Todo list

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, 61

Lay Abstract

Important: Lay abstract.

Abstract

Important: Abstract.

Acknowledgements

Important: Acknowledgements.

Contents

List of Figures

List of Tables

List of Source Codes

List of Abbreviations and Symbols

DAC	Differential Assertion Checking
GOOL	Generic Object-Oriented Language
HREF	Hypertext REFerence
IDE	Integrated Development Environment
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
C-use	Computational Use
DblPend	Double Pendulum
GamePhysics	Game Physics
P-use	Predicate Use
Projectile	Projectile
SglPend	Single Pendulum
SSP	Slope Stability analysis Program
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

Congratulations! If you're seeing this, it means you've managed to compile the PDF, which also means you can get started on typesetting your thesis[1].

This template is adapted from my thesis. If you'd like to see an example of this template in practice, please feel free to use my thesis as an example.

1.1 Template Organization

I've broken up the template according to my preferred organization: chapters in separate files, various kinds of assets (images, tables, code snippets, macros, etc.) in separate files, etc. The split is approximately according to Table 1.1.

Table 1.1: Template Organization

File/Folder	Intended Usage & Description
`thesis.tex`	Focal LaTeX file that collects everything and is used to build your thesis/report document.
`Makefile`	A basic `Makefile` configuration. See `make help` for a list of helpful commands.
`build/`	When you build your PDF, this folder is used as the working directory of LuaLaTeX. Using this allows us to quickly get rid of LaTeX build files that can cause problems when we re-build documents.
`manifest.tex`	Basic options that you should certainly configure according to your needs.
`chapters.tex`	All chapters of your thesis should be included here.

[1] Or report or ...

`chapters/`	Enumeration of the chapters of your thesis. I prefer using a two-digit indexing pattern for the prefix of file names so that I can quickly open up by chapter number using VS Codium.
`assets.tex`	Enumeration of the various kinds of "assets" in the `assets/` folder. See the file for examples on how you can write your extra utility macros.
`assets/`	Enumeration of various kinds of "assets," with subdirectories for images and figures, tables, and code snippets.
`front.tex`	All front matter of your thesis should be included here.
`front/`	Enumeration of the front chapters of your thesis. These chapters should all be numbered using Roman numerals.
`back.tex`	All back matter of your thesis should be included here.
`back/`	Enumeration of the back matter content.
`acronyms.tex`	List of acronyms you intend to use in your thesis. This uses the "acro" LaTeX package.
`macros.tex`	Helpful macros!
`unicode_chars.tex`	At times, you might find issues with unicode characters, especially in verbatim environments, where you might need to manually define them using other font glyphs.
`mcmaster_colours.tex`	Macros for the McMaster colour palette.
`README.md`	Read it!
`.gitignore`	List of files in the working directory that should be ignored by git.
`latexmkrc`	Used for setting the timezone for latexmk, but can be used for other options.

1.2 Writing Tips

When drafting chapters, I:

1. wrote "writing directives" for each chapter to understand what I need to write about (see Section 2.1),

2. wrote "todo" notes for tedious things that I might want to do later (such as citations, figures, code snippets, etc., see Section 2.4), and

3. regularly built my thesis using `make debug` to make sure that whatever I wrote didn't break the LaTeX code.

For workflow recommendations, you should speak with your supervisor as they might prefer you work in a specific way with them.

1.3 Development Recommendations

Other than the basic tools I used for this template, I enjoyed using the following tools while writing my thesis:

1. VS Codium/VS Code[2] with the following extensions:

   (a) LaTeX Workshop, for LaTeX syntax highlighting, code formatting (this is highly recommended), and code completion,

   (b) LTeX - LanguageTool grammar/spell checking, for grammar checking using LangaugeTool, and

   (c) Todo Tree, for quickly listing all of my TODO notes in my IDE (in addition to the list at the top of the PDF).

2. texcount (which should come with your LaTeX installation) to quickly check the word count of individual LaTeX files, and

3. Zotero for collecting my references and quickly exporting bib entries that I could use.

In particular, when writing, I found it particularly helpful to use VS Code's "Zen Mode" (to see your keybind, press `CTRL+ALT+P` and search for "Zen"), which enters a stripped-down full-screen version of the current working file, keeping your eyes purely focused on the document in front of you. Being comfortable with the keybinds is particularly helpful for working effectively in this setup. For example, I found the following[3] to be helpful: `CTRL+TAB` and `CTRL+SHIFT+TAB` to scroll between open files, `CTRL+P` to quickly open up recent files, `CTRL+ALT+P` to run commands you forgot the keybind for, `CTRL+O` to open up files out of the current working directory.

While writing, I enjoyed:

1. using "TODO" notes

   Easy: Such as this one, but check out Section 2.4 for more options.

   to collect notes that I would want to do later,

[2] I prefer VS Codium simply because I prefer libre software.

[3] If you're not using Linux, I cannot guarantee that these will be the same for you, so you should use `CTRL+ALT+P` to look for your appropriate bound keybinds.

2. formatting the LaTeX code to make it easier to read (the LaTeX Workshop plugin has functionality for this),
3. breaking the non-textual content into separate files and "include"-ing them in the LaTeX code so that they didn't cause large visual interruptions,
4. using git to version control copies of my thesis, chapters, etc.,
5. using TikZ and draw.io/diagrams.net to build graphics and diagrams, and
6. building the thesis often using `make debug` to quickly debug issues in the written code.

1.4 Troubleshooting

"StackOverflow" is a great area to look for solutions to common LaTeX issues. Otherwise, feel free to use create a ticket or sending an email to me.

Chapter 2

Extras

Writing Directives

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

2.1 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 2.1: 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.

2.2 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 2.2: Pseudocode: exPHref

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

2.3 Pseudocode Code Snippets

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

2.4 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 #1: Questions I might have?

