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:	xv, 127

Lay Abstract

Important: Lay abstract.

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 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 graphs to visualize and analyze these data.

This process uncovers 560 test approaches and four in-scope methods for deriving test approaches, such as those related to 75 software qualities. We also build a tool for generating graphs that illustrate relations between test approaches and track flaws captured by this tool and manually through the research process. This reveals 271 flaws, including nine terms used as synonyms to two (or more) disjoint test approaches and 15 pairs of test approaches that may either be synonyms or have a parent-child relationship. This also highlights notable confusion surrounding functional, operational acceptance, recovery, 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. We provide some preliminary advice on how to achieve this standardization.

Acknowledgements

ChatGPT was used to help generate supplementary Python code for constructing graphs 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. Jason Balaci's McMaster thesis template provided many helper LaTeX functions. Finally, Dr. Spencer Smith and Dr. Jacques Carette have been great supervisors and valuable sources of guidance and feedback.

Contents

List of Figures

List of Tables

List of Source Codes

List of Abbreviations and Symbols

c-use/C-use	Computation data Use
DAC	Differential Assertion Checking
DblPend	Double Pendulum
DOM	Document Object Model
du-path/DU-path	Definition-Use path
EMSEC	EManations SECurity
FIST	Fault Injection Security Tool
GlassBR	Glass BReaking
HREF	Hypertext REFerence
IEC	International Electrotechnical Commission
ISTQB	International Software Testing Qualifications Board
ML	Machine Learning
MR	Metamorphic Relation
OAT	Operational (Acceptance)/Orthogonal Array Testing
p-use/P-use	Predicate data Use
PAR	Product Anomaly Report
PDF	Portable Document Format
PIR	Product Incident Report
QAI	Quality Assurance Institute
RAC	Runtime Assertion Checking
RQ	Research Question
SglPend	Single Pendulum
SRS	Software Requirements Specification
SSP	Slope Stability analysis Program
SUT	System Under Test
SV	Software Verification
SWEBOK Guide	Guide to the SoftWare Engineering Body Of Knowledge
SWHS	Solar Water Heating System

TOAT	Taguchi's Orthogonal Array Testing
V&V	Verification and Validation

Reading Notes

Before reading this thesis, I encourage you to read through these notes, keeping them in mind while reading.

- The source code of this thesis is publicly available.
- This thesis template is primarily intended for usage by the computer science community[1]. However, anyone is free to use it.
- I've tried my best to make this template conform to the thesis requirements as per those set forth in 2021 by McMaster University. However, you should double-check that your usage of this template is compliant with whatever the "current" rules are.

Important: Replace reading notes.

[1] Hence why there are some LaTeX macros for "code" snippets.

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 (see Section 2.3.1), and supplementary terms—as well as the tools for data visualization and automated analysis outlined in Chapter 3. 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). Perhaps more surprisingly, "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. 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 (2011, p. 7). 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!

We noticed this lack of a standard language for software testing while working on our own software framework, Drasil (Carette et al., 2021), with the goal of "generating all of the software artifacts for (well understood) research software". Currently, these include Software Requirements Specifications (SRSs), READMEs, Makefiles, and code in up to six languages, depending on the specific case study (Hunt et al., 2021). To improve the quality, functionality, and maintainability of Drasil, we want to add test cases to this list (Smith, 2024; Hunt et al., 2021). This process would be a part of our "continuous integration system, [sic] so that the generated code for the case studies is automatically tested with each build" and would be a big step forward, since we currently only "test that the generated code compiles" (Smith, 2024). However, before we can include test cases as a generated artifact, the underlying domain—software testing—needs to be "well understood", which requires a "stable knowledge base" (Hunt et al., 2021).

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 testing approaches themselves:

ProWritingAid suggests that including "and incomplete" is redundant

- 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 2.1.1) and terminology collections (see Section 2.1.2) 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, 2010, p. 128; van Vliet, 2000, p. 399).
- **Fault:** "an incorrect step, process, or data definition in a computer program" (ISO/IEC and IEEE, 2010, p. 140) inserted when a developer makes an error (pp. 128, 140; Washizaki, 2024, p. 12-3; van Vliet, 2000, pp. 399–400).
- **Failure:** the inability of a system "to perform a required function or … within previously specified limits" (ISO/IEC and IEEE, 2019, p. 7; 2010, p. 139; similar in van Vliet, 2000, p. 400) that is "externally visible" (ISO/IEC and IEEE, 2019, p. 7; similar in van Vliet, 2000, p. 400) and caused by a fault (Washizaki, 2024, p. 12-3; van Vliet, 2000, p. 400).

OG ISO/IEC, 2005

- **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).

This distinction is sometimes important, but not always (Bourque and Fairley, 2014, p. 4-3). The term "fault" is "overloaded with too many meanings, as engineers and others use the word to refer to all different types of anomalies" (Washizaki, 2024, p. 12-3), and "defect" may be used as a "generic term that can refer to either a fault (cause) or a failure (effect)" (ISO/IEC and IEEE, 2017, p. 124; 2010, p. 96). 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"!

OG IEEE, 1996

OG 2005

These decisions are not inherently wrong, since they may be useful in certain contexts or for certain teams (cf. Section 2.2.2). 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? Our previous list of terms "error", "fault", "failure", and "defect" are used to describe many test approaches, including:

Q #1: Should I include a list of sources that describe each, or would that make this too cluttered?

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.

Q #2: Is this OK to use here w.r.t. #140?

These types of ambiguities 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 categorize experience-based testing as both a test design technique and a test practice on the same page—twice (2022, Fig. 2, p. 34)! 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). Load testing is performed with loads "between anticipated conditions of low, typical, and peak usage" (ISO/IEC and IEEE, 2022, p. 5) or loads that are as large as possible (Patton, 2006, p. 86). Alpha testing is performed by "users within the organization developing the software" (ISO/IEC and IEEE, 2017, p. 17), "a small, selected group of potential users" (Washizaki, 2024, 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 and fill. 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.

See #22

Based on the observed gap in software testing specifically 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. We provide more detailed discussion on what is and is not in scope later in this document, including practices we exclude in full, such as hardware testing (Appendix A.1), or in part, such as the parts of error seeding, fault injection testing, and mutation testing that do not directly test code (Appendix A.2). Additionally, static testing is a useful component of software testing and is therefore included at this level of analysis, despite it not being relevant to our original motivation (Appendix A.3).

Q #3: I want to provide a general reference to "Appendix A", but `\Cref{app-scope}` displays as "Section 8.4" for some reason. Is this worth pursuing, or is my workaround of more specific pointers OK?

This document describes our process, as well as our results, in more detail. We start by documenting the 560 test approaches mentioned by 66 sources (described

in Section 2.1), recording their name, category[1] (see Section 2.2.1), definition, synonyms (see Section 2.2.2), parents (see Section 2.2.3), and flaws (in a separate document) as applicable. We also record any other relevant notes, such as prerequisites, uncertainties, and other resources. We follow the procedure laid out in Section 2.3 and use these Research Questions (RQs) as a guide:

Later: Define in Terminology; #140

1. What testing approaches do the literature describe?
2. Are these descriptions consistent?
3. Can we systematically resolve any of these inconsistencies?

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 3). Despite the amount of well understood and organized knowledge, the literature is still quite flawed (Chapter 4). This reinforces the need for a proper taxonomy! We then provide some potential solutions covering some of these flaws (Chapter 5).

[1]There may be more than one category given for a single test approach (for example, A/B Testing in Table 1.1) which is indicative of a flaw (see Section 4.2.1).

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

Name	**Approach Category**	**Definition**	**Parent(s)**	**Synonym(s)**
A/B Testing	Practice (ISO/IEC and IEEE, 2022, p. 22), Type (implied by Firesmith, 2015, p. 58)	Testing "that allows testers to determine which of two systems or components performs better" (ISO/IEC and IEEE, 2022, p. 1)	Statistical Testing (ISO/IEC and IEEE, 2022, pp. 1, 35), Usability Testing (Firesmith, 2015, p. 58)	Split-Run Testing (ISO/IEC and IEEE, 2022, pp. 1, 35)
Big-Bang Testing	Level (inferred from integration testing)	"Testing in which … [components of a system] are combined all at once into an overall system, rather than in stages" (ISO/IEC and IEEE, 2017, p. 45)	Integration Testing (ISO/IEC and IEEE, 2017, p. 45; Washizaki, 2024, p. 5-7; Sharma et al., 2021, p. 603; Kam, 2008, p. 42)	—
Classification Tree Method	Technique (ISO/IEC and IEEE, 2022, p. 22; 2021, pp. 2, 12, Fig. 2; Hamburg and Mogyorodi, 2024)	Testing "based on exercising classes in a classification tree" (ISO/IEC and IEEE, 2021, p. 22)	Specification-based Testing (ISO/IEC and IEEE, 2022, p. 22; 2021, pp. 2, 12, Fig. 2; Hamburg and Mogyorodi, 2024; Firesmith, 2015, p. 47), Model-based Testing (ISO/IEC and IEEE, 2022, p. 13; 2021, pp. 6, 12)	Classification Tree Technique (Hamburg and Mogyorodi, 2024)

Chapter 2

Methodology

At a high level, our methodology follows the following steps:

Important: Say what the methodology is for

1. Identify authoritative sources on software testing (Section 2.1)
2. Identify software testing terminology from each source, including test approaches and terms that imply them (Section 2.3.1)
3. For each test approach, record its: (Section 2.3)
   (a) Name
   (b) Category[1] (Section 2.2.1)
   (c) Definition
   (d) Synonyms (Section 2.2.2)
   (e) Parents (Section 2.2.3)
   (f) Flaws (in a separate document)
   (g) Other relevant notes (e.g., prerequisites, uncertainties, and other resources)

Later: Define in Terminology; #140

4. Repeat steps 1 to 3 on any subsets of terminology that are missing or unclear (Section 2.3.2) until some stopping criteria

   Important: Define/add pointer

5. Analyze recorded test approach data for additional flaws
   (a) Generate relation graphs (Section 3.1)
   (b) Automatically detect certain classes of flaws (Section 3.2.1)
   (c) Automatically analyze manually recorded flaws from step 3f (Section 3.2.2)

[1]There may be more than one category given for a single test approach (for example, A/B Testing in Table 1.1) which is indicative of a flaw (see Section 4.2.1).

6. Report results of flaw analysis (Chapter 4)
7. Provide examples of how to resolve these flaws (Chapter 5)

2.1 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. Since we are particularly interested in software engineering, we start from the vocabulary document for systems and software engineering (ISO/IEC and IEEE, 2017) and two versions of the Guide to the SoftWare Engineering Body Of Knowledge (SWEBOK Guide)—the newest one (Bourque and Fairley, 2014) and one submitted for public review[2] (Washizaki, 2024). 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" (see Section 5.2); we verified this by looking at this original source. We similarly "snowball" on terminology itself; when a term requires more investigation (e.g., its definition is missing or unclear), we perform a miniature literature review on this subset to "fill in" this missing information (see Section 2.3.2). If these additional sources provide more information and are "trustworthy", we may then investigate them in their entirety (as opposed to just the original subset of interest). We define a source to be "trustworthy" if it:

Q #4: Better name for this?

Present tense?

1. has gone through a peer-review process,
2. is written by numerous, well-respected authors,
3. is informed by many sources, and
4. is accepted and used in the field of software.

For ease of discussion and analysis, we group the complete set of sources into "tiers" based on their format, method of publication, and this metric of "trustworthiness". We therefore create the following tiers, given in order of descending trustworthiness:

1. established standards (Section 2.1.1),
2. terminology collections (Section 2.1.2),
3. textbooks (Section 2.1.3), and
4. papers and other documents (Section 2.1.4).

[2](Washizaki, 2024) has been published since we investigated these sources; if time permits, we will revisit this published version.

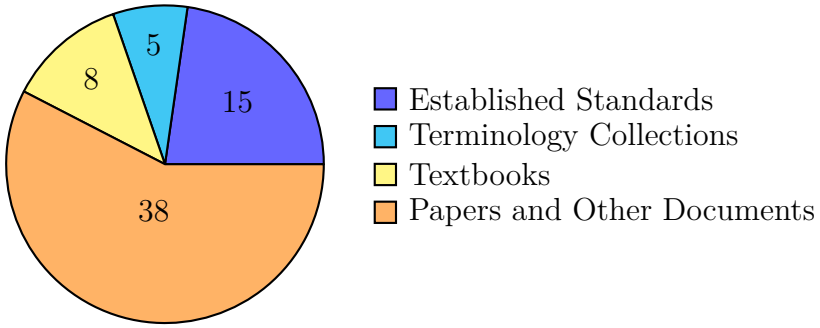


Figure 2.1: Summary of how many sources comprise each source tier.

A summary of how many sources comprise each tier is given in Figure 2.1. We often use papers to "fill in" missing information in a more specific subdomain and do not always investigate them entirely (see Section 2.3.2), which results in a large number of them. We use standards the second most frequently due to their high trustworthiness 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.

2.1.1 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. 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). For these reasons, they are the most trustworthy sources. However, this does not imply perfection, as we identify 50 flaws within these standards (see Tables 4.1 and 4.2)! Only standards for software development and testing are in scope for this research (see Chapter 1). For example, "the purpose of the ISO/IEC/IEEE 29119 series is to define an internationally agreed set of standards for software testing that can be used by any organization when performing any form of software testing" (2022, p. vii; similar in 2016, p. ix). This tier is composed of (ISO/IEC and IEEE, 2022; 2021; 2019; 2017; 2016; 2013; 2010; IEEE, 2012; ISO/IEC, 2023a;b; 2018; 2015; 2011; ISO, 2022; 2015) .

2.1.2 Terminology Collections

These are collections of software testing terminology built up from multiple sources (such as the established standards outlined in Section 2.1.1) that are made to be widely applicable. 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, 2025). 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. We include Firesmith's taxonomy (2015) because it presents relations between many test approaches and Doğan et al.'s literature review (2014) because it cites many of sources from which we can "snowball" if desired (see Section 2.1). This tier is composed of (Hamburg and Mogyorodi, 2024; Doğan et al., 2014; Firesmith, 2015; Washizaki, 2024; Bourque and Fairley, 2014).

2.1.3 Textbooks

We consider textbooks to be more trustworthy than papers (see Section 2.1.4) 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" (see Section 5.2); we verified this by looking at this original source. This tier is composed of (Ammann and Offutt, 2017; Dennis et al., 2012; Gerrard and Thompson, 2002; Kaner et al., 2011; Patton, 2006; Perry, 2006; Peters and Pedrycz, 2000; van Vliet, 2000).

Present tense?

2.1.4 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 documents are journal articles and conference papers, the following document types are also present. Some of these are not peer-reviewed works, but they show how terms are used in practice:

See #89

- 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))

The full set of sources that comprise this tier is (Bajammal and Mesbah, 2018; Barbosa et al., 2006; Baresi and Pezzè, 2006; Barr et al., 2015; Bas, 2024; Berdine

et al., 2006; Bocchino and Hamilton, 1996; Chalin et al., 2006; ChatGPT (GPT-4o), 2024; Choudhary et al., 2010; Dhok and Ramanathan, 2016; Engström and Petersen, 2015; Forsyth et al., 2004; Gerrard, 2000a;b; Ghosh and Voas, 1999; Godefroid and Luchaup, 2011; Intana et al., 2020; Jard et al., 1999; Kam, 2008; Kanewala and Yueh Chen, 2019; Kuļešovs et al., 2013; Lahiri et al., 2013; LambdaTest, 2024; Mandl, 1985; Moghadam, 2019; Pandey, 2023; Preuße et al., 2012; Rus et al., 2008; Knüvener Mackert GmbH, 2022; Sakamoto et al., 2013; Sangwan and LaPlante, 2006; Sharma et al., 2021; Sneed and Göschl, 2000; Souza et al., 2017; Tsui, 2007; Valcheva, 2013; Yu et al., 2011).

2.2 Terminology

Our research aims to describe the current state of software testing literature. To reduce potential bias, we do not invent or add our own classifications or kinds of relations. Instead, the notions of test approach categories (Section 2.2.1), synonyms (Section 2.2.2), and parent-child relations (Section 2.2.3) presented here follow logically from the literature. We define them here for clarity since we use them throughout this thesis, even though they are "results" of our research. While most information is presented explicitly in the sources we investigate, some appears more implicitly. This is a useful distinction to make, as implicit claims carry less weight than explicit ones. We call this property "rigidity" and define it in Section 2.2.4.

2.2.1 Approach Categories

While there are many ways to categorize software testing approaches, perhaps the most widely used is the one given by ISO/IEC and IEEE (2022). This schema categorizes test approaches as levels, types, techniques, practices, or static testing (2022, Fig. 2; see Table 2.1). However, we consider static testing to be a less prominent category, since these test approaches can often be grouped into one of the other four categories. For example, static assertion checking (mentioned by Lahiri et al., 2013, p. 345; Chalin et al., 2006, p. 343) is a subapproach of assertion checking, which may also be performed dynamically. Therefore, static assertion checking may inherit assertion checking's inferred category of "practice", which is a more meaningful classification than one based solely on "whether it is performed by running code or not". We track static testing approaches at this level of analysis (see Appendix A.3), but may fold them into other categories as appropriate, although some teams may wish to keep them separate based on their specific context.

Of the five primary categories we use (see Table 2.1), "level" and "type" are particularly pervasive in the literature; for example, six non-IEEE sources also give unit testing, integration testing, system testing, and acceptance testing as examples of test levels (Washizaki, 2024, pp. 5-6 to 5-7; Hamburg and Mogyorodi, 2024; Perry, 2006, pp. 807–808; Peters and Pedrycz, 2000, pp. 443–445; Kuļešovs et al., 2013, p. 218; Gerrard, 2000a, pp. 9, 13), although they may use a different term for "test level" (see Table 2.1). Because of their widespread use and their

OG Black, 2009

usefulness in dividing the domain of software testing into more manageable subsets, we use these categories for now. These four subcategories of test approaches can be loosely described by what they specify as follows:

See #21

- **Level**: What code is tested
- **Practice**: How the test is structured and executed
- **Technique**: How inputs and/or outputs are derived
- **Type**: Which software quality is evaluated

For example, boundary value analysis is a test technique since its inputs are "the boundaries of equivalence partitions" (ISO/IEC and IEEE, 2022, p. 2; 2021, p. 1; similar on p. 12 and in Hamburg and Mogyorodi, 2024). Similarly, acceptance testing is a test level since its goal is to "enable a user, customer, or other authorized entity to determine whether to accept a system or component" (ISO/IEC and IEEE, 2017, p. 5; similar in 2021, p. 6; Sakamoto et al., 2013, p. 344), which requires the system or component to be developed and ready for testing.

While the vast majority of identified test approaches can be categorized in this way, there are some outliers. For example, an "artifact" category is an addition option, since some terms can refer to the application of a test approach and/or the resulting document(s). Because of this, a test approach being categorized as a category from Table 2.1 *and* an artifact is *not* a flaw (see Section 4.2.1). Additionally, "test metrics" were also identified, but tracking them is out-of-scope, since they describe methods for *evaluating* testing as opposed to methods for *performing* it. The related test approaches that seek to maximize these metrics are instead captured as kinds of coverage-driven testing (see Section 2.3.1) and experience-based testing (ISO/IEC and IEEE, 2022, p. 34).

See #44, #119, and #39

See #119

See #21 and #22

Excluding our proposed "artifact" category and the "approach" supercategory, the categories given in Table 2.1 seem to be orthogonal. 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; 2021, p. 7), and "Keyword-Driven Testing [sic] can be applied at all testing levels … and for various types of testing" (2016, p. 4). This means that a specific test approach can be derived by combining multiple test approaches from different categories; for example, formal reviews are a combination of formal testing and reviews. The following examples are given in the literature; the two subapproaches for each are omitted for brevity as the name of the combination makes it clear what they are:

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 verification (Preuße et al., 2012, p. 6)
5. Closed-loop protection system testing (Forsyth et al., 2004, p. 331)

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

Term	**Definition**	**Examples**
Test Approach	A "high-level test implementation choice" that includes "test level, test type, test technique, test practice and … static testing" (ISO/IEC and IEEE, 2022, p. 10) and is used to "pick the particular test case values" (2017, p. 465)	black or white box, minimum and maximum boundary value testing (ISO/IEC and IEEE, 2017, p. 465)
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; 2021, p. 6) with their "own documentation and resources" (2017, p. 469)	unit/component testing, integration testing, system testing, acceptance testing (ISO/IEC and IEEE, 2022, p. 12; 2021, p. 6; 2017, p. 467)
Test Practice	A "conceptual framework that can be applied to … [a] test process to facilitate testing" (ISO/IEC and IEEE, 2022, p. 14; 2017, p. 471; OG IEEE 2013)	scripted testing, exploratory testing, automated testing (ISO/IEC and IEEE, 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; similar in 2017, p. 467) that "generate evidence that test item requirements have been met or that defects are present in a test item" (2021, p. vii)	equivalence partitioning, boundary value analysis, branch testing (ISO/IEC and IEEE, 2022, p. 11)
Test Type	"Testing that is focused on specific quality characteristics" (ISO/IEC and IEEE, 2022, p. 15; 2021, p. 7; 2017, p. 473; OG IEEE 2013)	security testing, usability testing, performance testing (ISO/IEC and IEEE, 2022, p. 15; 2017, p. 473)

[a] Also called "test phase" (see Synonyms Flaw 2) or "test stage" (see Synonyms Flaw 4).

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

Table 2.2: 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, 2024, 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, 2024, pp. 5-7 to 5-9)	Type?
Phase	none given	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; 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	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 for generating or selecting the most suitable test suites" (Washizaki, 2024, p. 5-10)	specification-based testing, structure-based testing, fault-based testing[c] (Washizaki, 2024, pp. 5-10, 5-13 to 5-15)	Technique

[a] See Synonyms Flaw 4.

[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).

6. Endurance stability testing (Firesmith, 2015, p. 55)
7. End-to-end functionality testing (ISO/IEC and IEEE, 2021, p. 20; Gerrard, 2000a, Tab. 2)
8. Formal reviews (Hamburg and Mogyorodi, 2024)
9. Grey-box integration testing (Sakamoto et al., 2013, p. 344)
10. Incremental integration testing (Sharma et al., 2021, pp. 601, 603, 605–606)
11. Informal reviews (Hamburg and Mogyorodi, 2024)
12. Infrastructure compatibility testing (Firesmith, 2015, p. 53)
13. Invariant-based automatic testing (Doğan et al., 2014, pp. 184–185, Tab. 21), including for "AJAX user interfaces" (p. 191)
14. Legacy system integration (testing) (Gerrard, 2000a, Tab. 2)
15. Manual procedure testing (Firesmith, 2015, p. 47)
16. Manual security audits (Gerrard, 2000b, p. 28)
17. Model-based GUI testing (Doğan et al., 2014, Tab. 1; implied by Sakamoto et al., 2013, p. 356)
18. Model-based web application testing (implied by Sakamoto et al., 2013, p. 356)
19. Non-functional search-based testing (Doğan et al., 2014, Tab. 1)
20. Offline MBT (Hamburg and Mogyorodi, 2024)
21. Online MBT (Hamburg and Mogyorodi, 2024)
22. Role-based reviews (Hamburg and Mogyorodi, 2024)
23. Scenario walkthroughs (Gerrard, 2000a, Fig. 4)
24. Scenario-based reviews (Hamburg and Mogyorodi, 2024)
25. Security attacks (Hamburg and Mogyorodi, 2024)
26. Security audits (ISO/IEC and IEEE, 2021, p. 40; Gerrard, 2000b, p. 28)
27. Statistical web testing (Doğan et al., 2014, p. 185)
28. Usability test script(ing) (Hamburg and Mogyorodi, 2024)
29. Web application regression testing (Doğan et al., 2014, Tab. 21)
30. White-box unit testing (Sakamoto et al., 2013, pp. 345–346)

OG [19]

There are some cases where the subapproaches of the "compound" approaches above are *not* from separate categories. However, these cases can be explained by insufficient data or by edge cases that require special care. Further analysis is necessary and may end up showing these categories to be orthogonal. 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 4.2.1), 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

One important side effect of the particularity of these terms is that they can be "overloaded"; for example, someone could reasonably yet imprecisely use any of these four categories as a synonym for "approach". Even our prompt in (ChatGPT (GPT-4o), 2024, emphasis added) was imprecise, asking for the "*type* of software testing that focuses on looking for bugs where others have already been found." Interestingly, ChatGPT later "corrected" this by calling defect-based testing an approach (although this may have been biased by our previous usage of the term "approach")! Because of this nuance, we need to carefully investigate these kinds of flaws. 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. For this reason, these "categorizations" are marked with a question mark (?) and included in Table 4.6 instead of in Table 4.4. In particular, other sources (such as Washizaki, 2024; Barbosa et al., 2006) propose similar yet distinct categories that clash or overlap with these categories. These are given in Table 2.2 and may be used to infer the category from its "IEEE Equivalent" column. Further analysis may reveal that these other categories provide new perspectives and may be useful in some contexts, either in place of or in tandem with ISO/IEC and IEEE's categorization (2022).

