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 (2023) 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: xiii, 30

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
HREF	Hypertext REFerence
IDE	Integrated Development Environment
MDD	Model-Driven Development
MOP	MetaObject Protocol
MSL	MultiStage Language
MSP	MultiStage Programming
PDF	Portable Document Format
PPTMP	PreProcessing-Time MetaProgramming
RTMP	RunTime MetaProgramming
SST	Skeleton Syntax Tree
DblPend	Double Pendulum
GamePhysics	Game Physics
Projectile	Projectile
SglPend	Single Pendulum
SSP	Slope Stability analysis Program

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. formatting the LaTeX code to make it easier to read (the LaTeX Workshop plugin has functionality for this),

[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.

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 [1, p. 113:6]
- Allows for improvements in:
  - "performance by generating efficient specialized programs based on specifications instead of using generic but inefficient programs" [1, 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" [1, p. 113:2]
  - code reuse through capturing "code patterns that cannot be abstracted" [1, 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" [1, p. 113:1]. "It constitutes a flexible and powerful reuse solution for the ever-growing size and complexity of software systems" [1, p. 113:31].

- Metalanguage: "the language in which the metaprogram is written" [1, p. 113:1]
- Object language: "the language in which the generated or transformed program is written" [1, p. 113:1]
- Homogeneous metaprogramming: when "the object language and the metalanguage are the same" [1, p. 113:1]
- Heterogeneous metaprogramming: when "the object language and the metalanguage are ... different" [1, p. 113:1]

3.1.2 Metaprogramming Models

Macro Systems [1, p. 113:3-7]

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

Lexical Macros

- Language agnostic [1, 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" [1, 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") [1, p. 113:6]

Syntactic Macros

- "Aware of the language syntax and semantics" [1, 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" [1, p. 113:5]

Reflection Systems [1, 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" [1, p. 113:7]
- Requires that the system can examine ("introspection") and modify ("intercession") how it is represented [1, p. 113:7]
  - The representation of a system can either be structural or behavioural (e.g., variable assignment) [1, 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" [1, p. 113:8]
- "Not limited to runtime systems", as some "compile-time systems ... rely on some form of structural introspection to perform code generation" [1, p. 113:9]

MetaObject Protocols (MOPs) [1, p. 113:9-11]

- "Interfaces to the language enabling one to incrementally transform the original language behavior and implementation" [1, 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" [1, p. 113:9]
  - Metaobject-based Approach: "Classes and metaobjects are distinct" [1, p. 113:9]
  - Message Reification Approach: used with message passing [1, 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" [1, 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" [1, 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" [1, p. 113:10].

Aspect-Oriented Programming (AOP) [1, 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*" [1, p. 113:12]
- Weaving: the process of "combining the base program with aspect code ... [to form] the final code" [1, 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 [1, 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 [1, 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") [1, 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" [1, p. 113:12]

Generative Programming [1, 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 [1, p. 113:31]
- Most "support code templates and quasi-quotes" [1, p. 113:31]
- Related to macro systems, but normal code and metacode are distinct

Template Systems [1, 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" [1, p. 113:13]

clarify what "free-form source code generation" means

- 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 [1, p. 113:14]

AST Transformations [1, 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" [1, p. 113:14]

Compile-Time Reflections [1, 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") [1, p. 113:15]

Class Compositions [1, p. 113:16-17]

- Offer "flexibility and expressiveness" through composition approaches [1, 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 [1, p. 113:16]

Investigate

Investigate?