Questions:

Chapter 3

Software Testing 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

3.1 Scope

This project is focused on the generation of test cases for code, so only the "testing" component of Verification and Validation (V&V) is considered (see #22). For example, design reviews (see ISO/IEC and IEEE, 2017, p. 132) and documentation reviews (see ISO/IEC and IEEE, 2017, p. 144) are out of scope, since they focus on the V&V of the design and documentation of the code, respectively, and not on the code itself. Likewise, ergonomics testing and proximity-based testing (see Hamburg and Mogyorodi, 2024) are out of scope since they are for testing hardware systems, and security audits are out of scope (see Hamburg and Mogyorodi, 2024) since they focus on "an organization's … processes and infrastructure". **NOTE:** 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., ML model testing seems domain-specific).

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).

This also means that only some aspects of some testing approaches are relevant. This mainly manifests as a testing approach that can verify both the V&V itself and 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 detec-

tion and removal", which is a part of V&V of the V&V itself, "and estimat[e] the number of faults remaining" (ISO/IEC and IEEE, 2017, p. 165), which helps verify the actual code.

2. *Fault injection testing*, where "faults are artificially introduced into the SUT", 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, 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.

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. 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).

4. 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.

5. Humorously, "loop testing" is in scope when referring to loops in software (Patton, 2006; van Vliet, 2000; Peters and Pedrycz, 2000; Dhok and Ramanathan, 2016; Godefroid and Luchaup, 2011), not loops in chemical systems (Dominguez-Pumar et al., 2020), wide-area damping controllers (Pierre et al., 2017; Trudnowski et al., 2017) or copper loops (Goralski, 1999)!

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[1] as well" (Washizaki, 2024, p. 5-2) (such as (Gerrard, 2000, pp. 8-9) and even (ISO/IEC and IEEE, 2017, p. 440)!), the scope of "testing" for the purpose of this project originally included both "static testing" and "dynamic testing", as done by ISO/IEC and IEEE (2022, p. 17). However, static testing tends to be less systematic/consistent and often requires human intervention, which makes it less relevant to this project's end goal: to generate test cases automatically. However, understanding the breadth of testing approaches provides a more complete picture of how software can be tested, how the various approaches are related to one another, and potentially how even parts of these "out-of-scope" approaches may be generated in the future! These "out-of-scope" approaches will be identified more systematically, but gathering information about them is an important precursor, making them within the scope of this research, although they will be excluded at a later phase. Even some dynamic methods, such as demonstrations

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

and dynamic analysis, which fall under the realm of "evaluation" as opposed to "testing" (Firesmith, 2015, p. 13) may be out of scope, due to their reliance on human intervention.

3.1.1 Derived Test Approaches

One group of test approaches given in IEEE Testing Terminology is "test types", which can be derived from software qualities: "capabilit[ies] of software product[s] to satisfy stated and implied needs when used under specified conditions" (ISO/IEC and IEEE, 2017, p. 424) . This is supported by Fenton and Pfleeger who say that reliability and performance testing are based on their underlying qualities (1997, p. 18) and that 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) (p. 36) where attributes must be defined before they can be measured (p. 38).

OG ISO/IEC 2014

After discussing this further (see #21 and #23), it was decided that tracking software qualities, in addition to testing approaches, would be worthwhile (see #27). This was done by capturing their definitions and any rationale for why it might be useful to consider an explicitly separate "test type" in a separate document, so this information could be captured without introducing clutter.

Similarly, since some types of requirements have associated types of testing (e.g., functional, non-functional, security), it was discussed whether each requirement type implies a related testing approach (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):

- **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)

Some test approaches seem to just be combinations of other (seemingly orthogonal) approaches, such as the following (all from (Hamburg and Mogyorodi, 2024) unless otherwise indicated):

- Checklist-based reviews
- Endurance stability testing (Firesmith, 2015, p. 55)
- End-to-end functionality testing (Gerrard, 2000, Tab. 2)
- Formal reviews
- Informal reviews
- Infrastructure compatibility testing (Firesmith, 2015, p. 53)

- Legacy system integration (testing) (Gerrard, 2000, Tab. 2)
- Offline MBT
- Online MBT
- Role-based reviews
- Scenario walkthroughs (Gerrard, 2000, Fig. 4)
- Scenario-based reviews
- Security attacks
- Usability test script(ing)

3.2 Methodology

This process initially involved looking through textbooks that were trusted at McMaster (Patton, 2006; Peters and Pedrycz, 2000; van Vliet, 2000). However, this process was somewhat ad hoc and arbitrary, meaning it wouldn't be as systematic as required. Going forward, this process will be more rigorous, starting from more established sources of software testing terminology in approximately the following order: (ISO/IEC and IEEE, 2022; Washizaki, 2024; Bourque and Fairley, 2014; ISO/IEC and IEEE, 2017, 2013; ISO/IEC, 2023b; IEEE, 2012; ISO/IEC, 2023a; Hamburg and Mogyorodi, 2024; Firesmith, 2015). Other sources that focused on cataloging and categorizing testing terminology, such as Kuļešovs et al. (2013), were also examined to see how well they agreed with the "standard" terminology.

I went through these resources by going through them looking for relevant terminology, taking special care with glossaries and lists of terms. Of particular note were terms that included "test(ing)", "validation", "verification", "review", "audit", or terms that had come up before as part of already-discovered testing approaches, such as "performance", "recovery", "component", "bottom-up", "boundary", and "configuration". If a term's definition had already been recorded, either the "new" one replaced it if the "old" one wasn't as clear/concise or parts of both were merged to paint a more complete picture. If any discrepancies or ambiguities arose, they were investigated to a reasonable extent and documented. If a testing approach was mentioned but not defined, it was still added to the glossary to indicate it should be investigated further. A similar methodology was used for tracking software qualities, albeit in a separate document (see Derived Test Approaches).

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)

However, over the course of this research, our scope was adjusted to include some terms for our initial list of test approaches to be filtered out later, such as types of attacks (see #55), meaning that some entries were missed during the first pass(es) of these resources. While reiterating over these resources would be ideal, this may not be possible due to time constraints.

Different sources categorized software testing approaches in different ways; while it is useful to record and think about these categorizations (see Categorizations), 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); more careful thought should be given to which are "best", and this should happen during the analysis stage.