Correct grammar?

Similarly, the literature gives many other ways to categorize test approaches that may be used in tandem with or in place of those given in Tables 2.1 and 2.2. In general, these are defined less systematically but are more fine-grained, seeming to

"specialize" categories from Table 2.1. The existence of these classifications are 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, 2024, p. 5-16). Unfortunately, even these alternate categories are not used consistently (see Categories Flaw 3 and Categories Flaw 7)!

These categorizations are given in Table 2.3 for completeness. The bases for these alternate categorizations, along with the example approaches listed, come from the source(s) indicated in the "Test Basis" column. When multiple sources are given for a test basis, the union of these sources all list the examples given. By looking at a test basis and its corresponding examples, we can infer its "Parent IEEE Category": the IEEE category from Table 2.1 it seems to subdivide. The sources provided in this column classify the example approaches given accordingly. For example, in the first row, ISO/IEC and IEEE (2021, pp. 4, 8) and Firesmith (2015, p. 46) both classify specification-based testing, structure-based testing, and grey-box testing as test techniques. A lack of source(s) in this column indicate that this category was inferred or that the example approaches were not categorized by the literature; additional context is sometimes provided in a footnote.

Another way that the IEEE categories can be subdivided is by grouping related testing approaches into a "class" or "family" with "commonalities and well-identified variabilities that can be instantiated", where "the commonalities are large and the variabilities smaller" (Carette, 2024). Examples of these are the classes of combinatorial (ISO/IEC and IEEE, 2021, p. 15) and data flow testing (p. 3) and the family of performance-related testing (Moghadam, 2019, p. 1187)[3], and is implied for security testing, a test type that consists of "a number of techniques[4]" (ISO/IEC and IEEE, 2021, p. 40). This is explored in more detail in Section 4.2.1. Note that "there is a lot of overlap between different classes of testing" (Firesmith, 2015, p. 8), meaning that "one category [of test techniques] might deal with combining two or more techniques" (Washizaki, 2024, p. 5-10). For example, "performance, load and stress testing might considerably overlap in many areas" (Moghadam, 2019, p. 1187). A side effect of this is that it is difficult to "untangle" these classes; for example, take the following sentence: "whitebox fuzzing extends dynamic test generation based on symbolic execution and constraint solving from unit testing to whole-application security testing" (Godefroid and Luchaup, 2011, p. 23)! This is, in part, why research on software testing terminology is so vital.

[3]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 Section 5.3).

[4]This may or may not be distinct from the notion of "test technique" described in Table 2.1.

Table 2.3: Alternate categorizations given by the literature.

Test Basis	**Example Approaches**	**Parent[a] IEEE Category**
Visibility of the SUT's Internal Structure (ISO/IEC and IEEE, 2021, p. 8; Washizaki, 2024, pp. 5-10, 5-16; Patton, 2006, pp. 53, 218; Perry, 2006, p. 69; Ammann and Offutt, 2017, pp. 57–58; Kuļešovs et al., 2013, p. 213) or Testing Approach (Sharma et al., 2021, p. 601; OG [8]) or Testing Method (Kam, 2008, pp. 4–5)	Specification-based Testing Structure-based Testing Grey-Box Testing	Technique (ISO/IEC and IEEE, 2021, pp. 4, 8; Firesmith, 2015, p. 46)
Source of Information for Designing Tests (ISO/IEC and IEEE, 2021, p. 8)	Specification-based Testing Structure-based Testing Experience-based Testing	Technique[b] (ISO/IEC and IEEE, 2022, p. 22; 2021, p. 4 Washizaki, 2024, pp. 5-10, 5-13; Hamburg and Mogyorodi, 2024; Firesmith, 2015, p. 46)
Selection Process (Washizaki, 2024, p. 5-16)	Deterministic Testing Random Testing	Technique (Washizaki, 2024, pp. 5-12, 5-16)
Question Answered: What? When? Where? Who? Why? How? How Well? (Firesmith, 2015, p. 17)	System Testing Smoke Testing Regression Testing Manual Testing Model-based Testing Scenario Testing Data Flow Testing[c]	Approach

Continued on next page

Table 2.3: Alternate categorizations given by the literature. (Continued)

Test Basis	**Example Approaches**	**Parent[a] IEEE Category**
Execution of Code[d] (Patton, 2006, p. 53; Kuļešovs et al., 2013, p. 214; Gerrard, 2000a, p. 12)	Static Testing Dynamic Testing	Approach
Goal of Testing (Perry, 2006, pp. 69–70; Kuļešovs et al., 2013, p. 214)	Verification Testing Validation Testing	Approach
Test Factor (also called Quality Factor or Quality Attribute) (Perry, 2006, pp. 40–41)	Correctness Testing Response-Time Testing Access Control Testing Compliance Testing Reliability Testing Maintainability Testing Portability Testing Performance Testing[e]	Type (ISO/IEC and IEEE, 2022, p. 22; and/or implied by its quality and/or Firesmith, 2015)
Source of Test Data (Peters and Pedrycz, 2000, p. 440)	Specification-based Testing Implementation-oriented Testing Error-oriented Testing	Technique
Adequacy Criterion (van Vliet, 2000, pp. 398–399)	Coverage-based Testing Fault-based Testing Error-based Testing	Technique (van Vliet, 2000, pp. 398–399)

Continued on next page

Table 2.3: Alternate categorizations given by the literature. (Continued)

Test Basis	**Example Approaches**	**Parent[a] IEEE Category**
Coverage Criteria (Ammann and Offutt, 2017, pp. 18–19)	Input Space Partitioning Graph Coverage Logic Coverage Syntax-based Testing	Technique (ISO/IEC and IEEE, 2022, p. 22; 2021, Fig. 2; Washizaki, 2024, p. 5-11; Firesmith, 2015, pp. 47–48)
Human Involvement (Kuļešovs et al., 2013, p. 214)	Manual Testing Automated Testing	Practice (ISO/IEC and IEEE, 2022, p. 22) Technique (implied by ISO/IEC and IEEE, 2022, p. 35; see Table 4.4)
Structuredness (Kuļešovs et al., 2013, p. 214)	Scripted Testing Exploratory Testing	Practice[f] (ISO/IEC and IEEE, 2022, pp. 20, 22)
Property of Code (Kuļešovs et al., 2013, p. 213) or Test Target (Kam, 2008, pp. 4–5)	Functional Testing Non-functional Testing	Ambiguous (see Section 4.3)
Coverage Requirement (Kam, 2008, pp. 4–5)	Data Flow Testing Control Flow Testing	Technique (Washizaki, 2024, p. 5-13)
Category of Test Type[g] (Gerrard, 2000a, p. 12)	Static Testing Test Browsing Functional Testing Non-functional Testing Large Scale Integration (Testing)	Ambiguous
Priority (in the context of testing e-business projects) (Gerrard, 2000a, p. 13)	Smoke Testing Usability Testing Performance Testing Functionality Testing	Type[h] (ISO/IEC and IEEE, 2022, p. 22; 2021, Tab. A.1; and/or implied by Firesmith, 2015, p. 53)

Continued on next page

Table 2.3: Alternate categorizations given by the literature. (Continued)

Test Basis	**Example Approaches**	**Parent[a] IEEE Category**
Purpose (Pan, 1999)	Correctness Testing Performance Testing Reliability Testing Security Testing	Type (ISO/IEC and IEEE, 2022, p. 22; and/or implied by Firesmith, 2015, p. 53)

[a] See Section 2.2.3.

[b] Experience-based testing may instead be a "practice" (see Table 4.4).

[c] This list is *quite* nonexhaustive.

[d] We also consider this categorization meaningful (see Appendix A.3).

[e] Other test factors are given that do not unambiguously map to corresponding test approaches: file integrity, authorization, audit trail, continuity of processing, service levels, ease of use, coupling (e.g., with other applications in a given environment), and ease of operation (e.g., documentation, training) (Perry, 2006, pp. 40–41).

[f] Exploratory testing may instead be a "technique" (see Table 4.4).

[g] "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.

[h] With the exception of smoke testing, which is categorized as a technique (Washizaki, 2024, p. 5-14; Sharma et al., 2021, pp. 601, 603, 605–606); performance testing is also sometimes categorized as a technique (ISO/IEC and IEEE, 2021, p. 38).

2.2.2 Synonym Relations

more in Umar2000

The same approach often has many names. For example, *specification-based testing* is also called:

1. Black-Box Testing (ISO/IEC and IEEE, 2022, p. 9; 2021, p. 8; 2017, p. 431; Washizaki, 2024, p. 5-10; Hamburg and Mogyorodi, 2024; Firesmith, 2015, p. 46 (without hyphen); Sakamoto et al., 2013, p. 344; van Vliet, 2000, p. 399)
2. Closed-Box Testing (ISO/IEC and IEEE, 2022, p. 9; 2017, p. 431)
3. Functional Testing[5] (ISO/IEC and IEEE, 2017, p. 196; Kam, 2008, p. 44; van Vliet, 2000, p. 399; implied by ISO/IEC and IEEE, 2021, p. 129; 2017, p. 431)
4. Domain Testing (Washizaki, 2024, p. 5-10)
5. Specification-oriented Testing (Peters and Pedrycz, 2000, p. 440, Fig. 12.2)
6. Input Domain-Based Testing (implied by Bourque and Fairley, 2014, pp. 4-7 to 4-8)

These synonyms are the same as synonyms in natural language; while they may emphasize different aspects or express mild variations, their core meaning is nevertheless the same. 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. Thus, synonyms are not inherently problematic, although they can be (see Section 4.2.2).

Synonym relations are often given explicitly in the literature. For example, ISO/IEC and IEEE (2022, p. 9) list "black-box testing" and "closed box testing" beneath the glossary entry for "specification-based testing", meaning they are synonyms. "Black-box testing" is likewise given under "functional testing" in (2017, p. 196), meaning it is also a synonym for "specification-based testing" through transitivity. However, these relations can also be less "rigid" (see Section 2.2.4); "functional testing" is listed in a *cf.* footnote to the glossary entry for "specification-based testing" (2017, p. 431), which supports the previous claim but would not necessarily indicate a synonym relation on its own.

Q #5: Is this clear/correct? Should I explain this more?

Similarly, Washizaki (2024, p. 5-10) says "*specification-based techniques* … [are] sometimes also called domain testing techniques" in the SWEBOK Guide V4, from which the synonym of "domain testing" follows logically. However, its predecessor V3 only *implies* the more specific "input domain-based testing" as a synonym. The section on test techniques says "the classification of testing techniques presented here is based on how tests are generated: from the software engineer's intuition and experience, the specifications, the code structure …" (Bourque and Fairley, 2014,

[5]This may be an outlier; see Section 4.3.1.

p. 4-7), and the first three subsections on the following page are "Based on the Software Engineer's Intuition and Experience", "Input Domain-Based Techniques", and "Code-Based Techniques" (p. 4-8). The order of the introductory list lines up with these sections, implying that "input domain-based techniques" are "generated[] from … the specifications" (i.e., that input domain-based testing is the same as specification-based testing). Furthermore, the examples of input domain-based techniques given—equivalence partitioning, pairwise testing, boundary-value analysis, and random testing—are all given as children[6] of specification-based testing (ISO/IEC and IEEE, 2022; 2021, Fig. 2; Hamburg and Mogyorodi, 2024); even V4 agrees with this (Washizaki, 2024, pp. 5-11 to 5-12)!

2.2.3 Parent-Child Relations

Many test approaches are multi-faceted and can be "specialized" into others; for example, there are many subtypes of performance-related testing, such as load testing and stress testing (see Section 5.3). These "specializations" will be referred to as "children" or "subapproaches" of the multi-faceted "parent". This nomenclature also extends to other categories (such as "subtype"; see Section 2.2.1 and Table 2.1) and software qualities ("subquality"; see Section 2.3.1). There are many reasons two approaches may have a parent-child relation, such as:

Q #6: Should I add more? This would require me to go through my glossary and reverse engineer why I considered parent-child relations to be explicit. In hindsight, I should have recorded this as I went, and I'm not sure how worth it it would be to do now

1. **The parent approach is part of a mutually exclusive set.** It is often trivial to classify a test approach as a child of one of a set of other, mutually exclusive test approaches. For example, ISO/IEC and IEEE say that "testing can take two forms: static and dynamic" (2022, p. 17) and provide examples of subapproaches of static and dynamic testing (Fig. 1). Likewise, Gerrard says "tests can be automated or manual" (2000a, p. 13) and gives subapproaches of automated and manual testing (Tab. 2; 2000b, Tab. 1).

2. **One is "stronger than" or "subsumes" the other.** When comparing adequacy criteria that "specif[y] requirements for testing" (van Vliet, 2000, p. 402), "criterion X is stronger than criterion Y if, for all programs P and all test sets T, X-adequacy implies Y-adequacy" (p. 432). While this relation only "compares the thoroughness of test techniques, not their ability to detect faults" (p. 434), it is sufficient to consider one a child of the other.

2.2.4 Rigidity

A consequence of the use of natural language and the lack of standardization is the considerable degree of nuance that can get lost when referring to information sources. While most information is presented explicitly in the sources we investigate, some appears more implicitly. This is a useful distinction to make, as implicit claims carry less weight than explicit ones. We call this property "rigidity" and

[6]Pairwise testing is given as a child of combinatorial testing, which is itself a child of specification-based testing, by (ISO/IEC and IEEE, 2021, Fig. 2) and (Washizaki, 2024, pp. 5-11 to 5-12), making it a "grandchild" of specification-based testing according to these sources.

capture it when citing sources. This allows us to provide a more complete picture of the state of the literature; for example, we can view implicit flaws separately in Tables 4.1 and 4.2, since additional context may rectify them. The following non-mutually exclusive reasons for information to be considered "implicit" emerged, and the given keywords are used to identify them (see the relevant source code):

1. **The information is implied.** The implicit categorizations of "test type" by Firesmith (2015, pp. 53–58) (see Tables 4.4 and 4.6) are an example of this. The given test approaches are not explicitly called "test types", as the term is used more loosely to refer to different kinds of testing—what should be called "test approaches" as per Table 2.1. However, this set of test approaches are "based on the associated quality characteristic and its associated quality attributes" (p. 53), implying that they are test types. Cases such as this are indicated by a question mark or one of the following keywords: "implied", "inferred", or "likely".

   Additionally, if a test approach in our glossary has a name ending in " (Testing)" with a space, 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 by 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.

2. **The information is not universal.** ISO/IEC and IEEE (2017, p. 372, emphasis added) define "regression testing" as "testing required to determine that a change to a system component has not adversely affected *functionality, reliability or performance* and has not introduced additional defects". While reliability testing, for example, is not *always* a subset of regression testing (since it may be performed in other ways), it *can be* accomplished by regression testing, so there is sometimes a parent-child relation (defined in Section 2.2.3) between them. Washizaki (2024, p. 5-8, emphasis added) provides a similar list: "regression testing … *may* involve functional and non-functional testing, such as reliability, accessibility, usability, maintainability, conversion, migration, and compatibility testing." Cases such as this are indicated by one of the following keywords: "can be", "should be", "ideally", "usually", "most", "likely", "often", or "if".

OG ISO/IEC, 2014

3. **The information is conditional.** As a more specific case of information not being universal, sometimes prerequisities must be satisfied for information to apply. For example, branch condition combination testing is equivalent to (and is therefore a synonym of) exhaustive testing *if* "each subcondition is viewed as a single input" (Peters and Pedrycz, 2000, p. 464). Likewise, statement testing can be used for (and is therefore a child of) unit testing *if* there are "less than 5000 lines of code" (p. 481). Cases such as this are indicated by the keyword "can be" or "if".

OG Miller et al., 1994

This can also apply more abstractly at the taxonomy level, where a parent-child relation only makes sense if the parent test approach exists. This occurs when a source gives a relation between qualities but at least one of them does not have an explicit approach associated with it (although it may be derived; see Section 2.3.1). For example, ISO/IEC (2023a) provides relations involving dependability and modifiability; these are tracked as qualities, not approaches, since only the qualities are described. Since the prerequisite of the relevant approach existing is *not* satisfied, these relations are omitted from any generated graphs.

4. **The information is dubious.** This happens when there is reason to doubt the information provided. If a source claims one thing that is not true, related claims lose credibility. For example, 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 Mistakes Flaw 21) casts doubt on the claim that "red-box testing" is a synonym for "acceptance testing" (Sneed and Göschl, 2000, p. 18) (see Mistakes Flaw 22). Doubts such as this can also originate from other sources. Kam (2008, p. 48) gives "user scenario testing" as a synonym of "use case testing", even though "an actor [in use case testing] can be … another system" (ISO/IEC and IEEE, 2021, p. 20), which does not fit as well with the label "user scenario testing". However, since a system can be seen as a "user" of the test item, this synonym relation is treated as implicit instead of as an outright flaw. Cases such as this are indicated by a question mark or one of the following keywords: "inferred", "should be", "ideally", "likely", "if", or "although".

OG Hetzel88

Flaws based on implicit information are themselves implicit. These are automatically detected when generating graphs and analyzing flaws (see Sections 3.1 and 3.2, respectively) by looking for the indicators of uncertainty mentioned above (see the relevant source code). These are used when creating the glossaries to capture varying degrees of nuance, such as when a test approach "can be" a child of another or is a synonym of another "most of the time" but not always. As an example, Table 4.5 contains relations that are explicit, implicit, and both; implicit relations are marked by the phrase "implied by".

2.3 Procedure

We track terminology used in the literature by building glossaries. The one most central to our research is our test approach glossary, where we give each test approach its own row to record its name and any given categories (see Section 2.2.1), synonyms (see Section 2.2.2), parents (see Section 2.2.3), and definitions. If no category is given, the "approach" category is assigned (with no accompanying citation) as a "catch-all" category. All other fields may be left blank, but a lack of definition indicates that the approach should be investigated further to see if its inclusion is meaningful (see Section 2.3.2). Any additional information from other

sources is added to or merged with the existing information in our glossary where appropriate. This includes the generic "approach" category being replaced with a more specific one, an additional synonym being mentioned, or another source describing an already-documented parent-child relation. If any new information contradicts existing information (or otherwise indicates something is wrong), this is investigated and documented (see Chapter 4), which may be done in a separate document and/or in the glossary itself. Sometimes, new information does not conflict with existing information, in which case the clearest and most concise version is kept, or they are merged to paint a more complete picture. Finally, we record any other notes, such as questions, prerequisites, and other resources to investigate.

We use similar procedures to track software qualities (see Section 2.3.1) and supplementary terminology (either shared by multiple approaches or too complicated to explain inline) in separate glossaries with a similar format. The name, definition, and synonym(s) of all terms are tracked, as well as any precedence for a related test type for a given software quality. We use heuristics to guide this process for all three glossaries to increase confidence that all terms are identified, paying special attention to the following when investigating a new source:

- glossaries and lists of terms,
- testing-related terms (e.g., terms containing "test(ing)", "review(s)", "audit(s)", "validation", or "verification"),
- terms that had emerged as part of already-discovered testing 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 implied testing approaches[7] (see Section 2.3.1).

We apply these heuristics to most investigated sources, especially established standards (see Section 2.1.1), in their entirety. Some sources, however, are only partially investigated, such as those chosen for a specific area of interest or based on a test approach that was determined to be out-of-scope. These include the following sources as described in Section 2.3.2: (ISO, 2022; 2015; Dominguez-Pumar et al., 2020; Pierre et al., 2017; Trudnowski et al., 2017; Yu et al., 2011; Tsui, 2007; Goralski, 1999).

During the first pass of data collection, we investigate and record all terminology related to software testing. Some of these terms are less applicable to test case automation—our original motivation—(such as static testing; see Appendix A.3) or quite broad (such as attacks; see Section 2.3.1), so they will be omitted during

See #39

See #55

[7]Since these methods for deriving test approaches only arose as research progressed, some examples would have been missed during the first pass(es) of resources investigated earlier in the process. While reiterating over them would be ideal, this may not be possible due to time constraints.

future analysis. Others 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. We decided that the following terms are too vague to merit tracking in our glossaries or analyzing further:

See #39, #44, and #28

- **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" (ISO/IEC and IEEE, 2017, p. 361)

OG PMBOK

- **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)

Throughout this process, information can be inferred from "surface-level" analysis that follows straightforwardly but isn't explicitly stated by any source. Examples of this are large scale integration testing and legacy system integration testing, described by Gerrard in (2000b, p. 30) and (2000a, Tab. 2; 2000b, Tab. 1), respectively. While he never explicitly says so, it can be inferred that these approaches are children of integration testing and system integration testing, respectively. Although these data do not come from the literature, they are documented for completeness; inferred flaws are given in Section 4.8 and inferred relations, if any, are included in Figures 5.1 to 5.3.

2.3.1 Derived Test Approaches

Throughout this research, we noticed many groups of test approaches that arise from some underlying area of software (testing) knowledge. The legitimacy of extrapolating new test approaches from these knowledge domains is heavily implied by the literature, but not explicitly stated as a general rule. Regardless, since the field of software is ever-evolving, it is crucial to be able to adapt to, talk about, and understand new developments in software testing. Bases for defining new test approaches suggested by the literature include coverage metrics, software qualities, and attacks. These are meaningful enough to merit analysis and are therefore in scope. Requirements may also imply related test approaches, but this mainly results in test approaches that would be out of scope. Other test approaches found in the literature are derived from programming languages or other orthogonal test approaches, but these are out of scope as this information is better captured by other approaches.

Coverage-driven Techniques

Test techniques are able to "identify test coverage items … and derive corresponding test cases" (ISO/IEC and IEEE, 2022, p. 11; similar in 2017, p. 467) in a "systematic" way (2017, p. 464). This allows for "the coverage achieved by a specific test design technique" to be calculated as a percentage of "the number of test coverage items covered by executed test cases" (2021, p. 30). Therefore, a given coverage metric implies a test approach aimed to maximize it. For example, path testing "aims to execute all entry-to-exit control flow paths in a SUT's control flow graph" (Washizaki, 2024, p. 5-13), thus maximizing the path coverage (see Sharma et al., 2021, Fig. 1).

See #63

Quality-driven Types

Since test types are "focused on specific quality characteristics" (ISO/IEC and IEEE, 2022, p. 15; 2021, p. 7; 2017, p. 473), 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 and IEEE, 2017, p. 424). This is supported by reliability and performance testing, which are both examples of test types (ISO/IEC and IEEE, 2022; 2021) that are based on their underlying qualities (Fenton and Pfleeger, 1997, p. 18). Given the importance of software qualities to defining test types, the definitions of 75 software qualities are also tracked in this current work. This was done by capturing their definitions, any precedent for the existence of an associated test type, and any synonyms (see Section 2.2.2) and additional notes in a glossary. Software qualities are "upgraded" to test types when mentioned (or implied) by a source by adding an associated test approach to this glossary (as outlined in Section 2.3) and removing the quality entry. Examples of this include conformance testing (Washizaki, 2024, 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).

OG IEEE 2013

OG ISO/IEC 2014

See #21, #23, and #27

Attacks

While attacks can be "malicious" (ISO/IEC and IEEE, 2017, p. 7), they are also given as a test practice (2022, p. 34; see Table 4.4). This means that software attacks, such as code injection and password cracking (Hamburg and Mogyorodi, 2024), can also be used for testing software, and only this kind of software attack is in scope. This is supported by the fact that penetration testing is also called "ethical hacking testing" (Washizaki, 2024, p. 13-4) or just "ethical hacking" (Gerrard, 2000b, p. 28); while hacking in general is not a test approach, doing so systematically to test and improve the software is.

