Todo list

Important: Lay abstract. . . . iii
Important: Replace reading notes. . . . xiii
Important: Declaration of Academic Achievement. . . . xiv
OG IEEE 2013 . . . 6
OG ISO/IEC 2014 . . . 6
Find examples? . . . 6
OG Alalfi et al., 2010 . . . 7
OG Artzi et al., 2008 . . . 7
OG [19] . . . 8
find original source for SouzaEtAl2017 technique examples: Mathur (2012) 14
OG PMBOK . . . 17
Should I add an example? . . . 21
Better place to include this, if at all? . . . 21
Ensure these tweaks are on an up-to-date version! . . . 22
Explain this better! . . . 22
more in Umar2000 . . . 25
OG Hetzel88 . . . 27
find source . . . 27
OG IEEE 2013 . . . 32
OG Fewster and Graham . . . 33
van Vliet (2000, p. 399) may list these as synonyms; investigate . . . 33
OG PMBOK 5th ed. . . . 34
find more academic sources . . . 35
OG ISO1984 . . . 38
OG Beizer . . . 43
OG IEEE 2013 . . . 51
investigate OG sources . . . 54
Should I include the definition of `Constraints`? . . . 54
cite Dr. Smith . . . 54
add refs to 'underlying Theory' comment and 'not all outputs be IMs' comment . . . 54
add constraints . . . 54
A justification for why we decided to do this should be added . . . 56
OG [3, 4, 5, 8] . . . 56
OG Black, 2009 . . . 57
add acronym? . . . 58

is this punctuation right? . . . 58
OG Myers 1976 . . . 59
OG ISO/IEC 2014 . . . 60
OG? . . . 60
OG ISO 26262 . . . 61
This should probably be explained after “test adequacy criterion” is defined 62
Q #1: Bring up! . . . 63
Expand on reliability testing (make own section?) . . . 63
see ISO 29119-11 . . . 64
Investigate . . . 64
OG [11, 6] . . . 65
OG Halfond and Orso, 2007 . . . 65
OG Artzi et al., 2008 . . . 65
OG ISO 25010? . . . 66
Originally used a *very* vague definition from (Peters and Pedrycz, 2000, p. 447); re-investigate! . . . 66
Does symbolic execution belong here? Investigate from textbooks . . . 66
OG Miller et al., 1994 . . . 67
OG Miller et al., 1994 . . . 67
OG Miller et al., 1994 . . . 67
OG Miller et al., 1994 . . . 67
Q #2: How do we decide on our definition? . . . 67
OG Miller et al., 1994 . . . 68
OG Beizer, 1990 . . . 69
Is this sufficient? . . . 69
Q #3: How is All-DU-Paths coverage stronger than All-Uses coverage according to (van Vliet, 2000, p. 433)? . . . 69
OG KA85 . . . 70
Investigate! . . . 71
Investigate these . . . 71
Add paragraph/section number? . . . 73
Add example . . . 73
Add source(s)? . . . 73
Important: “Important” notes. . . . 75
Generic inlined notes. . . . 75
Later: TODO notes for later! For finishing touches, etc. . . . 75
Easy: Easier notes. . . . 75
Needs time: Tedious notes. . . . 75
Q #4: Questions I might have? . . . 75

THE GENERATION OF TEST CASES IN DRASIL

THE GENERATION OF TEST CASES IN DRASIL

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 (2024) McMaster University
(Department of Computing and Software) Hamilton, Ontario

TITLE:	The Generation of Test Cases in Drasil
AUTHOR:	Samuel Crawford, B.Eng.
SUPERVISOR:	Dr. Carette and Dr. Smith
PAGES:	xiv, 88

Lay Abstract

Important: Lay abstract.

Abstract

Testing is a pervasive software development activity that is often complicated and expensive (if not simply overlooked), partly due to the lack of a standardized and consistent taxonomy for software testing. This hinders precise communication, leading to discrepancies and ambiguities across the literature and even within individual documents! In this paper, we systematically examine the current state of software testing terminology. We 1) identify established standards and prominent testing resources, 2) capture relevant testing terms from these sources, along with their definitions and relationships—both explicit and implicit—and 3) construct graphs to visualize and analyze this data. Our research uncovered 526 test approaches and four in-scope methods for describing "implied" test approaches. We also build a tool for generating graphs that illustrate relations between test approaches and track ambiguities captured by this tool and manually through the research process. Our results reveal 146 discrepancies or ambiguities, including ten terms used as synonyms to two (or more) disjoint test approaches and 11 pairs of test approaches may either be synonyms or have a parent-child relationship. They also reveal notable confusion surrounding functional, operational acceptance, recovery, and scalability testing. These 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 for proofreading and assistance with LaTeX formatting and supplementary Python code for constructing graphs and generating LaTeX code, including regex. Jason Balaci's McMaster thesis template provided many helper LaTeX functions.

Contents

Todo list i

Lay Abstract iii

Abstract iv

Acknowledgements v

Contents vi

List of Figures ix

List of Tables x

List of Source Codes xi

List of Abbreviations and Symbols xii

Reading Notes xiii

Declaration of Academic Achievement xiv

1 Introduction 1

2 Scope 3
- 2.1 Static Testing . . . 5
- 2.2 Derived Test Approaches . . . 5
  - 2.2.1 Coverage-driven Techniques . . . 5
  - 2.2.2 Quality-driven Types . . . 6
  - 2.2.3 Requirements-driven Approaches . . . 6
  - 2.2.4 Attacks . . . 6
  - 2.2.5 Language-specific Approaches . . . 7
  - 2.2.6 Orthogonally Derived Approaches . . . 7

3 Methodology 9
- 3.1 Sources . . . 9
  - 3.1.1 Established Standards . . . 9
  - 3.1.2 “Meta-level” Collections . . . 10

3.1.3 Textbooks . . . 10
3.1.4 Papers and Other Documents . . . 10
3.1.5 Inferences . . . 11
3.2 Terminology . . . 11
3.2.1 Categories of Testing Approaches . . . 11
3.2.2 Parent-Child Relations . . . 15
3.2.3 Categories of Discrepancies . . . 15
3.2.4 Rigidity . . . 15
3.3 Procedure . . . 16
3.4 Undefined Terms . . . 17

4 Tools **19**
4.1 Approach Graph Generation . . . 19
4.2 Discrepancy Analysis . . . 22
4.2.1 Automated Discrepancy Analysis . . . 23
4.2.2 Augmented Discrepancy Analysis . . . 23

5 Discrepancies and Ambiguities **25**
5.1 Synonyms . . . 25
5.2 Parent Relations . . . 29
5.3 Categories of Testing Approaches . . . 32
5.4 Functional Testing . . . 33
5.4.1 Specification-based Testing . . . 33
5.4.2 Correctness Testing . . . 34
5.4.3 Conformance Testing . . . 34
5.4.4 Functional Suitability Testing . . . 34
5.4.5 Functionality Testing . . . 35
5.5 Operational (Acceptance) Testing (OAT) . . . 35
5.6 Recovery Testing . . . 35
5.7 Scalability Testing . . . 37
5.8 Other Discrepancies . . . 37
5.8.1 Other Discrepancies from Established Standards . . . 37
5.8.2 Other Discrepancies from "Meta-level" Collections . . . 40
5.8.3 Other Discrepancies from Textbooks . . . 42
5.8.4 Other Discrepancies from Papers and Other Documents . . . 43
5.9 Inferred Discrepancies . . . 44
5.9.1 Inferred Synonym Discrepancies . . . 44
5.9.2 Inferred Parent Discrepancies . . . 45
5.9.3 Other Inferred Discrepancies . . . 45

6 Recommendations **46**
6.1 Recovery Testing . . . 46
6.2 Scalability Testing . . . 48
6.3 Performance(-related) Testing . . . 49

7 Development Process **52**
7.1 Improvements to Manual Test Code . . . 53
7.1.1 Testing with Mocks . . . 53
7.2 The Use of Assertions in Code . . . 54
7.3 Generating Requirements . . . 54

8 Research **56**
8.1 Categorizations . . . 56
8.2 Existing Taxonomies, Ontologies, and the State of Practice . . . 58
8.3 Definitions . . . 59
8.3.1 Documentation . . . 60
8.4 General Testing Notes . . . 61
8.4.1 Steps to Testing . . . 62
8.4.2 Testing Stages . . . 62
8.4.3 Test Oracles . . . 63
8.4.4 Generating Test Cases . . . 64
8.5 Dynamic Black-Box (Behavioural) Testing . . . 65
8.5.1 Other Black-Box Testing . . . 66
8.6 Static White-Box Testing (Structural Analysis) . . . 66
8.6.1 Correctness Proofs . . . 66
8.7 Dynamic White-Box (Structural) Testing . . . 66
8.7.1 Code Coverage or Control-Flow Coverage . . . 66
8.7.2 Data Coverage . . . 68
8.7.3 Fault Seeding . . . 69
8.7.4 Mutation Testing . . . 69
8.8 Gray-Box Testing . . . 70
8.9 Regression Testing . . . 70
8.10 Metamorphic Testing (MT) . . . 71
8.10.1 Benefits of MT . . . 71
8.10.2 Examples of MT . . . 72
8.11 Roadblocks to Testing . . . 72
8.11.1 Roadblocks to Testing Scientific Software . . . 73

9 Extras **74**
9.1 Writing Directives . . . 74
9.2 HREFs . . . 74
9.3 Pseudocode Code Snippets . . . 75
9.4 TODOs . . . 75

Bibliography **76**

Appendix **86**

List of Figures

3.1 Summary of how many sources comprise each source category. . . . 11

4.1 Example generated graphs. 20

5.1 Sources of discrepancies based on source category. 26

6.1 Current relations between “recovery testing” terms. 47
6.2 Proposed relations between rationalized “recovery testing” terms. . 47
6.3 Current relations between “scalability testing” terms. 50
6.4 Proposed relations between rationalized “scalability testing” terms. 50
6.5 Proposed relations between rationalized “performance-related testing” terms. 51

List of Tables

3.1 IEEE Testing Terminology 13
3.2 Other Testing Terminology 14

4.1 Example glossary entries demonstrating how parent relations are tracked. 19
4.2 Example glossary entries demonstrating how synonym relations are tracked. 21

5.1 Breakdown of identified discrepancies by source and type. 30
5.2 Pairs of test approaches with both parent-child and synonym relations. 30

8.1 Types of Data Flow Coverage 70

List of Source Codes

9.1 Pseudocode: exWD 74
9.2 Pseudocode: exPHref 75

A.3 Code for determining a source's category based on its citation. . . . 86
A.4 Tests for main with an invalid input file 86
A.5 Projectile's choice for constraint violation behaviour in code 87
A.6 Projectile's manually created input verification requirement 87
A.7 "MultiDefinitions" (MultiDefn) Definition 87
A.8 Pseudocode: Broken QuantityDict Chunk Retriever 88

List of Abbreviations and Symbols

DAC	Differential Assertion Checking
DOM	Document Object Model
EMSEC	EManations SECurity
HREF	Hypertext REFerence
ISTQB	International Software Testing Qualifications Board
ML	Machine Learning
MR	Metamorphic Relation
MT	Metamorphic Testing
PDF	Portable Document Format
QAI	Quality Assurance Institute
RAC	Runtime Assertion Checking
SUT	System Under Test
SV	Software Verification
TOAT	Taguchi's Orthogonal Array Testing
C-use	Computational Use
DblPend	Double Pendulum
GamePhysics	Game Physics
OAT	Operational (Acceptance)/Orthogonal Array Testing
P-use	Predicate Use
Projectile	Projectile
SglPend	Single Pendulum
SSP	Slope Stability analysis Program
SWEBOK Guide	Guide to the SoftWare Engineering Body Of Knowledge
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

Important: Declaration of Academic Achievement.

Chapter 1

Introduction

Testing software is complicated, expensive, and often overlooked. Improving the productivity of testing and testing research requires a standard language for communication. For example, "complete testing" could mean that the tester has "completed the discovery of every bug in the product…[,] the agreed-upon tests…[, or] the time period assigned to testing", which can lead to miscommunication and, ultimately, the tester getting "blamed for not doing … [their] job" (Kaner et al., 2011, p. 7). Unfortunately, a search for a systematic, rigorous, and "complete" taxonomy for software testing revealed that the existing ones are inadequate:

- Tebes et al. (2020a) focus on *parts* of the testing process (e.g., test goal, testable entity),
- Souza et al. (2017) prioritize organizing testing approaches over defining them, and
- Unterkalmsteiner et al. (2014) provide a foundation for classification but not how it applies to software testing terminology.

Some existing collections of software testing terminology were found, but in addition to being incomplete, they also contained many oversights. For example, (ISO/IEC and IEEE, 2017) provides the following incomplete/nonsensical definitions for the following terms:

1. **Event Sequence Analysis:** "per" (p. 170)
2. **Operable:** "state of" (p. 301)

Additionally, it also defines "device" as a "mechanism or piece of equipment designed to serve a purpose or perform a function" (p. 136), but does not define "equipment" and only defines "mechanism" in the software sense as how "a function … transform[s] input into output" (p. 270).

Thus we set about closing this gap in the literature. We first define the scope of what kinds of "software testing" are of interest (Chapter 2) and examine the existing literature (Chapter 3), partially through the use of tools created for analysis (Chapter 4). Despite the amount of well understood and organized knowledge,

there are still many discrepancies and ambiguities in the literature, either within the same source or between various sources (Chapter 5). This reinforces the need for a proper taxonomy! We also provide some potential solutions covering some of these discrepancies (Chapter 6). The following three research questions guide this process:

1. What testing approaches are described in the literature?
2. What discrepancies exist between descriptions of these testing approaches?
3. Can any of these discrepancies be resolved/reduced systematically?