3.3 Observations

3.3.1 Categories of Testing Approaches

For classifying different kinds of tests, ISO/IEC and IEEE (2022) provide some terminology (see Table 3.1). 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 Scope, #21, and #22).

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[1]) 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); "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[2] (sometimes "Phase"[3] or "Stage"[4])	A stage of testing "typically associated with the achievement of particular objectives and used to treat particular risks" (ISO/IEC and IEEE, 2022, p. 12) with "its 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; 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; 2017, p. 473; OG IEEE 2013; similar in Hamburg and Mogyorodi, 2024)	security testing, usability testing, performance testing (ISO/IEC and IEEE, 2022, p. 15; 2017, p. 473)

[1] "Design technique" is sometimes abbreviated to "technique" (ISO/IEC and IEEE, 2022, p. 11; Hamburg and Mogyorodi, 2024).

[2] "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).

[3] "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).

[4] Washizaki, 2024, pp. 5-6 to 5-7; Hamburg and Mogyorodi, 2024; Gerrard, 2000, 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, 2000, p. 13); see "Level" in IEEE Testing Terminology	desktop development testing, infrastructure testing, system testing, large scale integration, and post-deployment monitoring (Gerrard, 2000, 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

3.3.2 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) so, for example, "one category [of test techniques] might deal with combining two or more techniques" (Washizaki, 2024, p. 5-10). 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 gray-box (functional, structural, or a mix of the two) (Washizaki, 2024, pp. 5-10, 5-16; Ammann and Offutt, 2017, pp. 57-58; Kuļešovs et al., 2013, p. 213; Patton, 2006, pp. 53, 218; Perry, 2006, p. 69)
- Level/stage[2] of testing: unit, integration, system, or acceptance (Washizaki, 2024, pp. 5-6 to 5-7; Hamburg and Mogyorodi, 2024; Kuļešovs et al., 2013, p. 218 ; Patton, 2006; Perry, 2006; Peters and Pedrycz, 2000; Gerrard, 2000, pp. 9, 13) (sometimes includes installation (van Vliet, 2000, p. 439) or regression (Barbosa et al., 2006, p. 3))

OG Black, 2009

- Key aspect: specification, structure, or experience (Washizaki, 2024, p. 5-10)

Originally in ISO/IEC/IEEE 29119-4:2021

- 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"[3] (Firesmith, 2015, p. 17)
- Execution of code: static or dynamic (Kuļešovs et al., 2013, p. 214; Gerrard, 2000, p. 12; Patton, 2006, p. 53)
- Goal of testing: verification or validation (Kuļešovs et al., 2013, p. 214; Perry, 2006, pp. 69-70)

[2] See IEEE Testing Terminology.

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

- Property of code: functional or non-functional (Kuļešovs et al., 2013, p. 213)
- Human involvement: manual or automated (Kuļešovs et al., 2013, p. 214)
- Structuredness: scripted or exploratory (Kuļešovs et al., 2013, p. 214)
- Source of test data: specification-, implementation-, or error-oriented (Peters and Pedrycz, 2000, p. 440)
- Adequacy criterion: coverage-, fault-, or error-based ("based on knowledge of the typical errors that people make") (van Vliet, 2000, pp. 398-399)
- Priority[4]: smoke, usability, performance, or functionality testing (Gerrard, 2000, p. 12)
- Category of test "type"[5]: static testing, test browsing, functional testing, non-functional testing, or large scale integration (testing) (Gerrard, 2000, p. 12)
- Purpose: correctness, performance, reliability, or security (Pan, 1999)

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 [6] 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)

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

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

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

3.3.3 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?

is this punctuation right?

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 "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[7] 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

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

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).

3.3.4 Discrepancies and Ambiguities

ISO/IEC and IEEE (2022) mentions the following 44 test approaches without defining them. This means that out of the 99 identified test approaches, almost 45% had no associated definition! However, the previous version of this standard, (ISO/IEC and IEEE, 2013), generally explained two, provided references for three, and explicitly defined three of these terms, for a total of eight definitions that could (should) have been included in (ISO/IEC and IEEE, 2022)! These are marked with underline, *italics*, and **bold**, respectively. Additionally, entries marked with an asterisk* were defined (at least partially) in (ISO/IEC and IEEE, 2017), which would have been available when creating (ISO/IEC and IEEE, 2022). These terms bring the total count of terms that could (should) have been defined in (ISO/IEC and IEEE, 2022) to eighteen; almost 20% of undefined test approaches could have been defined!

- Acceptance Testing*
- All Combinations Testing
- All-C-Uses Testing (Data Definition C-Use Pair*)
- All-Definitions Testing
- All-DU-Paths Testing (Data Definition-Use Path*)
- All-P-Uses Testing (Data Definition P-Use Pair*)
- All-Uses Testing (Data Definition-Use Pair*)
- Alpha Testing*
- Base Choice Testing (also mentioned but not defined in (ISO/IEC and IEEE, 2017))
- Beta Testing*
- Branch Condition Combination Testing
- Branch Condition Testing
- **Capacity Testing***
- Capture-Replay Driven Testing
- Cause-Effect Testing

- Classification Tree Method (also mentioned but not defined in (ISO/IEC and IEEE, 2013))
- Conversion Testing
- *Data Flow Testing** (ISO/IEC/IEEE 29119-4)
- Data-driven Testing
- Disaster/Recovery Testing (Disaster Recovery*)
- Each Choice Testing
- Factory Acceptance Testing
- Fault Injection Testing
- Functional Suitability Testing (also mentioned but not defined in (ISO/IEC and IEEE, 2017))
- **Installability Testing***
- Integration Testing*
- Localization Testing
- Model Verification
- Negative Testing
- Operational Acceptance Testing
- Performance-related Testing (although Performance Testing is defined in (ISO/IEC and IEEE, 2022); see the "performance testing" ambiguity below)
- Production Verification Testing
- Recovery Testing (**Backup and Recovery Testing***, Recovery*)
- Response-Time Testing
- *Reviews* (ISO/IEC 20246) (Code Reviews*)
- Scalability Testing
- Statistical Testing
- Syntax 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!)
- Usability Testing* (also mentioned but not defined in (ISO/IEC and IEEE, 2013); in (ISO/IEC and IEEE, 2017), "usability test" is defined with a note to compare it to "usability testing" (p. 493), but no corresponding entry is present)
- Use Case Testing (also mentioned but not defined in (ISO/IEC and IEEE, 2013))
- User Acceptance Testing