Requirements-driven Approaches

While not as universally applicable, some types of requirements have associated types of testing (e.g., functional, non-functional, security). This may mean that categories of requirements *also* imply related testing approaches (such as "technical

testing"). Even assuming this is the case, some types of requirements do not apply to the code itself, and as such are out of scope, such as:

See #43

- **Nontechnical Requirement:** a "requirement affecting product and service acquisition or development that is not a property of the product or service" (ISO/IEC and IEEE, 2017, p. 293)
- **Physical Requirement:** a "requirement that specifies a physical characteristic that a system or system component must possess" (ISO/IEC and IEEE, 2017, p. 322)

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 examples:

See #63

- "An approach … for JavaScript testing (referred to as Randomized)" (Doğan et al., 2014, p. 192) is really just random testing used within JavaScript.
- "SQL statement coverage" is really just statement coverage used specifically for SQL statements (Doğan et al., 2014, Tab. 13).

OG Alalfi et al., 2010

- 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" are not exclusive to PHP (Doğan et al., 2014, Tab. 27).

OG Artzi et al., 2008

- While "HTML testing" is listed or implied by Gerrard (2000a, Tab. 2; 2000b, Tab. 1, p. 3) and Patton (2006, p. 220), it seems to be a combination of syntax testing, functionality testing, hyperlink testing/link checking, cross-browser compatibility testing, performance testing, content checking (Gerrard, 2000b, p. 3), and grey-box testing (Patton, 2006, pp. 218–220).

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. We consider these out of scope if their details are captured by their in-scope subapproaches, but record them as support for the orthogonality of test approach categories in Section 2.2.1. If a source describes an orthogonally derived approach in more detail, such as

security audits, we also record it as a distinct approach in our test approach glossary with its related information.

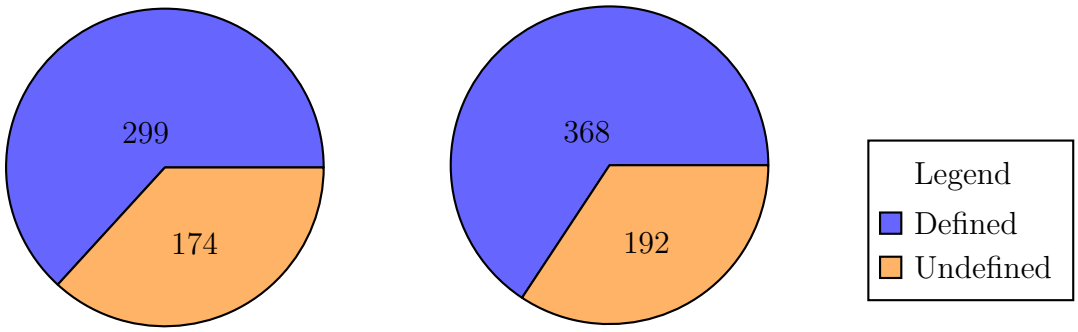
The existence of orthogonal combinations could allow for other test approaches to be extrapolated from them. For example, Moghadam (2019) uses the phrase "machine learning-assisted performance testing"; since performance testing is a known test approach, this may imply the existence of the test approach "machine learning-assisted testing". Likewise, Jard et al. (1999) use the phrases "local synchronous testing" and "remote asynchronous testing". While these can be decomposed, for example, into local testing and synchronous testing, the two resulting approaches may not be orthogonal, potentially even having a parent-child relation (defined in Section 2.2.3).

2.3.2 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, many undefined test approaches are given by (ISO/IEC and IEEE, 2022) and (Firesmith, 2015). Once we exhaust the standards in Section 2.1.1, we perform miniature literature reviews on these subsets to "fill in" the missing definitions (along with any relations), essentially "snowballing" on these terms. This process uncovers even more 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 Section 2.3.1). The following terms (and their respective related terms) were explored in the sources given:

- **Assertion Checking:** Lahiri et al. (2013); Chalin et al. (2006); Berdine et al. (2006)
- **Loop Testing**[8]**:** 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)

[8](ISO, 2022) and (ISO, 2015) were used as reference for terms but not fully investigated, (Pierre et al., 2017) and (Trudnowski et al., 2017) were added as potentially in scope, and (Dominguez-Pumar et al., 2020) and (Goralski, 1999) were added as out-of-scope examples.



(a) The 473 approaches before investigating undefined terms.

(b) The 560 approaches after investigating undefined terms.

Figure 2.2: Breakdown of how many test approaches are undefined.

- **Sandwich Testing:** Sharma et al. (2021); Sangwan and LaPlante (2006)
- **Orthogonal Array Testing**[9]**:** Mandl (1985); Valcheva (2013)
- **Backup Testing**[10]**:** Bas (2024)

Applying our procedure from Section 2.3 to these sources brings the number of testing approaches from 473 to 560 and the number of *undefined* terms from 174 to 192 (see Figure 2.2). This implies that about 79% of added test approaches are defined, which helps verify that our procedure constructively uncovers new terminology.

In addition to terms with missing definitions, some terms do not appear in the literature at all! While most test approaches arise as a result of our snowballing approach, we each have preexisting knowledge of what test approaches exist (a form of experience-based testing, if you will). As an example, we are surprised that property-based testing is not mentioned in any sources investigated, even using it as a target "stopping point" throughout this process. Test approaches such as these that arise independently of snowballing may serve as starting points for continuing research if they are not mentioned by the literature. 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:

Q #7: Is this "joke" too distracting?

See #57, #81, #88, and #125

Q #8: I think these issue refs, along with some others may actually be worth keeping in our final thesis/paper; thoughts?

1. Chaos engineering
2. Chosen-ciphertext attacks
3. Concolic testing
4. Concurrent testing
5. Destructive testing
6. Dogfooding
7. Implementation-based testing
8. Interaction-based testing

[9](Yu et al., 2011) and (Tsui, 2007) were added as out-of-scope examples.

[10]See Section 4.5.

[11]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 some-

9. Lunchtime attacks[11]
10. Parallel testing
11. Property-based testing
12. Pseudo-random bit testing
13. Rubber duck testing
14. Shadow testing

thing 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".

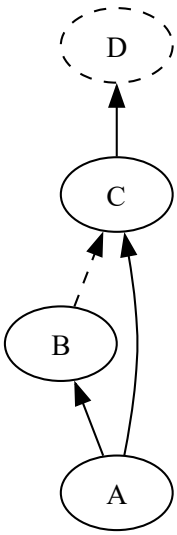
Chapter 3

Tools

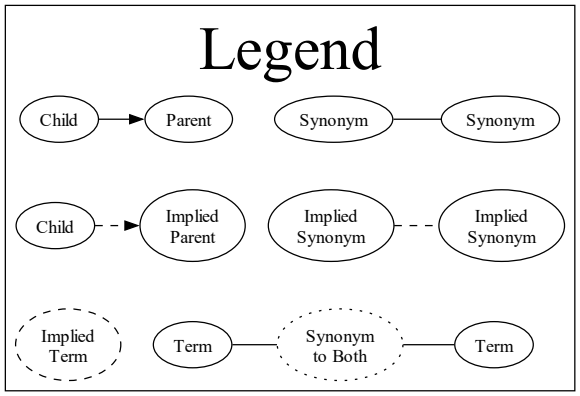
To better understand our findings, we build tools to more intuitively visualize relations between test approaches (Section 3.1) and automatically track their flaws (Section 3.2). Doing this manually would be error-prone due to the amount of data involved (for example, we identify 560 test approaches) and the number of situations where the underlying data would change, including more detailed analysis, error corrections, and the addition of data. These all require tedious updates to the corresponding graphs that may be overlooked or done incorrectly. Besides being more systematic, automating these processes also allows us to observe the impacts of smaller changes, such as unexpected flaws that arise from a new relation between two approaches. It also helps verify the tools themselves; for example, tracking a flaw manually should affect relevant flaw counts, and we can double-check this. We also define macros to help achieve our goals of maintainability, traceability, and reproducibility (Section 3.3).

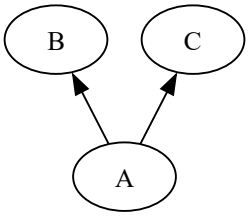
3.1 Approach Graph Generation

To better visualize how test approaches relate to each other, we develop a tool to automatically generate graphs of these relations. Since synonym (see Section 2.2.2) and parent-child relations (see Section 2.2.3) between approaches are tracked in our test approach glossary in a consistent format, they can be parsed systematically. For example, if the entries in Table 3.1 appear in the glossary, then their parent relations are displayed as Figure 3.1a in the generated graph. Relevant citation information is also captured in our glossary following the author-year citation format, including "reusing" information from previous citations. For example, the first row of Table 3.1 contains the citation "(Author, 0000; 0001)", which means that this information was present in two documents by "Author": one written in the year 0000, and one in 0001. The following citation, "(0000)", contains no author, which means it was written by the same one as the previous citation. These citations are processed according to this logic (see the relevant source code) so they can be consistently tracked throughout the analysis.

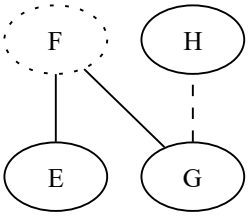


(a) Graph from Table 3.1.

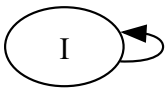




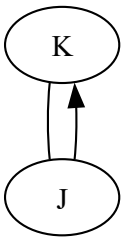
(b) Rigid graph from Table 3.1.



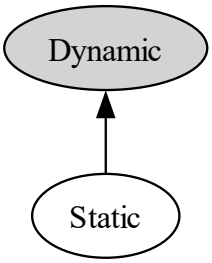
(c) Graph from Table 3.2.



(d) Self-loop graph.



(e) Graph of a pair of terms with a parent-child *and* synonym relation.



(f) Static graph.

Figure 3.1: Example generated graphs.

Table 3.1: Example glossary entries demonstrating how we track parent-child relations (see Section 2.2.3).

Name[a]	**Parent(s)**
A	B (Author, 0000; 0001), C (0000)
B	C (implied by Author, 0000)
C	D (Author, 0002)
D (implied by Author, 0002)	

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

All parent-child relations are graphed, since they are guaranteed to be visually meaningful. Synonym relations, however, are either excluded from or included in graphs as follows. For each synonym pair, at least one term will have its own row (or else it would not appear in the glossary at all), so the following cases are possible:

Is this correct grammar?

1. **(Excluded) Only one synonym has its own row.** This is a "typical" synonym relation (see Section 2.2.2) where the terms are interchangeable. The synonym *could* be included as an alternate name inside the node of its partner, but this would unnecessarily clutter the graphs.

2. **(Included) Both synonyms have their own row in the glossary.** This may indicate that the synonym relation is incorrect, since separate rows in the glossary define separate approaches (with their own definitions, nuances, etc.).

3. **(Included) Two synonym pairs share a synonym without its own row.** This is a transitive extension to the previous case. If two distinct approaches share a synonym, that implies that they are synonyms themselves, resulting in the same possibility of the relation being incorrect.

Is this multicolumn formatting OK?

Table 3.2: Example glossary entries demonstrating how we track synonym relations (see Section 2.2.2).

Name[a]	**Synonym(s)**
E	F (Author, 0000; implied by 0001)
G	F (Author, 0002), ↪ H (implied by 0000)
H	X

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

These conditions are deduced from the information parsed from the glossary. For example, if the entries in Table 3.2 appear in the glossary, then they are displayed as Figure 3.1c in the generated graph (note that X does not appear since it does not meet the criteria given above).

This allows for automatic detection of some classes of flaws. The most trivial to automate is "multi-synonym" relations, given in Case 3 above, since these are already found in order to generate the graph as desired. The list found in Section 4.2.2 is automatically generated based on glossary entries such as those found in Table 3.2. The self-referential definitions in Section 4.2.3 were also trivial, found by simply looking for lines in the generated .tex files with prefixes of the form `I -> I` (where `I` is the label used for a test approach node in these graphs). This process results in output similar to Figure 3.1d. A similar process is used to detect instances where two approaches have a synonym *and* a parent-child relation. A dictionary of each term's synonyms is built to evaluate which synonym relations are notable enough to include in the graph, and these mappings are then checked to see if one appears as a parent of the other. For example, if `J` and `K` are synonyms, a generated .tex file with a parent line starting with `J -> K` would result in these approaches being graphed as shown in Figure 3.1e.

The visual nature of these graphs makes it possible to represent both explicit and implicit relations without double counting them during the analysis in Section 3.2. If a relation is both explicit *and* implicit, the implicit relation is only shown in the graph if it is from a more "trusted" source tier (see Section 2.1). For example, note that only the explicit synonym relation between E and F from Table 3.1 is shown in Figure 3.1c. Implicit approaches and relations are denoted by dashed lines, as shown in Figures 3.1a and 3.1c; explicit approaches are *always* denoted by solid lines, even if they are also implicit. "Rigid" versions of these graphs that exclude implicit approaches and relations can also be generated; the rigid version of Figure 3.1a is given in Figure 3.1b.

Since these graphs tend to be large, it is useful to focus on specific subsets of them. To do this, we generated graphs limited to approaches in a selected approach category (see Section 2.2.1) as well as a graph of static approaches. The latter is done because ISO/IEC and IEEE (2022, Fig. 2) consider static testing to be a separate approach category and because static testing is quite different from dynamic testing (see Appendix A.3). Generated graphs focused on static testing include any relations with dynamic approaches (since they are our primary focus) and these dynamic approach nodes are colored grey, as shown in Figure 3.1f.

Additionally, more specific subsets of these graphs can be generated from a given subset of approaches, such as those pertaining to recovery or scability. These areas are of particular note, with their own sections for discussing flaws (Sections 4.5 and 4.6, respectively). Graphs of just these subsets help visualize relations between relevant test approaches, so these are generated and given in Figures 5.1a and 5.2a, respectively. By specifying sets of approaches and relations to add or remove, these generated graphs can then be updated in accordance with our recommendations; applying those given in Sections 5.1 and 5.2 results in the updated graphs in Figures 5.1b and 5.2b, respectively. Any added approaches or relations are colored orange. Recommendations can also be inherited; for example, Figure 5.3 was generated based on the modifications from Figures 5.1b and 5.2b along with other changes mentioned in Section 5.3.

3.2 Flaw Analysis

In addition to analyzing specific flaws, an overview of their amounts, sources, rigidities (see Section 2.2.4), classes, and categories is also useful. Subsets of this task can be automated (Section 3.2.1) and the remaining manual portion can be augmented with automated tools (Section 3.2.2).

To understand where flaws exist in the literature, they are grouped based on the source tiers (as described in Section 2.1) responsible for them. Each flaw is then counted *once* per source category if it appears within it *and/or* between it and a more "trusted" category. This avoids counting the same flaw more than once for a given category, which would result in the number of *occurrences* of all flaws, instead of the number of flaws *themselves*, which is more useful. The exception to this is Figure 4.1, which counts the following sources of flaws separately:

See #83

1. those that appear once in (or consistently throughout) a document (i.e., are "self-contained"),

See #137 and #138

2. those between two parts of a single document (i.e., internal conflicts),

See #137 and #138

3. those between documents by the same author(s) or standards organization(s), and

4. those within a source tier.

As before, these are not double counted, meaning that the maximum number of counted flaws possible within a *single* source tier in Figure 4.1 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 fulfill all four criteria, it would only be counted once. Note that while the different versions of the Guides to the SoftWare Engineering Body Of Knowledge (SWEBOK Guides) have different editors (Washizaki, 2024; Bourque and Fairley, 2014), we consider them to be written by the same organization: the IEEE Computer Society (Washizaki, 2025; see Section 2.1.2).

As an example of this process, consider a flaw *within* an IEEE document (e.g., two different definitions are given for a term within the same IEEE document) *and* between another IEEE document, the ISTQB glossary *and* two papers. This would add one to the following rows of Tables 4.1 and 4.2 in the relevant column:

- **Established Standards**: this flaw occurs:

  1. within one standard and
  2. between two standards.

  This increments the count by just one to avoid double counting and would do so even if only one of the above conditions was true. A more nuanced breakdown of flaws that identifies those within a singular document and those between documents by the same author is given in Figure 4.1 and explained in more detail in Section 3.2.2.

- **Terminology Collections**: this flaw occurs between a source in this category and a "more trusted" one (the IEEE standards).
- **Papers and Other Documents**: this flaw occurs between a source in this category and a "more trusted" one. Even though there are two sources in this category *and* two "more trusted" categories involved, this increments the count by just one to avoid double counting.

3.2.1 Automated Flaw Analysis

As outlined in Section 3.1, some types of flaws can be detected automatically. While just counting the total number of these types of flaws is trivial, tracking the source(s) of these flaws is more involved. Since the appropriate citations for each piece of information is tracked (see Tables 3.1 and 3.2 for examples of how these citations are formatted in the glossaries), they can be used to find the offending source tiers. This comes with the added benefit of these citations being available to be formatted for use with LaTeX's citation commands for inclusion in this document.

Comparing the authors and years of each source related to a given flaw can determine if it manifests within a single document and/or between documents by the same author(s) when creating Figure 4.1. Then, the relevant sources can be sorted into their categories based on their citations (see the relevant source code). This determines the appropriate row of Tables 4.1 and 4.2 and the appropriate graph and slice in Figure 4.1. These lists of sources can then be distilled down to sets of categories which are compared against each other to determine how many times a given flaw manifests between source tiers. Examples of this process are described in more detail in Section 3.2.2.

Alongside this citation information are the keywords relevant for assessing a piece of information's rigidity (see Section 2.2.4). This is useful when counting flaws, since they can be both explicit and implicit, but should not be double counted as both! When counting flaws in Tables 4.1 and 4.2, each one is counted only for its most "rigid" manifestation (i.e., it will only increment a value in the "Implicit" column if it is *not* also explicit).

See #83

3.2.2 Augmented Flaw Analysis

While some subsets of flaws can be deduced automatically from analyzing the testing approach glossary, most need to be tracked manually. This is done by adding comments to the relevant LaTeX files (generated or not) of the form

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

which can then be parsed to detemine where flaws occur. `CAT` is a placeholder for the flaw's category (see Section 4.2) identifier and `CLS` is a placeholder for its class (see Section 4.1) identifier. These designations are ommitted from the following examples of these comments.

Each group of sources is separated with a pipe symbol to be compared with the others, so any number of groups are permitted. We make a distinction between "self-contained" flaws and "internal" flaws. Self-contained flaws are those that manifest by comparing a document to a source of ground truth. Sometimes, these do not require an explicit comparison; for example, Omissions often fall in this category, since the lack of information is contained within a single source and does not need to be cross-checked against a source of ground truth. If only one group of sources is present in a flaw's comment, such as the first line below, it is considered to be a self-contained flaw. On the other hand, internal flaws arise when a document disagrees with itself by containing two conflicting pieces of information and include many Contradictions and Overlaps. These can even occur on the same page, such as when an acronym is given to two distinct terms (see Overlaps Flaw 8 and Overlaps Flaw 9)! If a source appears in multiple groups in a flaw's comment, it is considered to be an internal flaw. 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}
```

Discrepancies between groups are not double counted; this means the following line adds discrepancies between X and Z *and* between Y and Z, without counting the discrepancy between X and Z twice.

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

Each source is given using its BibTeX key wrapped in curly braces to mimic LaTeX's citation commands for ease of parsing, with the exception of the ISTQB glossary, due to its use of custom commands via `\citealias`. For example, the line

```
% Flaw count: {IEEE2022} | {IEEE2022} {IEEE2017}
↪  ISTQB {Kam2008} {Bas2024}
```

would be parsed as the example given in Section 3.2. Since the IEEE documents are written by the same standards organizations (ISO/IEC and IEEE), they are counted as a discrepancy between documents by the same author(s) in Figure 4.1.

The rigidity (see Section 2.2.4) of flaws can also be manually specified by inserting the phrase "implied by" after the sources of explicit information and before those of implicit information. Parsing this information follows the same rules as the automatic flaw analysis (see Section 3.2.1).

```
% Flaw count: {IEEE2022} implied by {Kam2008} |
↪  {IEEE2017} implied by {IEEE2022}
```

For example, the above line indicates that the flaws given below are present. The second flaw only affects Figure 4.1 since it is less "rigid" than the first flaw within standards (see Section 2.1.1). The rest increment their corresponding count in Figure 4.1 and Tables 4.1 and 4.2 by only one:

- an explicit discrepancy between documents by ISO/IEC and IEEE,
- an implicit discrepancy within a single document, and
- an implicit discrepancy between a paper and a standard.

Occasionally, a source from a lower tier is used as the "ground truth" for a flaw. For example, the terms "acceleration tolerance testing" and "acoustic tolerance testing" seem to only refer to software testing in (Firesmith, 2015, p. 56); 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 don't exactly seem relevant... This flaw is supported by additional papers found via a miniature literature review (described in Section 2.3.2) from a lower source tier than (Firesmith, 2015) (which is a terminology collection; see Section 2.1). However, it is really based in (Firesmith, 2015) and not in these additional papers, but if these sources where included as detailed above, this is the way it would be counted. Therefore, we document these "ground truth" sources separately to track them for traceability without incorrectly counting flaws. This is done for this specific example (and similarly for other cases) as follows:

Is this a scope flaw?

```
% Flaw count (TERMS, WRONG): {Firesmith2015}
% Ground truth: {LiuEtAl2023} {MorgunEtAl1999} {HolleyEtAl1996}
↪  {HoweAndJohnson1995}
```

3.3 LaTeX 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, many values are calculated by a Python script and saved to a file. We then assign these values to a corresponding LaTeX macro, which we can use instead of manually replacing the value throughout our documents every time it changes. Table 3.3 lists these macros, along with descriptions of what they define and their current values.

Table 3.3: LaTeX macros for calculated values.

Macro	What it Counts	Value
`\approachCount{}`	Identified test approaches	560
`\qualityCount{}`	Identified software qualities	75
`\srcCount{}`[a]	Sources used in glossaries	66
`\flawCount{}`[b]	Identified flaws	271

Continued on next page

Table 3.3: LaTeX macros for calculated values. (Continued)

Macro	What it Counts	Value
`\TotalBefore{}`[c]	Test approaches identified before process in Section 2.3.2	473
`\UndefBefore{}`[c]	Undefined test approaches identified before process in Section 2.3.2	174
`\TotalAfter{}`[c]	Test approaches identified after process in Section 2.3.2	560
`\UndefAfter{}`[c]	Undefined test approaches identified after process in Section 2.3.2	192
`\parSynCount{}`	Pairs of test approaches with a child-parent *and* synonym relation	15
`\selfCycleCount{}`	Test approaches that are a parent of themselves	3

[a] Calculated in LaTeX from source tier lists; see Section 3.3.

[b] Alias for `\totalSmntcFlawBrkdwn{13}`; see Section 3.3.

[c] These macros are defined as counters to allow them to be used in calculations within LaTeX (such as in Section 2.3.2 and Fig. 2.2).

Additionally, we count flaws based on their rigidity, source tier, and whether they are syntactic or semantic (see Sections 2.1, 2.2.4, 3.2.1, and 3.2.2 and Chapter 4). We save these counts to files, a syntactic and semantic version for each source tier, then read them in to macros to populate Tables 4.1 and 4.2. For example, `\stdSntxFlawBrkdwn{1}` corresponds to the number of explicit mistakes in standards documents, and `\stdSntxFlawBrkdwn{2}` to the number of implicit ones. These macros also include `\totalSntxFlawBrkdwn{13}` and `\totalSntxFlawBrkdwn{13}`, which are identical and track the total number of identified flaws.

Just as with calculated values, it is important that repeated text is updated consistently, which we accomplish by defining more macros. Some of these are generated by Python scripts in a similar fashion to calculated values, such as the lists of sources in Section 2.1. These are built by extracting all sources cited in our three glossaries, categorizing, sorting, and formatting them (including handling edge cases), and saving them to a file. These are then defined as `\stdSources{}`, `\metaSources{}`, `\textSources{}`, and `\paperSources{}`. The numbers of sources in each tier are also saved to build Figure 2.1 and calcuate `\srcCount{}` (see Table 3.3). However, most of the macros for reused text are created manually when the reuse is first noticed. Some of these macros account for context-specific formatting, such as capitalization, depending on how they are used; we omit these details here for brevity. We create macros to reuse many types of information throughout our documents as shown in Table 3.4.

Should I explicitly display these lists in a table here? That feels redundant and might make things unnecessarily cluttered.

Table 3.4: LaTeX macros for reused text.

Type	Macro	Used in
Flaws	\bugPattonFlaw{}	Chapter 1 and Synonyms Flaw 5
	\alphaFlaw{}	Chapter 1 and Definitions Flaw 10
	\loadFlaw{}	Chapter 1 and Definitions Flaw 12
	\expBasedCatMain{}	Chapter 1 and Section 4.2.1
	\tourFlaw{}	Chapters 1 and 4 and Definitions Flaw 9
	\perfAsFamily{}	Sections 2.2.1 and 4.2.1
	\tolTestFlaw{}	Section 3.2.2 and Terminology Flaw 4
Footnotes	\ftrnote{} \specfn{} \ucstn{}	Thesis (automated) and paper (manual) versions of Table 4.5
	\notDefDistinctIEEE{}	Definitions Flaw 4 and Section 7.1
	\gerrardDistinctIEEE{}	Table 2.3 and Definitions Flaw 41
	\distinctIEEE{}	\gerrardDistinctIEEE{}, \notDefDistinctIEEE{}, and Sections 2.2.1 and 4.2.1
Links	\ourApproachGlossary{}	Sections 2.3, 2.3.1, and 3.1
	\seeSrcCode{}	Sections 2.2.4, 3.1, 3.2.1, and 3.3
	\recFigs{}	Section 2.3 and Chapter 5
Source Tiers[a]	\stds{}	Figures 2.1 and 4.1a, Section 3.2, and Tables 4.1 and 4.2
	\metas{}	Figures 2.1 and 4.1b, Section 3.2, and Tables 4.1 and 4.2
	\texts{}	Figures 2.1 and 4.1c and Tables 4.1 and 4.2
	\papers{}	Figures 2.1 and 4.1d and Section 3.2
	\papersTbl{}	Tables 4.1 and 4.2
RQs	\rqatext{} \rqbtext{} \rqctext{}	Chapter 1 and seminar slides

Continued on next page

Table 3.4: LaTeX macros for reused text. (Continued)

Type	Macro	Used in
Misc.	`\accelTolTest{}`	`\tolTestFlaw{}` and Appendix A.1
	`\addTextEx{}`	Sections 2.1 and 2.1.3
	`\rigidBlurb{}`	Sections 2.2 and 2.2.4
	`\supersAck{}`	Acknowledgements and seminar slides
	`\supers{}`	`\supersAck{}` and Declaration of Academic Achievement
	`\displayNL{}`	Sections 3.1, 3.2.2, and 3.3 and Table 3.5

[a] See Section 2.1.

In addition to this thesis, we also prepare a conference paper based on our research. While we can reuse most content without modifying it, there are some formatting differences between the two document types. For example, our thesis uses the `natbib` package for citations while the IEEE guidelines for paper submissions suggest the use of `cite` (Shell, 2015, p. 8); we define aliases so that we can reuse text that includes citations (see the relevant source code).

Is this a "command"?

In general, we use the command `\ifnotpaper` to allow for manual distinctions between the two documents' formats, such as how they handle citations, using this basic format:

```
\ifnotpaper <thesis code> \else <paper code> \fi
```

For example, in Section 2.2.1, we provide a list of non-IEEE sources that support a claim made by the IEEE. Since we sort sources based on trustworthiness (see Section 2.1), publication year, and number of authors, the relevant thesis code is:

```
(\citealp[pp.~5\=/6 to 5\=/7]{SWEBOK2024};
↪  \citealpISTQB{}; \citealp[pp.~807\==808]{Perry2006};
↪  \citealp[pp.~443\==445]{PetersAndPedrycz2000};
↪  \citealp[p.~218]{KuļešovsEtAl2013}\todo{OG Black, 2009};
↪  \citealp[pp.~9, 13]{Gerrard2000a})
```

Meanwhile, IEEE guidelines prefer that sources are kept in separate brackets, sorted in order of their first appearance in the document. Therefore, paper code for this list of sources is:

```
\cite[pp.~443\==445]{PetersAndPedrycz2000},
↪  \cite[pp.~5\=/6 to 5\=/7]{SWEBOK2024}, \cite{ISTQB},
↪  \cite[pp.~807\==808]{Perry2006},
↪  \cite[pp.~9, 13]{Gerrard2000a},
↪  \cite[p.~218]{KuļešovsEtAl2013}
```

Later: Ensure this code matches the final version!

In particular, note the usage of the `\cite{}` command, the *lack* of use of the custom alias for citing the ISTQB glossary, the different order, and the lack of

the `\todo{}`, since these are only rendered for reference in the thesis. For other edge cases with different formatting styles for these document types, we define the macros given in Table 3.5 via an example usage of the macro(s) and how the code is rendered in our thesis *and* paper[1]:

Does this footnote make sense?

Table 3.5: LaTeX macros for handling formatting differences between thesis and paper.

Context	Displayed as
Code	`\refHelper{} \citet{IEEE2022} \multiAuthHelper{form}` `↪ the basis of this \docType{}.`
Thesis	ISO/IEC and IEEE (2022) form the basis of this thesis.
Paper	Reference [16] forms the basis of this paper.
Code	`\flawref{cat-acro}`
Thesis	Overlaps Flaw 8
Paper	Section III-B2
Code	`\reduns{}`
Thesis	Redunancies
Paper	Redundancies[a]

[a] Section omitted for brevity.

[1]Since this document is the thesis version, some of the paper renderings are hardcoded.

Chapter 4

Flaws

After gathering all these data[1], we find many flaws. To better understand and analyze them, we group them by their syntax and their semantics. Syntactic flaws (Section 4.1) describe *how* a flaw manifests, such as information that is wrong (a "mistake"; Section 4.1.1) or missing (an "omission"; Section 4.1.2). On the other hand, semantic flaws (Section 4.2) describe the knowledge domain in which a flaw manifests, such as a flaw between synonyms (Section 4.2.2) or parent-child relations (Section 4.2.3). As an example, 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). This is a case of contradictory definitions, so it appears with both the contradictions (Section 4.1.3) *and* the definition flaws (Section 4.2.4). Within these sections, "more significant" flaws are listed first, followed by "less significant" ones omitted from the paper version of this thesis. They are then sorted based on their source tier (see Section 2.1).

A summary of how many flaws there are by syntax and by semantics is shown in Tables 4.1 and 4.2, respectively, where a given row corresponds to the number of flaws either within that source tier (see Section 2.1) and/or with a "more trusted" one (i.e., a previous row in the table). The numbers of (Exp)licit and (Imp)licit (see Section 2.2.4) flaws are also presented in these tables. Since each flaw is grouped by syntax *and* by semantics, the totals per source and grand totals in these tables are equal. However, the numbers of flaws listed in Sections 4.1 and 4.2 are not equal, as those automatically uncovered based on semantics (see Section 3.2.1) are only listed in the corresponding category section for clarity; they still contribute to the counts in Table 4.1!

Moreover, certain "subsets" of testing revealed many interconnected flaws. These are given in their respective sections as a "third view" to keep related information together, but still count towards the syntaxes and semantics of flaws listed above (Section 3.2.2 outlines how this is done), causing a further mismatch between the counts in Tables 4.1 and 4.2 and the counts in Sections 4.1 and 4.2. The "problem" subsets of testing include Functional Testing, Operational (Acceptance) Testing (OAT), Recovery Testing, Scalability Testing, and Compatibility

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

Testing. Finally, Inferred Flaws are also given for completeness, despite being less certain and thus not contributing to any counts.

Table 4.1: Breakdown of identified Syntactic Flaws by Source Tier.

	Mistakes		**Omissions**		**Contradictions**		**Ambiguities**		**Overlaps**		**Redunancies**		
Source Tier	**Exp**	**Imp**	**Exp**	**Imp**	**Exp**	**Imp**	**Exp**	**Imp**	**Exp**	**Imp**	**Exp**	**Imp**	**Total**
Established Standards	7	1	2	0	18	10	4	0	8	0	0	0	50
Terminology Collections	11	0	1	0	34	17	14	3	5	1	2	0	88
Textbooks	7	1	2	0	38	4	5	0	1	0	0	0	58
Papers and Others	9	1	4	0	24	20	9	3	2	1	2	0	75
Total	34	3	9	0	114	51	32	6	16	2	4	0	271

Table 4.2: Breakdown of identified Semantic Flaws by Source Tier.

	Categories		**Synonyms**		**Parents**		**Definitions**		**Terminology**		**Citations**		
Source Tier	**Exp**	**Imp**	**Exp**	**Imp**	**Exp**	**Imp**	**Exp**	**Imp**	**Exp**	**Imp**	**Exp**	**Imp**	**Total**
Established Standards	10	7	4	2	6	0	14	2	5	0	0	0	50
Terminology Collections	10	14	9	3	9	2	22	0	13	2	4	0	88
Textbooks	2	0	16	0	9	4	19	1	7	0	0	0	58
Papers and Others	10	13	14	9	11	1	8	0	7	2	0	0	75
Total	32	34	43	14	35	7	63	3	32	4	4	0	271

Table 4.3: Sources of flaws based on source tier.

Flaw between a document from a source tier below and a ...	**source of ground truth**	**part of the same document**	**document with the same author**	**standard**	**collection**	**textbook**	**paper**
Established Standards	11	29	25	5	—	—	—
Terminology Collections	18	10	0	44	17	—	—
Textbooks	0	4	0	26	14	14	—
Papers and Other Documents	9	4	0	22	23	10	7

4.1 Syntactic Flaws

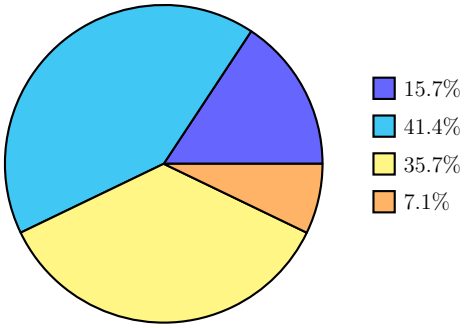
The following sections list observed flaws grouped by *how* they manifest. These include Mistakes, Omissions, Contradictions, Ambiguities, Overlaps, and Redunancies.

4.1.1 Mistakes

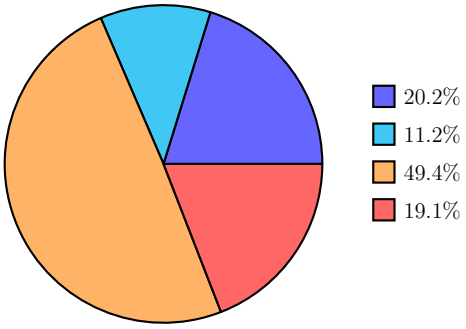
The following are cases where information is incorrect; this includes cases Terminology included that should *not* have been, untrue claims about Citations, and simple typos:

1. Since errors are distinct from defects/faults (ISO/IEC and IEEE, 2010, pp. 128, 140; Washizaki, 2024, p. 12-3; van Vliet, 2000, pp. 399–400), error guessing should instead be called "defect guessing" if it is based on a "checklist of potential defects" (ISO/IEC and IEEE, 2021, p. 29) 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, 2024, p. 5-13). One (or both) of these proposed terms may be useful in tandem with "error guessing", which would focus on errors as traditionally defined; this would be a subapproach of error-based testing (implied by van Vliet, 2000, p. 399).

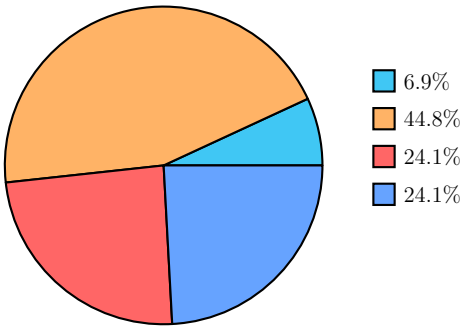
2. Similarly, "fault seeding" is not a synonym of "error seeding" as claimed by (ISO/IEC and IEEE, 2017, p. 165; van Vliet, 2000, p. 427). The term "error seeding", used by (ISO/IEC and IEEE, 2017, p. 165; Firesmith, 2015, p. 34; van Vliet, 2000, p. 427), should be abandoned in favour of "fault seeding", as 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



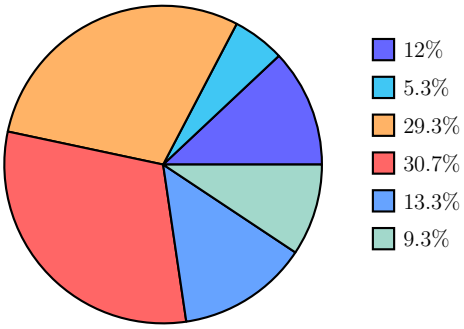
(a) Flaws found in Established Standards.



(b) Flaws found in Terminology Collections.



(c) Flaws found in Textbooks.



(d) Flaws found in Papers and Other Documents.

Legend

- With a source of ground truth
- Within a single document
- Between documents by the same author(s) or standards organization(s)
- Between a document from this category and a standard
- Between a document from this category and a collection
- Between a document from this category and a textbook
- Between a document from this category and a paper

Figure 4.1: Sources of flaws based on source tier.

the number of new faults and the number of introduced faults that were discovered (van Vliet, 2000, p. 427).

3. Hamburg and Mogyorodi (2024) classify ML model testing as a test level, which they define as "a specific instantiation of a test process": a vague definition that does not match the one in Table 2.1.

4. The terms "acceleration tolerance testing" and "acoustic tolerance testing" seem to only refer to software testing in (Firesmith, 2015, p. 56); 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 don't exactly seem relevant…

Is this a scope flaw?

5. The differences between the terms "error", "failure", "fault", "defect" are significant and meaningful (ISO/IEC and IEEE, 2010, pp. 128, 139–140; Washizaki, 2024, p. 12-3; van Vliet, 2000, pp. 399–400), 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"!

6. Peters and Pedrycz claim that "structural testing subsumes white box testing" but they seem to describe the same thing: they 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 (2000, p. 447)!

OG Beizer

7. Kam (2008, p. 46) says that the goal of negative testing is "showing that a component or system does not work" which is not true; if robustness is an important quality for the system, then testing the system "in a way for which it was not intended to be used" (Hamburg and Mogyorodi, 2024) (i.e., negative testing) is one way to help test this!

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

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

10. Since keyword-driven testing can be used for automated *or* manual testing (ISO/IEC and IEEE, 2016, pp. 4, 6), the claim that "test cases can be either manual test cases or keyword test cases" (p. 6) is incorrect.

11. The definition of "math testing" given by Hamburg and Mogyorodi (2024) is too specific to be useful, likely taken from an example instead of a general

definition: "testing to determine the correctness of the pay table implementation, the random number generator results, and the return to player computations".

12. A similar issue exists with multiplayer testing, where its definition specifies "the casino game world" (Hamburg and Mogyorodi, 2024).

13. "Par sheet testing" from (Hamburg and Mogyorodi, 2024) seems to refer to the specific example mentioned in Mistakes Flaw 11 and does not seem more widely applicable, since a "PAR sheet" is "a list of all the symbols on each reel of a slot machine" (Bluejay, 2024).

Does this belong here?

14. The source that Hamburg and Mogyorodi (2024) cite for the definition of "test type" does not seem to actually provide a definition.

15. The same is true for "visual testing" (Hamburg and Mogyorodi, 2024).

16. The same is true for "security attack" (Hamburg and Mogyorodi, 2024).

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

18. Peters and Pedrycz imply that decision coverage is a child of both c-use coverage *and* p-use coverage (2000, Fig. 12.31); this seems incorrect, since decisions are the result of p-uses (*not* c-uses) and only the p-use relation is implied by (ISO/IEC and IEEE, 2021, Fig. F.1).

OG Reid 1996

19. 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

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

21. Sneed and Göschl (2000, p. 18) 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, Sakamoto et al. (2013, pp. 345–346) describe "black-box integration testing".

OG Hetzel88

find more examples

22. The previous flaw makes the claim that "red-box testing" is a synonym for "acceptance testing" (Sneed and Göschl, 2000, p. 18) lose credibility.

23. 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, 2024, p. 5-15; similar in van Vliet, 2000, pp. 428–429) which is not a requirement of back-to-back testing.

4.1.2 Omissions

The following are cases where information (usually Definitions) *should have* been included but was not:

1. Integration testing, system testing, and system integration testing are all listed as "common test levels" (ISO/IEC and IEEE, 2022, p. 12; 2021, p. 6), but no definitions are given for the latter two, making it unclear what "system integration testing" is; it is a combination of the two? somewhere on the spectrum between them? It is listed as a child of integration testing by Hamburg and Mogyorodi (2024) and of system testing by Firesmith (2015, p. 23).

2. Similarly, component testing, integration testing, and component integration testing are all listed in (ISO/IEC and IEEE, 2017), but "component integration testing" is only defined as "testing of groups of related components" (ISO/IEC and IEEE, 2017, p. 82); it is a combination of the two? somewhere on the spectrum between them? As above, it is listed as a child of integration testing by Hamburg and Mogyorodi (2024).

3. Kam (2008, p. 42) says "See *boundary value analysis*," for the glossary entry of "boundary value testing" but does not provide this definition.

4. The acronym "SoS" is used but not defined by Firesmith (2015, p. 23).

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

6. 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).

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 isn't 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); I think usability and/or performance would have made sense.

[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.

4.1.3 Contradictions

The following are cases where multiple sources of information (sometimes within the same document!) disagree; note that cases where all sources of information are incorrect are considered contradictions and not Mistakes, since this would require analysis that has not been performed yet:

1. Regression testing and retesting are sometimes given as two distinct approaches (ISO/IEC and IEEE, 2022, p. 8; Firesmith, 2015, p. 34), but sometimes regression testing is defined as a form of "selective retesting" (ISO/IEC and IEEE, 2017, p. 372; Washizaki, 2024, pp. 5-8, 6-5, 7-5 to 7-6; 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. "Software testing" is often defined to exclude static testing (Firesmith, 2015, p. 13; Ammann and Offutt, 2017, p. 222; Peters and Pedrycz, 2000, p. 439), restricting "testing" to mean dynamic validation (Washizaki, 2024, 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, 2024, p. 5-2). This is done by ISO/IEC and IEEE (2022, p. 17) and Gerrard (2000a, pp. 8–9); the former even explicitly *exclude* static testing in another document (2017, p. 440)!

3. 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 [sic] 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.

See #14

4. Performance testing and security testing are given as subtypes of reliability testing by ISO/IEC (2023a), but these are all listed separately by Firesmith (2015, p. 53).

5. Similarly, random testing is a subtechnique of specification-based testing (ISO/IEC and IEEE, 2022, pp. 7, 22; 2021, pp. 5, 20, Fig. 2; Washizaki,

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

2024, p. 5-12; Hamburg and Mogyorodi, 2024) but is listed separately by Firesmith (2015, p. 46).

6. Path testing "aims to execute all entry-to-exit control flow paths in a SUT's control flow graph" (Washizaki, 2024, 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."

7. 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).

8. Alpha testing is performed by "users within the organization developing the software" (ISO/IEC and IEEE, 2017, p. 17), "a small, selected group of potential users" (Washizaki, 2024, p. 5-8), or "roles outside the development organization" conducted "in the developer's test environment" (Hamburg and Mogyorodi, 2024).

9. "Use case testing" is given as a synonym of "scenario testing" by Hamburg and Mogyorodi (2024) but listed separately by ISO/IEC and IEEE (2022, Fig. 2) and described as a "common form of scenario testing" in (2021, p. 20). This implies that use case testing may instead be a child of user scenario testing (see Table 4.5).

OG Hass, 2008

10. The terms "test level" and "test stage" are given as synonyms (Hamburg and Mogyorodi, 2024; implied by Gerrard, 2000a, p. 9), but Washizaki (2024, 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 2.2.3) to "test level" instead. However, the examples listed—unit testing, integration testing, system testing, and acceptance testing (Washizaki, 2024, pp. 5-6 to 5-7)—are commonly categorized as "test levels" (see Section 2.2.1).

11. 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.

12. Load testing is performed with loads "between anticipated conditions of low, typical, and peak usage" (ISO/IEC and IEEE, 2022, p. 5) or loads that are as large as possible (Patton, 2006, p. 86).

13. State testing requires that "all states in the state model … [are] 'visited' " in (ISO/IEC and IEEE, 2021, p. 19) which is only one of its possible criteria in (Patton, 2006, pp. 82-83).

14. ISO/IEC and IEEE (2017, p. 456) say system testing is "conducted on a complete, integrated system" (which Peters and Pedrycz (2000, Tab. 12.3)

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

15. "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.

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

OG Beizer

17. 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 (2021) 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).

OG 2015

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

OG ISO1984

19. 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" (2021, p. 18) and one where "the conditions and actions are generally described but are incomplete" (2017, p. 175).

OG 2015

20. ISO/IEC and IEEE's 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" (2017, p. 83)—is *much* more vague than the definition given in (2021, p. 27): testing that exercises "control flow sub-paths from each variable definition to each c-use of that definition (with no intervening definitions)" (similar in van Vliet, 2000, p. 425; Peters and Pedrycz, 2000, p. 479).

21. 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]

22. The claim that "test cases can be either manual test cases or keyword test cases" (ISO/IEC and IEEE, 2016, p. 6) implies that "keyword-driven testing" could be a synonym of "automated testing" instead of its child, which seems more reasonable (2016, p. 4; 2022, p. 35).

[4]For simplicity, this implied categorization as "technique" is omitted from Table 4.4.

23. Different capitalizations of the abbreviations of "computation data use" and "predicate data use" are used: the lowercase "c-use" and "p-use" (ISO/IEC and IEEE, 2021, pp. 3, 27-29, 35-36, 114-155, 117-118, 129; 2017, p. 124; Peters and Pedrycz, 2000, p. 477, Tab. 12.6) and the uppercase "C-use" and "P-use" (van Vliet, 2000, pp. 424-425).

24. Similarly for "definition-use" (such as in "definition-use path"), both the lowercase "du" (ISO/IEC and IEEE, 2021, pp. 3, 27, 29, 35, 119-121, 129; Peters and Pedrycz, 2000, pp. 478-479) and the uppercase "DU" (van Vliet, 2000, p. 425) are used.

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

26. 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, 2021, p. 29; similar in 2017, p. 125; Washizaki, 2024, p. 5-13; Peters and Pedrycz, 2000, p. 479); however, paths containing simple cycles may also be required (van Vliet, 2000, p. 425).

27. Van Vliet says that all-p-uses testing is only stronger than all-edges (branch) testing if there are infeasible paths (2000, pp. 432-433), but ISO/IEC and IEEE do not specify this caveat (2021, Fig. F.1).

OG Reid, 1996

28. Similarly, van Vliet says that all-du-paths testing is only stronger than all-uses testing if there are infeasible paths (2000, pp. 432-433), but Washizaki does not specify this caveat (2024, p. 5-13).

29. 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).

Does this merit counting this as an Ambiguity as well as a Contradiction?

30. Although ad hoc testing is classified as a "technique" (Washizaki, 2024, p. 5-14), it is one in which "no recognized test design technique is used" (Kam, 2008, p. 42).

4.1.4 Ambiguities

The following are cases where information (usually Definitions or distinctions between Terminology) is unclear:

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

Is this a def flaw?

2. 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.

3. "Installability testing" is given as a test type (ISO/IEC and IEEE, 2022, p. 22; 2021, p. 38; 2017, p. 228), while "installation testing" is given as a test level (van Vliet, 2000, p. 439; implied by Washizaki, 2024, p. 5-8). Since "installation testing" is not given as an example of a test level throughout the sources that describe them (see Section 2.2.1), it is likely that the term "installability testing" with all its related information should be used instead.

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

5. "Data definition" is defined as a "statement where a variable is assigned a value" (ISO/IEC and IEEE, 2021, 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.

OG 2015

6. While "error" is defined as "a human action that produces an incorrect result" (Washizaki, 2024, pp. 12-3; van Vliet, 2000, p. 399), Washizaki does not use this consistently, sometimes implying that errors can be instrinsic to software itself (2024, pp. 4-9, 6-5, 7-3, 12-4, 12-9, 12-13).

OG [14]

7. Washizaki (2024, 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 match the definition of "failure": the inability of a system "to perform a required function or … within previously specified limits" (ISO/IEC and IEEE, 2019, p. 7; 2010, p. 139; similar in van Vliet, 2000, p. 400) that is "externally visible" (ISO/IEC and IEEE, 2019, p. 7; similar in van Vliet, 2000, p. 400).

Q #9: I ignore "syntax errors, runtime errors, and logical errors" (Washizaki, 2024, p. 16-13, cf. p. 18-13) since they seem to be in different domains. Does that make sense? How should I document this?

8. Similarly, Washizaki (2024, 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." This should either be called "*error* tolerance" or be described as "detecting *faults* and then recovering from them", since "error" and "fault" have distinct meanings (Washizaki, 2024, p. 5-3; van Vliet, 2000, pp. 399–400).

OG ISO/IEC, 2005

The intent of this term-definition pair is unclear, as the strategies given—"backing up and retrying, using auxiliary code and voting algorithms, and replacing an erroneous value with a phony value that will have a benign effect" (Washizaki, 2024, p. 4-11)—could be used for errors or faults.

9. While ergonomics testing is out of scope (as it tests hardware, not software), 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*.

10. Similarly, end-to-end testing is defined as testing "in which business processes are tested from start to finish under production-like circumstances" (Hamburg and Mogyorodi, 2024); it is unclear whether this tests the business processes themselves or the system's role in performing them.

11. 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 testing approach or a tool that supports testing?

12. 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).

13. Similarly, it is ambiguous whether "tool/environment testing" refers to 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.

14. Retesting and regression testing seem to be separated from the rest of the testing approaches (ISO/IEC and IEEE, 2022, p. 23), but it is not clearly detailed why; Barbosa et al. (2006, p. 3) consider regression testing to be a test level, but since it is not given as an example of a test level throughout the sources that describe them (see Section 2.2.1), it is likely that it is at best not universal and at worst incorrect.

15. "Conformance testing" is implied to be a synonym of "compliance testing" by Kam (2008, p. 43) which only makes sense because of the vague definition of the latter: "testing to determine the compliance of the component or system".

4.1.5 Overlaps

The following are cases where information overlaps, such as nonatomic Definitions and Terminology:

1. ISO/IEC and IEEE (2022, p. 34) give the "landmark tour" as an example of "a tour used for exploratory testing", but they also use the analogy of "a tour guide lead[ing] a tourist through the landmarks of a big city" to describe tours in general. Is the distinction between them the fact that landmark tours are pre-planned and follow a decided-upon sequence (p. 34)?

2. ISO/IEC and IEEE say that "test level" and "test phase" are synonyms, both meaning a "specific instantiation of [a] test sub-process" (2017, pp. 469, 470; 2013, p. 9), 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 (2017, pp. 420, 509; Perry, 2006, p. 56).

3. ISO/IEC and IEEE define "error" as "a human action that produces an incorrect result", but also as "an incorrect result" itself (2010, p. 128). Since faults are inserted when a developer makes an error (2010, pp. 128, 140; Washizaki, 2024, p. 12-3; van Vliet, 2000, pp. 399–400), this means that they are "incorrect results", making "error" and "fault" synonyms and the distinction between them less useful.

4. Additionally, "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" (ISO/IEC and IEEE, 2010, p. 128; similar in Washizaki, 2024, 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".

5. The SWEBOK Guide V4 defines "privacy testing" as testing that "assess[es] the security and privacy of users' personal data to prevent local attacks" (Washizaki, 2024, p. 5-10); 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, [sic] are protected so that" only "authorized persons or systems" can use them as intended.

6. "Orthogonal array testing" (Washizaki, 2024, 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").

7. ISO/IEC and IEEE provide a definition for "inspections and audits" (2017, p. 228), 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.

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

9. The same is true for "hardware-in-the-loop testing" and "human-in-the-loop testing" ("HIL") (Firesmith, 2015, p. 23), although Preuße et al. (2012, p. 2) use "HiL" for the former.

10. "Visual browser validation" is described as both static *and* dynamic in the same table (Gerrard, 2000a, Tab. 2), even though they are implied to be orthogonal classifications: "test types can be static *or* dynamic" (p. 12, emphasis added).

4.1.6 Redunancies

The following are cases of redundant information:

1. "Ethical hacking testing" is given as a synonym of penetration testing by Washizaki (2024, p. 13-4), which seems redundant; Gerrard (2000b, p. 28) uses the term "ethical hacking" which is clearer.

2. While correct, ISTQB's definition of "specification-based testing" is not helpful: "testing based on an analysis of the specification of the component or system" (Hamburg and Mogyorodi, 2024).

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

4.2 Semantic Flaws

The following sections list observed flaws grouped by *what area* they manifest in. These include Approach Category Flaws, Synonym Relation Flaws, Parent-Child Relation Flaws, Definition Flaws, Terminology Flaws, and Citation Flaws.

4.2.1 Approach Category Flaws

While the IEEE categorization of testing 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 2.2). 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! Additionally, the boundaries between approaches within a category may be unclear: "although each technique is defined independently of all others, in practice [sic] some can be used in combination with other techniques" (ISO/IEC

and IEEE, 2021, p. 8). For example, "the test coverage items derived by applying equivalence partitioning can be used to identify the input parameters of test cases derived for scenario testing" (p. 8). Even the categories themselves are not consistently defined, and some approaches are categorized differently by different sources; these differences are tracked so they can be analyzed more systematically.

See #21

1. Hamburg and Mogyorodi (2024) classify ML model testing as a test level, which they define as "a specific instantiation of a test process": a vague definition that does not match the one in Table 2.1.

OG Beizer

2. 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 (2021) 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).

3. Since keyword-driven testing can be used for automated *or* manual testing (ISO/IEC and IEEE, 2016, pp. 4, 6), the claim that "test cases can be either manual test cases or keyword test cases" (p. 6) is incorrect.

4. 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.[5]

5. Retesting and regression testing seem to be separated from the rest of the testing approaches (ISO/IEC and IEEE, 2022, p. 23), but it is not clearly detailed why; Barbosa et al. (2006, p. 3) consider regression testing to be a test level, but since it is not given as an example of a test level throughout the sources that describe them (see Section 2.2.1), it is likely that it is at best not universal and at worst incorrect.

6. Although ad hoc testing is classified as a "technique" (Washizaki, 2024, p. 5-14), it is one in which "no recognized test design technique is used" (Kam, 2008, p. 42).

7. "Visual browser validation" is described as both static *and* dynamic in the same table (Gerrard, 2000a, Tab. 2), even though they are implied to be orthogonal classifications: "test types can be static *or* dynamic" (p. 12, emphasis added).

Some category flaws can be detected automatically, such as test approaches with more than one category. These are given in Table 4.4 and include experience-based testing, which is of particular note. ISO/IEC and IEEE categorize experience-based testing as both a test design technique and a test practice on the same

[5]For simplicity, this implied categorization as "technique" is omitted from Table 4.4.

page—twice (2022, Fig. 2, p. 34)! These authors say "experience-based testing practices like exploratory testing … are not … techniques for designing test cases", although they "can use … test techniques" (2021, p. viii), which they support in (2022, p. 33) along with scripted testing. This implies that "experience-based test design techniques" are used *by* the practice of experience-based testing which is not *itself* a test technique (and similarly with scripted testing). If this is the case, it blurs the line between "practice" and "technique", which may explain why experience-based testing is categorized inconsistently in the literature.

See #64

However, this might mean that a practice such as experience-based testing can be viewed as a "class of test case design techniques" (ISO/IEC and IEEE, 2022, p. 4). If *this* is the case, then test approaches that are a collection of specific subtechniques may be considered practices. The following test approaches are each described as a "class", "family", or "collection" of techniques by the sources given, which seems to support this:

- Combinatorial testing (ISO/IEC and IEEE, 2022, p. 3; 2021, p. 2; Washizaki, 2024, p. 5-11)
- Data flow testing (2021, p. 3; implied by Washizaki, 2024, p. 5-13)
- Performance(-related) testing (ISO/IEC and IEEE, 2021, p. 38; Moghadam, 2019, p. 1187[6])
- Security testing (implied by ISO/IEC and IEEE, 2021, p. 40)
- Fault tolerance testing (implied by Washizaki, 2024, p. 4-11)

Of the above, security testing is an outlier, since it is consistently categorized as a test type (ISO/IEC and IEEE, 2022, pp. 9, 22, 26–27; 2021, pp. 7, 40, Tab. A.1; 2017, p. 405; implied by its quality (ISO/IEC, 2023a; Washizaki, 2024, p. 13-4); Firesmith, 2015, p. 53) despite consisting of "a number of techniques" (ISO/IEC and IEEE, 2021, p. 40), although this may be distinct from the notion of "test technique" described in Table 2.1. In addition, specification-based testing and structure-based testing may also be considered "families" since they are quite broad with many subtechniques and are described as "complementary" alongside experience-based testing (ISO/IEC and IEEE, 2021, p. 8, Fig. 2).

OG 2013

Subapproaches of experience-based tesing, such as error guessing and exploratory testing, are also categorized ambiguously, causing confusion on how categories and parent-child relations (see Section 2.2.3) interact. ISO/IEC and IEEE (2022, p. 34, emphasis added) say that a previous standard (2021) "describes the experience-based test *design technique* of error guessing. Other experience-based test *practices* include (but are not limited to) exploratory testing …, tours, attacks, and checklist-based testing". This seems to imply that error guessing is both a technique *and* a practice, which does not make sense if these categories are orthogonal. Similarly,

[6]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 Section 5.3).

the conflicting categorizations of beta testing in Table 4.4 may propagate to its children closed beta testing and open beta testing; since this is an inferrence, it is omitted from that table and instead included in Table 4.6. These kinds of inconsistencies between parent and child test approach categorizations may indicate that categories are not transitive or that more thought must be given to them.

Table 4.4: Test approaches with more than one category.

Approach	**Category 1**	**Category 2**
Capacity Testing	Technique (ISO/IEC and IEEE, 2021, p. 38)	Type (ISO/IEC and IEEE, 2022, p. 22; 2013, p. 2; Firesmith, 2015, p. 53; implied by its quality (ISO/IEC, 2023a; ISO/IEC and IEEE, 2021, Tab. A.1))
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)
Endurance Testing	Technique (ISO/IEC and IEEE, 2021, p. 38)	Type (ISO/IEC and IEEE, 2013, p. 2; implied by Firesmith, 2015, p. 55)
Experience-based Testing	Practice (ISO/IEC and IEEE, 2022, pp. 22, 34; 2021, p. viii)	Technique (ISO/IEC and IEEE, 2022, pp. 4, 22; 2021, p. 4; Washizaki, 2024, p. 5-13; Hamburg and Mogyorodi, 2024; Firesmith, 2015, pp. 46, 50; implied by ISO/IEC and IEEE, 2022, p. 34)
Exploratory Testing	Practice (ISO/IEC and IEEE, 2022, pp. 20, 22, 34; 2021, p. viii)	Technique (ISO/IEC and IEEE, 2022, p. 34; Washizaki, 2024, p. 5-14; Firesmith, 2015, p. 50; implied by ISO/IEC and IEEE, 2022, p. 33; inferred from experience-based testing)
Load Testing	Technique (ISO/IEC and IEEE, 2021, p. 38)	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	Practice (ISO/IEC and IEEE, 2022, p. 22; 2021, p. viii)	Technique (Engström and Petersen, 2015, pp. 1-2; Kam, 2008, p. 4; implied by ISO/IEC and IEEE, 2022, p. 32; 2021, p. 7; 2017, p. 469)

Continued on next page

Table 4.4: Test approaches with more than one category. (Continued)

Approach	**Category 1**	**Category 2**
Mutation Testing	Methodology (ISO/IEC and IEEE, 2017, p. 286)	Technique (Washizaki, 2024, p. 5-15; van Vliet, 2000, pp. 428-429)
Performance Testing	Technique (ISO/IEC and IEEE, 2021, p. 38)	Type (ISO/IEC and IEEE, 2022, pp. 7, 22, 26-27; 2021, p. 7; implied by Firesmith, 2015, p. 53)
Stress Testing	Technique (ISO/IEC and IEEE, 2021, p. 38)	Type (ISO/IEC and IEEE, 2022, pp. 9, 22; 2017, p. 442; implied by Firesmith, 2015, p. 54)
A/B Testing	Practice (ISO/IEC and IEEE, 2022, p. 22)	Type (implied by Firesmith, 2015, p. 58)
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)	Technique (implied by Hamburg and Mogyorodi, 2024)
Automated Testing	Practice (ISO/IEC and IEEE, 2022, pp. 20, 22)	Technique (implied by ISO/IEC and IEEE, 2022, p. 35)
Back-to-Back Testing	Practice (ISO/IEC and IEEE, 2022, p. 22)	Technique (implied by ISO/IEC and IEEE, 2022, p. 35)
Beta Testing	Type (implied by Firesmith, 2015, p. 58)	Level (ISO/IEC and IEEE, 2022, p. 22; inferred from acceptance testing)
Continuous Testing	Practice (Washizaki, 2024, p. 6-13)	Technique (implied by ISO/IEC and IEEE, 2022, p. 35)

Continued on next page

Table 4.4: Test approaches with more than one category. (Continued)

Approach	**Category 1**	**Category 2**
Error Guessing	Practice (implied by ISO/IEC and IEEE, 2022, p. 34)	Technique (ISO/IEC and IEEE, 2022, pp. 4, 22, 34; 2021, pp. 4, 11, 29, Fig. 2; 2013, p. 3; Washizaki, 2024, p. 5-13; Firesmith, 2015, p. 50)
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; 2021, p. 6; Washizaki, 2024, p. 5-7; Hamburg and Mogyorodi, 2024; Sakamoto et al., 2013, p. 343; Peters and Pedrycz, 2000, Tab. 12.3; van Vliet, 2000, p. 438; implied by Barbosa et al., 2006, p. 3)
Manual Testing	Practice (ISO/IEC and IEEE, 2022, p. 22)	Technique (implied by ISO/IEC and IEEE, 2022, p. 35)
Mathematical-based Testing	Practice (ISO/IEC and IEEE, 2022, pp. 22, 36)	Technique (implied by ISO/IEC and IEEE, 2022, p. 36)
Penetration Testing	Technique (ISO/IEC and IEEE, 2021, p. 40; Hamburg and Mogyorodi, 2024)	Type (implied by Firesmith, 2015, p. 57; inferred from security testing)
Procedure Testing	Technique (implied by Firesmith, 2015, p. 47)	Type (ISO/IEC and IEEE, 2022, pp. 7, 22; 2021, 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; 2021, p. 6; 2017, p. 467; 2016, p. 4; Washizaki, 2024, p. 5-6; Hamburg and Mogyorodi, 2024; Sakamoto et al., 2013, p. 343; Peters and Pedrycz, 2000, Tab. 12.3; van Vliet, 2000, p. 438; implied by Barbosa et al., 2006, p. 3)

Continued on next page

Table 4.4: Test approaches with more than one category. (Continued)

Approach	**Category 1**	**Category 2**
Volume Testing	Technique (ISO/IEC and IEEE, 2021, p. 38)	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)
Regression Testing	Technique (implied by ISO/IEC and IEEE, 2022, p. 35)	Level (implied by Barbosa et al., 2006, p. 3)

4.2.2 Synonym Relation Flaws

As mentioned in Section 2.2.2, synonyms do not inherently signify a flaw. Unfortunately, there are many instances of incorrect or ambiguous synonyms, such as the following:

1. 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 [sic] 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.

See #14

2. ISO/IEC and IEEE say that "test level" and "test phase" are synonyms, both meaning a "specific instantiation of [a] test sub-process" (2017, pp. 469, 470; 2013, p. 9), 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 (2017, pp. 420, 509; Perry, 2006, p. 56).

3. "Use case testing" is given as a synonym of "scenario testing" by Hamburg and Mogyorodi (2024) but listed separately by ISO/IEC and IEEE (2022, Fig. 2) and described as a "common form of scenario testing" in (2021, p. 20). This implies that use case testing may instead be a child of user scenario testing (see Table 4.5).

OG Hass, 2008