This research and analysis (including the development of tools for analysis outlined in Chapter 4) was performed by Samuel Crawford under the guidance, supervision, and recommendations of Drs. Spencer Smith and Jacques Carette. The resulting contributions are three glossaries of test approaches, software qualities (which may imply Quality-driven Types), and supplementary terms, as well as tools for automated analysis and visualization of these data. These are all available online at https://github.com/samm82/TestGen-Thesis for independent analysis and, ideally, extension as more test approaches are discovered and documented.

In our own project Drasil (Carette et al., 2021), which is aimed at "generating all of the software artifacts for (well understood) research software", we wanted to automate the generation of tests. We did not want to do this in an ad hoc manner, so we sought to fully understand the target domain: testing. The goal was to uncover the various approaches towards testing, as well as which prerequisites (e.g., input files, oracles) are needed for each. Then we could evaluate which prerequisites are already "known" by Drasil, as well as which were possible to "teach", and generate test cases from them. The lack of a consistent, comprehensive source of this information caused the focus of this project to shift drastically.

Chapter 2

Scope

Since our motivation is restricted to testing of code, only the "testing" component of Verification and Validation (V&V) is considered (see #22). For example, design reviews and documentation reviews (see ISO/IEC and IEEE, 2017, pp. 132, 144, respectively) are out of scope, as they focus on the V&V of design and documentation, respectively. Likewise, ergonomics testing and proximity-based testing (see Hamburg and Mogyorodi, 2024) are out of scope as they fundamentally involve hardware. Similarly removed is 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). Sometimes, wider decisions must be made on whether a whole category of testing is in scope or not. For example, while all the examples of domain-specific testing given by Firesmith (2015, p. 26) are focused on hardware, this might not be representative of all types (e.g., Machine Learning (ML) model testing seems domain-specific). Conversely, the examples of environmental tolerance testing (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 seem to focus on environment-specific factors that would not impact the code, this category of testing is also out of scope.

It is also interesting to note that different test approaches seem to be more specific to certain domains. For example, the terms "software qualification testing" and "system qualification testing" show up throughout (Knüvener Mackert GmbH, 2022), which was written for the automotive industry, and 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).

Furthermore, only some aspects of some testing approaches are relevant. This mainly manifests as a testing approach that applies to both the V&V itself and to the code. For example:

1. *Error seeding* is the "process of intentionally adding known faults 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).

2. *Fault injection testing*, where "faults are artificially introduced 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).

3. "*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).

4. 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).

5. Orthogonal Array Testing (OAT) can be used when testing software (Mandl, 1985) (in scope) but can also be used for 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.

6. Even though *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), only reliability and maintainability testing done *on* code is in scope.

7. 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)

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

(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).

2.1 Static Testing

Sometimes, the term "testing" excludes static testing (Ammann and Offutt, 2017, p. 222; Firesmith, 2015, p. 13), restricting it to "dynamic validation" (Washizaki, 2024, p. 5-1) or "dynamic verification" "in which a system or component is executed" (ISO/IEC and IEEE, 2017, p. 427). Since "terminology is not uniform among different communities, and some use the term *testing* to refer to static techniques[2] as well" (Washizaki, 2024, p. 5-2), the scope of "testing" for the purpose of this project will include both "static testing" and "dynamic testing", as done by ISO/IEC and IEEE (2022, p. 17), Gerrard (2000a, pp. 8-9), and even a source that explicitly excluded static testing (ISO/IEC and IEEE, 2017, p. 440)!

Static testing generally seems more ad hoc and less relevant for our original goal (the automatic generation of tests). In particular, some techniques may generate false positives which require human intervention, 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 "out-of-scope" approaches may even be generated in the future! Therefore, we keep static testing in-scope at this stage of the analysis.

2.2 Derived Test Approaches

Since the field of software is ever-evolving, being able to adapt to new developments, as well as being able to talk about and understand them, is crucial. In addition to methods of categorizing test approaches, the literature also provides the following methods of deriving new ones. For completeness, these are also considered in scope, except for Language-specific Approaches and Orthogonally Derived Approaches.

2.2.1 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 "the number of test coverage items covered by executed test cases" divided by "the total number of test coverage items identified" (2021, p. 30). "Coverage levels can range from 0% to 100%" and may or may not include "infeasible" test coverage items, which are "not … executable or [are] impossible to be covered by a test case" (p. 30). Perhaps more interestingly, the

[2]Not formally defined, but distinct from the notion of "test technique" described in IEEE Testing Terminology.

further implication is that a given coverage metric implies a test approach aimed to maximize it; for example, "path testing" is testing that "aims to execute all entry-to-exit control flow paths in a SUT's control flow graph" (Washizaki, 2024, p. 5013), thus maximizing the path coverage (see also #63, (Sharma et al., 2021, Fig. 1)).

2.2.2 Quality-driven Types

OG IEEE 2013

OG ISO/IEC 2014

Since test types are "focused on specific quality characteristics" (ISO/IEC and IEEE, 2022, p. 15; 2021, p. 7; 2017, p. 473), they can 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). For quantifying quality-driven testing, measurements should include an entity to be measured, a specific attribute to measure, and the actual measure (i.e., units, starting state, ending state, what to include) (1997, p. 36) where attributes must be defined before they can be measured (p. 38).

Given the importance of software qualities to defining test types, the definitions of 75 software qualities are also tracked in this current work (see #21, #23, and #27). This was done by capturing their definitions, any precedent for the existence of an associated test type, and any additional notes in a glossary. Over time, software qualities were "upgraded" to test types when mentioned (or implied) by a source.

Find examples?

2.2.3 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 (see #43), such as:

- **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)

2.2.4 Attacks

Since attacks are given as a test practice (ISO/IEC and IEEE, 2022, p. 34), different kinds of software attacks, such as code injection and password cracking, can also be used as test approaches.

2.2.5 Language-specific Approaches

Specific programming languages are sometimes used to define test approaches. If the reliance on a specific programming language is intentional, then this really implies an underlying test approach that may be generalized to other languages. These are therefore considered out-of-scope (see #63), including the following examples:

- "They implemented an approach … for JavaScript testing (referred to as Randomized)" (Doğan et al., 2014, p. 192); this really refers to random testing used within JavaScript
- "SQL statement coverage" is really just statement coverage used specifically for SQL statements (Doğan et al., 2014, Tab. 13)
- "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)
- 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, and content checking (Gerrard, 2000b, p. 3)

OG Alalfi et al., 2010

OG Artzi et al., 2008

2.2.6 Orthogonally Derived Approaches

Some test approaches appear to be combinations of other (seemingly orthogonal) approaches. These are considered out of scope, since they are just trivial specializations of documented test approaches that are in scope. For the following examples; we indicate the combination for the first item, but omit the rest for brevity as the name makes it clear what the combination is:

- Black box conformance testing (Jard et al., 1999, p. 25) (combining black box and conformance testing)
- Black-box integration testing (Sakamoto et al., 2013, p. 345-346)
- Checklist-based reviews (Hamburg and Mogyorodi, 2024)
- Closed-loop HiL verification (Preuße et al., 2012, p. 6)
- Closed-loop protection system testing (Forsyth et al., 2004, p. 331)
- Endurance stability testing (Firesmith, 2015, p. 55)
- End-to-end functionality testing (ISO/IEC and IEEE, 2021, p. 20; Gerrard, 2000a, Tab. 2)

- Formal reviews (Hamburg and Mogyorodi, 2024)
- Gray-box integration testing (Sakamoto et al., 2013, p. 344)
- Incremental integration testing (Sharma et al., 2021, p. 601)
- Informal reviews (Hamburg and Mogyorodi, 2024)
- Infrastructure compatibility testing (Firesmith, 2015, p. 53)
- Invariant-based automatic testing (Doğan et al., 2014, pp. 184-185, Tab. 21), including for "AJAX user interfaces" (p. 191)
- Legacy system integration (testing) (Gerrard, 2000a, Tab. 2)
- Machine learning-assisted performance testing? (Moghadam, 2019)
- Manual procedure testing (Firesmith, 2015, p. 47)
- Manual security audits (Gerrard, 2000b, p. 28)
- Model-based GUI testing (Doğan et al., 2014, Tab. 1; implied by Sakamoto et al., 2013, p. 356)
- Model-based web application testing (implied by Sakamoto et al., 2013, p. 356)
- Non-functional search-based testing (Doğan et al., 2014, Tab. 1)
- Offline MBT (Hamburg and Mogyorodi, 2024)
- Online MBT (Hamburg and Mogyorodi, 2024)
- Role-based reviews (Hamburg and Mogyorodi, 2024)
- Scenario walkthroughs (Gerrard, 2000a, Fig. 4)
- Scenario-based reviews (Hamburg and Mogyorodi, 2024)
- Security attacks (Hamburg and Mogyorodi, 2024)
- Statistical web testing (Doğan et al., 2014, p. 185)
- Usability test script(ing) (Hamburg and Mogyorodi, 2024)
- Web application regression testing (Doğan et al., 2014, Tab. 21)
- White-box unit testing (Sakamoto et al., 2013, pp. 345-346)

OG [19]

While some approaches are closely associated, such as remote testing with asynchronous testing, and local with synchronous (Jard et al., 1999), these seem to imply a parent-child relation rather than a combination.

Chapter 3

Methodology

3.1 Sources

As there is no single authoritative source on software testing terminology, we need to look at many to see how various terms are used in practice. Starting from some set of sources, we then use "snowball sampling" (a "method of … sample selection … used to locate hidden populations" (Johnson, 2014)) to gather further sources (see Section 3.4). Sources with a similar degree of "trustworthiness" are grouped into categories; sources that are more "trustworthy":

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

We therefore create the following categories, given in order of descending trustworthiness: Established Standards, "Meta-level" Collections, Textbooks, and Papers and Other Documents. Additionally, some information comes from Inferences. Each category is given a unique colour to better track how their information appears in relevant graphs (see Figures 6.1 to 6.5). A summary of how many sources comprise each category is given in Figure 3.1.

3.1.1 Established Standards

(ISO/IEC and IEEE, 2022; 2021; 2019; 2017; 2013; IEEE, 2012; ISO, 2022; 2015; ISO/IEC, 2023a;b; 2018; 2015; 2011)

- Colored green
- Information on software development and testing from standards bodies
- Written by reputable organizations for use in software engineering

3.1.2 "Meta-level" Collections

(Hamburg and Mogyorodi, 2024; Washizaki, 2024; Firesmith, 2015; Doğan et al., 2014; Bourque and Fairley, 2014)

- Colored blue
- Collections of relevant terminology (such as ISTQB's glossary, the SWEBOK Guide, and Doğan et al.'s literature review (2014))
- Built up from various sources, including Established Standards, and often written by a large organization (such as ISTQB); 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)

3.1.3 Textbooks

(Dennis et al., 2012; Kaner et al., 2011; Patton, 2006; Perry, 2006; Gerrard and Thompson, 2002; Peters and Pedrycz, 2000; van Vliet, 2000)

- Colored maroon
- Textbooks trusted at McMaster (Patton, 2006; Peters and Pedrycz, 2000; van Vliet, 2000) were the original (albeit ad hoc and arbitrary) starting point
- Written by smaller sets of authors, but with a formal review process before publication
- Used as resources for teaching software engineering and may be used as guides in industry

3.1.4 Papers and Other Documents

(Bas, 2024; LambdaTest, 2024; Pandey, 2023; Knüvener Mackert GmbH, 2022; Sharma et al., 2021; Moghadam, 2019; Bajammal and Mesbah, 2018; Souza et al., 2017; Dhok and Ramanathan, 2016; Barr et al., 2015; Lahiri et al., 2013; Sakamoto et al., 2013; Valcheva, 2013; Preuße et al., 2012; Godefroid and Luchaup, 2011; Yu et al., 2011; Choudhary et al., 2010; Kam, 2008; Tsui, 2007; Barbosa et al., 2006; Baresi and Pezzè, 2006; Berdine et al., 2006; Chalin et al., 2006; Sangwan and LaPlante, 2006; Forsyth et al., 2004; Sneed and Göschl, 2000; Gerrard, 2000a;b; Jard et al., 1999; Bocchino and Hamilton, 1996; Mandl, 1985)

- Colored black
- Mainly consists of academic papers: journal articles, conference papers, reports (Kam, 2008; Gerrard, 2000a;b), and a thesis (Bas, 2024)
- Written by much smaller sets of authors with unknown peer review processes

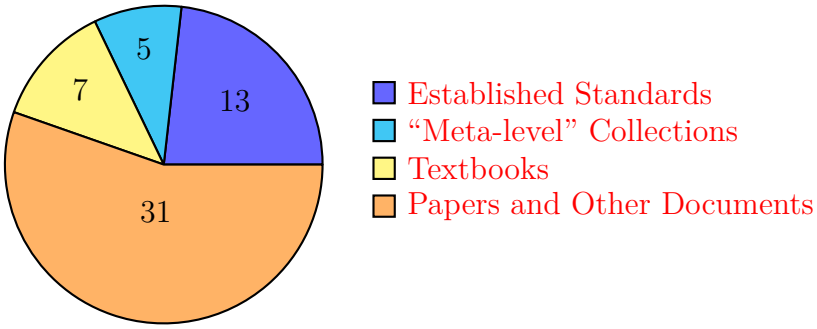


Figure 3.1: Summary of how many sources comprise each source category.