Additionally, discrepancies and ambiguities exist both among sources and within individual ones; these may be areas for further investigation:

1. "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)!
2. Experience-based testing is categorized as both a test design technique and a test practice on the same page (ISO/IEC and IEEE, 2022, p. 22)! This also causes confusion about its children, such as error guessing and exploratory testing; ISO/IEC and IEEE (2022, p. 34) say error guessing is an "experience-based test design technique" and "experience-based test practices include …exploratory testing, tours, attacks, and checklist-based testing". There are several instances of inconsistencies between parent and child test approach categorizations (which may indicate they aren't necessarily the same, or that more thought must be given to classification/organization).
3. "Fuzz testing" is "tagged" (?) as "artificial intelligence" (ISO/IEC and IEEE, 2022, p. 5), although I don't think this is a set-in-stone requirement.
4. "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 (2006, p. 86) says the loads should as large as possible.

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

6. 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? Likewise, it is listed as a child of integration testing by Hamburg and Mogyorodi, 2024.

7. "Disaster/recovery testing" and "recovery testing" (as a subset of performance-related testing) are both listed as test types (ISO/IEC and IEEE, 2022, p. 22) but not defined, making it unclear what distinguishes them. ISO/IEC and IEEE (2013, p. 2) define "backup and recovery testing" as testing "that 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", which may be what is meant by "recovery testing" in the context of performance-related testing. Meanwhile, SWEBOK V4 defines "recovery testing" as the testing of "software restart capabilities after a system crash or other disasters [sic]" (Washizaki, 2024, p. 5-9), which may be what is meant *outside* of the context of performance.

8. Similarly, "branch condition testing" and "branch condition combination testing" are both listed as subsets of structure-based testing (ISO/IEC and IEEE, 2022, p. 22) but are not defined, making it unclear what distinguishes them.

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

10. 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) considers regression testing to be a test level.

11. 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.

12. SWEBOK V4 says "scalability testing evaluates the capability to use and learn the system and the user documentation. It also focuses on the system's effectiveness in supporting user tasks and the ability to recover from user errors" (Washizaki, 2024, p. 5-9). This description seems to describe "usability testing" instead, despite earlier defining/describing "scalability" as …

    (a) "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)

    (b) "connected to the complexity of the platform and environment in which the program runs, such as distributed, wireless networks and virtualized environments, large-scale clusters, and mobile clouds" (Washizaki, 2024, p. 5-5)

    Other definitions of "scalability" support these definitions, so the definition of "scalability testing" follows trivially from there (sometimes explicitly (Hamburg and Mogyorodi, 2024)):

    - The "capability of a product to handle growing or shrinking workloads or to adapt its capacity to handle variability" (ISO/IEC, 2023a)
    - "The degree to which a component or system can be adjusted for changing" (Hamburg and Mogyorodi, 2024)

13. SWEBOK V4 says that one objective of elasticity testing is "to evaluate scalability" (Washizaki, 2024, p. 5-9), which seems like an objective of scalability testing instead.

14. SWEBOK 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 with ISO/IEC and IEEE's definition of "security testing", which is "conducted to evaluate the degree to which a test item, and associated data and information, are protected so that" only "authorized persons or systems" can use them as intended (2022), both in scope and name.

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

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

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

18. 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).

19. 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).

20. Sometimes "condition testing" and "decision testing" are treated as synonyms (Washizaki, 2024, p. 5-13), but sometimes they are kept separate, with the former only involving atomic conditions (Hamburg and Mogyorodi, 2024), implying that it may be a sub-approach of the latter?

21. "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 are at best ambiguous and at worst incorrect.

22. 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*.

23. 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".

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

25. 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).

26. The definition of "scalability" given by Hamburg and Mogyorodi, 2024 is "the degree to which a component or system can be adjusted for changing *capacity*" (emphasis added), but the source that it references says that scalability is "the measure of a system's ability to be upgraded to accommodate increased *loads*" (Gerrard and Thompson, 2002, p. 381, emphasis added); this is accomplished *through* increasing capacity, not in reaction to it (p. 192).

27. 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?

28. The source cited for the definition of "test type" from Hamburg and Mogyorodi, 2024 does not seem to provide a definition itself.

29. The same is true for "visual testing".

30. The same is true for "security attack".

31. 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).

32. "Scenario testing" and "use case testing" are given as synonyms by Hamburg and Mogyorodi, 2024, but listed separately by ISO/IEC and IEEE (2022, p. 22) (although the latter is not defined).

33. 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).

34. "System testing" is listed as a subtype of "system testing" by Firesmith (2015, p. 23).

35. "Hardware-" and "human-in-the-loop testing" have the same acronym: "HIL" (Firesmith, 2015, p. 23).

36. The same is true for "customer" and "contract(ual) acceptance testing" ("CAT") (Firesmith, 2015, p. 30).

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

38. 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.

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

40. "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 are defined.

41. Firesmith makes a distinction between organization- and role-based testing (2015, pp. 17, 37, 39), which seems arbitrary, but further investigation may prove it to be meaningful (see #59).

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

43. While Firesmith says that fault tolerance testing is a subtype[8] of robustness testing (2015, p. 56) (which is distinct from reliability testing (p. 53)), other sources say it is a subtype of reliability testing (ISO/IEC and IEEE, 2017, p. 375; Washizaki, 2024, p. 7-10) or a synonym of "robustness testing" (Hamburg and Mogyorodi, 2024).

44. 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, 2000, p. 11) are unclear and seem miniscule.