4. The terms "test level" and "test stage" are given as synonyms (Hamburg and Mogyorodi, 2024; implied by Gerrard, 2000a, p. 9), but Washizaki (2024, 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 2.2.3) to "test level" instead. However, the examples listed—unit testing, integration testing, system testing, and acceptance testing (Washizaki, 2024, pp. 5-6 to 5-7)—are commonly categorized as "test levels" (see Section 2.2.1).

5. The differences between the terms "error", "failure", "fault", "defect" are significant and meaningful (ISO/IEC and IEEE, 2010, pp. 128, 139–140; Washizaki, 2024, p. 12-3; van Vliet, 2000, pp. 399–400), 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”!

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

7. “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.

8. Peters and Pedrycz claim that “structural testing subsumes white box testing” but they seem to describe the same thing: they 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 (2000, p. 447)!

9. The claim that “test cases can be either manual test cases or keyword test cases” (ISO/IEC and IEEE, 2016, p. 6) implies that “keyword-driven testing” could be a synonym of “automated testing” instead of its child, which seems more reasonable (2016, p. 4; 2022, p. 35).

10. 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

OG Hetzel88

11. Sneed and Göschl (2000, p. 18) 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, Sakamoto et al. (2013, pp. 345–346) describe “black-box integration testing”.

find more examples

12. The previous flaw makes the claim that “red-box testing” is a synonym for “acceptance testing” (Sneed and Göschl, 2000, p. 18) lose credibility.

13. 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, 2024, p. 5-15; similar in van Vliet, 2000, pp. 428–429) which is not a requirement of back-to-back testing.

14. “Conformance testing” is implied to be a synonym of “compliance testing” by Kam (2008, p. 43) which only makes sense because of the vague definition of the latter: “testing to determine the compliance of the component or system”.

There are also cases in which a term is given as a synonym to two (or more) terms that are not synonyms themselves. Sometimes, these terms *are* synonyms; for example, Hamburg and Mogyorodi (2024) say "use case testing", "user scenario testing", and "scenario testing" are all synonyms (although there may be a slight distinction; see Table 4.5 and Synonyms Flaw 3). However, this does not always make sense. We identify nine such cases through automatic analysis of the generated graphs, 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. **Condition Testing:**
   - *Branch Condition Combination Testing* (Patton, 2006, p. 120; Sharma et al., 2021, Fig. 1)
   - Branch Condition Testing (Hamburg and Mogyorodi, 2024)
   - *Decision Testing* (Washizaki, 2024, p. 5-13)
2. **Soak Testing:**
   - Endurance Testing (ISO/IEC and IEEE, 2021, p. 39)
   - Reliability Testing[7] (Gerrard, 2000a, Tab. 2; 2000b, Tab. 1, p. 26)
3. **Functional Testing:**
   - Behavioural Testing (van Vliet, 2000, p. 399)
   - Correctness Testing (Washizaki, 2024, p. 5-7)
   - Specification-based Testing (ISO/IEC and IEEE, 2017, p. 196; Kam, 2008, pp. 44-45; van Vliet, 2000, p. 399; implied by ISO/IEC and IEEE, 2021, p. 129; 2017, p. 431)
4. **Link Testing:**
   - Branch Testing (implied by ISO/IEC and IEEE, 2021, p. 24)
   - Component Integration Testing (Kam, 2008, p. 45)
   - Integration Testing (implied by Gerrard, 2000a, p. 13)
5. **Exhaustive Testing:**
   - Branch Condition Combination Testing (Peters and Pedrycz, 2000, p. 464)
   - Path Testing (van Vliet, 2000, p. 421)
6. **State-based Testing:**
   - State Transition Testing (Firesmith, 2015, p. 47; Barbosa et al., 2006, p. 3)

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

- State-based Web Browser Testing (Doğan et al., 2014, p. 193)

7. **Static Verification:**
   - Static Assertion Checking[8] (Chalin et al., 2006, p. 343)
   - Static Testing (implied by Chalin et al., 2006, p. 343)
8. **Testing-to-Fail:**
   - Forcing Exception Testing (Patton, 2006, pp. 66-67, 78)
   - Negative Testing (Patton, 2006, pp. 67, 78, 84-87)
9. **Invalid Testing:**
   - Error Tolerance Testing (implied by Kam, 2008, p. 45)
   - Negative Testing (Hamburg and Mogyorodi, 2024; implied by ISO/IEC and IEEE, 2021, p. 10)

4.2.3 Parent-Child Relation Flaws

Parent-Child Relations are also not immune to difficulties, as shown by the following flaws:

1. Regression testing and retesting are sometimes given as two distinct approaches (ISO/IEC and IEEE, 2022, p. 8; Firesmith, 2015, p. 34), but sometimes regression testing is defined as a form of "selective retesting" (ISO/IEC and IEEE, 2017, p. 372; Washizaki, 2024, pp. 5-8, 6-5, 7-5 to 7-6; 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. Performance testing and security testing are given as subtypes of reliability testing by ISO/IEC (2023a), but these are all listed separately by Firesmith (2015, p. 53).
3. Similarly, random testing is a subtechnique of specification-based testing (ISO/IEC and IEEE, 2022, pp. 7, 22; 2021, pp. 5, 20, Fig. 2; Washizaki, 2024, p. 5-12; Hamburg and Mogyorodi, 2024) but is listed separately by Firesmith (2015, p. 46).
4. 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.

[8] Chalin et al. (2006, p. 343) list Runtime Assertion Checking (RAC) and Software Verification (SV) 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 RAC instead.

5. ISO/IEC and IEEE provide a definition for "inspections and audits" (2017, p. 228), 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.

6. Peters and Pedrycz imply that decision coverage is a child of both c-use coverage *and* p-use coverage (2000, Fig. 12.31); this seems incorrect, since decisions are the result of p-uses (*not* c-uses) and only the p-use relation is implied by (ISO/IEC and IEEE, 2021, Fig. F.1).

OG Reid 1996

testing and security testing are given as subtypes of reliability testing by (ISO/IEC, 2023a), but these are all listed separately by (Firesmith, 2015, p. 53).

Additionally, some self-referential definitions imply that a test approach is a parent of itself. Since these are by nature self-contained within a given source, these are counted *once* as explicit flaws within their sources in Tables 4.1 and 4.2. The following examples were identified through automatic analysis of the generated graphs:

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, performance testing is *not* described as a subapproach of usability testing by (Gerrard, 2000a;b), which would have been more meaningful information to capture.

There are also pairs of synonyms where one is described as a subapproach of the other, abusing the meaning of "synonym" and causing confusion. We identify 15 of these pairs through automatic analysis of the generated graphs, which are given in Table 4.5 (additional pairs where a flaw is inferred are given in Section 4.8.2 for completeness). 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"[9], this overlooks the effects of input data (ISO/IEC and IEEE, 2021, pp. 129–130; Patton, 2006, p. 121; Peters and 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).

[9]The contradictory definitions of path testing given in Definitions Flaw 8 add another layer of complexity to this claim.

Table 4.5: Pairs of test approaches with a parent-child *and* synonym relation.

"Child" → "Parent"	**Parent-Child Source(s)**	**Synonym Source(s)**
All Transitions Testing → State Transition Testing	(ISO/IEC and IEEE, 2021, p. 19)	(Kam, 2008, p. 15)
Co-existence Testing → Compatibility Testing	(ISO/IEC, 2023a; ISO/IEC and IEEE, 2022, p. 3; 2021, Tab. A.1)	(ISO/IEC and IEEE, 2021, p. 37)
Domain Testing → Specification-based Testing	(Peters and Pedrycz, 2000, Tab. 12.1)	(Washizaki, 2024, 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, 2021, p. 38)	(ISO/IEC and IEEE, 2017, p. 196; Kam, 2008, pp. 44-45; van Vliet, 2000, p. 399; implied by ISO/IEC and IEEE, 2021, p. 129; 2017, p. 431)
Orthogonal Array Testing → Pairwise Testing	(Mandl, 1985, p. 1055)	(Washizaki, 2024, p. 5-11; Valcheva, 2013, p. 473)
Path Testing → Exhaustive Testing	(Peters and Pedrycz, 2000, pp. 466-467, 476; implied by Patton, 2006, pp. 120-121)	(van Vliet, 2000, p. 421)
Performance Testing → Performance-related Testing	(ISO/IEC and IEEE, 2022, p. 22; 2021, p. 38)	(Moghadam, 2019, p. 1187)

Continued on next page

Table 4.5: Pairs of test approaches with a parent-child *and* synonym relation. (Continued)

"Child" → "Parent"	**Parent-Child Source(s)**	**Synonym Source(s)**
Static Analysis → Static Testing	(ISO/IEC and IEEE, 2022, pp. 9, 17, 25, 28; Hamburg and Mogyorodi, 2024; Gerrard, 2000a, Fig. 4, p. 12; 2000b, p. 3)	(Peters and Pedrycz, 2000, p. 438)
Structural Testing → Structure-based Testing	(Patton, 2006, pp. 105-121)	(ISO/IEC and IEEE, 2022, p. 9; 2017, pp. 443-444; Hamburg and Mogyorodi, 2024; implied by Barbosa et al., 2006, p. 3)
Use Case Testing → Scenario Testing[c]	(ISO/IEC and IEEE, 2021, p. 20; OG Hass, 2008)	(Hamburg and Mogyorodi, 2024)
Branch Condition Combination Testing → Decision Testing	(implied by the caveat of "atomic conditions" in Hamburg and Mogyorodi, 2024)	(Washizaki, 2024, p. 5-13)
Branch Condition Combination Testing → Exhaustive Testing	(implied by Patton, 2006, p. 121)	(Peters and Pedrycz, 2000, p. 464)
Reviews → Structural Analysis	(Patton, 2006, p. 92)	(implied by Patton, 2006, p. 92)
Beta Testing → User Testing	(implied by Firesmith, 2015, p. 39)	(implied by Firesmith, 2015, p. 39)

[a] Fault tolerance testing may also be a subapproach of reliability testing (ISO/IEC and IEEE, 2017, p. 375; Washizaki, 2024, p. 7-10), which is distinct from robustness testing (Firesmith, 2015, p. 53).

[b] See Section 4.3.1.

[c] See Synonyms Flaw 3.

4.2.4 Definition Flaws

Perhaps the most interesting category for those seeking to understand how to apply a given test approach, there are many flaws with how test approaches, as well as supporting terms, are defined:

1. ISO/IEC and IEEE (2022, p. 34) give the "landmark tour" as an example of "a tour used for exploratory testing", but they also use the analogy of "a tour guide lead[ing] a tourist through the landmarks of a big city" to describe tours in general. Is the distinction between them the fact that landmark tours are pre-planned and follow a decided-upon sequence (p. 34)?

2. Integration testing, system testing, and system integration testing are all listed as "common test levels" (ISO/IEC and IEEE, 2022, p. 12; 2021, p. 6), but no definitions are given for the latter two, making it unclear what "system integration testing" is; it is a combination of the two? somewhere on the spectrum between them? It is listed as a child of integration testing by Hamburg and Mogyorodi (2024) and of system testing by Firesmith (2015, p. 23).

3. Similarly, component testing, integration testing, and component integration testing are all listed in (ISO/IEC and IEEE, 2017), but "component integration testing" is only defined as "testing of groups of related components" (ISO/IEC and IEEE, 2017, p. 82); it is a combination of the two? somewhere on the spectrum between them? As above, it is listed as a child of integration testing by Hamburg and Mogyorodi (2024).

4. "Software testing" is often defined to exclude static testing (Firesmith, 2015, p. 13; Ammann and Offutt, 2017, p. 222; Peters and Pedrycz, 2000, p. 439), restricting "testing" to mean dynamic validation (Washizaki, 2024, 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[10] as well" (Washizaki, 2024, p. 5-2). This is done by ISO/IEC and IEEE (2022, p. 17) and Gerrard (2000a, pp. 8–9); the former even explicitly *exclude* static testing in another document (2017, p. 440)!

5. ISO/IEC and IEEE define "error" as "a human action that produces an incorrect result", but also as "an incorrect result" itself (2010, p. 128). Since faults are inserted when a developer makes an error (2010, pp. 128, 140; Washizaki, 2024, p. 12-3; van Vliet, 2000, pp. 399–400), this means that they are "incorrect results", making "error" and "fault" synonyms and the distinction between them less useful.

6. Additionally, "error" can also be defined as "the difference between a computed, observed, or measured value or condition and the true, specified, or

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

theoretically correct value or condition" (ISO/IEC and IEEE, 2010, p. 128; similar in Washizaki, 2024, 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".

7. The SWEBOK Guide V4 defines "privacy testing" as testing that "assess[es] the security and privacy of users' personal data to prevent local attacks" (Washizaki, 2024, p. 5-10); 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, [sic] are protected so that" only "authorized persons or systems" can use them as intended.

8. Path testing "aims to execute all entry-to-exit control flow paths in a SUT's control flow graph" (Washizaki, 2024, 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."

9. 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).

10. Alpha testing is performed by "users within the organization developing the software" (ISO/IEC and IEEE, 2017, p. 17), "a small, selected group of potential users" (Washizaki, 2024, p. 5-8), or "roles outside the development organization" conducted "in the developer's test environment" (Hamburg and Mogyorodi, 2024).

11. 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.

12. Load testing is performed with loads "between anticipated conditions of low, typical, and peak usage" (ISO/IEC and IEEE, 2022, p. 5) or loads that are as large as possible (Patton, 2006, p. 86).

13. State testing requires that "all states in the state model … [are] 'visited' " in (ISO/IEC and IEEE, 2021, p. 19) which is only one of its possible criteria in (Patton, 2006, pp. 82-83).

14. ISO/IEC and IEEE (2017, p. 456) say system testing is "conducted on a complete, integrated system" (which Peters and Pedrycz (2000, Tab. 12.3)

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

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

OG Beizer

16. Kam (2008, p. 46) says that the goal of negative testing is "showing that a component or system does not work" which is not true; if robustness is an important quality for the system, then testing the system "in a way for which it was not intended to be used" (Hamburg and Mogyorodi, 2024) (i.e., negative testing) is one way to help test this!

17. Kam (2008, p. 42) says "See *boundary value analysis*," for the glossary entry of "boundary value testing" but does not provide this definition.

OG 2015

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

OG ISO1984

19. 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" (2021, p. 18) and one where "the conditions and actions are generally described but are incomplete" (2017, p. 175).

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

OG 2015

21. ISO/IEC and IEEE's 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" (2017, p. 83)—is *much* more vague than the definition given in (2021, p. 27): testing that exercises "control flow sub-paths from each variable definition to each c-use of that definition (with no intervening definitions)" (similar in van Vliet, 2000, p. 425; Peters and Pedrycz, 2000, p. 479).

OG 2015

22. "Data definition" is defined as a "statement where a variable is assigned a value" (ISO/IEC and IEEE, 2021, 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.

OG [14]

23. While "error" is defined as "a human action that produces an incorrect result" (Washizaki, 2024, pp. 12-3; van Vliet, 2000, p. 399), Washizaki does not use this consistently, sometimes implying that errors can be instrinsic to software itself (2024, pp. 4-9, 6-5, 7-3, 12-4, 12-9, 12-13).

Q #10: I ignore "syntax errors, runtime errors, and logical errors" (Washizaki, 2024, p. 16-13, cf. p. 18-13) since they seem to be in different

24. While ergonomics testing is out of scope (as it tests hardware, not software), 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*.

25. Similarly, end-to-end testing is defined as testing "in which business processes are tested from start to finish under production-like circumstances" (Hamburg and Mogyorodi, 2024); it is unclear whether this tests the business processes themselves or the system's role in performing them.

26. 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 testing approach or a tool that supports testing?

27. The definition of "math testing" given by Hamburg and Mogyorodi (2024) is too specific to be useful, likely taken from an example instead of a general definition: "testing to determine the correctness of the pay table implementation, the random number generator results, and the return to player computations".

28. A similar issue exists with multiplayer testing, where its definition specifies "the casino game world" (Hamburg and Mogyorodi, 2024).

29. While correct, ISTQB's definition of "specification-based testing" is not helpful: "testing based on an analysis of the specification of the component or system" (Hamburg and Mogyorodi, 2024).

30. 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).

31. Similarly, it is ambiguous whether "tool/environment testing" refers to 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.

32. The acronym "SoS" is used but not defined by Firesmith (2015, p. 23).

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

34. Van Vliet specifies that every successor of a data definition use needs to be executed as part of all-uses testing (2000, pp. 424-425), but this condition is not included elsewhere (ISO/IEC and IEEE, 2021, 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, 2021, p. 29; similar in 2017, p. 125; Washizaki, 2024, p. 5-13; Peters and Pedrycz, 2000, p. 479); however, paths containing simple cycles may also be required (van Vliet, 2000, p. 425).

36. Van Vliet says that all-p-uses testing is only stronger than all-edges (branch) testing if there are infeasible paths (2000, pp. 432-433), but ISO/IEC and IEEE do not specify this caveat (2021, Fig. F.1).

OG Reid, 1996

37. Similarly, van Vliet says that all-du-paths testing is only stronger than all-uses testing if there are infeasible paths (2000, pp. 432-433), but Washizaki does not specify this caveat (2024, p. 5-13).

38. 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).

Does this merit counting this as an Ambiguity as well as a Contradiction?

39. 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).

40. 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.

41. Availability testing isn't assigned to a test priority (Gerrard, 2000a, Tab. 2), despite the claim that "the test types[11] have been allocated a slot against the four test priorities" (p. 13); I think usability and/or performance would have made sense.

Also of note: (ISO/IEC and IEEE, 2022; 2021), from the ISO/IEC/IEEE 29119 family of standards, mention the following 23 test approaches without defining them. This means that out of the 114 test approaches they mention, about 20% have no associated definition!

However, the previous version of this standard, (2013), generally explained two, provided references for two, and explicitly defined one of these terms, for a total of five definitions that could (should) have been included in (2022)! These terms have been underlined, *italicized*, and **bolded**, respectively. Additionally, entries marked with an asterisk* were defined (at least partially) in (2017), which would have been available when creating this family of standards. These terms bring the

[11] "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.

total count of terms that could (should) have been defined to nine; almost 40% of undefined test approaches could have been defined!

- Acceptance Testing*
- Alpha Testing*
- Beta Testing*
- Capture-Replay Driven Testing
- Data-driven Testing
- Error-based Testing
- Factory Acceptance Testing
- Fault Injection Testing
- Functional Suitability Testing (also mentioned but not defined in (ISO/IEC and IEEE, 2017))
- Integration Testing*
- Model Verification
- Operational Acceptance Testing
- Orthogonal Array Testing
- Production Verification Testing
- Recovery Testing* (Failover/Recovery Testing, Back-up/Recovery Testing, **Backup and Recovery Testing***, Recovery*; see Section 4.5)
- Response-Time Testing
- *Reviews* (ISO/IEC 20246) (Code Reviews*)
- Scalability Testing (defined as a synonym of "capacity testing"; see Section 4.6)
- Statistical Testing
- System Integration Testing (System Integration*)
- System Testing* (also mentioned but not defined in (ISO/IEC and IEEE, 2013))
- *Unit Testing** (IEEE Std 1008-1987, IEEE Standard for Software Unit Testing implicitly listed in the bibliography!)
- User Acceptance Testing

4.2.5 Terminology Flaws

While some flaws exist because the definition of a term is wrong, others exist because term's *name* or *label* is wrong! This could be considered a "sister" category of Definition Flaws, but these seem different enough to merit their own category. This most often manifests as terms that are included in reference material that should not have been, terms that share the same acronym, and terms that have typos or are redundant. The following flaws are presented in that order:

1. Since errors are distinct from defects/faults (ISO/IEC and IEEE, 2010, pp. 128, 140; Washizaki, 2024, p. 12-3; van Vliet, 2000, pp. 399–400), error guessing should instead be called "defect guessing" if it is based on a "checklist of potential defects" (ISO/IEC and IEEE, 2021, p. 29) 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, 2024, p. 5-13). One (or both) of these proposed terms may be useful in tandem with "error guessing", which would focus on errors as traditionally defined; this would be a subapproach of error-based testing (implied by van Vliet, 2000, p. 399).

2. Similarly, "fault seeding" is not a synonym of "error seeding" as claimed by (ISO/IEC and IEEE, 2017, p. 165; van Vliet, 2000, p. 427). The term "error seeding", used by (ISO/IEC and IEEE, 2017, p. 165; Firesmith, 2015, p. 34; van Vliet, 2000, p. 427), should be abandoned in favour of "fault seeding", as 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).

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

Is this a def flaw?

4. The terms "acceleration tolerance testing" and "acoustic tolerance testing" seem to only refer to software testing in (Firesmith, 2015, p. 56); 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 don't exactly seem relevant…

Is this a scope flaw?

5. "Orthogonal array testing" (Washizaki, 2024, 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. "Installability testing" is given as a test type (ISO/IEC and IEEE, 2022, p. 22; 2021, p. 38; 2017, p. 228), while "installation testing" is given as a

test level (van Vliet, 2000, p. 439; implied by Washizaki, 2024, p. 5-8). Since "installation testing" is not given as an example of a test level throughout the sources that describe them (see Section 2.2.1), it is likely that the term "installability testing" with all its related information should be used instead.

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

8. Washizaki (2024, 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 match the definition of "failure": the inability of a system "to perform a required function or … within previously specified limits" (ISO/IEC and IEEE, 2019, p. 7; 2010, p. 139; similar in van Vliet, 2000, p. 400) that is "externally visible" (ISO/IEC and IEEE, 2019, p. 7; similar in van Vliet, 2000, p. 400).

OG ISO/IEC, 2005

9. Similarly, Washizaki (2024, 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." This should either be called "*error* tolerance" or be described as "detecting *faults* and then recovering from them", since "error" and "fault" have distinct meanings (Washizaki, 2024, p. 5-3; van Vliet, 2000, pp. 399–400). The intent of this term-definition pair is unclear, as the strategies given—"backing up and retrying, using auxiliary code and voting algorithms, and replacing an erroneous value with a phony value that will have a benign effect" (Washizaki, 2024, p. 4-11)—could be used for errors or faults.

10. "Ethical hacking testing" is given as a synonym of penetration testing by Washizaki (2024, p. 13-4), which seems redundant; Gerrard (2000b, p. 28) uses the term "ethical hacking" which is clearer.

11. "Par sheet testing" from (Hamburg and Mogyorodi, 2024) seems to refer to the specific example mentioned in Mistakes Flaw 11 and does not seem more widely applicable, since a "PAR sheet" is "a list of all the symbols on each reel of a slot machine" (Bluejay, 2024).

Does this belong here?

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

13. The same is true for "hardware-in-the-loop testing" and "human-in-the-loop testing" ("HIL") (Firesmith, 2015, p. 23), although Preuße et al. (2012, p. 2) use "HiL" for the former.

14. Different capitalizations of the abbreviations of "computation data use" and "predicate data use" are used: the lowercase "c-use" and "p-use" (ISO/IEC and IEEE, 2021, pp. 3, 27-29, 35-36, 114-155, 117-118, 129; 2017, p. 124; Peters and Pedrycz, 2000, p. 477, Tab. 12.6) and the uppercase "C-use" and "P-use" (van Vliet, 2000, pp. 424-425).

15. Similarly for "definition-use" (such as in "definition-use path"), both the lowercase "du" (ISO/IEC and IEEE, 2021, pp. 3, 27, 29, 35, 119-121, 129; Peters and Pedrycz, 2000, pp. 478-479) and the uppercase "DU" (van Vliet, 2000, p. 425) are used.

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

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

4.2.6 Citation Flaws

Sometimes a document cites another for a piece of information that does not appear! The following flaws are examples of this:

1. The source that Hamburg and Mogyorodi (2024) cite for the definition of "test type" does not seem to actually provide a definition.

2. The same is true for "visual testing" (Hamburg and Mogyorodi, 2024).

3. The same is true for "security attack" (Hamburg and Mogyorodi, 2024).

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

4.3 Functional Testing

"Functional testing" is described alongside many other, likely related, terms. This leads to confusion about what distinguishes these terms, as shown by the following five:

4.3.1 Specification-based Testing

This is defined as "testing in which the principal test basis is the external inputs and outputs of the test item" (ISO/IEC and IEEE, 2022, p. 9). This agrees with a definition of "functional testing": "testing that … focuses solely on the outputs generated in response to selected inputs and execution conditions" (ISO/IEC and IEEE, 2017, p. 196). Notably, ISO/IEC and IEEE (2017) lists both as synonyms of "black-box testing" (pp. 431, 196, respectively), despite them sometimes being defined separately. For example, the International Software Testing Qualifications Board (ISTQB) defines "specification-based testing" as "testing based on an analysis of the specification of the component or system" (and gives "black-box testing" as a synonym) and "functional testing" as "testing performed to evaluate if a component or system satisfies functional requirements" (specifying no synonyms) (Hamburg and Mogyorodi, 2024); the latter references ISO/IEC and IEEE (2017,

van Vliet (2000, p. 399) may list these as synonyms; investigate

p. 196) ("testing conducted to evaluate the compliance of a system or component with specified functional requirements") which *has* "black-box testing" as a synonym, and mirrors ISO/IEC and IEEE (2022, p. 21) (testing "used to check the implementation of functional requirements"). Overall, specification-based testing (ISO/IEC and IEEE, 2022, pp. 2-4, 6-9, 22) and black-box testing (Washizaki, 2024, p. 5-10; Souza et al., 2017, p. 3) are test design techniques used to "derive corresponding test cases" (ISO/IEC and IEEE, 2022, p. 11) from "selected inputs and execution conditions" (ISO/IEC and IEEE, 2017, p. 196).

4.3.2 Correctness Testing

Washizaki (2024, p. 5-7) says "test cases can be designed to check that the functional specifications are correctly implemented, which is variously referred to in the literature as conformance testing, correctness testing or functional testing"; this mirrors previous definitions of "functional testing" (ISO/IEC and IEEE, 2022, p. 21; 2017, p. 196) but groups it with "correctness testing". Since "correctness" is a software quality (ISO/IEC and IEEE, 2017, p. 104; Washizaki, 2024, p. 3-13) which is what defines a "test type" (ISO/IEC and IEEE, 2022, p. 15) (see Section 2.3.1), it seems consistent to label "functional testing" as a "test type" (ISO/IEC and IEEE, 2022, pp. 15, 20, 22; 2021, pp. 7, 38, Tab. A.1; 2016, p. 4). However, this conflicts with its categorization as a "technique" if considered a synonym of Specification-based Testing. Additionally, "correctness testing" is listed separately from "functionality testing" by Firesmith (2015, p. 53).

4.3.3 Conformance Testing

Testing that ensures "that the functional specifications are correctly implemented", and can be called "conformance testing" or "functional testing" (Washizaki, 2024, p. 5-7). "Conformance testing" is later defined as testing used "to verify that the SUT conforms to standards, rules, specifications, requirements, design, processes, or practices" (Washizaki, 2024, p. 5-7). This definition seems to be a superset of testing methods mentioned earlier as the latter includes "standards, rules, requirements, design, processes, … [and]" practices in *addition* to specifications!

A complicating factor is that "compliance testing" is also (plausibly) given as a synonym of "conformance testing" (Kam, 2008, p. 43). However, "conformance testing" can also be defined as testing that evaluates the degree to which "results … fall within the limits that define acceptable variation for a quality requirement" (ISO/IEC and IEEE, 2017, p. 93), which seems to describe something different.

OG PMBOK 5th ed.

4.3.4 Functional Suitability Testing

Procedure testing is called a "type of functional suitability testing" (ISO/IEC and IEEE, 2022, p. 7) but no definition of that term is given. "Functional suitability" is the "capability of a product to provide functions that meet stated and implied needs of intended users when it is used under specified conditions", including meeting "the functional specification" (ISO/IEC, 2023a). This seems to align with the

definition of "functional testing" as related to "black-box/specification-based testing". "Functional suitability" has three child terms: "functional completeness" (the "capability of a product to provide a set of functions that covers all the specified tasks and intended users' objectives"), "functional correctness" (the "capability of a product to provide accurate results when used by intended users"), and "functional appropriateness" (the "capability of a product to provide functions that facilitate the accomplishment of specified tasks and objectives") (ISO/IEC, 2023a). Notably, "functional correctness", which includes precision and accuracy (ISO/IEC, 2023a; Hamburg and Mogyorodi, 2024), seems to align with the quality/ies that would be tested by "correctness" testing.

4.3.5 Functionality Testing

"Functionality" is defined as the "capabilities of the various … features provided by a product" (ISO/IEC and IEEE, 2017, p. 196) and is said to be a synonym of "functional suitability" (Hamburg and Mogyorodi, 2024), although it seems like it should really be a synonym of "functional completeness" based on (ISO/IEC, 2023a), which would make "functional suitability" a subapproach. Its associated test type is implied to be a subapproach of build verification testing (Hamburg and Mogyorodi, 2024) and made distinct from "functional testing"; interestingly, security is described as a subapproach of both non-functional and functionality testing (Gerrard, 2000a, Tab. 2). "Functionality testing" is listed separately from "correctness testing" by Firesmith (2015, p. 53).

4.4 Operational (Acceptance) Testing (OAT)

Some sources refer to "operational acceptance testing" (ISO/IEC and IEEE, 2022, p. 22; Hamburg and Mogyorodi, 2024) while some refer to "operational testing" (Washizaki, 2024, p. 6-9, in the context of software engineering operations; ISO/IEC, 2018; ISO/IEC and IEEE, 2017, p. 303; Bourque and Fairley, 2014, pp. 4-6, 4-9). A distinction is sometimes made (Firesmith, 2015, p. 30) but without accompanying definitions, it is hard to evaluate its merit. Since this terminology is not standardized, I propose that the two terms are treated as synonyms (as done by other sources (LambdaTest, 2024; Bocchino and Hamilton, 1996)) as 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).

find more academic sources

A summary of definitions of "operational (acceptance) testing" is that it is "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 "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).

4.5 Recovery Testing

"Recovery testing" is "testing … aimed at verifying software restart capabilities after a system crash or other disaster" (Washizaki, 2024, p. 5-9) including "recover[ing] the data directly affected and re-establish[ing] the desired state of the system" (ISO/IEC, 2023a; similar in Washizaki, 2024, p. 7-10) so that the system "can perform required functions" (ISO/IEC and IEEE, 2017, p. 370). It is also called "recoverability testing" (Kam, 2008, p. 47) and potentially "restart & recovery (testing)" (Gerrard, 2000a, Fig. 5). The following terms, along with "recovery testing" itself (ISO/IEC and IEEE, 2022, p. 22) are all classified as test types, and the relations between them can be found in Figure 5.1a.

- **Recoverability Testing:** Testing "how well a system or software can recover data during an interruption or failure" (Washizaki, 2024, p. 7-10; similar in ISO/IEC, 2023a) and "re-establish the desired state of the system" (ISO/IEC, 2023a). Synonym for "recovery testing" in Kam (2008, p. 47).
- **Disaster/Recovery Testing** serves to evaluate 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" (2021, p. 37). 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.
- **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 and seems to correspond to the definition of "disaster/recovery testing" in (2017, p. 140).
- **Backup/Recovery Testing:** Testing that determines the ability "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, 2021, p. 37). This seems to correspond to the definition of "disaster/recovery testing" in (2021, p. 37). It is also given as a subtype of "disaster/recovery testing", even though that tests if "operation of the test item can be transferred to a different operating site" (p. 37). It also seems to overlap with "backup and recovery testing", which adds confusion.
- **Failover/Recovery Testing:** Testing that 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, 2021, p. 37). This is given as a subtype of "disaster/recovery testing", even though that tests if "operation of the test item can be transferred to a different operating site" (p. 37).

- **Failover Testing:** Testing that "validates the SUT's ability to manage heavy loads or unexpected failure to continue typical operations" (Washizaki, 2024, p. 5-9) by entering a "backup operational mode in which [these responsibilities] … are assumed by a secondary system" (Hamburg and Mogyorodi, 2024). While not *explicitly* related to recovery, "failover/recovery testing" also describes the idea of "failover", and Firesmith (2015, p. 56) uses the term "failover and recovery testing", which could be a synonym of both of these terms.

4.6 Scalability Testing

There were three ambiguities around the term "scalability testing", listed below. The relations between these test approaches (and other relevant ones) are shown in Figure 5.2a.

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

2. ISO/IEC and IEEE give the external modification of the system as part of "scalability" (2021, p. 39), while ISO/IEC (2023a) imply that it is limited to the system itself

3. The SWEBOK Guide V4's definition of "scalability testing" (Washizaki, 2024, p. 5-9) is really a definition of usability testing!

4.7 Compatibility Testing

"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". The term "interoperability testing" is not defined, but is used three times (ISO/IEC and IEEE, 2022, pp. 22, 43) (although the third usage seems like it should be "portability testing"). This implies that "co-existence testing" and "interoperability testing" should be defined as their own terms, which is supported by definitions of "co-existence" and "interoperability" often being separate (Hamburg and Mogyorodi, 2024; ISO/IEC and IEEE, 2017, pp. 73, 237), the definition of "interoperability testing" from ISO/IEC and IEEE (2017, p. 238), and the decomposition of "compatibility" into "co-existence" and "interoperability" by ISO/IEC (2023a)! The "interoperability" element of "compatibility testing" is explicitly excluded by ISO/IEC and IEEE (2021, p. 37), (incorrectly) implying that "compatibility testing" and "co-existence testing" are synonyms. 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.

4.8 Inferred Flaws

Along the course of this analysis, we inferred many potential flaws. Some of these have a conflicting source while others do not. These are excluded from any counts of the numbers of flaws, since they are more subjective, but are given below for completeness.

4.8.1 Inferred Synonym Flaws

See Section 4.2.2.

1. **Production Acceptance Testing:**
    - Operational Testing (Hamburg and Mogyorodi, 2024)[12]
    - Production Verification Testing[13]
2. **Operational Testing:**
    - Field Testing
    - Qualification Testing

Additionally, Kam (2008, p. 46) gives "program testing" as a synonym of "component testing" but it probably should be a synonym of "system testing" instead.

4.8.2 Inferred Parent Flaws