- Much less widespread than other categories of sources
- Many of these sources were investigated to "fill in" missing definitions (see Section 3.4)
- Also included (for brevity) are some less-than-academic sources to investigate how terms are used in practice, such as websites (LambdaTest, 2024; Pandey, 2023) and a booklet (Knüvener Mackert GmbH, 2022)

3.1.5 Inferences

While not as clear-cut as the other source categories, some information is inferred from various sources, such as "surface-level" analysis that follows straightforwardly without being explicitly stated in the text. Inferred relations are colored gray and inferred discrepancies are given in Section 5.9.

3.2 Terminology

This research was intended to describe the current state of testing terminology instead of prematurely applying any classifications to reduce bias. Therefore, the notions of Categories of Testing Approaches, Parent-Child Relations, and Categories of Discrepancies arose naturally from the literature. Even though these are "results" of this research, they are defined here for clarity since they are used throughout this thesis. We also define the notion of Rigidity.

3.2.1 Categories of Testing Approaches

Different sources categorize software testing approaches in different ways; while it is useful to record and think about these categorizations (see Section 8.1), following one (or more) during the research stage could lead to bias and a prescriptive categorization, instead of letting one emerge descriptively during the analysis stage. Since these categorizations are not mutually exclusive, it also means that more than one could be useful (both in general and to this specific project).
ISO/IEC and IEEE (2022) provide a classification for different kinds of tests (see

Table 3.1). A deeper rationale for a proposed classification will be given during the analysis stage.

However, other sources (Barbosa et al., 2006; Souza et al., 2017) provide alternate categories (see Table 3.2) which may be beneficial to investigate to determine if this categorization is sufficient.

A "metric" categorization was considered at one point, but was decided to be out of the scope of this project (see Chapter 2, #21, and #22). Related testing approaches may be grouped into a "class" or "family" to group those with "commonalities and well-identified variabilities that can be instantiated", where "the commonalities are large and the variabilities smaller" (see #64). 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)[1], and may also be implied for security testing, a test type that consists of "a number of techniques[2]" (ISO/IEC and IEEE, 2021, p. 40).

[1]The original source describes "performance testing … as a family of performance-related testing techniques", but it makes more sense to consider "performance-related testing" as the "family" with "performance testing" being one of the variabilities (see Section 6.3).

[2]This may or may not be distinct from the notion of "test technique" described in IEEE Testing Terminology.

Table 3.1: IEEE Testing Terminology

Term	Definition	Examples
Approach	A "high-level test implementation choice, typically made as part of the test strategy design activity" that includes "test level, test type, test technique, test practice and the form of static testing to be used" (ISO/IEC and IEEE, 2022, p. 10); described by a *test strategy* (2017, p. 472) and is also 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)
(Design)[a] Technique	A "defined" and "systematic" (ISO/IEC and IEEE, 2017, p. 464) "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); "a variety … is typically required to suitably cover any system" (2022, p. 33) and is "often selected based on team skills and familiarity, on the format of the test basis", and on expectations (2022, p. 23)	equivalence partitioning, boundary value analysis, branch testing (ISO/IEC and IEEE, 2022, p. 11)
Level[b] (sometimes "Phase"[c] or "Stage"[d])	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); more generally, "designat[es] … the coverage and detail" (2017, p. 249)	unit/component testing, integration testing, system testing (ISO/IEC and IEEE, 2022, p. 12; 2021, p. 6; 2017, p. 467)
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); more generally, a "specific type of activity that contributes to the execution of a process" (2017, p. 331)	scripted testing, exploratory testing, automated testing (ISO/IEC and IEEE, 2022, p. 20)
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] "Design technique" is sometimes abbreviated to "technique" (ISO/IEC and IEEE, 2022, p. 11; Hamburg and Mogyorodi, 2024).

[b] "Test level" can also refer to the scope of a test process; for example, "across the whole organization" or only "to specific projects" (ISO/IEC and IEEE, 2022, p. 24).

[c] "Test phase" can be a synonym for "test level" (ISO/IEC and IEEE, 2017, p. 469; 2013, p. 9) but 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).

[d] Used by (Washizaki, 2024, pp. 5-6 to 5-7; Hamburg and Mogyorodi, 2024; Gerrard, 2000a, pp. 9, 13).

Table 3.2: Other Testing Terminology

Term	Definition	Examples	IEEE Equiv.
Guidance	none given (Barbosa et al., 2006, p. 3)	none given	Technique?
Level	"distinguished based on the object of testing, the *target*, or on the purpose or *objective*" (Washizaki, 2024, p. 5-6); these are "orthogonal" and "determine how the test suite is identified … regarding its consistency … and its composition" (Washizaki, 2024, p. 5-2)	Target: unit, integration, system (Washizaki, 2024, pp. 5-6 to 5-7; Souza et al., 2017, p. 3), acceptance testing (Washizaki, 2024, p. 5-7) Objective: conformance, installation, regression, performance, reliability, security (Washizaki, 2024, pp. 5-7 to 5-9)	Target: Level Obj.: Mainly type
Method	none given (Barbosa et al., 2006, p. 3)	none given	Practice?
Phase	none given (Perry, 2006, p. 221; Barbosa et al., 2006, p. 3)	unit, integration, system, 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 (Barbosa et al., 2006, p. 3); categorized in[to] testing methods, testing guidances and testing techniques (Barbosa et al., 2006, p. 3)	none given generally; see examples of "Technique"	Approach
Process	"A sequence of testing steps" (Barbosa et al., 2006, p. 2) that is "based on a development technology and … paradigm, as well as on a testing procedure" (Barbosa et al., 2006, p. 3)	none given	Practice
Stage	An alternative to the "traditional … test stages" that is based on "clear technical groupings" (Gerrard, 2000a, p. 13); see "Level" in IEEE Testing Terminology	desktop development testing, infrastructure testing, system testing, large scale integration, and 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) "on a sound theoretical basis" (Barbosa et al., 2006, p. 3)	specification-, structure-, experience-, fault-, usage-based testing (Washizaki, 2024, pp. 5-10, 5-13 to 5-15); black-box, white-box, defect/fault-based, model-based testing (Souza et al., 2017, p. 3); functional, structural, error-based, state-based testing (Barbosa et al., 2006, p. 3)	Technique

find original source for S EtAl2017 te nique exam

It also seems that these categories are 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). Due to this, a specific test approach can be derived by combining test approaches from different categories; see Section 2.2.6 for some examples of this.

3.2.2 Parent-Child Relations

Many test approaches are multi-faceted and can be "specialized" into others, such as Performance(-related) Testing. These "specializations" will be referred to as "children" or "sub-approaches" of the multi-faceted "parent". This nomenclature also extends to other Categories of Testing Approaches from Table 3.1, such as "sub-type".

3.2.3 Categories of Discrepancies

Since there are many discrepancies in the literature, it is useful to decompose them into categories to count, analyze, and discuss them in a more focused way. The following categories are used, based on the types of discepancies observed:

- **Synonyms:** discrepancies with synonym relations of test approaches,
- **Parents:** discrepancies with Parent-Child Relations of test approaches,
- **Categories:** discrepancies with Categories of Testing Approaches,
- **Definitions:** discrepancies with definitions of test approaches and related supplementary terminology,
- **Terminology:** discrepancies with the terms used for test approaches and related supplementary terminology (these are different from "definition discrepancies" since they focus on the labels used for terms, such as when one name is used when another should have been), and
- **Sources:** discrepancies with the citations given for referenced information (such as when a source is claimed to explain something but does not).

3.2.4 Rigidity

Since there is a considerable degree of nuance introduced by the use of natural language, not all discrepancies are equal! To capture this nuance and provide a more complete picture, we make a distinction between explicit and implicit discrepancies, such as in Table 5.1. A piece of information is "implicit" if:

- it is not directly given by a source but seems to be implied, and/or
- it is only true some of the time (e.g., under certain conditions).

Discrepancies based on implicit information are themselves implicit. These are automatically detected when generating graphs and analyzing discrepancies by looking for indicators of uncertainty, such as question marks, " (Testing)" (which indicates that a test approach isn't explicitly denoted as such), and the keywords "implied", "inferred", "can be", "ideally", "usually", "most", "likely", "often", "if", and "although" (see the relevant source code). These words were 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 isn't always. As an example, Table 5.2 contains relations that are explicit, implicit, and both; implicit relations are marked by the phrase "implied by".

3.3 Procedure

To track terminology used in the literature, we build a glossary of test approaches, including the term itself, its definition, and any synonyms or parents. Any additional notes, such as questions or sources to investigate further, are also recorded. Approach categorizations, such as those found in Table 3.1 and some outliers (e.g., "artifact"), are tracked for future investigation.

Most relevant sources are analyzed in their entirety to systematically extract terminology, with the exception of some sources that were only partially investigated. This is the case for sources chosen for a specific area of interest or based on a test approach that was determined to be out-of-scope, such as some sources given in Section 3.4. Heuristics are used to guide this process, by investigating:

- 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[3] (see Section 2.2).

When terms have multiple definitions, either the clearest and most concise version is kept, or they are merged to paint a more complete picture. If any discrepancies or ambiguities arise, they are reasonably investigated and always documented. If a testing approach is mentioned but not defined, it is added to the glossary to indicate it should be investigated further (see Section 3.4). A similar

[3]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.

methodology is used for tracking software qualities, albeit in a separate document (see Section 2.2.2).

During the first pass of data collection, all software-testing-focused terms are included. Some of them are less applicable to test case automation (such as Section 2.1, #39) or too broad(such as Section 2.2.4, #55), so they will be omitted over the course of analysis.

During this investigation, some terms came up that seemed to be relevant to testing but were so vague, they didn't provide any new information. These were decided to be not worth tracking (see #39, #44, #28) and are listed below:

- **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)

3.4 Undefined Terms

The search process led to some testing approaches being mentioned without definition; (ISO/IEC and IEEE, 2022) and (Firesmith, 2015) in particular introduced many. Once Established Standards had been exhausted, we devised a strategy to look for sources that explicitly define these terms, consistent with our snowballing approach. This uncovers new approaches, both in and out of scope (such as EManations SECurity (EMSEC) testing, HTML testing, and aspects of loop testing and orthogonal array testing; see Chapter 2).

The following terms (and their respective related terms) were exploredin the following sources, bringing the number of testing approaches from 442 to 526 and the number of *undefined* terms from 156 to 172 (the assumption can be made that about 81% of added terms also included a definition):

- **Assertion Checking:** Lahiri et al. (2013); Chalin et al. (2006); Berdine et al. (2006)
- **Loop Testing**[4]**:** Dhok and Ramanathan (2016); Godefroid and Luchaup (2011); Preuße et al. (2012); Forsyth et al. (2004)

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

- **EMSEC Testing:** Zhou et al. (2012); ISO (2021)
- **Asynchronous Testing:** Jard et al. (1999)
- **Performance(-related) Testing:** Moghadam (2019)
- **Web Application Testing:** Doğan et al. (2014); Kam (2008)
  - **HTML Testing:** Choudhary et al. (2010); Sneed and Göschl (2000); Gerrard (2000b)
  - **Document Object Model (DOM) Testing:** Bajammal and Mesbah (2018)
- **Sandwich Testing:** Sharma et al. (2021); Sangwan and LaPlante (2006)
- **Orthogonal Array Testing**[5]**:** Mandl (1985); Valcheva (2013)
- **Backup Testing**[6]**:** Bas (2024)

[5] (Yu et al., 2011) and (Tsui, 2007) were added as out-of-scope examples.
[6] See Section 5.6.

Chapter 4

Tools

To better understand our findings, we build tools to visualize the relations between test approaches more intuitively (Section 4.1) and track discrepancies surrounding them automatically (Section 4.2).

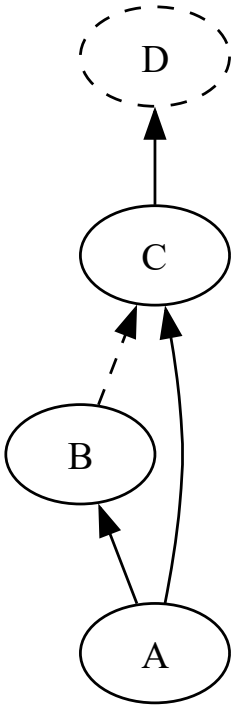
4.1 Approach Graph Generation

To better visualize how test approaches relate to each other, we developed a tool to automatically generate graphs of these relations. Since parent-child and synonym relations between approaches are tracked in our approach glossary in a consistent format, they can be parsed systematically. For example, if the entries in Table 4.1 appear in the glossary, then their parent relations are displayed as Figure 4.1a 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 4.1 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 citationg. These citations are processed according to this logic (see the relevant source code) so they can be consistently tracked throughout the analysis.