45. 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.

46. Availability testing isn't assigned to a test priority (Gerrard, 2000, 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.

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

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

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

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

49. The phrase "continuous automated testing" (Gerrard, 2000, 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).

50. Gerrard very helpfully indicates that performance testing is performance testing and that usability testing is usability testing; interestingly, performance testing is *not* given as a sub-category of usability testing (2000, Tab. 2).

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

Functional Testing

Throughout the literature, "functional testing" seems to be described in many ways, alongside other, potentially related, terms:

- **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), which 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). However, they are sometimes defined as separate terms: "specification-based testing" as "testing based on an analysis of the specification of the component or system" (including "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). This definition of "functional testing" 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, 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 given "selected inputs and execution conditions" (ISO/IEC and IEEE, 2017, p. 196)).

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

- **Correctness Testing:** Washizaki (2024, p. 5-7) says "test cases can be designed to check that the functional specifications are correctly implemented, which is variously referred to in the literature as conformance testing, correctness testing or functional testing"; this mirrors previous definitions of "functional testing" (ISO/IEC and IEEE, 2022, p. 21; 2017, p. 196) but groups it with "correctness testing". Since "correctness" is a software quality (ISO/IEC and IEEE, 2017, p. 104; Washizaki, 2024, p. 3-13) which is

what defines a "test type" (ISO/IEC and IEEE, 2022, p. 15), it seems consistent to label "functional testing" as a "test type" (ISO/IEC and IEEE, 2022, pp. 15, 20, 22). This is listed separately from "functionality testing" by Firesmith (2015, p. 53).

- **Conformance Testing:** As mentioned above, Washizaki (2024, p. 5-7) says testing "that the functional specifications are correctly implemented" can be called "conformance testing" or "functional testing". The definition of "conformance testing" is later given 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 the testing mentioned earlier; the former only lists "specifications" while the latter also includes "standards", "rules", "requirements", "design", "processes", and "practices"!

- **Functional Suitability Testing:** Procedure testing is called a "type of functional suitability testing" (ISO/IEC and IEEE, 2022, p. 7), but no definition of "functional suitability testing" 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.

- **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 its "parent". 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, 2000, Tab. 2). This is listed separately from "correctness testing" by Firesmith (2015, p. 53).

Operational (Acceptance) Testing

Some sources refer to "operational acceptance testing" (ISO/IEC and IEEE, 2022, p. 22; Hamburg and Mogyorodi, 2024) while some refer to "operational test-

ing" (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 given 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).

Performance Testing

"Performance testing" is defined as testing "conducted to evaluate the degree to which a test item accomplishes its designated functions within given constraints of time and other resources" (ISO/IEC and IEEE, 2022, p. 7). It is listed as a subset of "performance-related testing", along with other approaches like load and capacity testing (p. 22); Washizaki (2024, p. 5-9) gives "capacity and response time" as examples of "performance characteristics" that performance testing would seek to "assess". I see two possible resolutions to this:

1. Assign the definition of "performance testing" to "performance-related testing" and give "performance testing" a more specific definition.

2. Replace the term "performance-related testing" with "performance testing"; this seems more logical, since I haven't found a definition of "performance-related testing" (at least yet) and most (if not all) definitions of "performance testing" seem to treat it as a category of tests.

Similarly, "performance" and "performance efficiency" are both given as software qualities by ISO/IEC and IEEE, with the latter defined as the "performance relative to the amount of resources used under stated conditions" (2017, p. 319) or the "capability of a product to perform its functions within specified time and throughput parameters and be efficient in the use of resources under specified conditions" (ISO/IEC, 2023a). Initially, there didn't seem to be any meaningful distinction between the two, although the term "performance testing" is defined (2017, p. 320) and used by ISO/IEC and IEEE and the term "performance efficiency testing" is *also* used by ISO/IEC and IEEE (but not defined explicitly). Further discussion (see #43) brought us to the conclusion that "performance efficiency testing" is a subset of "performance testing", and the difference of "relative to the amount of resources used" or "be efficient in the use of resources" between the two is meaningful.

3.4 Definitions

- Software testing: "the process of executing a program with the intent of finding errors" (Peters and Pedrycz, 2000, p. 438) . "Testing can reveal failures, but the faults causing them are what can and must be removed" (Washizaki, 2024, p. 5-3); it can also include certification, quality assurance, and quality improvement (Washizaki, 2024, p. 5-4). Involves "specific preconditions [and] …stimuli so that its actual behavior can be compared with its expected or required behavior", including control flows, data flows, and postconditions (Firesmith, 2015, p. 11)

OG Myers 1976

- 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)

3.4.1 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)

3.5 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)

- Since "there is no uniform best test technique", it is advised to use many techniques when testing (van Vliet, 2000, 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

3.5.1 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

3.5.2 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 generate[] 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)*
  - 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)

Q #2: Bring up!

- 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)

3.5.3 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). SWEBOK 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)

3.5.4 Generating Test Cases

- "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)

- 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)

3.6 Static Black-Box (Specification) Testing (Patton, 2006, pp. 56-62)

Describe anyway

Most of this section is irrelevant to generating test cases, as they require human involvement (e.g., Pretend to Be the Customer (Patton, 2006, pp. 57-58), Research Existing Standards and Guidelines (Patton, 2006, pp. 58-59)). However, it provides a "Specification Terminology Checklist" (Patton, 2006, p. 61) that includes some keywords that, if found, could trigger an applicable warning to the user (similar to the idea behind the correctness/consistency checks project). In general, each requirement should be unambiguous, testable, binding, and "acceptable to all stakeholders", and the "overall collection" should be complete, consistent, and feasible (Washizaki, 2024, p. 1-8):

- **Potentially unrealistic:** always, every, all, none, every, certainly, therefore, clearly, obviously, evidently
- **Potentially vague:** some, sometimes, often, usually, ordinarily, customarily, most, mostly, good, high-quality, fast, quickly, cheap, inexpensive, efficient, small, stable
- **Potentially incomplete:** etc., and so forth, and so on, such as, handled, processed, rejected, skipped, eliminated, if …then …(without "else" or "otherwise"), to be determined (van Vliet, 2000, p. 408)

3.6.1 Coverage-Based Testing of Specification (van Vliet, 2000, pp. 425-426)

Requirements can be "depicted as a graph, where the nodes denote elementary requirements and the edges denote relations between [them]" from which test cases can be derived (van Vliet, 2000, p. 425). However, it can be difficult to assess whether a set of equivalence classes are truly equivalent, since the specific data available in each node is not apparent (van Vliet, 2000, p. 426).

