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, 59

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

AOP	Aspect-Oriented Programming
AST	Abstract Syntax Tree
CSP	Cross-Stage Persistence
CTMP	Compile-Time MetaProgramming
DSL	Domain-Specific Language
GOOL	Generic Object-Oriented Language
HREF	Hypertext REFerence
IDE	Integrated Development Environment
MDD	Model-Driven Development
MOP	MetaObject Protocol
MR	Metamorphic Relation
MSL	MultiStage Language
MSP	MultiStage Programming
MT	Metamorphic Testing
PDF	Portable Document Format
PPTMP	PreProcessing-Time MetaProgramming
QAI	Quality Assurance Institute
RTMP	RunTime MetaProgramming
SST	Skeleton Syntax Tree
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

Notes

3.1 A Survey of Metaprogramming Languages

investigate more: Steele 1990?

- Often done with Abstract Syntax Trees (ASTs), although other bases are used:
  - Skeleton Syntax Trees (SSTs), used by Dylan (Lilis and Savidis, 2019, p. 113:6)
- Allows for improvements in:
  - "performance by generating efficient specialized programs based on specifications instead of using generic but inefficient programs" (Lilis and Savidis, 2019, p. 113:2)
  - reasoning about object programs through "analyzing and discovering object-program characteristics that enable applying further optimizations as well as inspecting and validating the behavior of the object program" (Lilis and Savidis, 2019, p. 113:2)
  - code reuse through capturing "code patterns that cannot be abstracted" (Lilis and Savidis, 2019, p. 113:2)

3.1.1 Definitions

"*Metaprogramming* is the process of writing computer programs, called *metaprograms*, that [can] …generate new programs or modify existing ones" (Lilis and Savidis, 2019, p. 113:1). "It constitutes a flexible and powerful reuse solution for the ever-growing size and complexity of software systems" (Lilis and Savidis, 2019, p. 113:31).

- Metalanguage: "the language in which the metaprogram is written" (Lilis and Savidis, 2019, p. 113:1)
- Object language: "the language in which the generated or transformed program is written" (Lilis and Savidis, 2019, p. 113:1)

- Homogeneous metaprogramming: when "the object language and the metalanguage are the same" (Lilis and Savidis, 2019, p. 113:1)
- Heterogeneous metaprogramming: when "the object language and the metalanguage are …different" (Lilis and Savidis, 2019, p. 113:1)

3.1.2 Metaprogramming Models

Macro Systems (Lilis and Savidis, 2019, p. 113:3-7)

- Map specified input sequences in a source file to corresponding output sequences ("macro expansion") until no input sequences remain (Lilis and Savidis, 2019, p. 113:3); this process can be:
  1. procedural (involving algorithms; this is more common (Lilis and Savidis, 2019, p. 113:31)), or
  2. pattern-based (only using pattern matching) (Lilis and Savidis, 2019, p. 113:4)
- Must avoid variable capture (unintended name conflicts) by being "hygienic" (Lilis and Savidis, 2019, p. 113:4); this may be overridden to allow for "intentional variable capture", such as Scheme's *syntax-case* macro (Lilis and Savidis, 2019, p. 113:5)

Lexical Macros

- Language agnostic (Lilis and Savidis, 2019, p. 113:3)
- Usually only sufficient for basic metaprogramming since changes to the code without considering its meaning "may cause unintended side effects or name clashes and may introduce difficult-to-solve bugs" (Lilis and Savidis, 2019, p. 113:5)
- Marco was the first safe, language-independent macro system that "enforce[s] specific rules that can be checked by special oracles" for given languages (as long as the languages "produce descriptive error messages") (Lilis and Savidis, 2019, p. 113:6)

Syntactic Macros

- "Aware of the language syntax and semantics" (Lilis and Savidis, 2019, p. 113:3)
- MS^2 "was the first programmable syntactic macro system for syntactically rich languages", including by using "a type system to ensure that all generated code fragments are syntactically correct" (Lilis and Savidis, 2019, p. 113:5)

Reflection Systems (Lilis and Savidis, 2019, p. 113:7-9)

- "Perform computations on [themselves] in the same way as for the target application, enabling one to adjust the system behavior based on the needs of its execution" (Lilis and Savidis, 2019, p. 113:7)
- Means that the system can "observe and possibly modify its structure and behaviour" (Štuikys and Damaševičius, 2013, p. 22); these processes are called "introspection" and "intercession", respectively (Lilis and Savidis, 2019, p. 113:7)
  - The representation of a system can either be structural or behavioural (e.g., variable assignment) (Lilis and Savidis, 2019, p. 113:7)
- "Runtime code generation based on source text can be impractical, inefficient, and unsafe, so alternatives have been explored based on ASTs and quasi-quote operators, offering a structured approach that is subject to typing for expressing and combining code at runtime" (Lilis and Savidis, 2019, p. 113:8)
- "Not limited to runtime systems", as some "compile-time systems …rely on some form of structural introspection to perform code generation" (Lilis and Savidis, 2019, p. 113:9)

MetaObject Protocols (MOPs) (Lilis and Savidis, 2019, p. 113:9-11)

- "Interfaces to the language enabling one to incrementally transform the original language behavior and implementation" (Lilis and Savidis, 2019, p. 113:9)
- Three different approaches:
  - Metaclass-based Approach: "Classes are considered to be objects of metaclasses, called metaobjects, that are responsible for the overall behavior of the object system" (Lilis and Savidis, 2019, p. 113:9)
  - Metaobject-based Approach: "Classes and metaobjects are distinct" (Lilis and Savidis, 2019, p. 113:9)
  - Message Reification Approach: used with message passing (Lilis and Savidis, 2019, p. 113:9)
- Can either be runtime (more common) or compile-time (e.g., OpenC++); the latter protocols "operate as advanced macro systems that perform code transformation based on metaobjects rather than on text or ASTs" (Lilis and Savidis, 2019, p. 113:11)

Dynamic Shells "Pseudo-objects with methods and instance variables that may be attached to other objects" that "offer efficient and type-safe MOP functionality for statically typed languages" (Lilis and Savidis, 2019, p. 113:10).