As discussed in Section 4.2.3, some pairs of synonyms also have a parent-child relation, abusing the meaning of "synonym" and causing confusion. While Table 4.5 gives the cases where both relations are supported by the literature, some are less explicit. For example, while "*dynamic testing* is sometimes called … dynamic analysis" (Peters and Pedrycz, 2000, p. 438; implied by ISO/IEC and IEEE, 2017, p. 149), it could be inferred from their static counterparts that dynamic analysis is a *child* of dynamic testing (see ISO/IEC and IEEE, 2022, pp. 9, 17, 25, 28; Hamburg and Mogyorodi, 2024). Additionally, the following automatically generated lists contain examples where at least one of these conflicting relations is *not* 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.

[12] "Operational" and "production acceptance testing" are treated as synonyms by Hamburg and Mogyorodi (2024) but listed separately by Firesmith (2015, p. 30).

[13] "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.

Pairs given a parent-child relation

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

Pairs given a synonym relation

1. Structured Walkthroughs → Walkthroughs[14] (Hamburg and Mogyorodi, 2024)
2. Functionality Testing → Functional Suitability Testing (implied by Hamburg and Mogyorodi, 2024; this seems wrong)

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

1. Computation Flow Testing → Computation Testing
2. Field Testing → Operational Testing
3. Monkey Testing → Fuzz Testing
4. Organization-based Testing → Role-based Testing[15]
5. Scenario-based Evaluations → Scenario-based Testing
6. System Qualification Testing → System Testing

See #59

In addition to this type of flaw, (Gerrard, 2000a, Tab. 2) does *not* give "functionality testing" as a parent of "end-to-end functionality testing". Finally, Patton (2006, p. 119) says that branch testing is "the simplest form of path testing" which is also implied by ISO/IEC and IEEE (2021, 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 Definitions Flaw 8)!

OG Reid, 1996

4.8.3 Inferred Category Flaws

See Section 4.2.1.

Table 4.6: Test approaches inferred to have more than one category.

Approach	Category 1	Category 2
Big-Bang Testing	Level (inferred from integration testing)	Technique (Sharma et al., 2021, pp. 601, 603, 605-606)

Continued on next page

[14]See Synonyms Flaw 7.

[15]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.

Table 4.6: Test approaches inferred to have more than one category. (Continued)

Approach	Category 1	Category 2
Bottom-Up Testing	Level (inferred from integration testing)	Technique (Sharma et al., 2021, pp. 601, 603, 605-606)
Fuzz Testing	Practice (inferred from mathematical-based testing)	Technique (Hamburg and Mogyorodi, 2024; Firesmith, 2015, p. 51; implied by ISO/IEC and IEEE, 2022, p. 36)
Memory Management Testing	Technique (ISO/IEC and IEEE, 2021, p. 38)	Type (inferred from performance-related testing)
Privacy Testing	Technique (ISO/IEC and IEEE, 2021, p. 40)	Type (inferred from security testing)
Sandwich Testing	Level (inferred from integration testing)	Technique (Sharma et al., 2021, pp. 601, 603, 605-606)
Security Audits	Technique (ISO/IEC and IEEE, 2021, p. 40)	Type (inferred from security testing)
Vulnerability Scanning	Technique (ISO/IEC and IEEE, 2021, p. 40)	Type (inferred from security testing)
Closed Beta Testing	Level (inferred from beta testing)	Type (implied by Firesmith, 2015, p. 58)
Open Beta Testing	Level (inferred from beta testing)	Type (implied by Firesmith, 2015, p. 58)
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)

4.8.4 Other Inferred Flaws

The following are flaws that, if were more concrete, would also be included alongside the other flaws:

- "Fuzz testing" is "tagged" (?) as "artificial intelligence" (ISO/IEC and IEEE, 2022, p. 5).
- Gerrard's definition for "security audits" seems too specific, only applying to "the products installed on a site" and "the known vulnerabilities for those products" (2000b, p. 28).

Chapter 5

Recommendations

As we have shown in Chapter 4, "testing is a mess" (Mosser, 2023, priv. comm.)! It will take a lot of time, effort, expertise, and training to organize these terms (and their relations) logically. However, the hardest step is often the first one, so we attempt to give some examples of how this "rationalization" can occur. These changes often 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 terms, we seek to make them:

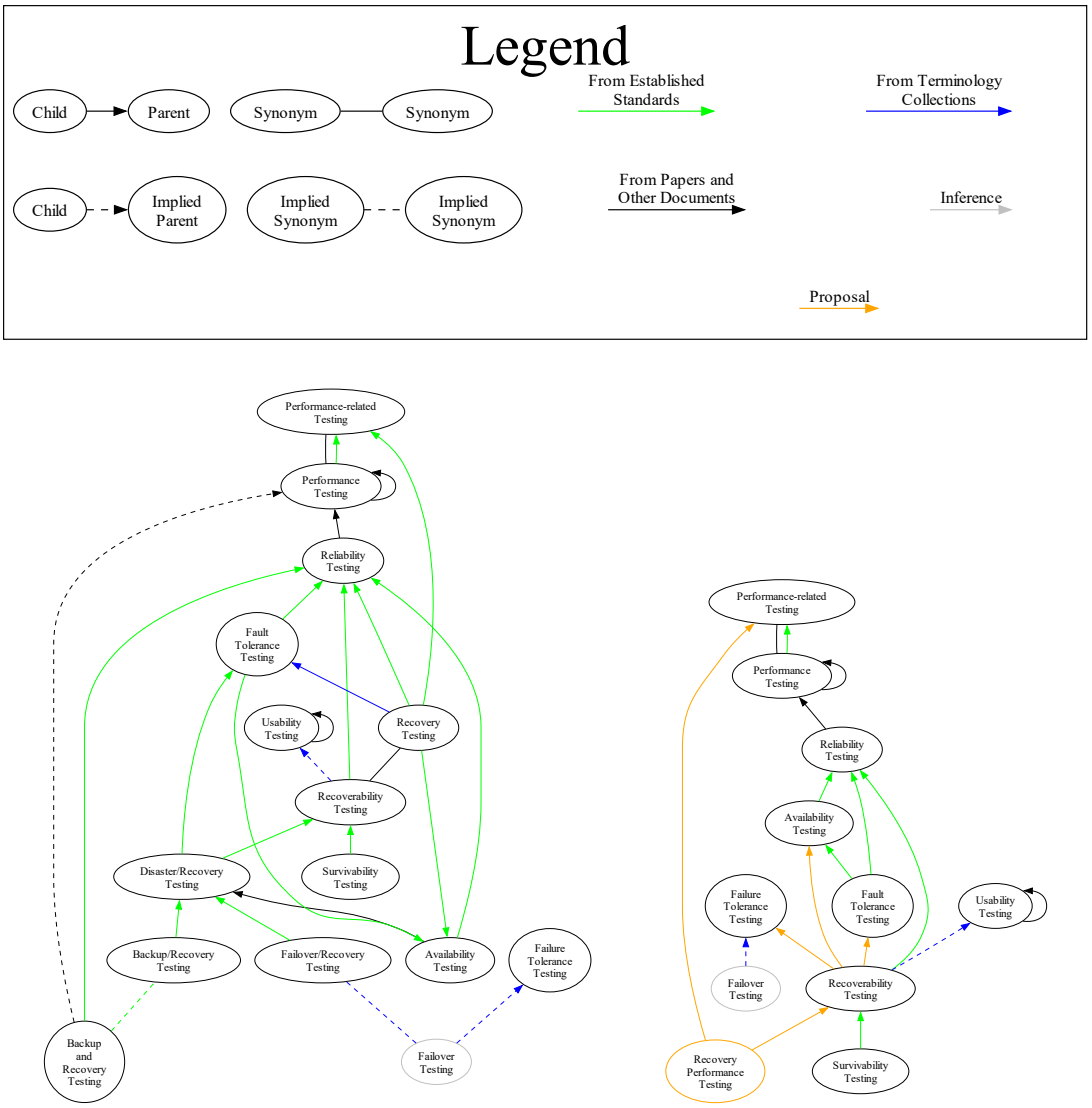
1. Atomic (e.g., disaster/recovery testing seems to have two disjoint definitions)
2. Straightforward (e.g., backup and recovery testing's definition implies the idea of performance, but its name does not ; failover/recovery testing, failover and recovery testing, and failover testing are all given separately)
3. Consistent (e.g., backup/recovery testing and failover/recovery testing explicitly exclude an aspect included in its parent disaster/recovery testing)

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 (Section 4.2.2) or as parents of themselves (Section 4.2.3), or pairs of approaches with both a parent-child *and* synonym relation (Section 4.2.3).

Same label to different phantomsections; is that OK?

We give recommendations for the areas of recovery testing (Section 5.1), scalability testing (Section 5.2), and performance-related testing (Section 5.3). Graphical representations (described in Section 3.1) of these subsets are given in Figures 5.1 to 5.3, in which arrows representing relations between approaches are coloured based on the source tier (see Section 2.1) that defines them. Any added approaches or relations are colored orange. Note that inferred relations (colored grey) are included for completeness, despite not coming from the literature (see Section 2.3).

Should this be the case?



(a) Graph of current relations. (b) Graph of proposed relations.

Figure 5.1: Graphs of relations between terms related to recovery testing.

5.1 Recovery Testing

The following terms should be used in place of the current terminology to more clearly distinguish between different recovery-related test approaches. The result of the proposed terminology, along with their relations, is demonstrated in Figure 5.1b.

- **Recoverability Testing:** "Testing … aimed at verifying software restart capabilities after a system crash or other disaster" (Washizaki, 2024, p. 5-9) including "recover[ing] the data directly affected and re-establish[ing] the desired state of the system" (ISO/IEC, 2023a; similar in Washizaki, 2024, p. 7-10) so that the system "can perform required functions" (ISO/IEC and

IEEE, 2017, p. 370). "Recovery testing" will be a synonym, as in (Kam, 2008, p. 47), since it is the more prevalent term throughout various sources, although "recoverability testing" is preferred to indicate that this explicitly focuses on the *ability* to recover, not the *performance* of recovering.

- **Failover Testing:** Testing that "validates the SUT's ability to manage heavy loads or unexpected failure to continue typical operations" (Washizaki, 2024, p. 5-9) by entering a "backup operational mode in which [these responsibilities] … are assumed by a secondary system" (Hamburg and Mogyorodi, 2024). This will replace "failover/recovery testing", since it is more clear, and since this is one way that a system can recover from failure, it will be a subset of "recovery testing".

- **Transfer Recovery Testing:** Testing to evaluate if, in the case of a failure, "operation of the test item can be transferred to a different operating site and … be transferred back again once the failure has been resolved" (2021, p. 37). This replaces the second definition of "disaster/recovery testing", since the first is just a description of "recovery testing", and could potentially be considered as a kind of failover testing. This may not be intrinsic to the hardware/software (e.g., may be the responsibility of humans/processes).

- **Backup Recovery Testing:** Testing that determines the ability "to restor[e] from back-up memory in the event of failure" (ISO/IEC and IEEE, 2021, p. 37). The qualification that this occurs "without transfer[ing] to a different operating site or back-up system" (p. 37) *could* be made explicit, but this is implied since it is separate from transfer recovery testing and failover testing, respectively.

- **Recovery Performance Testing:** Testing "how well a system or software can recover … [from] an interruption or failure" (Washizaki, 2024, p. 7-10; similar in ISO/IEC, 2023a) "within specified parameters of time, cost, completeness, and accuracy" (ISO/IEC and IEEE, 2013, p. 2). The distinction between the performance-related elements of recovery testing seemed to be meaningful, but was not captured consistently by the literature. This will be a subset of "performance-related testing" (see Section 5.3) as "recovery testing" is in (ISO/IEC and IEEE, 2022, p. 22). This could also be extended into testing the performance of specific elements of recovery (e.g., failover performance testing), but this be too fine-grained and may better be captured as an orthogonally derived test approach.

See #40

5.2 Scalability Testing

The ambiguity around scalability testing found in the literature is resolved and/or explained by other sources! ISO/IEC and IEEE (2021, p. 39) give "scalability testing" as a synonym of "capacity testing", defined as the testing of a system's ability to "perform under conditions that may need to be supported in the future",

which "may include assessing what level of additional resources (e.g. memory, disk capacity, network bandwidth) will be required to support anticipated future loads". 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", such as the number of concurrent users, transaction throughput, or database size (ISO/IEC, 2023a). Because of this nuance, it makes more sense to consider these terms separate and *not* synonyms, as done by Firesmith (2015, p. 53) and Bas (2024, pp. 22-23).

Unfortunately, only focusing on future capacity requirements still leaves room for ambiguity. While the previous definition of "scalability testing" includes the external modification of the system, 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. The potential reason for this is implied by Washizaki (2024, p. 5-9)'s claim that one objective of elasticity testing is "to evaluate scalability": (ISO/IEC, 2023a)'s notion of "scalability" likely refers more accurately to "elasticity"! This also makes sense in the context of other definitions provided by Washizaki (2024):

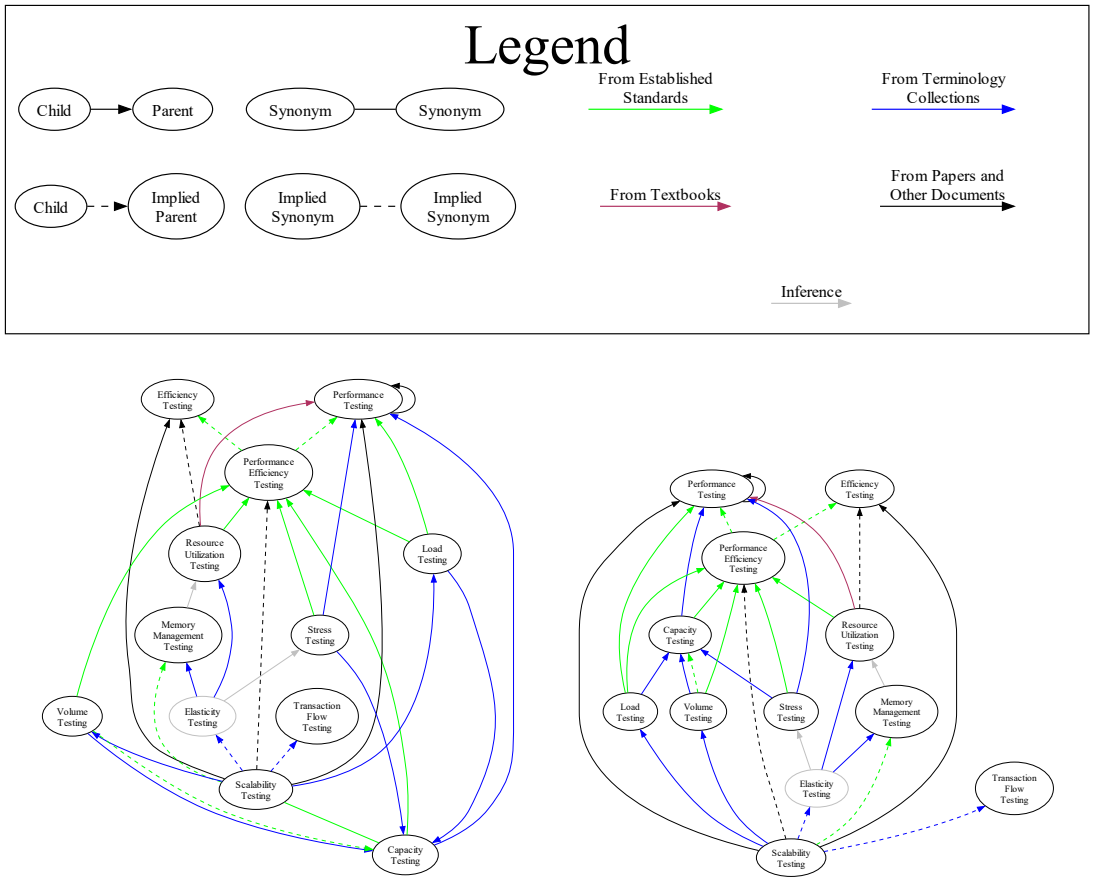
- **Scalability:** "the software's ability to increase and scale up on its nonfunctional requirements, such as load, number of transactions, and volume of data" (p. 5-5). Based on this definition, scalability testing is then a subtype of load testing and volume testing, as well as potentially transaction flow testing.
- **Elasticity Testing[1]:** testing that "assesses the ability of the SUT … to rapidly expand or shrink compute, memory, and storage resources without compromising the capacity to meet peak utilization" (p. 5-9). Based on this definition, elasticity testing is then a subtype of memory management testing (with both being a subtype of resource utilization testing) and stress testing.

See #35

This distinction is also 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".

To make things even more confusing, the SWEBOK Guide V4 says "scalability testing evaluates the capability to use and learn the system and the user documentation" and "focuses on the system's effectiveness in supporting user tasks and the ability to recover from user errors" (Washizaki, 2024, p. 5-9). This seems to define "usability testing" with elements of functional and recovery testing, which is completely separate from the definitions of "scalability", "capacity", and "elasticity

[1]While this definition seems correct, it only cites a single source **that doesn't contain the words "elasticity" or "elastic"**!



(a) Graph of current relations. (b) Graph of proposed relations.

Figure 5.2: Graphs of relations between terms related to scalability testing.

testing"! This definition should simply be disregarded, since it is inconsistent with the rest of the literature. The removal of the previous two synonym relations is demonstrated in Figure 5.2b.

5.3 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; 2017, p. 320; similar in 2021, 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, 2024, p. 5-9). It is performed either …

1. "within given constraints of time and other resources" (ISO/IEC and IEEE,

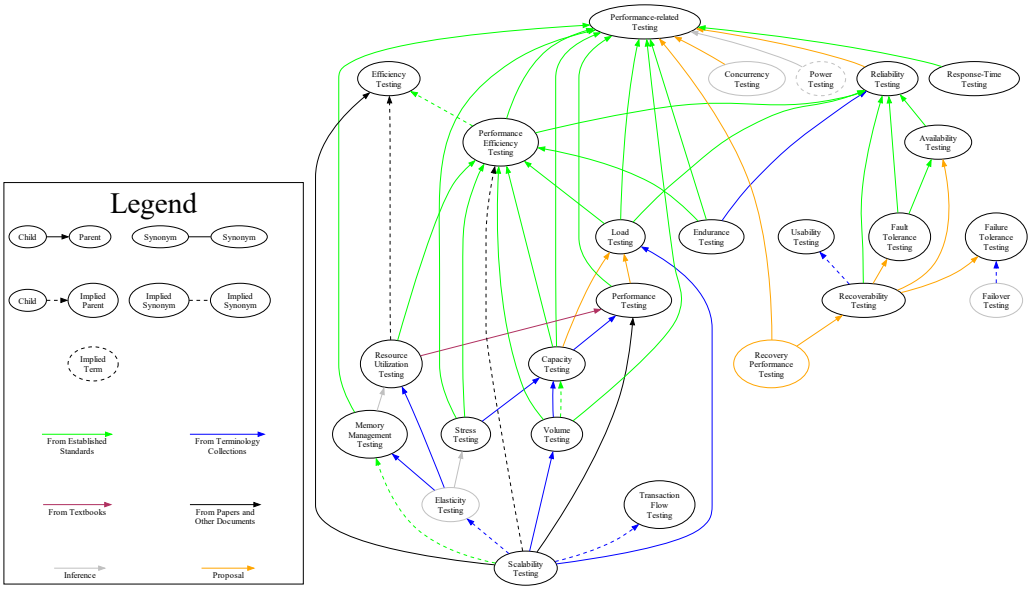


Figure 5.3: Proposed relations between rationalized "performance-related testing" terms.

2022, p. 7; 2017, p. 320; similar in Moghadam, 2019, p. 1187), or

2. "under a 'typical' load" (ISO/IEC and IEEE, 2021, 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' " (2021, 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 (2024, p. 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, 2024, p. 5-9; Moghadam, 2019, p. 1187), as noted in Section 4.2.2. 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; 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, 2021, 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; 2021, p. 39; 2017, p. 253; Hamburg and Mogyorodi, 2024), so it is a child of

OG IEEE 2013

"performance-related testing" and a parent of "performance testing".

After these changes, some finishing touches remain. The "self-loops" mentioned in Section 4.2.3 provide no new information and can be removed. Similarly, the term "soak testing" can be removed. Since it is given as a synonym to both "endurance testing" *and* "reliability testing" (see Section 4.2.2), it makes sense to just use these terms instead of one that is potentially ambiguous. These changes (along with those from Sections 5.1 and 5.2 made implicitly) result in the relations shown in Figure 5.3.

Chapter 6

Development Process

The following is a rough outline of the steps I have gone through this far for this project:

- Start developing system tests (this was pushed for later to focus on unit tests)
- Test inputting default values as `float`s and `int`s
- Check constraints for valid input
- Check constraints for invalid input
- Test the calculations of:
  - `t_flight`
  - `p_land`
  - `d_offset`
  - `s`
- Test the writing of valid output
- Test for projectile going long
- Integrate system tests into existing unit tests
- Test for assumption violation of `g`
  - Code generation could be flawed, so we can't assume assumptions are respected
  - Test cases shouldn't necessarily match what is done by the code; for example, g = 0 shouldn't really give a `ZeroDivisionError`; it should be a `ValueError`
  - This inspired the potential for The Use of Assertions in Code
- Test that calculations stop on a constraint violation; this is a requirement should be met by the software (see Section 6.3)

- Test behaviour with empty input file
- Start creation of test summary (for `InputParameters` module)
  - It was difficult to judge test case coverage/quality from the code itself
  - This is not really a test plan, as it doesn't capture the testing philosophy
  - Rationale for each test explains why it supports coverage and how Drasil derived (would derive) it
- Start researching testing
- Implement generation of `__init__.py` files (#3516)
- Start the Generating Requirements subproject

6.1 Improvements to Manual Test Code

Even though this code will eventually be generated by Drasil, it is important that it is still human-readable, for the benefit of those reading the code later. This is one of the goals of Drasil (see #3417 for an example of a similar issue). As such, the following improvements were discovered and implement in the manually created testing code:

- use pytest's parameterization
- reuse functions/data for consistency
- improve import structure
- use `conftest` for running code before all tests of a module

6.1.1 Testing with Mocks

When testing code, it is common to first test lower-level modules, then assume that these modules work when testing higher-level modules. An example would be using an input module to set up test cases for a calculation module after testing the input module. This makes sense when writing test cases manually since it reduces the amount of code that needs to be written and still provides a reasonably high assurance in the software; if there is an issue with the input module that affects the calculation module tests, the issue would be revealed when testing the input module.

However, since these test cases will be generated by Drasil, they can be consistently generated with no additional effort. This means that the testing of each module can be done completely independently, increasing the confidence in the tests.

6.2 The Use of Assertions in Code

While assertions are often only used when testing, they can also be used in the code itself to enforce constraints or preconditions; they act like documentation that determines behaviour! For example, they could be used to ensure that assumptions about values (like the value for gravitational acceleration) are respected by the code, which gives a higher degree of confidence in the code. This process is known as "assertion checking" (Lahiri et al., 2013) .

investigate OG sources

6.3 Generating Requirements

I structured my manually created test cases around Projectile's functional requirements, as these are the most objective aspects of the generated code to test automatically. One of these requirements was "Verify-Input-Values", which said "Check the entered input values to ensure that they do not exceed the data constraints. If any of the input values are out of bounds, an error message is displayed and the calculations stop." This led me to develop a test case to ensure that if an input constraint was violated, the calculations would stop (Source Code B.1).

However, this test case failed, since the actual implementation of the code did *not* stop upon an input constraint violation. This was because the code choice for what to do on a constraint violation (Source Code B.2) was "disconnected" from the manually written requirement (Source Code B.3), as described in #3523.

Should I include the definition of `Constraints`?

This problem has been encountered before (#3259) and presented a good opportunity for generation to encourage reusability and consistency. However, since it makes sense to first verify outputs before actually outputting them and inserting generated requirements among manually created ones seemed challenging, it made sense to first generate an output requirement.

While working on Drasil in the summer of 2019, I implemented the generation of an input requirement across most examples (#1844). I had also attempted to generate an output requirement, but due to time constraints, this was not feasible. The main issue with this change was the desire to capture the source of each output for traceability; this source was attached to the `InstanceModel` (or rarely, `DataDefinition`) and not the underlying `Quantity` that was used for a program's outputs. The way I had attempted to do this was to add the reference as a `Sentence` in a tuple.

Taking another look at this four years later allowed us to see that we should be storing the outputs of a program as their underlying models, allowing us to keep the source information with it . While there is some discussion about how this might change in the future , for now, all outputs of a program should be `InstanceModel`s. Since this change required adding the `Referable` constraints to the output field of `SystemInformation`, the outputs of all examples needed to be updated to satisfy this constraint; this meant that generating the output requirement of each example was nearly trivial once the outputs were specified correctly. After modifying `DataDefinition`s in GlassBR that were outputs to be `InstanceModel`s (#3569; #3583), reorganizing the requirements of SWHS (#3589; #3607), and clarifying

cite Dr. Smith

add refs to 'underlying Theory' comment and 'not all outputs be IMs' comment

add constraints

the outputs of SWHS (#3589), SglPend (#3533), DblPend (#3533), GamePhysics (#3609), and SSP (#3630), the output requirement was ready to be generated.

Chapter 7

Research

It was realized early on in the process that it would be beneficial to understand the different kinds of testing (including what they test, what artifacts are needed to perform them, etc.). This section provides some results of this research, as well as some information on why and how it was performed.

A justification for why we decided to do this should be added

7.1 Existing Taxonomies, Ontologies, and the State of Practice

One thing we may want to consider when building a taxonomy/ontology is the semantic difference between related terms. For example, one ontology found that the term " 'IntegrationTest' is a kind of Context (with semantic of stage, but not a kind of Activity)" while " 'IntegrationTesting' has semantic of Level-based Testing that is a kind of Testing Activity [or] … of Test strategy" (Tebes et al., 2019, p. 157).

A note on testing artifacts is that they are "produced and used throughout the testing process" and include test plans, test procedures, test cases, and test results (Souza et al., 2017, p. 3). The role of testing artifacts is not specified in (Barbosa et al., 2006); requirements, drivers, and source code are all treated the same with no distinction (Barbosa et al., 2006, p. 3).

add acronym?

In (Souza et al., 2017), the ontology (ROoST) is made to answer a series of questions, including "What is the test level of a testing activity?" and "What are the artifacts used by a testing activity?" (Souza et al., 2017, pp. 8-9).

is this punctuation right?

The question "How do testing artifacts relate to each other?" (Souza et al., 2017, p. 8) is later broken down into multiple questions, such as "What are the test case inputs of a given test case?" and "What are the expected results of a given test case?" (Souza et al., 2017, p. 21). *These questions seem to overlap with the questions we were trying to ask about different testing techniques.* Tebes et al. (2019, pp. 152-153) may provide some sources for software testing terminology and definitions (this seems to include those suggested by Dr. Carette) in addition to a list of ontologies (some of which have been investigated).

One software testing model developed by the Quality Assurance Institute (QAI) includes the test environment ("conditions …that both enable and constrain how

testing is performed", including mission, goals, strategy, "management support, resources, work processes, tools, motivation"), test process (testing "standards and procedures"), and tester competency ("skill sets needed to test software in a test environment") (Perry, 2006, pp. 5-6).

Another source introduced the notion of an "intervention": "an act performed (e.g. use of a technique[1] or a process change) to adapt testing to a specific context, to solve a test issue, to diagnose testing or to improve testing" (Engström and Petersen, 2015, p. 1) and noted that "academia tend[s] to focus on characteristics of the intervention [while] industrial standards categorize the area from a process perspective" (Engström and Petersen, 2015, p. 2). It provides a structure to "capture both a problem perspective and a solution perspective with respect to software testing" (Engström and Petersen, 2015, pp. 3-4), but this seems to focus more on test interventions and challenges rather than approaches (Engström and Petersen, 2015, Fig. 5).

7.2 Definitions

OG Myers 1976

OG ISO/IEC 2014

- Software testing: "the process of executing a program with the intent of finding errors" (Peters and Pedrycz, 2000, p. 438) . "Testing can reveal failures, but the faults causing them are what can and must be removed" (Washizaki, 2024, p. 5-3); it can also include certification, quality assurance, and quality improvement (Washizaki, 2024, p. 5-4). Involves "specific preconditions [and] … stimuli so that its actual behavior can be compared with its expected or required behavior", including control flows, data flows, and postconditions (Firesmith, 2015, p. 11), and "an evaluation … of some aspect of the system or component" based on "results [that] are observed or recorded" (ISO/IEC and IEEE, 2022, p. 10; 2021, p. 6; 2017, p. 465)
- Test case: "the specification of all the entities that are essential for the execution, such as input values, execution and timing conditions, testing procedure, and the expected outcomes" (Washizaki, 2024, pp. 5-1 to 5-2)
- Verification: "the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase" (van Vliet, 2000, p. 400)
- Validation: "the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements" (van Vliet, 2000, p. 400)
- Test Suite Reduction: the process of reducing the size of a test suite while maintaining the same coverage (Barr et al., 2015, p. 519); can be accomplished through mutation testing

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

- Test Case Reduction: the process of "removing side-effect free functions" from an individual test case to "reduc[e] test oracle costs" (Barr et al., 2015, p. 519)

7.2.1 Documentation

- Verification and Validation (V&V) Plan: a document for the "planning of test activities" described by IEEE Standard 1012 (van Vliet, 2000, p. 411)
- Test Plan: "a document describing the scope, approach, resources, and schedule of intended test activities" in more detail that the V&V Plan (van Vliet, 2000, pp. 412-413); should also outline entry and exit conditions for the testing activities as well as any risk sources and levels (Peters and Pedrycz, 2000, p. 445)
- Test Design documentation: "specifies … the details of the test approach and identifies the associated tests" (van Vliet, 2000, p. 413)
- Test Case documentation: "specifies inputs, predicted outputs and execution conditions for each test item" (van Vliet, 2000, p. 413)
- Test Procedure documentation: "specifies the sequence of actions for the execution of each test" (van Vliet, 2000, p. 413)
- Test Report documentation: "provides information on the results of testing tasks", addressing software verification and validation reporting (van Vliet, 2000, p. 413)

7.3 General Testing Notes

See #54

OG ISO 26262

- The scope of testing is very dependent on what type of software is being tested, as this informs what information/artifacts are available, which approaches are relevant, and which tacit knowledge is present. For example, a method table is a tool for tracking the "test approaches, testing techniques and test types that are required depending … on the context of the test object" (Hamburg and Mogyorodi, 2024) , although this is more specific to the automotive domain
- If faults exist in programs, they "must be considered faulty, even if we cannot devise test cases that reveal the faults" (van Vliet, 2000, p. 401)
- "There is no established consensus on which techniques … are the most effective. The only consensus is that the selection will vary as it should be dependent on a number of factors" (ISO/IEC and IEEE, 2021, p. 128; similar in van Vliet, 2000, p. 440), and it is advised to use many techniques when testing (p. 440). This supports the principle of *independence of testing*: the "separation of responsibilities, which encourages the accomplishment of objective testing" (Hamburg and Mogyorodi, 2024)

7.3.1 Steps to Testing (Peters and Pedrycz, 2000, p. 443)

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
6. Compare the expected results to the actual results

7.3.2 Test Oracles

A test oracle is a "source of information for determining whether a test has passed or failed" (ISO/IEC and IEEE, 2022, p. 13) or that "the SUT behaved correctly … and according to the expected outcomes" and can be "human or mechanical" (Washizaki, 2024, p. 5-5). Oracles provide either "a 'pass' or 'fail' verdict"; otherwise, "the test output is classified as inconclusive" (Washizaki, 2024, p. 5-5). This process can be "deterministic" (returning a Boolean value) or "probabilistic" (returning "a real number in the closed interval $[0, 1]$") (Barr et al., 2015, p. 509). Probabilistic test oracles can be used to reduce the computation cost (since test oracles are "typically computationally expensive") (Barr et al., 2015, p. 509) or in "situations where some degree of imprecision can be tolerated" since they "offer a probability that [a given] test case is acceptable" (Barr et al., 2015, p. 510). The SWEBOK Guide V4 lists "unambiguous requirements specifications, behavioral models, and code annotations" as examples (Washizaki, 2024, p. 5-5), and Barr et al. provides four categories (2015, p. 510):

- Specified test oracle: "judge[s] all behavioural aspects of a system with respect to a given formal specification" (Barr et al., 2015, p. 510)
- Derived test oracle: any "artefact[] from which a test oracle may be derived—for instance, a previous version of the system" or "program documentation"; this includes regression testing, metamorphic testing (Barr et al., 2015, p. 510), and invariant detection (either known in advance or "learned from the program") (Barr et al., 2015, p. 516)
  - This seems to prove "relative correctness" as opposed to "absolute correctness" (Lahiri et al., 2013, p. 345) since this derived oracle may be wrong!
  - "Two versions can be checked for semantic equivalence to ensure the correctness of [a] transformation" in a process that can be done "incrementally" (Lahiri et al., 2013, p. 345)
  - Note that the term "invariant" may be used in different ways (see (Chalin et al., 2006, p. 348))

- Pseudo-oracle: a type of derived test oracle that is "an alternative version of the program produced independently" (by a different team, in a different language, etc.) (Barr et al., 2015, p. 515) . *We could potentially use the programs generated in other languages as pseudo-oracles!*

see ISO 29119-11

- Implicit test oracles: detect "'obvious' faults such as a program crash" (potentially due to a null pointer, deadlock, memory leak, etc.) (Barr et al., 2015, p. 510)

- "Lack of an automated test oracle": for example; a human oracle generating sample data that is "realistic" and "valid", (Barr et al., 2015, pp. 510-511), crowdsourcing (Barr et al., 2015, p. 520), or a "Wideband Delphi": "an expert-based test estimation technique that … uses the collective wisdom of the team members" (Hamburg and Mogyorodi, 2024)

7.3.3 Generating Test Cases

- "Impl[ies] a reduction in human effort and cost, with the potential to impact the test coverage positively", and a given "policy could be reused in analogous situations which leads to even more efficiency in terms of required efforts" (Moghadam, 2019, p. 1187)

- "A **test adequacy criterion** … specifies requirements for testing … and can be used … as a test case generator. … [For example, i]f a 100% statement coverage has not been achieved yet, an additional test case is selected that covers one or more statements yet untested" (van Vliet, 2000, p. 402)

- "Test data generators" are mentioned on (van Vliet, 2000, p. 410) but not described

Investigate

- "Dynamic test generation consists of running a program while simultaneously executing the program symbolically in order to gather constrains on inputs from conditional statements encountered along the execution (Godefroid and Luchaup, 2011, p. 23)

OG [11, 6]

- "Generating tests to detect [loop inefficiencies]" is difficult due to "virtual call resolution", reachability conditions, and order-sensitivity (Dhok and Ramanathan, 2016, p. 896)

- Can be facilitated by "testing frameworks such as JUnit [that] automate the testing process by writing test code" (Sakamoto et al., 2013, p. 344)

- Assertion checking requires "auxiliary invariants", and while "many … can be synthesized automatically by invariant generation methods, the undecidable nature (or the high practical complexity) of assertion checking precludes complete automation for a general class of user-supplied assertions" (Lahiri et al., 2013, p. 345)

  - Differential Assertion Checking (DAC) can be supported by "automatic invariant generation" (Lahiri et al., 2013, p. 345)

- *Automated interface discovery* can be used "for test-case generation for web applications" (Doğan et al., 2014, p. 184)

OG Halfond and Orso, 2007

- "Concrete and symbolic execution" can be used in "a dynamic test generation technique … for PHP applications" (Doğan et al., 2014, p. 192)

OG Artzi et al., 2008

- COBRA is a tool that "generates test cases automatically and applies them to the simulated industrial control system in a SiL Test" (Preuße et al., 2012, p. 2)
- Test case generation is useful for instances where one kind of testing is difficult, but can be generated from a different, simpler kind (e.g., asynchronous testing from synchronous testing (Jard et al., 1999))
- Since some values may not always be applicable to a given scenario (e.g., a test case for zero doesn't make sense if there is a constraint that the value in question cannot be zero), the user should likely be able to select categories of tests to generate instead of Drasil just generating all possible test cases based on the inputs (Smith and Carette, 2023)
- "Test suite augmentation techniques specialise in identifying and generating" new tests based on changes "that add new features", but they could be extended to also augment "the expected output" and "the existing *oracles*" (Barr et al., 2015, p. 516)

Investigate!

- The Fault Injection Security Tool (FIST) "automates fault injection analysis of software using program inputs, fault injection functions, and assertions in programs written in C and C++" (Ghosh and Voas, 1999, pp. 40–41). "For example, Booleans are corrupted to their opposite value during execution, integers are corrupted using a random function with a uniform distribution centered around their current value, character strings are corrupted using random values" (p. 41). Unfortunately, "we do not have automatic learning systems for protecting software states, though it is the subject of ongoing research" (p. 44).

7.4 Examples of Metamorphic Relations (MRs)

- The distance between two points should be the same regardless of which one is the "start" point (ISO/IEC and IEEE, 2021, p. 22)
- "If a person smokes more cigarettes, then their expected age of death will probably decrease (and not increase)" (ISO/IEC and IEEE, 2021, p. 22)
- "For a function that translates speech into text[,] … the same speech at different input volume levels … [should result in] the same text" (ISO/IEC and IEEE, 2021, p. 22)

- The average of a list of numbers should be equal (within floating-point errors) regardless of the list's order (Kanewala and Yueh Chen, 2019, p. 67)
- For matrices, if $B = B_1 + B_2$, then $A \times B = A \times B_1 + A \times B_2$ (Kanewala and Yueh Chen, 2019, pp. 68-69)
- Symmetry of trigonometric functions; for example, $\sin(x) = \sin(-x)$ and $\sin(x) = \sin(x + 360°)$ (Kanewala and Yueh Chen, 2019, p. 70)
- Modifying input parameters to observe expected changes to a model's output (e.g., testing epidemiological models calibrated with "data from the 1918 Influenza outbreak"); by "making changes to various model parameters … authors identified an error in the output method of the agent based epidemiological model" (Kanewala and Yueh Chen, 2019, p. 70)
- Using machine learning to predict likely MRs to identify faults in mutated versions of a program (about 90% in this case) (Kanewala and Yueh Chen, 2019, p. 71)

7.5 Roadblocks to Testing

- Intractability: it is generally impossible to test a program exhaustively (ISO/IEC and IEEE, 2022, p. 4; Washizaki, 2024, p. 5-5; Peters and Pedrycz, 2000, pp. 439, 461; van Vliet, 2000, p. 421)
- Adequacy: to counter the issue of intractability, it is desirable "to reduce the cardinality of the test suites while keeping the same effectiveness in terms of coverage or fault detection rate" (Washizaki, 2024, p. 5-4) which is difficult to do objectively; see also "minimization", the process of "removing redundant test cases" (Washizaki, 2024, p. 5-4)
- Undecidability (Peters and Pedrycz, 2000, p. 439): it is impossible to know certain properties about a program, such as if it will halt (i.e., the Halting Problem (Gurfinkel, 2017, p. 4)), so "automatic testing can't be guaranteed to always work" for all properties (Nelson, 1999)

Add paragraph/section number?

7.5.1 Roadblocks to Testing Scientific Software (Kanewala and Yueh Chen, 2019, p. 67)

- "Correct answers are often unknown": if the results were already known, there would be no need to develop software to model them (Kanewala and Yueh Chen, 2019, p. 67); in other words, complete test oracles don't exist "in all but the most trivial cases" (Barr et al., 2015, p. 510), and even if they are, the "automation of mechanized oracles can be difficult and expensive" (Washizaki, 2024, p. 5.5)
- "Practically difficult to validate the computed output": complex calculations and outputs are difficult to verify (Kanewala and Yueh Chen, 2019, p. 67)

- "Inherent uncertainties": since scientific software models scenarios that occur in a chaotic and imperfect world, not every factor can be accounted for (Kanewala and Yueh Chen, 2019, p. 67)
- "Choosing suitable tolerances": difficult to decide what tolerance(s) to use when dealing with floating-point numbers (Kanewala and Yueh Chen, 2019, p. 67)
- "Incompatible testing tools": while scientific software is often written in languages like FORTRAN, testing tools are often written in languages like Java or C++ (Kanewala and Yueh Chen, 2019, p. 67)

Out of this list, only the first two apply. The scenarios modelled by Drasil are idealized and ignore uncertainties like air resistance, wind direction, and gravitational fluctuations. There are not any instances where special consideration for floating-point arithmetic must be taken; the default tolerance used for relevant testing frameworks has been used and is likely sufficient for future testing. On a related note, the scientific software we are trying to test is already generated in languages with widely-used testing frameworks.

Add example

Add source(s)?

Chapter 8

Extras

Writing Directives

- What macros do I want the reader to know about?

8.1 Writing Directives

I enjoy writing directives (mostly questions) to navigate what I should be writing about in each chapter. You can do this using:

Source Code 8.1: Pseudocode: exWD