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

This is the process of "entering inputs, receiving outputs, and checking the results" (Patton, 2006, p. 64). (van Vliet, 2000, p. 399) also calls this "functional testing".

Requirements

- Requirements documentation (definition of what the software does) (Patton, 2006, p. 64); relevant information could be:
  - Requirements: Input-Values and Output-Values
  - Input/output data constraints

3.7.1 Exploratory Testing (Patton, 2006, p. 65)

An alternative to dynamic black-box testing when a specification is not available (Patton, 2006, p. 65). The software is explored to determine its features, and these features are then tested (Patton, 2006, p. 65). Finding any bugs using this method is a positive thing (Patton, 2006, p. 65), since despite not knowing what the software *should* do, you were able to determine that something is wrong.

This is not applicable to Drasil, because not only does it already generate a specification, making this type of testing unnecessary, there is also a lot of human-based trial and error required for this kind of testing (Smith and Carette, 2023).

3.7.2 Equivalence Partitioning/Classing (Patton, 2006, pp. 67-69)

The process of dividing the infinite set of test cases into a finite set that is just as effective (i.e., that reveals the same bugs) (Patton, 2006, p. 67). The opposite of this, testing every combination of inputs, is called "exhaustive testing" and is "probably not feasible" (Washizaki, 2024, p. 5-5; ISO/IEC and IEEE, 2022, p. 4; van Vliet, 2000, p. 421; Peters and Pedrycz, 2000, pp. 439, 461).

Requirements

- Ranges of possible values (Patton, 2006, p. 67); could be obtained through:
  - Input/output data constraints
  - Case statements

3.7.3 Data Testing (Patton, 2006, pp. 70-79)

The process of "checking that information the user inputs [and] results", both final and intermediate, "are handled correctly" (Patton, 2006, p. 70). This type of testing can also occur at the white-box level, such as the implementation of boundaries (van Vliet, 2000, p. 431) or intermediate values within components.

Boundary Conditions (Patton, 2006, pp. 70-74)

"[S]ituations at the edge of the planned operational limits of the software" (Patton, 2006, p. 72). Often affects types of data (e.g., numeric, speed, character, location, position, size, quantity (Patton, 2006, p. 72)) each with its own set of (e.g., first/last, min/max, start/finish, over/under, empty/full, shortest/longest, slowest/fastest, soonest/latest, largest/smallest, highest/lowest, next-to/farthest-from (Patton, 2006, pp. 72-73)). Data at these boundaries should be included in an equivalence partition, but so should data in between them (Patton, 2006, p. 73). Boundary conditions should be tested using "the valid data just inside the boundary, …the last possible valid data, and …the invalid data just outside the boundary" (Patton, 2006, p. 73), and values at the boundaries themselves should still be tested even if they occur "with zero probability", in case there actually *is* a case where it can occur; this process of testing may reveal it (Peters and Pedrycz, 2000, p. 460).

Requirements

- Ranges of possible values (Patton, 2006, p. 67, 73); could be obtained through:
  - Case statements
  - Input/output data constraints (e.g., inputs that would lead to a boundary output)

Buffer Overruns (Patton, 2006, pp. 201-205) *Buffer overruns* are "the number one cause of software security issues" (Patton, 2006, p. 75). They occur when the size of the destination for some data is smaller than the data itself, causing existing data (including code) to be overwritten and malicious code to potentially be injected (Patton, 2006, p. 202, 204-205). They often arise from bad programming practices in "languages [sic] such as C and C++, that lack safe string handling functions" (Patton, 2006, p. 201). Any unsafe versions of these functions that are used should be replaced with the corresponding safe versions (Patton, 2006, pp. 203-204).

Sub-Boundary Conditions (Patton, 2006, pp. 75-77)

Boundary conditions "that are internal to the software [but] aren't necessarily apparent to an end user" (Patton, 2006, p. 75). These include powers of two (Patton, 2006, pp. 75-76) and ASCII and Unicode tables (Patton, 2006, pp. 76-77).

While this is of interest to the domain of scientific computing, this is too involved for Drasil right now, and the existing software constraints limit much of the potential errors from over/underflow (Smith and Carette, 2023). Additionally, strings are not really used as inputs to Drasil and only occur in output with predefined values, so testing these values are unlikely to be fruitful.

There also exist sub-boundary conditions that arise from "complex" requirements, where behaviour depends on multiple conditions (van Vliet, 2000, p. 430).

These "error prone" points around these boundaries should be tested (van Vliet, 2000, p. 430) as before: "the valid data just inside the boundary, …the last possible valid data, and …the invalid data just outside the boundary" (Patton, 2006, p. 73). 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).

Requirements

- Increased knowledge of data type structures (e.g., monoids, rings, etc. (Smith and Carette, 2023)); this would capture these sub-boundaries, as well as other information like relevant tests cases, along with our notion of these data types (`Space`)

Default, Empty, Blank, Null, Zero, and None (Patton, 2006, pp. 77-78)

These should be their own equivalence class, since "the software usually handles them differently" than "the valid cases or …invalid cases" (Patton, 2006, p. 78).

Since these 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).

Requirements

- Knowledge of an "empty" value for each `Space` (stored alongside each type in `Space`?)
- Knowledge of how input data could be omitted from an input (e.g., a missing command line argument, an empty line in a file); could be obtained from:
  - User responsibilities
- Knowledge of how a programming language deals with `Null` values and how these can be passed as arguments

Invalid, Wrong, Incorrect, and Garbage Data (Patton, 2006, pp. 78-79)

This is testing-to-fail (Patton, 2006, p. 77).