Dynamic Extensions "Offer similar functionality [to dynamic shells] but for classes, allowing a program to replace the methods of a class and its subclasses by the methods of another class at runtime" (Lilis and Savidis, 2019, p. 113:10).

Aspect-Oriented Programming (AOP) (Lilis and Savidis, 2019, p. 113:11-13)

- The use of *aspects*: "modular units …[that] contain information about the additional behavior, called *advice*, that will be added to the base program by the aspect as well as the program locations, called *join points*, where this extra behavior is to be inserted based on some matching criteria, called *pointcuts*" (Lilis and Savidis, 2019, p. 113:12)
- Weaving: the process of "combining the base program with aspect code …[to form] the final code" (Lilis and Savidis, 2019, p. 113:12)
- Two variants:
  1. Static AOP: when weaving takes place at compile time, usually with "a separate language and a custom compiler, called [an] *aspect weaver*"; results in better performance (Lilis and Savidis, 2019, p. 113:12)
  2. Dynamic AOP: when weaving takes place at runtime by instrumenting "the bytecode …to be able to weave the aspect code"; provides more flexibility (Lilis and Savidis, 2019, p. 113:12)
- This model originates from reflecting and MOPs (AspectS and AspectL "support AOP by building respectively on the runtime MOPs of Smalltalk and Lisp") (Lilis and Savidis, 2019, p. 113:12)
- While "AOP can support metaprogramming by inserting code before, after, or around matched join points, as well as introducing data members and methods through intertype declarations", it is usually done the other way around, as most AOP frameworks "rely on metaprogramming techniques" (Lilis and Savidis, 2019, p. 113:12)

Generative Programming (Lilis and Savidis, 2019, p. 113:13-17)

- "A software development paradigm based on modeling software system families such that, given a particular requirements specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge"

get original source from Czarnecki and Eisenecker 2000

- Often done with using ASTs (Lilis and Savidis, 2019, p. 113:31)
- Most "support code templates and quasi-quotes" (Lilis and Savidis, 2019, p. 113:31)
- Related to macro systems, but normal code and metacode are distinct

Template Systems (Lilis and Savidis, 2019, p. 113:13-14)

- Template code is instantiated with specific parameters to generate ALL code in a target language; "no free-form source code generation is allowed" (Lilis and Savidis, 2019, p. 113:13)
- It is possible, though complex, to express any "to express any generative metaprogram", as long as "the appropriate metaprogramming logic for type manipulation" is present (Lilis and Savidis, 2019, p. 113:14)

clarify what "free-form source code generation" means

AST Transformations (Lilis and Savidis, 2019, p. 113:14-15)

- "Offer code templates through quasi-quotation to support AST creation and composition and complement them with AST traversal or transformation features" (Lilis and Savidis, 2019, p. 113:14)

Compile-Time Reflections (Lilis and Savidis, 2019, p. 113:15-16)

- "Offer compile-time reflection features to enable generating code based on existing code structures" while trying to ensure that "the generator will always produce well-formed code" (this is not always fully possible; for example, Genoupe "cannot guarantee that the generated code is always well typed") (Lilis and Savidis, 2019, p. 113:15)

Class Compositions (Lilis and Savidis, 2019, p. 113:16-17)

- Offer "flexibility and expressiveness" through composition approaches (Lilis and Savidis, 2019, p. 113:16)
  - *Mixins*:
  - *Traits*: "support a uniform, expressive, and type-safe way for metaprogramming without resorting to ASTs" and offer "compile-time pattern-based reflection" through parameterization (Lilis and Savidis, 2019, p. 113:16)

Investigate

- Includes *feature-oriented programming* approaches

Investigate?

MultiStage Programming (MSP) (Lilis and Savidis, 2019, p. 113:17-20)

- "Makes ...[levels of evaluation] accessible to the programmer through ...*staging annotations*" to "specify the evaluation order of the program computations" and work with these computation stages (Lilis and Savidis, 2019, p. 113:17)
- Related to program generation and procedural macro systems (Lilis and Savidis, 2019, p. 113:17); macros are often implemented as multistage computations (Lilis and Savidis, 2019, p. 113:18)

- Languages that use MSP are called *MultiStage Languages (MSLs)* or *two-stage languages*, depending on how many stages of evaluation are offered (Lilis and Savidis, 2019, p. 113:17); MSLs are more common (Lilis and Savidis, 2019, p. 113:31)
  - C++ first instantiates templates, then translates nontemplate code (Lilis and Savidis, 2019, p. 113:19)
  - Template Haskell evaluates "the top-level splices to generate object-level code" at compile time, then executes the object-level code at runtime (Lilis and Savidis, 2019, p. 113:19)
- Often involves *Cross-Stage Persistence (CSP)*, which allows "values …available in the current stage" to be used in future stages (Lilis and Savidis, 2019, p. 113:17)
  - If this is used, *cross-stage safety* is often also used to prevent "variables bound at some stage …[from being] used at an earlier stage" (Lilis and Savidis, 2019, p. 113:17)
- Usually homogeneous, but there are exceptions; MetaHaskell, a modular framework (Lilis and Savidis, 2019, p. 113:19) with a type system, allows for "heterogeneous metaprogramming with multiple object languages" (Lilis and Savidis, 2019, p. 113:18)
- "Type safety …comes at the cost of expressiveness" (Lilis and Savidis, 2019, p. 113:19)