```
\begin{writingdirectives}
    \item What macros do I want the reader to know about?
\end{writingdirectives}
```

Personally, I put them at the top of chapter files, just after chapter declarations.

8.2 HREFs

For PDFs, we have (at least) 2 ways of viewing them: on our computers, and printed out on paper. If you choose to view through your computer, reading links (as they are linked in this example, inlined everywhere with "clickable" links) is fine. However, if you choose to read it on printed paper, you will find trouble clicking on those same links. To mitigate this issue, I built the "porthref" macro (see `macros.tex` for the definition) to build links that appear as clickable text when "compiling for computer-focused reading," and adds links to footnotes when "compiling for printing-focused reading." There is an option (`compilingforprinting`) in the `manifest.tex` file that controls whether PDF builds should be done for

computers or for printers. For example, by default, McMaster is made with clickable functionaity, but if you change the `manifest.tex` option as mentioned, then you will see the link in a footnote (try it out!).

Source Code 8.2: Pseudocode: exPHref

```
\porthref{McMaster}{https://www.mcmaster.ca/}
```

8.3 Pseudocode Code Snippets

For pseudocode, you can also use the pseudocode environment, such as that used in Source Code B.5.

8.4 TODOs

While writing, I plastered my thesis with notes for future work because, for whatever reason, I just didn't want to, or wasn't able to, do said work at that time. To help me sort out my notes, I used the `todonotes` package with a few extra macros (defined in `macros.tex`). For example,...

Important notes:

Important: "Important" notes.

Generic inlined notes:

Generic inlined notes.

Later: TODO notes for later! For finishing touches, etc.

Notes for later:

Some "easy" notes:

Easy: Easier notes.

Tedious work:

Needs time: Tedious notes.

Q #11: Questions I might have?

Questions:

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.

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.

Jacques Carette, July 2024. URL https://github.com/samm82/TestingTesting/issues/64#issuecomment-2217320875.

Jacques Carette, Spencer Smith, Jason Balaci, Ting-Yu Wu, Samuel Crawford, Dong Chen, Dan Szymczak, Brooks MacLachlan, Dan Scime, and Maryyam Niazi. Drasil, February 2021. URL https://github.com/JacquesCarette/Drasil/tree/v0.1-alpha.

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.

Arie Gurfinkel. Testing: Coverage and Structural Coverage, 2017. URL https://ece.uwaterloo.ca/~agurfink/ece653w17/assets/pdf/W03-Coverage.pdf.

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

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.

Anthony Hunt, Peter Michalski, Dong Chen, Jason Balaci, and Spencer Smith. Drasil - Generate All the Things!, 2021. URL https://jacquescarette.github.io/Drasil/.

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.

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 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 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 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 - 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 2019. doi: 10.1109/IEEESTD.2019.8657410.

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 2021. 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.

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.

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.

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.

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.

Randal C. Nelson. Formal Computational Models and Computability, January 1999. URL https://www.cs.rochester.edu/u/nelson/courses/csc_173/computability/undecidable.html.

Jiantao Pan. Software Testing, 1999. URL http://users.ece.cmu.edu/~koopman/des_s99/sw_testing/.

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.

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.

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.

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.

Michael Shell. How to Use the IEEEtran LaTeX Class. *Journal of LaTeX Class Files*, 14(8), August 2015.

Spencer Smith. Potential Projects, June 2024. URL https://github.com/JacquesCarette/Drasil/wiki/Potential-Projects.

Spencer Smith and Jacques Carette. Private Communication, July 2023.

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(20000009)1:13.3.CO;2-#. URL https://www.researchgate.net/publication/220116945_Testing_software_for_Internet_application.

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.

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, Denis Peppino, Pablo Becker, Gerardo Matturro, Martín Solari, and Luis Olsina. A Systematic Review on Software Testing Ontologies. pages 144–160. August 2019. ISBN 978-3-030-29237-9. doi: 10.1007/978-3-030-29238-6_11.

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. URL https://waseda.app.box.com/v/SWEBOK4-book.

Hironori Washizaki. Software Engineering Body of Knowledge (SWEBOK), February 2025. 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 testing 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" (Zhou et al., 2012, p. 95), is also out of scope.
- 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, 2024, p. 5-13) or through a model, such as a finite-state machine (as a subset of model-based testing) (Doğan et al., 2014, p. 184).

A.2 V&V of Other Artifacts

While many artifacts produced by the software life cycle can be tested, only testing performed on the code *itself* 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).

[1]Humorously, the testing of loops in chemical systems (Dominguez-Pumar et al., 2020) and copper loops (Goralski, 1999) are out of 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 [System Under Test]", can be used to evaluate the effectiveness of a test suite (Washizaki, 2024, 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, 2024, p. 5-15), 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 (Washizaki, 2024, p. 5-15) (in scope).

A.3 Static Testing

Throughout the literature, static testing is more ambiguous than dynamic testing, with more ad hoc processes and inconsistent in/exclusion from the scope of software testing in general (see Definitions Flaw 4). Furthermore, it seems less relevant to our original goal (the automatic generation of tests). 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 testing approaches requires a "complete" picture of how software can be tested and how the various approaches relate to one another. 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 work, even though static testing will likely be removed at a later stage of analysis based on our original motivation.

See #41 and #44

[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 4.2.2 as a synonym flaw.

Appendix B: Code Snippets

Source Code B.1: Tests for main with an invalid input file