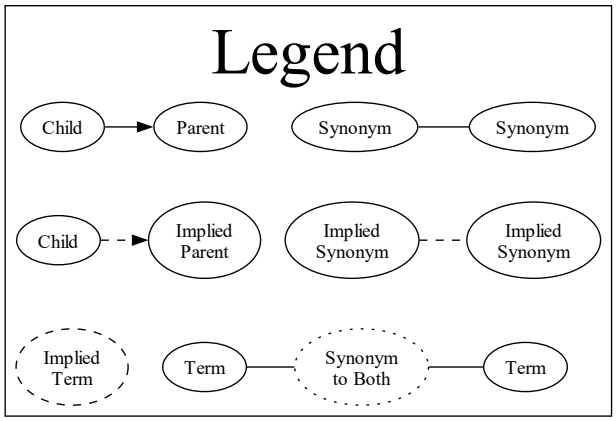
Name	Parent(s)
A	B (Author, 0000; 0001), C (0000)
B	C (implied by Author, 0000)
C	D (Author, 0002)
D (implied by Author, 0002)	

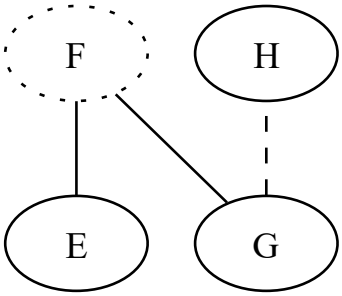
Table 4.1: Example glossary entries demonstrating how parent relations are tracked.

All parent-child relations are graphed, as well as synonym relations where either:

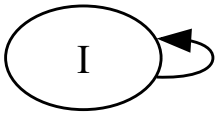


(a) Graph from Table 4.1.

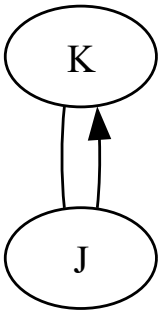




(b) Graph from Table 4.2.



(c) Graph generated from a .tex file containing a line starting with `I -> I`.



(d) Graph generated from a .tex file containing a line starting with `J -> K` if J and K are synonyms.

Figure 4.1: Example generated graphs.

1. both synonyms have their own rows in the glossary, or
2. a term is a synonym to more than one term with its own row in the glossary.

These conditions are also deduced from the information parsed from the glossary. For example, if the entries in Table 4.2 appear in the glossary, then they are displayed as Figure 4.1b in the generated graph (note that X does not appear since it does not meet the criteria given above).

Name	Synonym(s)
E	F (Author, 0000; implied by 0001)
G	F (Author, 0002), H (implied by 0000)
H	X

Table 4.2: Example glossary entries demonstrating how synonym relations are tracked.

This allows for automatic detection of some classes of discrepancies. The most trivial to automate is "multi-synonym" relations, which are already found to generate the graph as desired. The list found in Section 5.1 is automatically generated based on glossary entries such as those found in Table 4.2. The self-referential definitions in Section 5.2 were also trivial, found by simply looking for lines the generated .tex files starting with `I -> I` which would result in the graph in Figure 4.1c. 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 4.1d.

The visual nature of these graphs makes it possible to represent both explicit and implicit relations without double counting them during analysis. 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 category; note that only the explicit synonym relation between E and F from Table 4.1 is shown in Figure 4.1b. Implicit approaches and relations are denoted by dashed lines, as shown in Figures 4.1a and 4.1b; explicit approaches are *always* denoted by solid lines, even if they are also implicit. "Rigid" versions, which exclude implicit approaches and relations, can also be generated for each graph.

Should I add an example?

Since these graphs tend to be large, it is useful to focus on specific subsets of them. Graphs limited to approaches from a given approach category are generated, as well as a graph of static approaches; interestingly, static testing seems to be considered a separate approach category in (ISO/IEC and IEEE, 2022, Fig. 2). Since static testing is out of scope, it is also helpful to see how it overlaps with the in-scope dynamic testing, so these "connecting" relations are also graphed. Additionally, more specific subsets of these graphs can be generated based on a given

Better place to include this, if at all?

subset of approaches to include, such as those pertaining to recovery or scability, as shown in Figures 6.1 and 6.3, respectively (albeit with manually created legends). By specifying sets of approaches and relations to add or remove, these generated graphs can be updated in accordance with our Recommendations, as shown in Figures 6.2 and 6.4, respectively. These modifications can also be inherited, as in Figure 6.5, which was generated based on the modifications from Figures 6.2 and 6.4 and then manually tweaked.

Ensure these tweaks are on an up-to-date version!

4.2 Discrepancy Analysis

In addition to analyzing specific discrepancies (or classes of discrepancies), an overview of the amounts, sources, rigidities, and severities of these discrepancies is also useful. Subsets of this task can be automated (Section 4.2.1) and the remaining manual portion (such as finding and categorizing Other Discrepancies) can be augmented with automated tools (Section 4.2.2).

To understand where discrepancies exist in the literature, they are grouped based on the source categories (as described in Section 3.1) responsible for them. Each discrepancy 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 discrepancy twice for a given category (see #83), which would result in the number of *occurrences* of all discrepancies, instead of the number of discrepancies *themselves*, which is more useful. An exception to this is Figure 5.1, which counts discrepancies within a single document and those between documents by the same author(s) or standards organization(s) separately from those within a source category. As before, these are not double counted, meaning that the maximum number of counted discrepancies possible within a source category in Figure 5.1 is three (once for each type). This only occurs if there is an example of each type of discrepancy source where a "stricter" type does not apply.

Explain this better!

As an example of this process, consider a discrepancy *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 Table 5.1 in the relevant column:

- **Established Standards**: this discrepancy 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 discrepancies that identifies those within a singular document and those between documents by the same author is given in Figure 5.1 and explained in more detail in Section 4.2.2.

- **"Meta-level" Collections**: this discrepancy occurs between a source in this category and a "more trusted" one (the IEEE standards).

- **Papers and Other Documents**: this discrepancy 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.

4.2.1 Automated Discrepancy Analysis

As outlined in Section 4.1, some types of discrepancies can be detected automatically. While just counting the total number of these types of discrepancies is trivial, tracking the source(s) of these discrepancies is more involved. Since the appropriate citations for each piece of information is tracked (see Tables 4.1 and 4.2 for examples of how these citations are formatted in the glossaries), they can be used to find the offending source categories. This comes with the added benefit of being available to format these citations to use LaTeX's citation commands for use in the lists of discrepancies in Sections 5.1 and 5.2, including Table 5.1.

Comparing the authors and years of each source related to a given discrepancy can determine if it manifests within a single document and/or between documents by the same author(s) when creating Figure 5.1. Then, the relevant sources can be sorted into their categories based on their citations, done by the function in Source Code A.3, since each source category outlined in Section 3.1 is comprised of a small number of authors (with the exception of Papers and Other Documents). This determines the appropriate row of Table 5.1 and the appropriate graph and slice in Figure 5.1. 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 discrepancy manifests between source categories. Examples of this process are described in more detail in Section 4.2.2.

Alongside this citation information are the keywords relevant for assessing a piece of information's Rigidity. This is useful when counting discrepancies, since a discrepancy can be both explicit and implicit, but should not be double counted as both (see #83)! When counting discrepancies in Table 5.1, a given discrepancy is counted only for its most "rigid" manifest (i.e., it will only increment a value in the "Implicit" column if it is *not* also explicit).

4.2.2 Augmented Discrepancy Analysis

While the discrepancies in Sections 5.1 and 5.2, including Table 5.1, could be deduced automatically from analyzing the testing approach glossary, other types of discrepancies (namely Other Discrepancies) needed to be tracked manually. This is done by adding comments to the relevant LaTeX files (generated or not) of the form

```
% Discrep count: {A1} {A2} … | {B1} {B2} … | {C1} {C2} …
```

which can then be parsed to detemine where discrepancies occur. Each group of sources is separated with a pipe symbol to be compared with the others, so any number of groups are permitted. If only one group is present, it is compared with

itself. For example, the first line below means that source X has a discrepancy with itself, while the second line adds a discrepancy between X and Y.

```
% Discrep count: {X}
% Discrep 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.

```
% Discrep 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

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

would be parsed as the example given in Section 4.2. 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 5.1.

The Rigidity of discrepancies 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 Automated Discrepancy Analysis. For example, the line

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

indicates that the following discrepancies are present. These all increment counts in both Figure 5.1 and Table 5.1 by only one, except for the second, which only affects Figure 5.1. This is because it is less "rigid" than another discrepancy within Established Standards (i.e., the first one) and as such is not double counted in Table 5.1:

- 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.

Chapter 5

Discrepancies and Ambiguities

After gathering all this data[1], we found many discrepancies and ambiguities. A summary of these is shown in Table 5.1, where a given row corresponds to the number of discrepancies either within that category and/or with a "more trusted" source category (i.e., a previous row in the table). Issues with Synonyms, Parent Relations, and Categories of Testing Approaches are (Exp)licit or (Imp)licit. Issues with Functional Testing, Operational (Acceptance) Testing (OAT), Recovery Testing, and Scalability Testing are also given, although not listed separately in Table 5.1; these are counted alongside Other Discrepancies, all grouped into degrees of severity as follows:

- High: Semantic differences between test approaches
- (Med)ium: Differences in supporting information about test approaches
- Low: Typos, redundancy, or issues with referencing

5.1 Synonyms

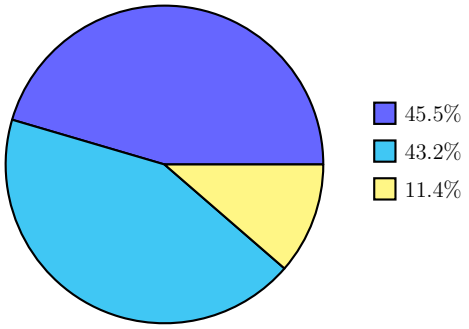
more in Umar2000

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

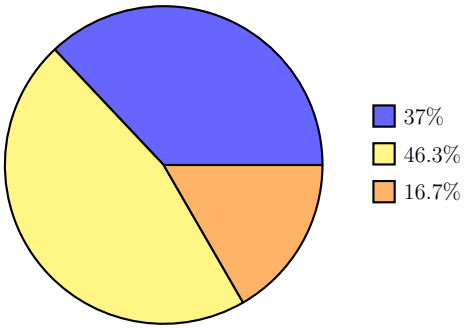
1. Black-Box Testing (ISO/IEC and IEEE, 2022, p. 9; 2021, p. 8; 2017, p. 431; Washizaki, 2024, p. 5-10; Hamburg and Mogyorodi, 2024; Firesmith, 2015, p. 46 (without hyphen); Sakamoto et al., 2013, p. 344; van Vliet, 2000, p. 399)
2. Closed-Box Testing (ISO/IEC and IEEE, 2022, p. 9; 2017, p. 431)
3. Functional Testing[2] (2017, p. 196; Kam, 2008, p. 44; van Vliet, 2000, p. 399; implied by 2021, p. 129; 2017, p. 431)

[1] Available in `ApproachGlossary.csv` and `QualityGlossary.csv` at https://github.com/samm82/TestGen-Thesis.

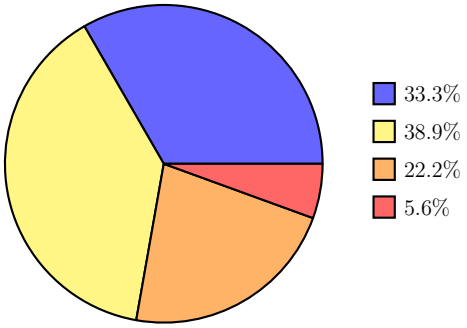
[2] This may be an outlier; see Section 5.4.1.



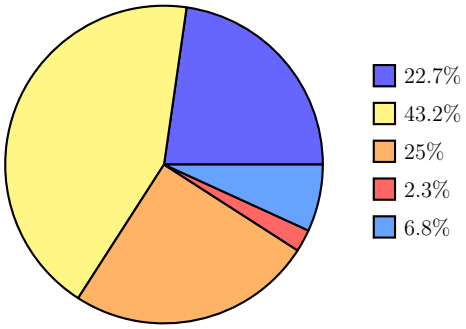
(a) Discrepancies found in Established Standards.



(b) Discrepancies found in "Meta-level" Collections.



(c) Discrepancies found in Textbooks.



(d) Discrepancies found in Papers and Other Documents.

Legend

- 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 "meta-level" document
- Between a document from this category and a textbook
- Between a document from this category and a paper

Figure 5.1: Sources of discrepancies based on source category.

4. Domain Testing (Washizaki, 2024, p. 5-10)
5. Input Domain-Based Testing (implied by (Bourque and Fairley, 2014, p. 4-8))

While some of these synonyms may express mild variations, their core meaning is nevertheless the same. Here we use the terms "specification-based" and "structure-based testing" as they articulate the source of the information for designing test cases, but a team or project also using gray-box testing may prefer the terms "black-box" and "white-box testing" for consistency. Thus, synonyms do not inherently signify a discrepancy. Unfortunately, there are many instances of incorrect or ambiguous synonyms, such as the following:

1. Sneed and Göschl give "white-", "grey-", and "black-box testing" as synonyms for "module", "integration", and "system testing", respectively (2000, p. 18), but this mapping is incorrect; black-box testing can be performed on a module, for example. This makes the claim that "red-box testing" is a synonym for "acceptance testing" (p. 18) lose credibility.

OG Hetzel88

find source

2. "Program testing" is given as a synonym of "component testing" (Kam, 2008, p. 46), although it probably should be a synonym of "system testing" instead.
3. Kam seems to imply that "mutation testing" is a synonym of "back-to-back testing" (2008, p. 46), but these are two quite distinct techniques.
4. "Conformance testing" is implied to be a synonym of "compliance testing" by Kam, which only makes sense because of the vague definition of "compliance testing": "testing to determine the compliance of the component or system" (2008, p. 43).