3.1.3 Phase of Evaluation

- "In theory, any combination of them [the phases of evaluation] is viable; however, in practice most metalanguages offer only one or two of the options" (Lilis and Savidis, 2019, p. 113:20)
- "The phase of evaluation does not necessarily dictate the adoption of a particular metaprogramming model; however, there is a correlation between the two" (Lilis and Savidis, 2019, p. 113:20)

Preprocessing-Time Evaluation (Lilis and Savidis, 2019, p. 113:20-21)

- In PreProcessing-Time MetaProgramming (PPTMP), "metaprograms present in the original source are evaluated during the preprocessing phase and the resulting source file contains only normal program code and no metacode" (Lilis and Savidis, 2019, p. 113:20)
- These systems are called *source-to-source preprocessors* (Lilis and Savidis, 2019, p. 113:20) and are usually examples of generative programming (Lilis and Savidis, 2019, p. 113:21)

  - "All such cases involve syntactic transformations" (Lilis and Savidis, 2019, p. 113:21), usually using ASTs

- "Translation can reuse the language compiler or interpreter without the need for any extensions" (Lilis and Savidis, 2019, p. 113:20)
- Varying levels of complexity (e.g., these systems "may be fully aware of the language syntax and semantics") (Lilis and Savidis, 2019, p. 113:20)
- Includes all lexical macro systems (Lilis and Savidis, 2019, p. 113:20) and some "static AOP and generative programming systems" (Lilis and Savidis, 2019, p. 113:31)
- Typically doesn't use reflection (Reflective Java is an exception), MOPs, or dynamic AOP (Lilis and Savidis, 2019, p. 113:21)

Compilation-Time Evaluation (Lilis and Savidis, 2019, p. 113:21-23)

- In Compile-Time MetaProgramming (CTMP), "the language compiler is extended to handle metacode translation and execution" (Lilis and Savidis, 2019, p. 113:22)
  - There are many ways of extending the compiler, including "plugins, syntactic additions, procedural or rewrite-based AST transformations, or multistage translation" (Lilis and Savidis, 2019, p. 113:22)
  - Metacode execution can be done by "interpreting the source metacode …or compiling the source metacode to binary and then executing it" (Lilis and Savidis, 2019, p. 113:22)
- These systems are usually examples of generative programming but can also use macros, MOPs, AOP (Lilis and Savidis, 2019, p. 113:22), and/or reflection (Lilis and Savidis, 2019, p. 113:23)

Execution-Time Evaluation (Lilis and Savidis, 2019, p. 113:23-25)

- RunTime MetaProgramming (RTMP) "involves extending the language execution system and offering runtime libraries to enable dynamic code generation and execution" and is "the only case where it is possible to extend the system based on runtime state and execution" (Lilis and Savidis, 2019, p. 113:23)
- Includes "most reflection systems, MOPs, MSP systems, and dynamic AOP systems" (Lilis and Savidis, 2019, p. 113:31)

3.1.4 Metaprogram Source Location

Embedded in the Subject Program (Lilis and Savidis, 2019, p. 113:25-26)

- Usually occurs with macros, templates, MSLs, reflection, MOPs, and AOP (Lilis and Savidis, 2019, p. 113:25)

Context Unaware (Lilis and Savidis, 2019, p. 113:25)

- Occurs when metaprograms only need to know their input parameters to generate ASTs (Lilis and Savidis, 2019, p. 113:25)
- Very common: supported by "most CTMP systems" (Lilis and Savidis, 2019, p. 113:31) and "for most macro systems…, generative programming systems …and MSLs …it is the only available option" (Lilis and Savidis, 2019, p. 113:25)

Context Aware (Lilis and Savidis, 2019, p. 113:25-26)

- "Typically involves providing access to the respective program AST node and allowing it to be traversed" as "an extra …parameter to the metaprogram" (Lilis and Savidis, 2019, p. 113:25)
- Allows for code transformation "at multiple different locations reachable from the initial context" (Lilis and Savidis, 2019, p. 113:25)
- Very uncommon (Lilis and Savidis, 2019, p. 113:25, 31)

Global (Lilis and Savidis, 2019, p. 113:26)

- Involves "scenarios that collectively introduce, transform, or remove functionality for the entire program" (Lilis and Savidis, 2019, p. 113:26)
- Usually occurs with reflection, MOPs, and AOP (Lilis and Savidis, 2019, p. 113:26); offered by "most RTMP systems" (Lilis and Savidis, 2019, p. 113:31)
- Can be used with "any PPTMP or CTMP system that provides access to the full program AST" (Lilis and Savidis, 2019, p. 113:26)
- "Can also be seen as a context-aware case where the context is the entire program" (Lilis and Savidis, 2019, p. 113:26)

External to the Subject Program (Lilis and Savidis, 2019, p. 113:27)

- Occurs when metaprograms "are specified as separate transformation programs applied through PPTMP systems or supplied to the compiler together with the target program to be translated as extra parameters" (Lilis and Savidis, 2019, p. 113:27)
- Includes many instances of AOP (Lilis and Savidis, 2019, p. 113:27)

3.1.5 Relation to the Object Language

- Each metaprogramming language has two layers:
  1. "The basic object language"
  2. "The metaprogramming elements for implementing the metaprograms" (the *metalayer*) (Lilis and Savidis, 2019, p. 113:27)
- Sometimes the metalayer of a language is added to a language later, independently of the object language (Lilis and Savidis, 2019, p. 113:27)