```
# from
 ↪  https://stackoverflow.com/questions/54071312/how-to-pass-command-line-argv
## \brief Tests main with invalid input file
# \par Types of Testing:
# Dynamic Black-Box (Behavioural) Testing
# Boundary Conditions
# Default, Empty, Blank, Null, Zero, and None
# Invalid, Wrong, Incorrect, and Garbage Data
# Logic Flow Testing
@mark.parametrize("filename", invalid_value_input_files)
@mark.xfail
def test_main_invalid(monkeypatch, filename):
    # from
     ↪  https://stackoverflow.com/questions/10840533/most-pythonic-way-to-del
    try:
        remove(output_filename)
    except OSError as e: # this would be "except OSError, e:"
     ↪  before Python 2.6
        if e.errno != ENOENT: # no such file or directory
            raise # re-raise exception if a different error
             ↪  occurred


    assert not path.exists(output_filename)


    with monkeypatch.context() as m:
        m.setattr(sys, 'argv', ['Control.py',
         ↪  str(Path("test/test_input") / f"{filename}.txt")])
        Control.main()

    assert not path.exists(output_filename)
```

Source Code B.2: Projectile's choice for constraint violation behaviour in code

```
    srsConstraints = makeConstraints Warning Warning,
```

Source Code B.3: Projectile's manually created input verification requirement

```
verifyParamsDesc = foldlSent [S "Check the entered", plural
↪  inValue,
    S "to ensure that they do not exceed the" +:+. namedRef (datCon
    ↪  [] []) (plural datumConstraint),
    S "If any of the", plural inValue, S "are out of bounds" `sC`
    S "an", phrase errMsg, S "is displayed" `S.andThe` plural
    ↪  calculation, S "stop"]
```

Source Code B.4: "MultiDefinitions" (MultiDefn) Definition

```
-- | 'MultiDefn's are QDefinition factories, used for showing one
↪  or more ways
--   we can define a QDefinition.
data MultiDefn e = MultiDefn{
  -- | UID
  _rUid :: UID,
  -- | Underlying quantity it defines.
  _qd :: QuantityDict,
  -- | Explanation of the different ways we can define a quantity.
  _rDesc :: Sentence,
  -- | All possible ways we can define the related quantity.
  _rvs :: NE.NonEmpty (DefiningExpr e)
}
```

Source Code B.5: Pseudocode: Broken QuantityDict Chunk Retriever

```
retrieveQD :: UID -> ChunkDB -> Maybe QuantityDict
retrieveQD u cdb = do
    (Chunk expectedQd) <- lookup u cdb
    pure expectedQd
```