There are also cases in which a term is given a synonym to two (or more) disjoint, unrelated terms, which would be a source of ambiguity to teams using these terms. Ten of these cases were identified through automatic analysis of the generated graphs, listed below:

1. **Invalid Testing:**
   - Error Tolerance Testing (Kam, 2008, p. 45)
   - Negative Testing (Hamburg and Mogyorodi, 2024; implied by ISO/IEC and IEEE, 2021, p. 10)
2. **Soak Testing:**
   - Endurance Testing (ISO/IEC and IEEE, 2021, p. 39)
   - Reliability Testing[3] (Gerrard, 2000a, Tab. 2; 2000b, Tab. 1, p. 26)

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

3. **User Scenario Testing:**
    - Scenario Testing (Hamburg and Mogyorodi, 2024)
    - Use Case Testing[4] (Kam, 2008, p. 48; although "an actor can be a user or another system" (ISO/IEC and IEEE, 2021, p. 20))
4. **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, p. 44; van Vliet, 2000, p. 399; implied by ISO/IEC and IEEE, 2021, p. 129; 2017, p. 431)
5. **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)
6. **(Multiple) Condition Testing:**
    - Branch Condition Combination Testing (Hamburg and Mogyorodi, 2024; Patton, 2006, p. 120)
    - Branch Condition Testing (Patton, 2006, p. 120)
7. **Extended Branch Coverage:**
    - Branch Condition Combination Testing (van Vliet, 2000, p. 422)
    - Branch Condition Testing (van Vliet, 2000, p. 422)
8. **State-based Testing:**
    - State Transition Testing (Firesmith, 2015, p. 47)
    - State-based Web Browser Testing (Doğan et al., 2014, p. 193)
9. **Static Verification:**
    - Static Assertion Checking[5] (Chalin et al., 2006, p. 343)
    - Static Testing (implied by Chalin et al., 2006, p. 343)

[4] "Scenario testing" and "use case testing" are given as synonyms by Hamburg and Mogyorodi (2024) and Kam (2008, pp. 47-49) but listed separately by ISO/IEC and IEEE (2022, p. 22), who also give "use case testing" as a "common form of scenario testing" (2021, p. 20). This implies that "use case testing" may instead be a child of "user scenario testing" (see Table 5.2).

[5] Chalin et al. list Runtime Assertion Checking (RAC) and Software Verification (SV) as "two complementary forms of assertion checking" (2006, p. 343); 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.

10. **Testing-to-Fail:**
    - Forcing Exception Testing (Patton, 2006, pp. 66-67, 78)
    - Negative Testing (Patton, 2006, pp. 67, 78, 84-87)

5.2 Parent Relations

Parent relations are not immune to difficulties, including self-referential definitions[6], which 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)

Performance testing is *not* described as a sub-approach 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 sub-approach of the other, abusing the meaning of "synonym" and causing confusion. We identified 11 of these pairs through automatic analysis of the generated graphs, which are given in Table 5.2. Finally, it is worth pointing out that fault tolerance testing may also be a sub-approach 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).

[6]Since these are by nature self-contained within a given source, these are counted *once* as explicit discrepancies within their sources in Table 5.1.

Table 5.1: Breakdown of identified discrepancies by source and type.

Source Category	Synonyms		Parents		Categories		Definitions		Terminology		Sources		Total
	Exp	Imp	Exp	Imp	Exp	Imp	Exp	Imp	Exp	Imp	Exp	Imp	
Established Standards	0	1	6	1	8	2	8	1	3	0	0	0	30
"Meta-level" Collections	7	3	7	4	3	6	13	0	8	0	3	0	54
Textbooks	10	0	2	1	1	0	3	1	0	0	0	0	18
Papers and Other Documents	11	5	10	0	5	2	6	0	5	0	0	0	44
Total	28	9	25	6	17	10	30	2	16	0	3	0	146

Table 5.2: Pairs of test approaches with both parent-child and synonym relations.

"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)
Fault Tolerance Testing → Robustness Testing	(Firesmith, 2015, p. 56)	(Hamburg and Mogyorodi, 2024)
Functional Testing → Specification-based Testing	(ISO/IEC and IEEE, 2021, p. 38)	(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)
Orthogonal Array Testing → Pairwise Testing	(Mandl, 1985, p. 1055)	(Washizaki, 2024, p. 5-11; Valcheva, 2013, p. 473)

Continued on next page

Table 5.2: Pairs of test approaches with both parent-child and synonym relations. (Continued)

"Child" → "Parent"	**Parent-Child Source(s)**	**Synonym Source(s)**
Performance Testing → Performance-related Testing	(ISO/IEC and IEEE, 2022, p. 22; 2021, p. 38)	(Moghadam, 2019, p. 1187)
Use Case Testing → Scenario Testing	(ISO/IEC and IEEE, 2021, p. 20; OG Hass, 2008)	(Hamburg and Mogyorodi, 2024; Kam, 2008, pp. 47-49)
Condition Testing → Decision Testing	(implied by Hamburg and Mogyorodi, 2024)	(Washizaki, 2024, p. 5-13)
Dynamic Analysis → Dynamic Testing	(implied by its static counterpart in ISO/IEC and IEEE, 2022, pp. 9, 17, 25, 28; Hamburg and Mogyorodi, 2024)	(ISO/IEC and IEEE, 2017, p. 149)
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)

5.3 Categories of Testing Approaches

While the IEEE categorization of testing approaches is useful, it is not without its faults. The boundaries between items 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:

1. 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)!
    - These authors previously say "experience-based testing practices like exploratory testing … are not … techniques for designing test cases", although they "can use … test techniques" (2021, p. viii). This implies that "experience-based test design techniques" are techniques used by the *practice* of experience-based testing, not that experience-based testing is *itself* a test technique. If this is the case, it is not always clearly articulated (ISO/IEC and IEEE, 2022, pp. 4, 22; 2021, p. 4; Washizaki, 2024, p. 5-13; Hamburg and Mogyorodi, 2024) and is sometimes contradicted (Firesmith, 2015, p. 46). However, this conflates the distinction between "practice" and "technique", making these terms less useful, so this may just be a mistake (see #64).
    - This also causes confusion about its children, such as error guessing and exploratory testing; again, on the same page, ISO/IEC and IEEE say error guessing is an "experience-based test design technique" and "experience-based test practices include … exploratory testing, tours, attacks, and checklist-based testing" (2022, p. 34). Other sources also do not agree whether error guessing is a technique (pp. 20, 22; 2021, p. viii) or a practice (Washizaki, 2024, p. 5-14).
2. The following test approaches are categorized as test techniques by (ISO/IEC and IEEE, 2021, p. 38) and as test types by the sources provided:
    (a) Capacity testing (ISO/IEC and IEEE, 2022, p. 22; 2013, p. 2; implied by its quality (ISO/IEC, 2023a; ISO/IEC and IEEE, 2021, Tab. A.1) and by (Firesmith, 2015, p. 53)),
    (b) Endurance testing (ISO/IEC and IEEE, 2013, p. 2; implied by (Firesmith, 2015, p. 55)),
    (c) Load testing (ISO/IEC and IEEE, 2022, pp. 5, 20, 22; 2017, p. 253; Hamburg and Mogyorodi, 2024; implied by (Firesmith, 2015, p. 54)),
    (d) Performance testing (ISO/IEC and IEEE, 2022, pp. 7, 22, 26-27; 2021, p. 7; implied by (Firesmith, 2015, p. 53)), and

OG IEEE 2013

(e) Stress testing (ISO/IEC and IEEE, 2022, pp. 9, 22; 2017, p. 442; implied by (Firesmith, 2015, p. 54)).

3. "Installability testing" is given as a test type (ISO/IEC and IEEE, 2022, p. 22; 2021, p. 38) but is sometimes called a test level as "installation testing" (Peters and Pedrycz, 2000, p. 445).

4. Model-based testing is categorized as both a test practice (ISO/IEC and IEEE, 2022, p. 22; 2021, p. viii) and a test technique (Kam, 2008, p. 4; implied by ISO/IEC and IEEE, 2021, p. 7; 2017, p. 469).

5. Data-driven testing is categorized as both a test practice (ISO/IEC and IEEE, 2022, p. 22) and a test technique (Kam, 2008, p. 43).

OG Fewster and Graham

6. Although ad hoc testing is sometimes 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).

There are also instances of inconsistencies between parent and child test approach categorizations. This may indicate they aren't necessarily the same, or that more thought must be given to this method of classification.

5.4 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:

5.4.1 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, 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,

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

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).

5.4.2 Correctness Testing

Washizaki 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" (2024, p. 5-7); 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 2.2.2), it seems consistent to label "functional testing" as a "test type" (ISO/IEC and IEEE, 2022, pp. 15, 20, 22); this conflicts with its categorization as a "technique" if considered a synonym of Specification-based Testing. "Correctness testing" is listed separately from "functionality testing" by Firesmith (2015, p. 53).

5.4.3 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.

5.4.4 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.

5.4.5 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 sub-approach. Its associated test type is implied to be a sub-approach of build verification testing (Hamburg and Mogyorodi, 2024) and made distinct from "functional testing"; interestingly, security is described as a sub-approach of both non-functional and functionality testing (Gerrard, 2000a, Tab. 2). "Functionality testing" is listed separately from "correctness testing" by Firesmith (2015, p. 53).

5.5 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).

5.6 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 6.1.

- **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 sub-type 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 sub-type 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 uses the term "failover and recovery testing" (2015, p. 56), which could be a synonym of both of these terms.

5.7 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 6.3.

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 implies that it is limited to the system itself (2023a)
3. The SWEBOK Guide V4's definition of "scalability testing" (Washizaki, 2024, p. 5-9) is really a definition of usability testing!

5.8 Other Discrepancies

We now outline discrepancies/ambiguities found in the literature that were not "large" enough to merit their own sections, grouped by the "categories" of sources outlined in Section 3.1.

5.8.1 Other Discrepancies from Established Standards

High Severity

- "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.

  - 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.

- 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). This means unit/component/module testing can refer to the testing of both a module and a specific function in a module (see #14). However, "component" is sometimes defined differently than "module": "components differ from classical modules for being re-used in different contexts independently of their development" (Baresi and Pezzè, 2006, p. 107), so distinguishing the two may be necessary.

Medium Severity

- 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.

- 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 ISO1984

- ISO/IEC and IEEE say that "test level" and "test phase" are synonyms[7], both meaning a "specific instantiation of [a] test sub-process" (2017, pp. 469, 470; 2013, p. 9), but there are also alternative definitions for them. "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), while "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).

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

Low Severity

- Integration, system, and system integration testing are all listed as "common test levels" (ISO/IEC and IEEE, 2022, p. 12; 2021, p. 6), but no definitions

[7] Although this is a discrepancy based on a synonym relation, the "synonyms" are supporting terms and not test approaches, which is why this is not included in Section 5.1 as a synonym discrepancy.

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).

- Similarly, component, integration, 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).

- 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".

- 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.

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 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 5.6)
- Response-Time Testing
- *Reviews* (ISO/IEC 20246) (Code Reviews*)
- Scalability Testing (defined as a synonym of "capacity testing"; see Section 5.7)
- 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

5.8.2 Other Discrepancies from "Meta-level" Collections

High Severity

- 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): testing "conducted to evaluate the degree to which a test item, [sic] and associated data and information, are protected so that" only "authorized persons or systems" can use them as intended.
- 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*.

- Hamburg and Mogyorodi (2024) describe the term "software in the loop" as a kind of testing, while the source it references 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?

- 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).

- It is ambiguous whether "tool/environment testing" refers to testing the tools/environment *themselves* or *using* them to test the object under test; the latter is implied, but the wording of its subtypes (Firesmith, 2015, p. 25) seems to imply the former.

- The terms "acceleration" 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…

- 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.

Medium Severity

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

- "Machine Learning (ML) model testing" and "ML functional performance" are defined 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" (Hamburg and Mogyorodi, 2024). The use of "performance" (or "correctness") in these definitions is at best ambiguous and at worst incorrect.

- 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".

- A similar issue exists with multiplayer testing, where its definition specifies "the casino game world" (Hamburg and Mogyorodi, 2024).
- Thirdly, "par sheet testing" from (Hamburg and Mogyorodi, 2024) seems to refer to this specific example 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).
- There is disagreement on the structure of tours; they can either be quite general (ISO/IEC and IEEE, 2022, p. 34) or "organized around a special focus" (Hamburg and Mogyorodi, 2024).
- Performance 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).

Low Severity

- 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).
- The source cited for the definition of "test type" from (Hamburg and Mogyorodi, 2024) does not seem to provide a definition itself.
- The same is true for "visual testing" (Hamburg and Mogyorodi, 2024).
- The same is true for "security attack" (Hamburg and Mogyorodi, 2024).
- "Hardware-" and "human-in-the-loop testing" have the same acronym: "HIL"[8] (Firesmith, 2015, p. 23).
- The same is true for "customer" and "contract(ual) acceptance testing" ("CAT") (Firesmith, 2015, p. 30).
- The acronym "SoS" is used but not defined by Firesmith (2015, p. 23).

5.8.3 Other Discrepancies from Textbooks

High Severity

- 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).
- "Load testing" is defined as using loads "usually between anticipated conditions of low, typical, and peak usage" (ISO/IEC and IEEE, 2022, p. 5), while Patton says the loads should as large as possible (2006, p. 86).

[8] "HiL" is used for the former by Preuße et al. (2012, p. 2).

- 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)!