Metalanguage Indistinguishable from the Object Language (Lilis and Savidis, 2019, p. 113:28-29)

- Two categories:
  1. "Object language and metalanguage …use the same constructs through the same syntax"
  2. "Metalanguage constructs …[are] modeled using object language syntax and applied through special language or execution system features" (Lilis and Savidis, 2019, p. 113:28)
     - Includes many examples of MOPs and AOP (Lilis and Savidis, 2019, p. 113:28)

Metalanguage Extends the Object Language (Lilis and Savidis, 2019, p. 113:29)

- Allows for reuse of "the original language['s] …well-known features instead of adopting custom programming constructs" (Lilis and Savidis, 2019, p. 113:29)
- "Typically involve new syntax and functionality used to differentiate normal code from metacode" (Lilis and Savidis, 2019, p. 113:29)
- Often used in quasi-quote constructs, two-stage and multistage languages, and MOPs (Lilis and Savidis, 2019, p. 113:29)
- Used with MSLs "as the base languages are extended with staging annotations to deliver MSP functionality" (Lilis and Savidis, 2019, p. 113:31)

Metalanguage Different from the Object Language (Lilis and Savidis, 2019, p. 113:29-31)

- Allows for "the metalanguage syntax and constructs …[to be] selected to better reflect the metalanguage concepts to ease their use in developing metaprograms and enable them to become more concise and understandable" (Lilis and Savidis, 2019, p. 113:29)

- However, it can lead to "different development practices and disable[s] the potential for design or code reuse between them [the languages]", as well as requiring users to know how to use both languages (Lilis and Savidis, 2019, p. 113:30)
- Used by some AOP and generative metaprogramming systems (Lilis and Savidis, 2019, p. 113:30)

3.2 Overview of Generative Software Development

"System family engineering seeks to exploit the commonalities among systems from a given problem domain while managing the variabilities among them in a systematic way" (Czarnecki, 2004, p. 326). "Generative software development is a system-family approach …that focuses on automating the creation of system-family members …from a specification written in [a Domain-Specific Language (DSL)]" (Czarnecki, 2004, p. 327). "DSLs come in a wide variety of forms, …[including] textual …[and] diagrammatic" (Czarnecki, 2004, p. 328).

"System family engineering distinguishes between at least two kinds of development processes: *domain engineering* and *application engineering*" (Czarnecki, 2004, p. 328). "Domain engineering …is concerned with the development of reusable assets such as components, generators, DSLs, analysis and design models, user documentation, etc." (Czarnecki, 2004, pp. 328-329). It includes "determining the scope of the family to be built, identifying the common and variable features among the family members", and "the development of a common architecture for all the members of the system family" (Czarnecki, 2004, p. 329). Application engineering includes "requirements elicitation, analysis, and specification" and "the manual or automated construction of the system from the reusable assets" (Czarnecki, 2004, p. 329). The assets from domain engineering are used to build the system development by application engineering, which provides domain engineering which the requirements to analyze for commonalities and create reusable assets for (Czarnecki, 2004, p. 329).

Aspect-Oriented Programming (AOP) "provides more powerful localization and encapsulation mechanisms than traditional component technologies" but there is still the need to "configure aspects and other components to implement abstract features" (Czarnecki, 2004, p. 338). AOP "cover[s] the solution space and only a part of the configuration knowledge", although "aspects can also be found in the problem space" (Czarnecki, 2004, p. 338).

3.2.1 Definitions

- Generative domain model: "a mapping between *problem space* and *solution space*" which "takes a specification and returns the corresponding implementation" (Czarnecki, 2004, p. 330)
  - Configuration view: "the problem space consists of domain-specific concepts and their features" such as "illegal feature combinations, default

settings, and default dependencies" (Czarnecki, 2004, p. 331). "An application programmer creates a configuration of features by selecting the desired ones, [sic] which then is mapped to a configuration of components" (Czarnecki, 2004, p. 331)

  - Transformational view: "a problem space is represented by a …[DSL], whereas the solution space is represented by an implementation language" (Czarnecki, 2004, p. 331). "A program in a …[DSL]" is transformed into "its implementation in the implementation language" (Czarnecki, 2004, p. 331)

- Problem space: "a set of domain-specific abstractions that can be used to specify the desired system-family member" (Czarnecki, 2004, p. 330)

- Solution space: "consists of implementation-oriented abstractions, which can be instantiated to create implementations of the [desired] specifications" (Czarnecki, 2004, p. 330)

- Network of domains: the graph built from "spaces and mappings …where each implementation of a domain exposes a DSL, which may be implemented by transformations to DSLs exposed by other domain implementations" (Czarnecki, 2004, pp. 332-333)

- Feature modeling: "a method and notation to elicit and represent common and variable features of the systems in a system family" (Czarnecki, 2004, p. 333). Can be used during domain analysis as "the starting point in the development of both system-family architecture and DSLs" (Czarnecki, 2004, p. 334)

- Model-Driven Development (MDD): uses "abstract representation[s] of a system and the portion[s] of the world that interact[] with it" to "captur[e] every important aspect of a software system" (Czarnecki, 2004, p. 336). Often uses DSLs and sometimes deals with system families, making it related to generative software development (Czarnecki, 2004, pp. 336-337)

3.3 Structured Program Generation Techniques

- Program transformer: something that "modifies an existing program, instead of generating a new one" (for example, by making a program's code adhere to style guides); the term "program generator" often includes program transformers (Smaragdakis et al., 2017, p. 1)

- Generators are used "to automate, elevate, modularize or otherwise facilitate program development" (Smaragdakis et al., 2017, p. 2)

- Why is it beneficial "to statically check the generator and be sure that no type error arises during its *run time*" (Smaragdakis et al., 2017, p. 2) instead of just checking the generated program(s)?