Requirements This seems to be the most open-ended category of testing.

- Specification of correct inputs that can be ignored; could be obtained through:
  - Input/output data constraints (e.g., inputs that would lead to a violated output constraint)
  - Type information for each input (e.g., passing a string instead of a number)

Syntax-Driven Testing (Peters and Pedrycz, 2000, pp. 448-449)

If the inputs to the system "are described by a certain grammar" (Peters and Pedrycz, 2000, p. 448), "test cases ...[can] be designed according to the syntax or constraint of input domains defined in requirement specification" (Intana et al., 2020, p. 260) .

Investigate this source more!

Decision Table-Based Testing (Peters and Pedrycz, 2000, pp. 448, 450-453)

"When the original software requirements have been formulated in the format of 'if-then' statements," a decision table can be created with a column for each test situation (Peters and Pedrycz, 2000, p. 448). "The upper part of the column contains conditions that must be satisfied. The lower portion of a decision table specifies the action that results from the satisfaction of conditions in a rule" (from the specification) (Peters and Pedrycz, 2000, p. 450).

3.7.4 State Testing (Patton, 2006, pp. 79-87)

The process of testing "the program's logic flow through its various states" (Patton, 2006, p. 79) by checking that state variables are correct after different transitions (p. 83). This is usually done by creating a state transition diagram that includes:

- Every possible unique state
- The condition(s) that take(s) the program between states
- The condition(s) and output(s) when a state is entered or exited

to map out the logic flow from the user's perspective (Patton, 2006, pp. 81-82). Next, these states should be partitioned using one (or more) of the following methods:

1. Test each state once
2. Test the most common state transitions
3. Test the least common state transitions
4. Test all error states and error return transitions
5. Test random state transitions (Patton, 2006, pp. 82-83)

For all of these tests, the values of the state variables should be verified (Patton, 2006, p. 83).

Requirements

- Knowledge of the different states of the program (Patton, 2006, p. 82); could be obtained through:
  - The program's modules and/or functions
  - The program's exceptions
- Knowledge about the different state transitions (Patton, 2006, p. 82); could be obtained through:
  - Testing the state transitions near the beginning of a workflow more?

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!

Testing States to Fail (Patton, 2006, pp. 84-87)

The goal here is to try and put the program in a fail state by doing things that are out of the ordinary. These include:

- Race Conditions and Bad Timing (Patton, 2006, pp. 85-86) (Is this relevant to our examples?)
- Repetition Testing: "doing the same operation over and over", potentially up to "thousands of attempts" (Patton, 2006, p. 86)
- Stress Testing: "running the software under less-than-ideal conditions" to see how it functions (Patton, 2006, p. 86)
- Load testing: running the software with as large of a load as possible (e.g., large inputs, many peripherals) (Patton, 2006, p. 86)

Requirements

- Repetition Testing: The types of operations that are likely to lead to errors when repeated (e.g., overwriting files?)
- Stress testing: can these be automated with pytest or are they outside our scope?

Investigate

- Load testing: Knowledge about the types of inputs that could overload the system (e.g., upper bounds on values of certain types)

3.7.5 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)!*)
- Think like a hacker (*likely out of scope*)
- Follow experience (*implicitly done by using Drasil*)

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

White-box testing is also called "glass box testing" (Peters and Pedrycz, 2000, p. 439). (Peters and Pedrycz, 2000, p. 447) claims that "structural testing subsumes white box testing", but I am unsure if this is a meaningful statement; they seem to describe the same thing to me, especially since it says "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!

Q #3: Is this true?

There are also some more specific categories of this, such as Scenario-Based Evaluation (van Vliet, 2000, pp. 417-418) and Stepwise Abstraction (van Vliet, 2000, pp. 419-420), that could be investigated further.

Do this!

- "The process of carefully and methodically reviewing the software design, architecture, or code for bugs without executing it" (Patton, 2006, p. 92)
- Less common than black-box testing, but often used for "military, financial, factory automation, or medical software, ...in a highly disciplined development model" or when "testing software for security issues" (Patton, 2006, p. 91); often avoided because of "the misconception that it's too time-consuming, too costly, or not productive" (Patton, 2006, p. 92)
- Especially effective early on in the development process (Patton, 2006, p. 92)
- Can "find bugs that would be difficult to uncover or isolate with dynamic black-box testing" and "gives the team's black-box testers ideas for test cases to apply" (Patton, 2006, p. 92)
- Largely "done by the language compiler" or by separate tools (van Vliet, 2000, pp. 413-414)

3.8.1 Reviews (Patton, 2006, pp. 92-95), (van Vliet, 2000, pp. 415-417), (Peters and Pedrycz, 2000, pp. 482-485)

- "The process under which static white-box testing is performed" (Patton, 2006, p. 92); consists of four main parts:

  1. Identify Problems: Find what is wrong or missing
  2. Follow Rules: There should be a structure to the review, such as "the amount of code to be reviewed …, how much time will be spent …, what can be commented on, and so on", to set expectations; "if a process is run in an ad-hoc fashion, bugs will be missed and the participants will likely feel that the effort was a waste of time"
  3. Prepare: Based on the participants' roles, they should know what they will be contributing during the actual review; "most of the problems found through the review process are found during preparation"
  4. Write a Report: A summary should be created and provided to the rest of the development team so that they know what problems exist, where they are, etc. (Patton, 2006, p. 93)

- Reviews improve communication, learning, and camaraderie, as well as the quality of code *even before the review*: if a developer "knows that his work is being carefully reviewed by his peers, he might make an extra effort to …make sure that it's right" (Patton, 2006, pp. 93-94)

- Many forms:

  - Peer Review: Also called "buddy review" (Patton, 2006, p. 94). The most informal review at the smallest scale (Patton, 2006, p. 94). One variation is where a group of two or three people go through code that one of them wrote (Patton, 2006, p. 94). Another is to have each person in a larger group submit "a 'best' program and one of lesser quality", randomly distribute all programs to be assessed by two people in the group, and return all feedback anonymously to the appropriate developer (van Vliet, 2000, p. 414)
  - Walkthrough: The author of the code presents it line by line to a small group that "question anything that looks suspicious" (Patton, 2006, p. 95); this is done by using test data to "walk through" the execution of the program (van Vliet, 2000, p. 416). A more structured walkthrough may have specific roles (presenter, coordinator, secretary, maintenance oracle, standards bearer, and user representative) (Peters and Pedrycz, 2000, p. 484)
  - Inspection: Someone who is *not* the author of the code presents it to a small group of people (Patton, 2006, p. 95); the author should be "a largely silent observer" who "may be consulted by the inspectors" (van Vliet, 2000, p. 415). Each member has a role, which may be tied

to a different perspective (e.g., designer, implementer, tester, (Peters and Pedrycz, 2000, p. 439) user, or product support person) (Patton, 2006, p. 95). Changes are made based on issues identified *after* the inspection (van Vliet, 2000, p. 415), and a reinspection may take place (Patton, 2006, p. 95); one guideline is to reinspect *100%* of the code "[i]f more than 5% of the material inspected has been reworked" (Peters and Pedrycz, 2000, p. 483).

- Can use various tools (see Coding Standards and Guidelines and Generic Code Review Checklist)
- *Could be used to evaluate Drasil and/or generated code, but couldn't be automated due to the human element*

This shouldn't really be at the same level as Reviews, but I didn't want to fight with more subsections yet

3.8.2 Coding Standards and Guidelines (Patton, 2006, pp. 96-99)

- Code may work but still be incorrect if it doesn't meet certain criteria, since these affect its reliability, readability, maintainability, and/or portability; e.g., the `goto`, `while`, and `if-else` commands in C can cause bugs if used incorrectly (Patton, 2006, p. 96)
- These guidelines can range in strictness and formality, as long as they are agreed upon and followed (Patton, 2006, p. 96)
- This could be checked using linters

This shouldn't really be at the same level as Reviews, but I didn't want to fight with more subsections yet

3.8.3 Generic Code Review Checklist (Patton, 2006, pp. 99-103)

- Data reference errors: "bugs caused by using a variable, constant, …[etc.] that hasn't been properly declared or initialized" for its context (Patton, 2006, p. 99)
- Data declaration errors: bugs "caused by improperly declaring or using variables or constants" (Patton, 2006, p. 100)
- Computation errors: "essentially bad math"; e.g., type mismatches, over/underflow, zero division, out of meaningful range (Patton, 2006, p. 101)
- Comparison errors: "very susceptible to boundary condition problems"; e.g., correct inclusion, floating point comparisons (Patton, 2006, p. 101)
- Control flow errors: bugs caused by "loops and other control constructs in the language not behaving as expected" (Patton, 2006, p. 102)
- Subroutine parameter errors: bugs "due to incorrect passing of data to and from software subroutines" (Patton, 2006, p. 102) (could also be called "interface errors" (van Vliet, 2000, p. 416))

- Input/output errors: e.g., how are errors handled? (Patton, 2006, pp. 102-103)
- ASCII character handling, portability, compilation warnings (Patton, 2006, p. 103)

Requirements

- Data reference errors: know what operations are allowed for each type and check that values are only used for those operations
- Data declaration errors: I think this will mainly be covered by checking for data reference errors and by our generator (e.g., no typos in type names)
- Computation errors: partially tested dynamically by system tests, but could also more formally check for things like type mismatches (does GOOL do this already?) or if divisors can ever be zero
- Comparison errors: I think this would mainly have to be done manually (maybe except for checking for (in)equality between values where it can never occur), but we may be able to generate a summary of all comparisons for manual verification
- Control flow errors: mostly irrelevant since we don't implement loops yet; would this include system tests?
- Subroutine parameter errors: we could check the types of values returned by a subroutine with the expected type (at least for languages like Python)
- Input/output errors: knowledge of (and more formal specification of) requirements would be needed here
- ASCII character handling, portability, compilation warnings: we could automatically check that the compiler (for languages that meaningfully have a compile stage) doesn't output any warnings (e.g., by saving output to a file and checking it is what is expected from a normal compilation); do we have any string inputs?

3.8.4 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

3.9 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).

3.9.1 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"[10] (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)

OG Miller et al., 1994

OG Miller et al., 1994

OG Miller et al., 1994

[10] 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).

  - 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)

OG Miller et al., 1994

  - 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))

Q #4: How do we decide on our definition?

  - 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)
  - 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)

3.9.2 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 3.7.3
- 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 3.3 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?

Table 3.3: 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

Q #5: How is All-DU-Paths coverage stronger than All-Uses coverage according to (van Vliet, 2000, p. 433)?

3.9.3 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).

3.9.4 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 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).

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!

OG KA85

3.10 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).

3.11 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

3.12 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).

3.12.1 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)

3.12.2 Examples of MT

- 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)

3.13 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?

3.13.1 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 int 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 4

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 Generating Requirements)

- 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

4.1 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

4.1.1 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.

4.2 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

4.3 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.1).

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

Should I include the definition of `Constraints`?

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

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

Taking another look at this four years later allowed us to see that we should be storing the outputs of a program as their underlying models, allowing us to keep the source information with it . While there is some discussion about how this might change in the future , for now, all outputs of a program should be `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.

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.

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.

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

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

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

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

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.

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.

Paul Gerrard. Risk-based E-business Testing - Part 1: Risks and Test Strategy. Technical report, Systeme Evolutif, London, UK, 2000. URL https://www.agileconnection.com/sites/default/files/article/file/2013/XUS129342file1_0.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, editors. ISTQB Glossary, v4.3, 2024. URL https://glossary.istqb.org/en_US/search.

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

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

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

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.

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/.

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.

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

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.

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].

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.

Appendix

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

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


    assert not path.exists(output_filename)


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

    assert not path.exists(output_filename)
```

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

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

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

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

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

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

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

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