Medium Severity

- Patton says that reviews are "*the* process[es] under which static white-box testing is performed" (2006, p. 92, emphasis added), but van Vliet also gives correctness proofs (2000, pp. 418-419).
- Hamburg and Mogyorodi claim that code inspections are related to peer reviews (2024), but Patton (2006, pp. 94-95) makes them quite distinct.
- Likewise, "walkthroughs" and "structured walkthroughs" are given as synonyms by Hamburg and Mogyorodi (2024) but Peters and Pedrycz imply that they are different by saying a more structured walkthrough may have specific roles (2000, p. 484).

5.8.4 Other Discrepancies from Papers and Other Documents

High Severity

OG Beizer

- Kam says that the goal of negative testing is "showing that a component or system does not work" (2008, p. 46) 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!
- Gerrard makes a distinction between "transaction verification" and "transaction testing" (2000a, Tab. 2) and uses the phrase "transaction flows" (Fig. 5) but doesn't explain them.

Medium Severity

- Bas lists "three [backup] location categories: local, offsite and cloud based [sic]" (2024, p. 16), but does not define or discuss "offsite backups" (pp. 16-17).
- Availability testing isn't assigned to a test priority (Gerrard, 2000a, Tab. 2), despite the claim that "the test types[9] have been allocated a slot against the four test priorities" (p. 13); I think usability and/or performance would have made sense.

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

- "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).

Low Severity

- Doğan et al. claim that (Sakamoto et al., 2013) defines "prime path coverage" (2014, p. 184), but it doesn't.
- "State-based" is misspelled by Kam as "state-base" (2008, pp. 13, 15) and "stated-base" (Tab. 1).
- Kam's definition of "boundary value testing" says "See *boundary value analysis*," but this definition is not present (2008).
- The phrase "continuous automated testing" (Gerrard, 2000a, p. 11) is redundant since continuous testing is a sub-category of automated testing (ISO/IEC and IEEE, 2022, p. 35, Hamburg and Mogyorodi, 2024).

5.9 Inferred Discrepancies

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

5.9.1 Inferred Synonym Discrepancies

See Section 5.1.

1. **Production Acceptance Testing:**
   - Operational Testing (Hamburg and Mogyorodi, 2024)[10]
   - Production Verification Testing[11]
2. **Operational Testing:**
   - Field Testing
   - Qualification Testing

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

[11] "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.

5.9.2 Inferred Parent Discrepancies

As in Table 5.2, some discrepancies occur when test approaches are classified as both children/parents *and* synonyms. The first two lists below give approaches that are given one of these relations by at least one source when it may make more sense for them to have the other relation. The third list gives approaches that could be inferred to have either relation, so more thought would have to be given before a recommendation can be made.

Pairs labelled as "children/parents"

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

Pairs labelled as "synonyms"

1. Structured Walkthroughs → Walkthroughs[12] (Hamburg and Mogyorodi, 2024)
2. Functionality Testing → Functional Suitability Testing (implied by Hamburg and Mogyorodi, 2024; this seems wrong)

Pairs that could be "children/parents" *or* "synonyms"

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

5.9.3 Other Inferred Discrepancies

The following are discrepancies that, if were more concrete, would be included in Section 5.8:

- "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).
- End-to-end functionality testing is *not* indicated to be functionality testing (Gerrard, 2000a, Tab. 2).

[12] See Section 5.8.3.

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

Chapter 6

Recommendations

We provide different recommendations for resolving various discrepancies (see Chapter 5). This was done with the goal of organizing them more logically and making 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)

The following are our recommendations for the areas of Recovery, Scalability, and Performance(-related) Testing, along with graphs of these subsets.

6.1 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 Figures 6.1 and 6.2.

- **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.

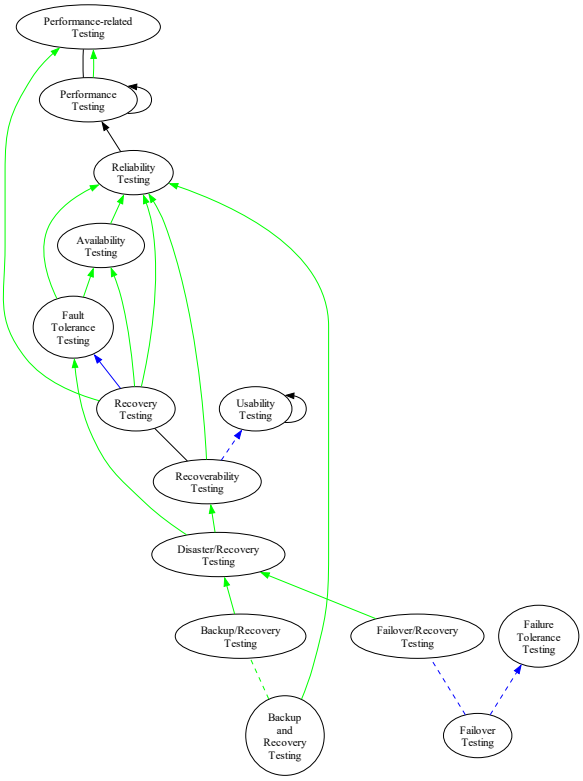


Figure 6.1: Current relations between "recovery testing" terms.

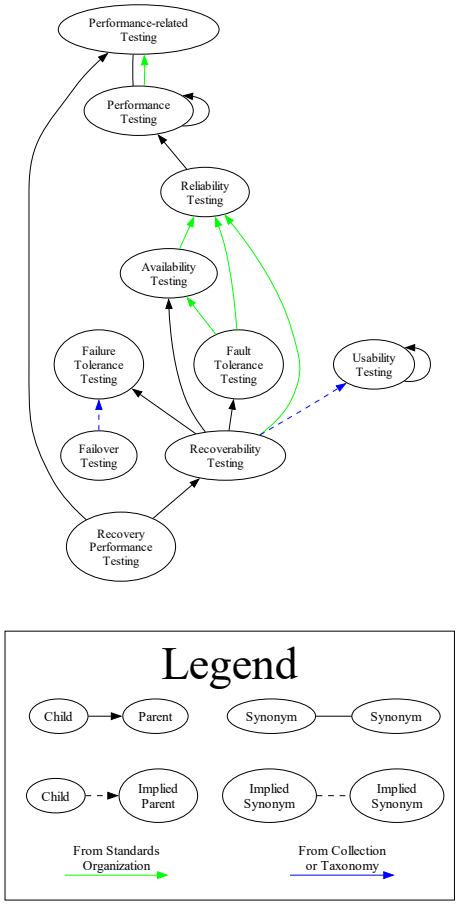


Figure 6.2: Proposed relations between rationalized "recovery testing" terms.

- **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 (see #40), but was not captured consistently by the literature. This will be a subset of "performance-related testing" (see Section 6.3) 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.

6.2 Scalability Testing

The ambiguity around scalability testing found in the literature is resolved and/or explained by other sources! ISO/IEC and IEEE 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" (2021, p. 39). This focus on "the future" is supported by Hamburg and Mogyorodi, who define "scalability" as "the degree to which a component or system can be adjusted for changing capacity" (2024); 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 describes it as testing the "capability of a product to handle growing or shrinking workloads or to adapt its capacity to handle variability" (2023a), implying that this is done by the system itself. The potential reason for this is implied by the SWEBOK Guide V4's claim that one objective of elasticity testing is "to evaluate scalability" (Washizaki, 2024, p. 5-9): ISO/IEC's notion of "scalability" likely refers more accurately to "elasticity"!

This also makes sense in the context of other definitions provided by the SWEBOK Guide V4:

- **Scalability:** "the software's ability to increase and scale up on its nonfunctional requirements, such as load, number of transactions, and volume of data" (Washizaki, 2024, 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" (Washizaki, 2024, 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.

This distinction is also consistent with how the terms are used in industry: Pandey 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" (2023; see #35).

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 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 Figures 6.3 and 6.4.

6.3 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, 2022, p. 7; 2017, p. 320; similar in Moghadam, 2019, p. 1187), or

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

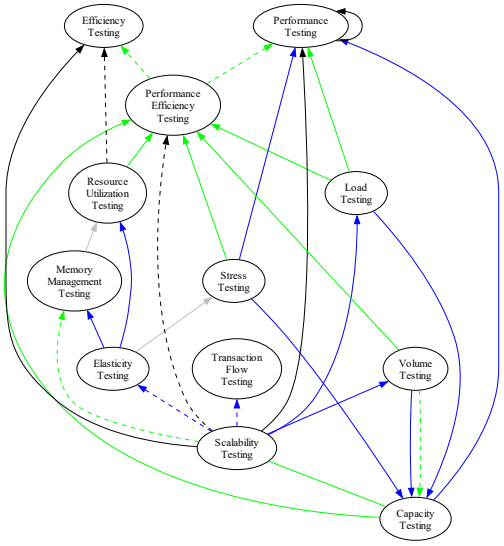


Figure 6.3: Current relations between "scalability testing" terms.

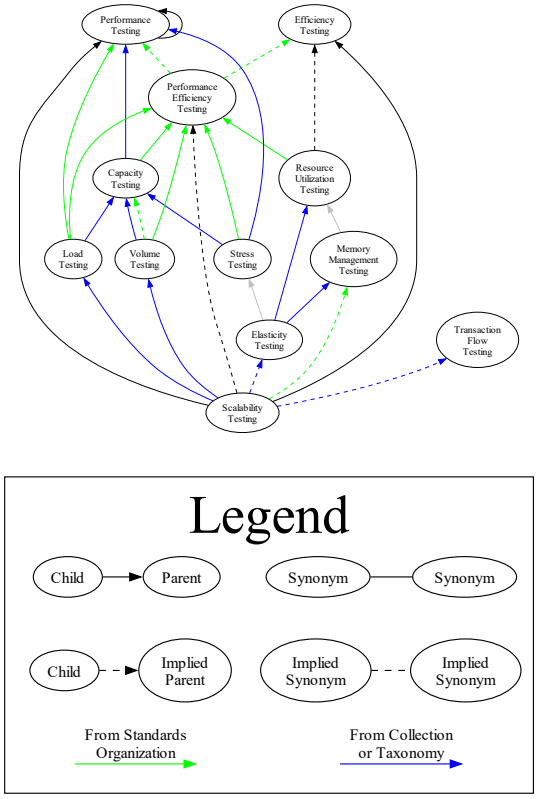


Figure 6.4: Proposed relations between rationalized "scalability testing" terms.

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). 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 sub-approaches 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 5.1. 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 sub-approach of this performed "under a 'typical' load"

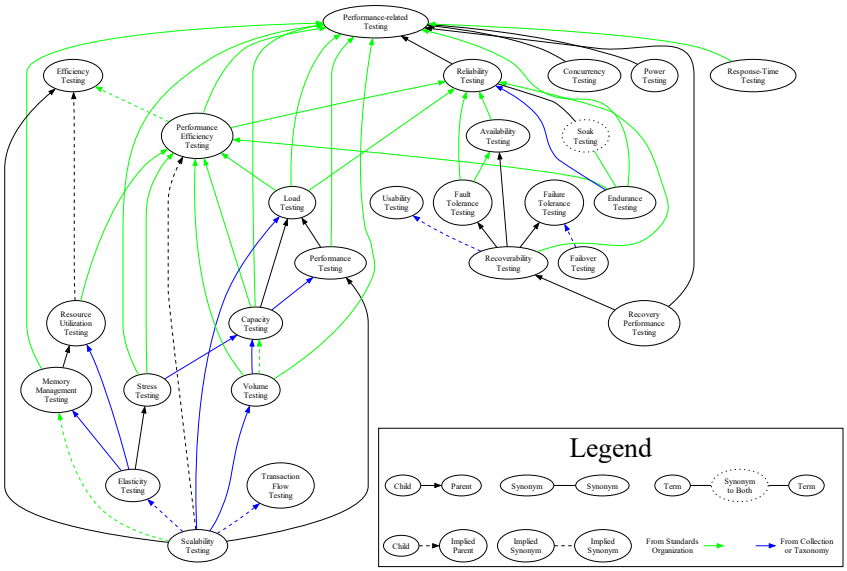


Figure 6.5: Proposed relations between rationalized "performance-related testing" terms.

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

OG IEEE 2013

Finally, the "self-loops" mentioned in Section 5.2 provide no new information and can be removed. These changes (along with those from Sections 6.1 and 6.2 made implicitly) result in the relations shown in Figure 6.5.

Chapter 7

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 `floats` and `ints`
- 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 7.3)

- 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

7.1 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

7.1.1 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.

7.2 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

7.3 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 A.4).

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 A.5) was "disconnected" from the manually written requirement (Source Code A.6), 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 `InstanceModels`. 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 `DataDefinitions` in GlassBR that were outputs to be `InstanceModels` (#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 8

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

8.1 Categorizations

Software testing approaches can be divided into the following categories. 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)!

Despite its challenges, it is useful to understand the differences between testing classes because tests from multiple subsets within the same category, such as functional and structural, "use different sources of information and have been shown to highlight different problems" (Washizaki, 2024, p. 5-16). However, some subsets, such as deterministic and random, may have "conditions that make one approach more effective than the other" (Washizaki, 2024, p. 5-16).

- Visibility of code: black-, white-, or grey-box (specificational/functional, structural, or a mix of the two) (ISO/IEC and IEEE, 2021, p. 8; Washizaki, 2024, pp. 5-10, 5-16; Sharma et al., 2021, p. 601, called "testing approaches" and (stepwise) code reading replaced "grey-box testing"; Ammann and Offutt, 2017, pp. 57-58; Kuļešovs et al., 2013, p. 213; Patton, 2006, pp. 53, 218; Perry, 2006, p. 69; Kam, 2008, pp. 4-5, called "testing methods")

OG [3, 4, 5, 8]

- Level/stage[1] of testing: unit, integration, system, or acceptance (Washizaki, 2024, pp. 5-6 to 5-7; Hamburg and Mogyorodi, 2024; Kuļešovs et al., 2013,

[1] See Table 3.1.