  - "An error in the generated program can be very hard to debug and may require full understanding of the generator itself" (Smaragdakis et al., 2017, p. 2)
  - Errors can occur in the generator from "mismatched assumptions"; for example, "the generator fails to take into account some input case, so that, even though the generator writer has tested the generator under several inputs, other inputs result in badly-formed programs" (Smaragdakis et al., 2017, p. 6)

3.3.1 Techniques for Program Generation (Smaragdakis et al., 2017, pp. 3-5)

1. Generation as text: "producing character strings containing the text of a program, which is subsequently interpreted or compiled" (Smaragdakis et al., 2017, p. 3)
2. Syntax tree manipulation: building up code using constructors in a syntactically meaningful way that preserves its structure
3. Code templates/quoting: involves "language constructs for generating program fragments in the target language …as well as for supplying values to fill in holes in the generated syntax tree" (Smaragdakis et al., 2017, p. 4)
4. Macros: "reusable code templates with pre-set rules for parameterizing them" (Smaragdakis et al., 2017, p. 4)
5. Generics: Mechanisms with "the ability to parameterize a code template with different static types" (Smaragdakis et al., 2017, p. 5)
6. Specialized languages: Languages with specific features for program generators, such as AOP and *inter-type declarations* (Smaragdakis et al., 2017, p. 5)

3.3.2 Kinds of Generator Safety (Smaragdakis et al., 2017, pp. 5-8)

- Lexical and syntactic well-formedness: "any generated/transformed program is guaranteed to pass the lexical analysis and parsing phases of a traditional compiler"; usually done "by encoding the syntax of the object language using the type system of the host language" (Smaragdakis et al., 2017, p. 6)
- Scoping and hygiene: avoiding issues with scope and unintentional variable capture
- Full well-formedness: ensuring that any generated/transformed program is guaranteed to be fully well-formed (e.g., "guaranteed to pass any static check in the target language" (Smaragdakis et al., 2017, p. 8))

3.3.3 Methods for Guaranteeing Fully Structured Generation (Smaragdakis et al., 2017, pp. 8-20)

1. MultiStage Programming (MSP): "the generator and the generated program …are type-checked by the same type system[] and some parts of the program are merely evaluated later (i.e., generated)"; similar to *partial evaluation* (Smaragdakis et al., 2017, p. 9)
2. Class Morphing: similar to MetaObject Protocols (MOPs)?
3. Reflection: (e.g., SafeGen (Smaragdakis et al., 2017, p. 15))
4. The use of "a powerful type system that can simultaneously express conventional type-level properties of a program and the logical structure of a generator under unknown inputs. This typically entails the use of dependent types" (e.g., Ur) (Smaragdakis et al., 2017, p. 16)
5. Macro systems, although "safety guarantees carry the cost of some manual verification effort by the programmer" (Smaragdakis et al., 2017, p. 19)

3.4 Taxonomy of Fundamental Concepts of Meta-Programming

3.4.1 Definitions

- Program transformation: "the process of changing one form of a program (source code, specification or model) into another, as well as a formal or abstract description of an algorithm that implements this transformation" (Štuikys and Damaševičius, 2013, p. 18)
  - It may or may not preserve the program's semantics (Štuikys and Damaševičius, 2013, p. 18)
  - In metaprogramming, "the transformation algorithm describes generation of a particular instance depending upon values of the generic parameters" (Štuikys and Damaševičius, 2013, p. 18)
  - Formal program transformation: "A stepwise manipulation, which (1) is defined on a programming language domain, (2) uses a formal model to support the refinement, and (3) simultaneously preserves the semantics" (Štuikys and Damaševičius, 2013, p. 18)
- Code generation: "the process by which a code generator converts a syntactically correct high-level program into a series of lower-level instructions"; the input can take many forms "typically consists of a parse tree, abstract syntax tree or intermediate language code" and "the output …could be in any language" (Štuikys and Damaševičius, 2013, p. 19)
- Generic component: "a software module …[that] abstractly and concisely represents a set of closely related ('look-alike') software components with slightly different properties" (Štuikys and Damaševičius, 2013, p. 19)

- Generative component: a generic component that has "explicitly added generative technology" (Štuikys and Damaševičius, 2013, p. 24)
- Separation of concerns: "the process of breaking a design problem into distinct tasks that are orthogonal and can be implemented separately" (Štuikys and Damaševičius, 2013, p. 21)

3.4.2 Other Notes

- Structural meta-programming concepts "are defined by the designer", "used during construction of the meta-programming systems and artefacts", and "depend upon [the] specific …meta-language" used (Štuikys and Damaševičius, 2013, p. 24)
- Most processes "are used in compile time or run time" except for generalization, which "is used during the creation of the meta-programming artefacts" (Štuikys and Damaševičius, 2013, pp. 24-25)

