capacity"; the original source they reference agrees, defining it as "the measure of a system's ability to be upgraded to accommodate increased loads" (Gerrard and Thompson, 2002, p. 381). In contrast, capacity testing focuses on the system's present state, evaluating the "capability of a product to meet requirements for the maximum limits of a product parameter" (ISO/IEC, 2023a). Therefore, these terms should *not* be synonyms, as done by Firesmith (2015, p. 53) and Bas (2024, pp. 22–23).

(CONTRA, DEFS)

The underlying reason that sources disagree on whether external modification of the system is part of scalability testing is its confusion with elasticity testing. Bertolino et al. say the two approaches are "closely related" (2019, p. 93:28), even claiming that one objective of elasticity testing is "to evaluate scalability" (p. 93:14)! However, Washizaki (2025a) (who cites Bertolino et al. (2019)) distinguishes between these approaches:

- **Scalability Testing:** testing that evaluates "the software's ability to scale up non-functional requirements such as load, number of transactions, and volume of data" (Washizaki, 2025a, p. 5-9; similar on p. 5-5).
- **Elasticity Testing:** testing that evaluates the ability of a system to "dynamically scal[e] up and down … resources as needed" (Bertolino et al., 2019, p. 93:18; similar on p. 93:13) "without compromising the capacity to meet peak utilization" (Washizaki, 2025a, p. 5-9).

This distinction is consistent with how the terms are used in industry: Pandey (2023) says that scalability is the ability to "increase … performance or efficiency as demand increases over time", while elasticity allows a system to "tackle changes in the workload [that] occur for a short period". Therefore, external modification of a system is part of scalability testing but *not* elasticity testing. This also implies that ISO/IEC's (2023a) notion of "scalability" actually refers to "elasticity".

F.4 Performance(-related) Testing

"Performance testing" is defined as testing "conducted to evaluate the degree to which a test item … accomplishes its designated functions" (ISO/IEC and IEEE, 2022, p. 7; 2021a, p. 2; 2017, p. 320; similar in 2021c, pp. 38-39; Moghadam, 2019, p. 1187). It does this by "measuring the performance metrics" (Moghadam, 2019, p. 1187; similar in Hamburg and Mogyorodi, 2024) (such as the "system's capacity for growth" (Gerrard, 2000b, p. 23)), "detecting the functional problems appearing under certain execution conditions" (Moghadam, 2019, p. 1187), and "detecting violations of non-functional requirements under expected and stress conditions" (Moghadam, 2019, p. 1187; similar in Washizaki, 2025a, pp. 5-8 to 5-9). It is performed either …

1. "within given constraints of time and other resources" (ISO/IEC and IEEE, 2022, p. 7; 2017, p. 320; similar in Moghadam, 2019, p. 1187), or
2. "under a 'typical' load" (ISO/IEC and IEEE, 2021c, p. 39).

It is listed as a subset of performance-related testing, which is defined as testing "to determine whether a test item performs as required when it is placed under various types and sizes of 'load' " (2021c, p. 38), along with other approaches like load and capacity testing (ISO/IEC and IEEE, 2022, p. 22). Note that "performance, load and stress testing might considerably overlap in many areas" (Moghadam, 2019, p. 1187). In contrast, Washizaki (2025a, pp. 5-8 to 5-9) gives "capacity and response time" as examples of "performance characteristics" that performance testing would seek to "assess", which seems to imply that these are subapproaches to performance testing instead. This is consistent with how some sources treat "performance testing" and "performance-related testing" as synonyms (Washizaki, 2025a, pp. 5-8 to 5-9; Moghadam, 2019, p. 1187), as noted in Section 5.2.2. This makes sense because of how general the concept of "performance" is; most definitions of "performance testing" seem to treat it as a category of tests.

However, it seems more consistent to infer that the definition of "performance-related testing" is the more general one often assigned to "performance testing" performed "within given constraints of time and other resources" (ISO/IEC and IEEE, 2022, p. 7; 2021a, p. 2; 2017, p. 320; similar in Moghadam, 2019, p. 1187), and "performance testing" is a subapproach of this performed "under a 'typical' load" (ISO/IEC and IEEE, 2021c, p. 39). This has other implications for relations between these types of testing; for example, "load testing" usually occurs "between anticipated conditions of low, typical, and peak usage" (ISO/IEC and IEEE, 2022, p. 5; 2021c, p. 39; 2017, p. 253; Hamburg and Mogyorodi, 2024), so it is a child of "performance-related testing" and a parent of "performance testing".

OG IEEE 2013

After these changes, some finishing touches remain. The reflexive parent relations are incorrect (as described in Section 5.2.3) and can be removed. Similarly, since "soak testing" is given as a synonym to both "endurance testing" *and* "reliability testing" (see Appendix D.2.2), it makes sense to just use these terms instead of one that is potentially ambiguous. These changes (along with those from Appendices F.2 and F.3) result in the proposed relations shown in Figure F.2.

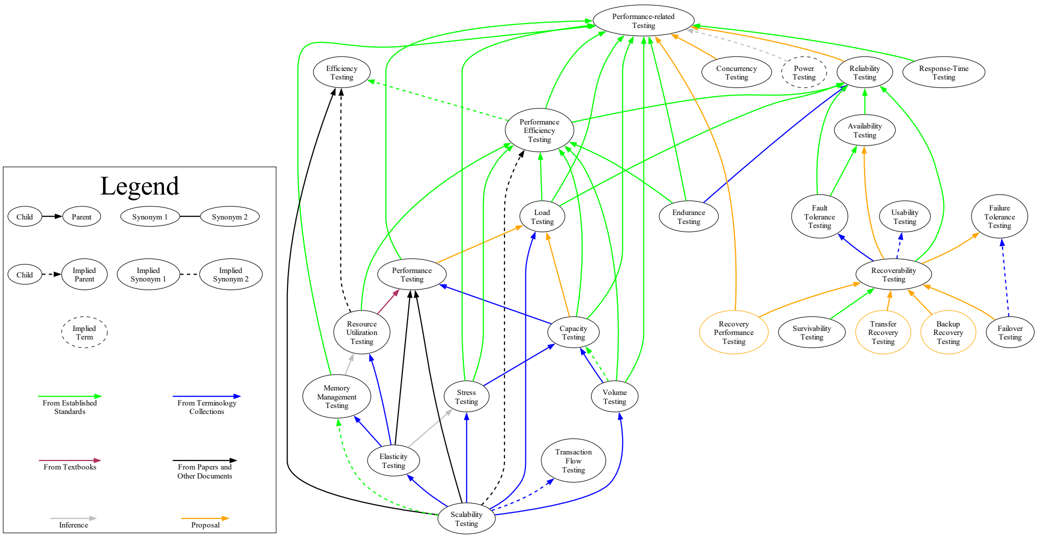


Figure F.2: Visualization of proposed relations in the subset of performance-related testing.

F.5 Compatibility Testing

“Co-existence” and “interoperability” are often defined separately (ISO/IEC and IEEE, 2017, pp. 73, 237; Hamburg and Mogyorodi, 2024), sometimes explicitly as a decomposition of “compatibility” (ISO/IEC, 2023a)! Following this precedent, “co-existence testing” and “interoperability testing” should be defined as their own test approaches to make their definitions atomic; ISO/IEC and IEEE (2017) define “interoperability testing” (p. 238) but not “co-existence testing”. The term “compatibility testing” may still be a useful test approach to define, but it should be defined independently of its children: “co-existence testing” and “interoperability testing”.