OG Black, 2009

p. 218 ; Patton, 2006; Perry, 2006; Peters and Pedrycz, 2000; Gerrard, 2000a, pp. 9, 13) (sometimes includes installation (van Vliet, 2000, p. 439) or regression (Barbosa et al., 2006, p. 3))

- Source of information for design: specification, structure, or experience (ISO/IEC and IEEE, 2021, p. 8)
  - Source of test data: specification-, implementation-, or error-oriented (Peters and Pedrycz, 2000, p. 440)
- Test case selection process: deterministic or random (Washizaki, 2024, p. 5-16)
- Coverage criteria: input space partitioning, graph coverage, logic coverage, or syntax-based testing (Ammann and Offutt, 2017, pp. 18-19)
- Question: what-, when-, where-, who-, why-, how-, and how-well-based testing; these are then divided into a total of "16 categories of testing types"[2] (Firesmith, 2015, p. 17)
- Execution of code: static or dynamic (Kuļešovs et al., 2013, p. 214; Gerrard, 2000a, p. 12; Patton, 2006, p. 53)
- Goal of testing: verification or validation (Kuļešovs et al., 2013, p. 214; Perry, 2006, pp. 69-70)
- Property of code (Kuļešovs et al., 2013, p. 213) or test target (Kam, 2008, pp. 4-5): functional or non-functional
- Human involvement: manual or automated (Kuļešovs et al., 2013, p. 214)
- Structuredness: scripted or exploratory (Kuļešovs et al., 2013, p. 214)
- Coverage requirement: data or control flow (Kam, 2008, pp. 4-5)
- Adequacy criterion: coverage-, fault-, or error-based ("based on knowledge of the typical errors that people make") (van Vliet, 2000, pp. 398-399)
- Priority[3]: smoke, usability, performance, or functionality testing (Gerrard, 2000a, p. 12)
- Category of test "type"[4]: static testing, test browsing, functional testing, non-functional testing, or large scale integration (testing) (Gerrard, 2000a, p. 12)
- Purpose: correctness, performance, reliability, or security (Pan, 1999)

[2]Not formally defined, but distinct from the notion of "test type" described in IEEE Testing Terminology.

[3]In the context of testing e-business projects.

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

Tests can also be tailored to "test factors" (also called "quality factors" or "quality attributes"): "attributes of the software that, if they are wanted, pose a risk to the success of the software" (Perry, 2006, p. 40). These include correctness, file integrity, authorization, audit trail, continuity of processing, service levels (e.g., response time), access control, compliance, reliability, ease of use, maintainability, portability, coupling (e.g., with other applications in a given environment), performance, and ease of operation (e.g., documentation, training) (Perry, 2006, pp. 40-41). *These may overlap with Derived Test Approaches and/or the "Results of Testing (Area of Confidence)" column in the summary spreadsheet.*

Engström "investigated classifications of research" (Engström and Petersen, 2015, p. 1) on the following four testing techniques. *These four categories seem like comparing apples to oranges to me.*

- **Combinatorial testing:** how the system under test is modelled, "which combination strategies are used to generate test suites and how test cases are prioritized" (Engström and Petersen, 2015, pp. 1-2)
- **Model-based testing:** the information represented and described by the test model (Engström and Petersen, 2015, p. 2)
- **Search-based testing:** "how techniques [5] had been empirically evaluated (i.e. objective and context)" (Engström and Petersen, 2015, p. 2)
- **Unit testing:** "source of information (e.g. code, specifications or testers intuition)" (Engström and Petersen, 2015, p. 2)

8.2 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). The question

is this punctuation right?

[5] Not formally defined, but distinct from the notion of "test technique" described in IEEE Testing Terminology.

"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.*

Most ontologies I can find seem to focus on the high-level testing process rather than the testing approaches themselves. For example, the terms and definitions (Tebes et al., 2020b) from TestTDO (Tebes et al., 2020a) provide *some* definitions of testing approaches, but mainly focus on parts of the testing process (e.g., test goal, test plan, testing role, testable entity) and how they relate to one another. Tebes et al. (2019, pp. 152-153) may provide some sources for software testing terminology and definitions (this seems to include the ones 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).

Unterkalmsteiner et al. (2014) provide a foundation to allow one "to classify and characterize alignment research and solutions that focus on the boundary between [requirements engineering and software testing]" but "do[] not aim at providing a systematic and exhaustive state-of-the-art survey of [either domain]" (p. A:2).

Another source introduced the notion of an "intervention": "an act performed (e.g. use of a technique[6] 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).

8.3 Definitions

OG Myers 1976

- 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

[6]Not formally defined, but distinct from the notion of "test technique" described in IEEE Testing Terminology.

(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)

OG ISO/IEC 2014

- 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)
- Defect: "an observable difference between what the software is intended to do and what it does" (Washizaki, 2024, p. 1-1); "can be used to refer to either a fault or a failure, [sic] when the distinction is not important" (Bourque and Fairley, 2014, p. 4-3)

OG?

- Error: "a human action that produces an incorrect result" (van Vliet, 2000, p. 399)
- Fault: "the manifestation of an error" in the software itself (van Vliet, 2000, p. 400); "the *cause* of a malfunction" (Washizaki, 2024, p. 5-3)
- Failure: incorrect output or behaviour resulting from encountering a fault; can be defined as not meeting specifications or expectations and "is a relative notion" (van Vliet, 2000, p. 400); "an undesired effect observed in the system's delivered service" (Washizaki, 2024, p. 5-3)
- 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
- 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)
- Probe: "a statement inserted into a program" for the purpose of dynamic testing (Peters and Pedrycz, 2000, p. 438)

8.3.1 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)

8.4 General Testing Notes

- 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 (see #54). 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

OG ISO 26262

- "Proving the correctness of software … applies only in circumstances where software requirements are stated formally" and assumes "these formal requirements are themselves correct" (van Vliet, 2000, p. 398)
- 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)
- Black-box test cases should be created based on the specification *before* creating white-box test cases to avoid being "biased into creating test cases based on how the module works" (Patton, 2006, p. 113)
- Simple, normal test cases (test-to-pass) should always be developed and run before more complicated, unusual test cases (test-to-fail) (Patton, 2006, p. 66)
- "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)

- When comparing adequacy criteria, "criterion X is stronger than criterion Y if, for all programs P and all test sets T, X-adequacy implies Y-adequacy" (the "stronger than" relation is also called the "subsumes" relation) (van Vliet, 2000, p. 432); this relation only "compares the thoroughness of test techniques, not their ability to detect faults" (van Vliet, 2000, p. 434)

This should probably be explained after "test adequacy criterion" is defined

8.4.1 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

8.4.2 Testing Stages

- Unit testing: "testing the individual modules [of a program]" (van Vliet, 2000, p. 438); also called "module testing" (Patton, 2006, p. 109) or "component testing" (Peters and Pedrycz, 2000, p. 444), although Baresi and Pezzè (2006, p. 107) say "components differ from classical modules for being re-used in different contexts independently of their development." Note that since a *component* is "a part of a system that can be tested in isolation" (Hamburg and Mogyorodi, 2024), this seems like it could apply to the testing of both modules *and* specific functions
- Integration testing: "testing the composition of modules"; done incrementally using *bottom-up* and/or *top-down* testing (van Vliet, 2000, pp. 438-439), although other paradigms for design, such as *big bang* and *sandwich* exist (Peters and Pedrycz, 2000, p. 489). See also (Patton, 2006, p. 109).
  - Bottom-up testing: uses *test drivers*: "tool[s] that e[] the test environment for a component to be tested" (van Vliet, 2000, p. 410) by "sending test-case data to the modules under test, read[ing] back the results, and verify[ing] that they're correct" (Patton, 2006, p. 109)
  - Top-down testing: uses *test stubs*: tools that "simulate[] the function of a component not yet available" (van Vliet, 2000, p. 410) by providing "fake" values to a given module to be tested (Patton, 2006, p. 110)

  - Big bang testing: the process of "integrat[ing] all modules in a single step and test[ing] the resulting system[]" (Peters and Pedrycz, 2000, p. 489). *Although this is "quite challenging and risky" (Peters and Pedrycz, 2000, p. 489), it may be made less so through the ease of generation, and may be more practical as a testing process for Drasil, although the introduction of the test cases themselves may be introduced, at least initially, in a more structured manner; also of note is its relative ease "to test paths" and "to plan and control" (Peters and Pedrycz, 2000, p. 490)*

Q #1: Bring up!

  - Sandwich testing: "combines the ideas of bottom-up and top-down testing by defining a certain target layer in the hierarchy of the modules" and working towards it from either end using the relevant testing approach (Peters and Pedrycz, 2000, p. 491)

- System testing: "test[ing] the whole system against the user documentation and requirements specification after integration testing has finished" (van Vliet, 2000, p. 439) ((Patton, 2006, p. 109) says this can also be done on "at least a major portion" of the product); often uses random, but representative, input to test reliability (van Vliet, 2000, p. 439)

Expand on reliability testing (make own section?)

- Acceptance testing: Similar to system testing that is "often performed under supervision of the user organization", focusing on usability (van Vliet, 2000, p. 439) and the needs of the customer(s) (Peters and Pedrycz, 2000, p. 492)

- Installation testing: Focuses on the portability of the product, especially "in an environment different from the one in which is has been developed" (van Vliet, 2000, p. 439); not one of the four levels of testing identified by the IEEE standard (Peters and Pedrycz, 2000, p. 445)

8.4.3 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 (MT) (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)

8.4.4 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).

8.5 Dynamic Black-Box (Behavioural) Testing (Patton, 2006, pp. 64-65)

"Error prone" points around boundaries—"the valid data just inside the boundary, … the last possible valid data, and … the invalid data just outside the boundary"(Patton, 2006, p. 73)—should be tested (van Vliet, 2000, p. 430). In this type of testing, the second type of data is called an "ON point", the first type is an "OFF point" for the domain on the *other* side of the boundary, and the third type is an "OFF point" for the domain on the *same* side of the boundary (van Vliet, 2000, p. 430).

Performance Testing

Testing to determine how efficiently software uses resources (including time and capacity) "when accomplishing its designated functions" (Hamburg and Mogyorodi, 2024).

OG ISO 25010?

Originally used a *very* vague definition from (Peters and Pedrycz, 2000, p. 447); re-investigate!

8.5.1 Other Black-Box Testing (Patton, 2006, pp. 87-89)

- Act like an inexperienced user (*likely out of scope*)
- Look for bugs where they've already been found (*keep track of previous failed test cases? This could pair well with Metamorphic Testing (MT)!*)

8.6 Static White-Box Testing (Structural Analysis) (Patton, 2006, pp. 91-104)

8.6.1 Correctness Proofs (van Vliet, 2000, pp. 418-419)

Requires a formal specification (van Vliet, 2000, p. 418) and uses "highly formal methods of logic" (Peters and Pedrycz, 2000, p. 438) to prove the existence of "an equivalence between the program and its specification" (p. 485). It is not often used and its value is "sometimes disputed" (van Vliet, 2000, p. 418). *Could be useful for Drasil down the road if we can specify requirements formally, and may overlap with others' interests in the areas of logic and proof-checking.*

Does symbolic execution belong here? Investigate from textbooks

8.7 Dynamic White-Box (Structural) Testing (Patton, 2006, pp. 105-121)

"Using information you gain from seeing what the code does and how it works to determine what to test, what not to test, and how to approach the testing" (Patton, 2006, p. 106).

8.7.1 Code Coverage (Patton, 2006, pp. 117-121) or Control-Flow Coverage (van Vliet, 2000, pp. 421-424)

"[T]est[ing] the program's states and the program's flow among them" (Patton, 2006, p. 117); allows for redundant and/or missing test cases to be identified (Patton, 2006, p. 118). Coverage-based testing is often based "on the notion of a control graph … [where] nodes denote actions, … (directed) edges connect actions with subsequent actions (in time) … [and a] path is a sequence of nodes connected by edges. The graph may contain cycles … [which] correspond to loops …" (van Vliet, 2000, pp. 420-421). "A cycle is called *simple* if its inner nodes are distinct and do not include [the node at the beginning/end of the cycle]" (van Vliet, 2000, p. 421, emphasis added). If there are multiple actions represented as nodes that

occur one after another, they may be collapsed into a single node (van Vliet, 2000, p. 421).

We discussed that generating infrastructure for reporting coverage may be a worthwhile goal, and that it can be known how to increase certain types of coverage (since we know the structure of the generated code, to some extent, beforehand), but I'm not sure if all of these are feasible/worthwhile to get to 100% (e.g., path coverage (van Vliet, 2000, p. 421)).