3.5 Roadblocks to Meta-Programming

- "Generators are often the technique of last resort" (Smaragdakis et al., 2017, p. 2)
- "A major stumbling block to achieving the promised benefits [of meta-programming] is the understanding and learning the meta-programming approach. One reason may be that we do not yet thoroughly understand the fundamental concepts that define meta-programming" (Štuikys and Damaševičius, 2013, p. 26)
- Meta-programming does not provide instant results; instead, the effort and design put in at the beginning of the process later pay off potentially large dividends that are not seen right away; "most …programmers and designers …like to reuse the existing software artefacts, but not much is done and [sic] invested into designing for reuse" (Štuikys and Damaševičius, 2013, p. 26) (example, meta-programming was proposed by McIlroy in 1968 but "software factories have not become a reality …partly due to …[this] significant initial investment") (Štuikys and Damaševičius, 2013, p. 27)
- Software development involves "work[ing] with multiple levels of abstraction", including "the syntax, semantics, abilities and limitations" of given languages, their implementation details, their communication details, and "impeding mismatches" between them (Štuikys and Damaševičius, 2013, p. 27)
- "Modification of the generated code usually removes the program from the scope of the meta-programming system" (Štuikys and Damaševičius, 2013, p. 27)

3.6 Software Metrics

- The following branches of testing started as parts of quality testing:
  - Reliability testing (Fenton and Pfleeger, 1997, p. 18, ch. 10)
  - Performance testing (Fenton and Pfleeger, 1997, p. 18, ch. 7)
- Reliability and maintainability can start to be tested even without code by "measur[ing] structural attributes of representations of the software" (Fenton and Pfleeger, 1997, p. 18)
- 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)
- 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) (Fenton and Pfleeger, 1997, p. 36)
  - These attributes must be defined before they can be measured (Fenton and Pfleeger, 1997, p. 38)

3.7 Software Testing

3.7.1 Methodology

It was realized early on in the process that it would be beneficial to understand the different types of testing (including what they test, what artifacts are needed to perform them, etc.). 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. These sources, as well as others, categorized these techniques 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.

Going forward, this process will be more rigorous, starting from more established sources of software testing terminology in the following order: (ISO/IEC and IEEE, 2022, 2017, 2013; ISO/IEC, 2023; IEEE, 2012; Bourque and Fairley, 2014; International Software Testing Qualifications Board, 2022).

should "author names" be acronyms or full?

Each test is made of at least one "approach", defined as a "high-level test implementation choice, typically made as part of the test strategy design activity" (ISO/IEC and IEEE, 2022, p. 10). This means that when looking at the different kinds of testing, each one is called an "approach". The different kinds of approach

include "test level, test type, test technique, test practice and the form of static testing to be used" (ISO/IEC and IEEE, 2022, p. 10), this provides a way to classify these testing approaches as one (or more) of the following kinds of approach:

- **(Design) Technique:** "procedure used to create or select a test model, identify test coverage items, and derive corresponding test cases" (e.g., equivalence partitioning, boundary value analysis, branch testing) (ISO/IEC and IEEE, 2022, p. 11)
- **Level:** A stage of testing "typically associated with the achievement of particular objectives and used to treat particular risks" (e.g., unit/component testing, integration testing, system testing) (ISO/IEC and IEEE, 2022, p. 12)
- **Practice:** A "conceptual framework that can be applied to …[a] test process to facilitate testing" (ISO/IEC and IEEE, 2022, p. 14) (e.g., scripted testing, exploratory testing, automated testing) (ISO/IEC and IEEE, 2022, p. 20)
- **Type:** "Testing that is focused on specific quality characteristics" (e.g., functional testing, usability testing, and performance testing) (ISO/IEC and IEEE, 2022, p. 15)

From ISO/IEC and IEEE (2022), many types of testing were mentioned but not defined. Additionally, there were some discrepancies or ambiguities (seven within just this source and four between other sources); 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 separate definitions of "co-existence"and "interoperability" being given in International Software Testing Qualifications Board (2022).
2. "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.
3. "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.
4. "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); however, on p. 22, it is listed as a subset of "performance-related testing", along with other approaches like load and capacity testing. It seems like the definition given should apply to "performance-related testing", and "performance testing" should be given a more specific definition, perhaps related to time?

5. Similarly, "performance" and "performance efficiency" are both listed as software qualities in as "degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy, or memory usage" (ISO/IEC and IEEE, 2017, p. 318) and "performance relative to the amount of resources used under stated conditions" (ISO/IEC and IEEE, 2017, p. 319), respectively. While the definition of "performance efficiency" doesn't seem to make any meaningful distinction between it and "performance", the term "performance testing" is defined (ISO/IEC and IEEE, 2017, p. 320) and used throughout ISO/IEC and IEEE (2017) while the term "performance efficiency testing" is used throughout ISO/IEC and IEEE (2017) (but not defined explicitly).

6. "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; how does it relate to functional testing?

7. 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?

8. 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?

9. "Functional testing" is given a loose definition of testing "often used to check the implementation of functional requirements" (ISO/IEC and IEEE, 2022, p. 21). While this source and International Software Testing Qualifications Board (2022) define this and "specification-based" testing as separate terms, van Vliet (2000, p. 399) lists them as synonyms.

10. "Disaster/recovery testing" and "recovery testing" (as a subset of performance-related testing) are both listed as types of testing (ISO/IEC and IEEE, 2022, p. 22) but not defined, making it unclear what distinguishes them. ISO/IEC and IEEE (2013, p. 2) defines "backup and recovery testing", which may be what is meant by "recovery testing" in the context of performance-related testing.

11. 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.

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

13. 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 testing level.

14. A component is an "entity with discrete structure …within a system considered at a particular level of analysis" (ISO/IEC, 2023) 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 this distinguishing the two may be necessary.

3.7.2 Software Testing Taxonomies, Ontologies, and 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).

There are many different concepts involved with software testing; (Barbosa et al., 2006) gives the following: testing process, testing phase, testing artifact, testing step, testing procedure, and testing resource (Barbosa et al., 2006, p. 2), as well as the relations between them (Barbosa et al., 2006, Fig. 2) (see also (Borges and Barbosa, 2009, Fig. 1), which I could only find as an example of an ontology in an otherwise unrelated paper). In addition to the concept of "test artifact", (Souza et al., 2017) also provides the following concepts: testing technique, test level (which seems to be comparable to "testing phase" from (Barbosa et al., 2006, p. 3)), and test environment (Souza et al., 2017, pp. 3-4). Particular things of note from these ontologies:

- The testing phases described by both are unit, integration, system, (Souza et al., 2017, p. 3), (Barbosa et al., 2006, p. 3), and regression testing (Barbosa

et al., 2006, p. 3)

- Testing artifacts 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)

- "Testing procedures can be categorized in testing methods, testing guidances and testing techniques", and "functional, structural, error-based and state-based" (called "black-box" "white-box", "defect-based", and "model-based" in (Souza et al., 2017, p. 3)) are examples of testing techniques (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 techniques themselves. For example, the terms and definitions (Tebes et al., 2020b) from TestTDO (Tebes et al., 2020a) provides *some* definitions of testing techniques, but mainly focuses 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) and also includes 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) provides 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 "does not aim at providing a systematic and exhaustive state-of-the-art survey of [either domain]" (Unterkalmsteiner et al., 2014, p. A:2).

Another source introduced the notion of an "intervention": "an act performed (e.g. use of a technique 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 to focus on characteristics of the intervention [while] industrial standards categorize the area from a

process perspective" (Engström and Petersen, 2015, p. 2). It provides a structure to "capture both a problem perspective and a solution perspective with respect to software testing" (Engström and Petersen, 2015, pp. 3-4), but this seems to focus more on test interventions and challenges rather than techniques (Engström and Petersen, 2015, Fig. 5).

Types of Testing Approaches

For classifying different types of tests, ISO/IEC and IEEE (2022) provides 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. It also may be helpful to introduce a "metric" categorization for things like error seeding, if they are determined to be within scope (see #21, #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)	any of the examples given below: equivalence partitioning, unit testing, scripted testing, functional testing
(Design) Technique	A "procedure used to create or select a test model, identify test coverage items, and derive corresponding test cases" (ISO/IEC and IEEE, 2022, p. 11)	equivalence partitioning, boundary value analysis, branch testing (ISO/IEC and IEEE, 2022, p. 11)
Level	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)	unit/component testing, integration testing, system testing (ISO/IEC and IEEE, 2022, p. 12)
Practice	A "conceptual framework that can be applied to …[a] test process to facilitate testing" (ISO/IEC and IEEE, 2022, p. 14)	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)	functional testing, usability testing, performance testing (ISO/IEC and IEEE, 2022, p. 15)

Table 3.2: Other Testing Terminology

Term	Definition	Examples	IEEE Equiv.
Guidance	none given (Barbosa et al., 2006, p. 3)	none given	Metric? Technique?
Level	none given	unit, integration, system testing (Souza et al., 2017, p. 3)	Level
Method	none given (Barbosa et al., 2006, p. 3)	none given	Practice?
Phase	none given (Barbosa et al., 2006, p. 3)	unit, integration, system, regression testing (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
Technique	The basis for how "to perform the tests in a systematic way and on a sound theoretical basis" (Barbosa et al., 2006, p. 3)	functional, structural, error-based, state-based testing (Barbosa et al., 2006, p. 3); black-box, white-box, defect-based, model-based (Souza et al., 2017, p. 3)	Technique

riginal
e for Souza-
017 tech-
examples:
ur (2012)

Categorizations

Software testing techniques can be divided into the following categories (note that these are not mutually exclusive):

- Execution of code: static or dynamic (Patton, 2006, p. 53)
- Visibility of code: black-, white-, or gray-box (functional, structural, or a mix of the two) (Patton, 2006, pp. 53, 218), (Perry, 2006, p. 69)
- Stage of testing: unit, integration, system, or acceptance (Patton, 2006), (Perry, 2006), (Peters and Pedrycz, 2000) (sometimes includes installation (van Vliet, 2000, p. 439) or regression (Barbosa et al., 2006, p. 3))
- Goal of testing: verification or validation (Perry, 2006, pp. 69-70)
- 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)
- 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 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 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)

3.7.3 Information Required for Different Types of Testing

The information contained in Table 3.3 outlines the required information for the types of testing listed in this document, as well as whether that information exists in Drasil already and if it can be added (or added *to*, in the case that it already exists).

Table 3.3: Testing Requirements

Testing Approach	Requirements	In Drasil?	Addable?
Unit testing	Code modules and their specifications	Yes	Yes
Integration testing	Code modules and their interfaces	Yes	???
System testing	Requirements specification; most of the code	Yes	Yes
Acceptance testing	Customer requirements and feedback	No	Partially
Installation testing	Algorithm for installation; environments to test in; method to check successful installation	Partially	Yes?

3.7.4 Definitions

Find original source: Myers 1976

- Software testing: "the process of executing a program with the intent of finding errors" (Peters and Pedrycz, 2000, p. 438)
- 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)
- 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)
- 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 (van Vliet, 2000, pp. 428-429)
- 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)

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)

3.7.5 General Testing Notes

- "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)
- 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

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

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" (International Software Testing Qualifications Board, 2022), 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)*

Q #2: Bring up!

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

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

Expand on reliability testing (make own section?)

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

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

Test Oracles

"A *test oracle* is a predicate that determines whether a given test activity sequence is an acceptable behaviour of the SUT [System Under Test] or not" (Barr et al., 2015, p. 509). They can either 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). They can be grouped into four categories:

- 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 is like the assertions we discussed earlier; documentation enforced by code!*

- 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!*
- 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), or crowdsourcing (Barr et al., 2015, p. 520)

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

3.7.6 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):

- **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)

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).