- Statement/line coverage: attempting to "execute every statement in the program at least once" (Patton, 2006, p. 119)
  - Weaker than (van Vliet, 2000, p. 421) and "only about 50% as effective as branch coverage" (Peters and Pedrycz, 2000, p. 481)
  - Requires 100% coverage to be effective (Peters and Pedrycz, 2000, p. 481)
  - "[C]an be used at the module level with less than 5000 lines of code"[7] (Peters and Pedrycz, 2000, p. 481)
  - Doesn't guarantee correctness (van Vliet, 2000, p. 421)
- Branch coverage: attempting to, "at each branching node in the control graph, … [choose] all possible branches … at least once" (van Vliet, 2000, p. 421)
  - Weaker than path coverage (van Vliet, 2000, p. 433), although (Patton, 2006, p. 119) says it is "the simplest form of path testing" (*I don't think this is true*)
  - Requires at least 85% coverage to be effective and is "most effective … at the module level" (Peters and Pedrycz, 2000, p. 481)
  - Cyclomatic-number criterion: an adequacy criterion that requires that "all linearly-independent paths are covered" (van Vliet, 2000, p. 423); results in complete branch coverage
  - Doesn't guarantee correctness (van Vliet, 2000, p. 421)
- Path coverage: "[a]ttempting to cover all the paths in the software" (Patton, 2006, p. 119); I always thought the "path" in "path coverage" was a path from program start to program end, but van Vliet seems to use the more general definition (which is, albeit, sometimes valid, like in "du-path") of being any subset of a program's execution (see (van Vliet, 2000, p. 420))
  - The number of paths to test can be bounded based on its structure and can be approached by dividing the system into subgraphs and computing the bounds of each individually (Peters and Pedrycz, 2000, pp. 471-473); this is less feasible if a loop is present (Peters and Pedrycz, 2000, pp. 473-476) since "a loop often results in an infinite number of possible paths" (van Vliet, 2000, p. 421)

OG Miller et al., 1994

OG Miller et al., 1994

OG Miller et al., 1994

OG Miller et al., 1994

Q #2: How do we decide on our definition?

[7] The US Software Engineering Institute has a checklist for determining which types of lines of code are included when counting (Fenton and Pfleeger, 1997, pp. 30-31).

  - van Vliet claims that if this is done completely, it "is equivalent to exhaustively testing the program" (van Vliet, 2000, p. 421); however, this overlooks the effect of inputs on behaviour as pointed out in (Peters and Pedrycz, 2000, pp. 466-467). Exhaustive testing requires both full path coverage *and* every input to be checked
  - Generally "not possible" to achieve completely due to the complexity of loops, branches, and potentially unreachable code (van Vliet, 2000, p. 421); even infeasible paths ("control flow paths that cannot be exercised by any input data" (Washizaki, 2024, p. 5-5)) must be checked for full path coverage to be achieved (Peters and Pedrycz, 2000, p. 439), presenting "a "significant problem in path-based testing" (Washizaki, 2024, p. 5-5)!
  - Usually "limited to a few functions with life criticality features (medical systems, real-time controllers)" (Peters and Pedrycz, 2000, p. 481)

OG Miller et al., 1994

- (Multiple) condition coverage: "takes the extra conditions on the branch statements into account" (e.g., all possible inputs to a Boolean expression) (Patton, 2006, p. 120)
  - "Also known as **extended branch coverage**" (van Vliet, 2000, p. 422)
  - Does not subsume and is not subsumed by path coverage (van Vliet, 2000, p. 433)
  - "May be quite challenging" since "if each subcondition is viewed as a single input, then this … is analogous to exhaustive testing"; however, there is usually a manageable number of subconditions (Peters and Pedrycz, 2000, p. 464)

8.7.2 Data Coverage (Patton, 2006, pp. 114-116)

In addition to Data Flow Coverage, there are also some minor forms of data coverage:

- Sub-boundaries: mentioned previously in notes that were moved to glossary
- Formulas and equations: related to computation errors
- Error forcing: setting variables to specific values to see how errors are handled; any error forced must have a chance of occurring in the real world, even if it is unlikely, and as such, must be double-checked for validity (Patton, 2006, p. 116)

Data Flow Coverage (Patton, 2006, p. 114), (van Vliet, 2000, pp. 424-425)

"[T]racking a piece of data completely through the software" (or a part of it), usually using debugger tools to check the values of variables (Patton, 2006, p. 114).

- "A variable is *defined* in a certain statement if it is assigned a (new) value because of the execution of that statement" (van Vliet, 2000, p. 424)
- "A definition in statement X is *alive* in statement Y if there exists a path from X to Y in which that variable does not get assigned a new value at some intermediate node" (van Vliet, 2000, p. 424)
- A path from a variable's definition to a statement where it is still alive is called **definition-clear** (with respect to this variable) (van Vliet, 2000, p. 424)
- Basic block: "[a] consecutive part[] of code that execute[s] together without any branching" (Peters and Pedrycz, 2000, p. 477)
- Predicate Use (P-use): e.g., the use of a variable in a conditional (van Vliet, 2000, p. 424)
- Computational Use (C-use): e.g., the use of a variable in a computation or I/O statement (van Vliet, 2000, p. 424)
- All-use: either a P-use or a C-use (Peters and Pedrycz, 2000, p. 478)
- DU-path: "a path from a variable definition to [one of] its use[s] that contains no redefinition of the variable" (Peters and Pedrycz, 2000, pp. 478-479)
- The three possible actions on data are defining, killing, and using; "there are a number of anomalies associated with these actions" (Peters and Pedrycz, 2000, pp. 478, 480) (see Data reference errors)

OG Beizer, 1990

Table 8.1 contains different types of data flow coverage criteria, approximately from weakest to strongest, as well as their requirements; all information is adapted from (van Vliet, 2000, pp. 424-425) .

Is this sufficient?

8.7.3 Fault Seeding (van Vliet, 2000, pp. 427-428)

The introduction of faults to estimate the number of undiscovered faults in the system based on the ratio between the number of new faults and the number of introduced faults that were discovered (which will ideally be small) (van Vliet, 2000, p. 427). Makes many assumptions, including "that both real and seeded faults have the same distribution" and requires careful consideration as to which faults are introduced and how (van Vliet, 2000, p. 427).

Q #3: How is All-DU-Paths coverage stronger than All-Uses coverage according to (van Vliet, 2000, p. 433)?

8.7.4 Mutation Testing (van Vliet, 2000, pp. 428-429)

"A (large) number of variants of a program is generated", each differing from the original "slightly" (e.g., by deleting a statement or replacing an operator with another) (van Vliet, 2000, p. 428). These *mutants* are then tested; if set of tests fails to expose a difference in behaviour between the original and many mutants, "then that test set is of low quality" (van Vliet, 2000, pp. 428-429). The goal

Table 8.1: Types of Data Flow Coverage

Criteria	Requirements
All-defs coverage	Each definition to be used at least once
All-P-uses coverage	A definition-clear path from each definition to each P-use
All-P-uses/Some-C-uses coverage	Same as All-P-uses coverage, but if a definition is only used in computations, at least one definition-clear path to a C-use must be included
All-C-uses/Some-P-uses coverage	A definition-clear path from each definition to each C-use; if a definition is only used in predicates, at least one definition-clear path to a P-use must be included
All-Uses coverage	A definition-clear path between each variable definition to each of its uses and each of these uses' successors
All-DU-Paths coverage	Same as All-Uses coverage, but each path must be cycle-free or a simple cycle

is to maximize the number of mutants identified by a given test set (van Vliet, 2000, p. 429). **Strong mutation testing** works at the program level while **weak mutation testing** works at the component level (and "is often easier to establish") (van Vliet, 2000, p. 429).

OG KA85

There is an unexpected byproduct of this form of testing. In some cases of one experiment , "the original program failed, while the modified program [mutant] yielded the right result" (van Vliet, 2000, p. 432)! In addition to revealing shortcomings of a test set, mutation testing can also point the developer(s) in the direction of a better solution!

8.8 Gray-Box Testing (Patton, 2006, pp. 218-220)

A type of testing where "you still test the software as a black-box, but you supplement the work by taking a peek (not a full look, as in white-box testing) at what makes the software work" (Patton, 2006, p. 218). An example of this is looking at HTML code and checking the tags used since "HTML doesn't execute or run, it just determines how text and graphics appear onscreen" (Patton, 2006, p. 220).

8.9 Regression Testing

Repeating "tests previously executed … at a later point in development and maintenance" (Peters and Pedrycz, 2000, p. 446) "to make sure there are no unwanted changes [to the software's behaviour]" (p. 481) (although allowing "some unwanted

differences to pass through" is sometimes desired, if tedious (p. 482)). See also (Patton, 2006, p. 232).

- Should be done automatically (Peters and Pedrycz, 2000, p. 481); "[t]est 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!

- Its "effectiveness … is expressed in terms of":
  1. difficulty of test suite construction and maintenance
  2. reliability of the testing system (Peters and Pedrycz, 2000, pp. 481-482)
- Various levels:
  - Retest-all: "all tests are rerun"; "this may consume a lot of time and effort" (van Vliet, 2000, p. 411) (*shouldn't take too much effort, since it will be automated, but may lead to longer CI runtimes depending on the scope of generated tests*)
  - Selective retest: "only some of the tests are rerun" after being selected by a *regression test selection technique*; "[v]arious strategies have been proposed for doing so; few of them have been implemented yet" (van Vliet, 2000, p. 411)

Investigate these

8.10 Metamorphic Testing (MT)

The use of Metamorphic Relations (MRs) "to determine whether a test case has passed or failed" (Kanewala and Yueh Chen, 2019, p. 67). "A[n] MR specifies how the output of the program is expected to change when a specified change is made to the input" (Kanewala and Yueh Chen, 2019, p. 67); this is commonly done by creating an initial test case, then transforming it into a new one by applying the MR (both the initial and the resultant test cases are executed and should both pass) (Kanewala and Yueh Chen, 2019, p. 68). "MT is one of the most appropriate and cost-effective testing techniques for scientists and engineers" (Kanewala and Yueh Chen, 2019, p. 72).

8.10.1 Benefits of MT

- Easier for domain experts; not only do they understand the domain (and its relevant MRs) (Kanewala and Yueh Chen, 2019, p. 70), they also may not have an understanding of testing principles (Kanewala and Yueh Chen, 2019, p. 69). *This majorly overlaps with Drasil!*
- Easy to implement via scripts (Kanewala and Yueh Chen, 2019, p. 69). *Again, Drasil*

- Helps negate the test oracle (Kanewala and Yueh Chen, 2019, p. 69) and output validation (Kanewala and Yueh Chen, 2019, p. 70) problems from Roadblocks to Testing Scientific Software (*i.e., the two that are relevant for Drasil*)
- Can extend a limited number of test cases (e.g., from an experiment that was only able to be conducted a few times) (Kanewala and Yueh Chen, 2019, pp. 70-72)
- Domain experts are sometimes unable to identify faults in a program based on its output (Kanewala and Yueh Chen, 2019, p. 71)

8.10.2 Examples of MT

- 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)

8.11 Roadblocks to Testing

- Intractability: it is generally impossible to test a program exhaustively (Washizaki, 2024, p. 5-5; ISO/IEC and IEEE, 2022, p. 4; van Vliet, 2000, p. 421; Peters and Pedrycz, 2000, pp. 439, 461)

- 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?

8.11.1 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 9

Extras

Writing Directives

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

9.1 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 9.1: 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.

9.2 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 9.2: Pseudocode: exPHref

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

9.3 Pseudocode Code Snippets

For pseudocode, you can also use the pseudocode environment, such as that used in Source Code A.8.

9.4 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 #4: 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, 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.

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.

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. ISTQB Glossary, v4.3, 2024. URL https://glossary.istqb.org/en_US/search.

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.

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.

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 –Software testing –Part 1: General concepts. *ISO/IEC/IEEE 29119-1:2013*, September 2013. doi: 10.1109/IEEESTD.2013.6588537.

ISO/IEC and IEEE. ISO/IEC/IEEE International Standard - Systems and software engineering–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.

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.

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.

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.

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 2020a. ISBN 978-1-71381-853-3.

Guido Tebes, Luis Olsina, Denis Peppino, and Pablo Becker. TestTDO_terms_definitions_vfinal.pdf, February 2020b. URL https://drive.google.com/file/d/19TWHd50HF04K6PPyVixQzR6c7HjW2kED/view.

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.

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

Source Code A.3: Code for determining a source's category based on its citation.

```
# rel == True if the SrcCat is used for coloring relations
def getSrcCat(s, rel: bool = False) -> SrcCat:
    if any(std in s for std in {"IEEE", "ISO", "IEC"}):
        return SrcCat.STD
    if any(metastd in s for metastd in
        {"Washizaki", "Bourque and Fairley", "SWEBOK",
            "Hamburg and Mogyorodi", "ISTQB", "Firesmith",
            "Doğan et al", "DoğanEtAl"}):
        return SrcCat.META
    if any(textbook in s for textbook in
        {"van Vliet", "vanVliet", "Patton", "Peters and Pedrycz",
            "PetersAndPedrycz", "Gerrard and Thompson",
            "GerrardAndThompson", "Dennis et al", "DennisEtAl",
            "Perry", "Ammann and Offutt", "AmmannAndOffutt",
            "Fenton and Pfleeger", "FentonAndPfleeger",
            "Kaner et al", "KanerEtAl"}):
        return SrcCat.TEXT
    return SrcCat.INFER if rel and not any(par in s for par in
     ↪  "()") else SrcCat.PAPER
```

Source Code A.4: Tests for main with an invalid input file

```
  # from
   ↪  https://stackoverflow.com/questions/54071312/how-to-pass-command-line-arg
  ## \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 A.5: Projectile's choice for constraint violation behaviour in code

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

Source Code A.6: 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 A.7: "MultiDefinitions" (MultiDefn) Definition

```haskell
-- | '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 A.8: Pseudocode: Broken QuantityDict Chunk Retriever

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