3.7.7 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

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).

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" (Peters and Pedrycz, 2000, p. 461).

Requirements

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

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).

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" (International Software Testing Qualifications Board, 2022).

Original source: ISO 25010?

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:

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

- 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)

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*)

3.7.8 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)

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 (Patton, 2006, pp. 96-99) and Generic Code Review Checklist (Patton, 2006, pp. 99-103))
- *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 (Patton, 2006, pp. 92-95), (van Vliet, 2000, pp. 415-417), (Peters and Pedrycz, 2000, pp. 482-485), but I didn't want to fight with more subsections yet

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 (Patton, 2006, pp. 92-95), (van Vliet, 2000, pp. 415-417), (Peters and Pedrycz, 2000, pp. 482-485), but I didn't want to fight with more subsections yet

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?

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

3.7.9 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).

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" (Peters and Pedrycz, 2000, p. 481)
  - Doesn't guarantee correctness (van Vliet, 2000, p. 421)
- Branch coverage: attempting to, "at each branching node in the control graph, …[choose] all possible branches …at least once" (van Vliet, 2000, p. 421)
  - Weaker than path coverage (van Vliet, 2000, p. 433), although (Patton, 2006, p. 119) says it is "the simplest form of path testing" (*I don't think this is true*)
  - Requires at least 85% coverage to be effective and is "most effective …at the module level" (Peters and Pedrycz, 2000, p. 481)
  - Cyclomatic-number criterion: an adequacy criterion that requires that "all linearly-independent paths are covered" (van Vliet, 2000, p. 423); results in complete branch coverage
  - Doesn't guarantee correctness (van Vliet, 2000, p. 421)

Find original source: Miller et al., 1994

Find original source: Miller et al., 1994

Find original source: Miller et al., 1994

Find original source: Miller et al., 1994

- 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 must be checked for full path coverage (Peters and Pedrycz, 2000, p. 439)!
  - Usually "limited to a few functions with life criticality features (medical systems, real-time controllers)" (Peters and Pedrycz, 2000, p. 481)

Find original source: 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)

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

In addition to Data Flow Coverage (Patton, 2006, p. 114), (van Vliet, 2000, pp. 424-425), there are also some minor forms of data coverage:

- Sub-boundaries: mentioned previously in 3.7.7
- 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)

get original source: Beizer, 1990

Table 3.4 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?

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

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

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

Table 3.4: 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

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).

add original source: KA85

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

3.7.10 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).

3.7.11 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

3.7.12 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).

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 (Kanewala and Yueh Chen, 2019, p. 67) (*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)

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)

3.8 Roadblocks to Testing

- Intractability: it is generally impossible to test a program exhaustively (van Vliet, 2000, p. 421), (Peters and Pedrycz, 2000, pp. 439, 461)
- 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?

3.8.1 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)
- "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.

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 `InstanceModel`s. Since this change required adding the `Referable` constraints to the output field of `SystemInformation`, the outputs of all examples needed to be updated to satisfy this constraint; this meant that generating the output requirement of each example was nearly trivial once the outputs were specified correctly. After modifying `DataDefinition`s in GlassBR that were outputs to be `InstanceModel`s (#3569; #3583), reorganizing the requirements of SWHS (#3589; #3607), and clarifying the outputs of SWHS (#3589), SglPend (#3533), DblPend (#3533), GamePhysics

cite Dr. Smith

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

add constraints

(#3609), and SSP (#3630), the output requirement was ready to be generated.

Bibliography

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.

Vanessa Borges and Ellen Barbosa. Using ontologies for modeling educational content. *SWEL*, 01 2009.

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.

Krzysztof Czarnecki. Overview of Generative Software Development. In Jean-Pierre Banâtre, Pascal Fradet, Jean-Louis Giavitto, and Olivier Michel, editors, *Unconventional Programming Paradigms*, Lecture Notes in Computer Science, pages 326–341, Le Mont Saint Michel, France, September 2004. Springer Berlin, Heidelberg. doi: https://doi.org/10.1007/11527800.

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, 2 edition, 1997. ISBN 0-534-95425-1.

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

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

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.

International Software Testing Qualifications Board. ISTQB Glossary, V4.2.1, 2022. URL https://glossary.istqb.org/en_US/search.

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 2023.

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.

Yannis Lilis and Anthony Savidis. A Survey of Metaprogramming Languages. In *ACM Computing Surveys*, volume 52, pages 113:1–40. Association for Computing Machinery, October 2019. doi: https://doi.org/10.1145/3354584.

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/.

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

William E. Perry. *Effective Methods for Software Testing*. Wiley Publishing, Inc., Indianapolis, IN, USA, 3 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.

Yannis Smaragdakis, Aggelos Biboudis, and George Fourtounis. Structured Program Generation Techniques. In Jácome Cunha, João P. Fernandes, Ralf Lämmel, João Saraiva, and Vadim Zaytsev, editors, *Grand Timely Topics in Software Engineering*, pages 154–178, Cham, 2017. Springer International Publishing. ISBN 978-3-319-60074-1.

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.

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.

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, 2 edition, 2000. ISBN 0-471-97508-7.

Vytautas Štuikys and Robertas Damaševičius. Taxonomy of Fundamental Concepts of Meta-Programming. In *Meta-Programming and Model-Driven Meta-Program Development: Principles, Processes and Techniques*, pages 17–29. Springer London, London, 2013. ISBN 978-1-4471-4126-6. doi: 10.1007/978-1-4471-4126-6_2. URL https://doi.org/10.1007/978-1-4471-4126-6_2.

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
```