- Includes *feature-oriented programming* approaches

MultiStage Programming (MSP) [1, 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 [1, p. 113:17]
- Related to program generation and procedural macro systems [1, p. 113:17]; macros are often implemented as multistage computations [1, 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 [1, p. 113:17]; MSLs are more common [1, p. 113:31]
  - C++ first instantiates templates, then translates nontemplate code [1, 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 [1, p. 113:19]
- Often involves *Cross-Stage Persistence (CSP)*, which allows "values . . . available in the current stage" to be used in future stages [1, 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" [1, p. 113:17]
- Usually homogeneous, but there are exceptions; MetaHaskell, a modular framework [1, p. 113:19] with a type system, allows for "heterogeneous metaprogramming with multiple object languages" [1, p. 113:18]
- "Type safety . . . comes at the cost of expressiveness" [1, 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" [1, 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" [1, p. 113:20]

Preprocessing-Time Evaluation [1, 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" [1, p. 113:20]
- These systems are called *source-to-source preprocessors* [1, p. 113:20] and are usually examples of generative programming [1, p. 113:21]
  - "All such cases involve syntactic transformations" [1, p. 113:21], usually using ASTs
- "Translation can reuse the language compiler or interpreter without the need for any extensions" [1, p. 113:20]
- Varying levels of complexity (e.g., these systems "may be fully aware of the language syntax and semantics") [1, p. 113:20]
- Includes all lexical macro systems [1, p. 113:20] and some "static AOP and generative programming systems" [1, p. 113:31]
- Typically doesn't use reflection (Reflective Java is an exception), MOPs, or dynamic AOP [1, p. 113:21]

Compilation-Time Evaluation [1, p. 113:21-23]

- In Compile-Time MetaProgramming (CTMP), "the language compiler is extended to handle metacode translation and execution" [1, p. 113:22]
  - There are many ways of extending the compiler, including "plugins, syntactic additions, procedural or rewrite-based AST transformations, or multistage translation" [1, p. 113:22]
  - Metacode execution can be done by "interpreting the source metacode ... or compiling the source metacode to binary and then executing it" [1, p. 113:22]
- These systems are usually examples of generative programming but can also use macros, MOPs, AOP [1, p. 113:22], and/or reflection [1, p. 113:23]

Execution-Time Evaluation [1, 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" [1, p. 113:23]
- Includes "most reflection systems, MOPs, MSP systems, and dynamic AOP systems" [1, p. 113:31]

3.1.4 Metaprogram Source Location

Embedded in the Subject Program [1, p. 113:25-26]

- Usually occurs with macros, templates, MSLs, reflection, MOPs, and AOP [1, p. 113:25]

Context Unaware [1, p. 113:25]

- Occurs when metaprograms only need to know their input parameters to generate ASTs [1, p. 113:25]
- Very common: supported by "most CTMP systems" [1, p. 113:31] and "for most macro systems. . . , generative programming systems . . . and MSLs . . . it is the only available option" [1, p. 113:25]

Context Aware [1, 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" [1, p. 113:25]
- Allows for code transformation "at multiple different locations reachable from the initial context" [1, p. 113:25]
- Very uncommon [1, p. 113:25, 31]

Global [1, p. 113:26]

- Involves "scenarios that collectively introduce, transform, or remove functionality for the entire program" [1, p. 113:26]
- Usually occurs with reflection, MOPs, and AOP [1, p. 113:26]; offered by "most RTMP systems" [1, p. 113:31]
- Can be used with "any PPTMP or CTMP system that provides access to the full program AST" [1, p. 113:26]
- "Can also be seen as a context-aware case where the context is the entire program" [1, p. 113:26]

External to the Subject Program [1, 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" [1, p. 113:27]
- Includes many instances of AOP [1, 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*) [1, p. 113:27]
- Sometimes the metalayer of a language is added to a language later, independently of the object language [1, p. 113:27]

Metalanguage Indistinguishable from the Object Language [1, 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" [1, p. 113:28]
     - Includes many examples of MOPs and AOP [1, p. 113:28]

Metalanguage Extends the Object Language [1, p. 113:29]

can I do this if it's said later?

- Allows for reuse of "the original [and well-known] language features in developing metaprograms . . . instead of adopting custom programming constructs" [1, p. 113:29]
- "Typically involve new syntax and functionality used to differentiate normal code from metacode" [1, p. 113:29]
- Often used in quasi-quote constructs, two-stage and multistage languages, and MOPs [1, p. 113:29]
- Used with MSLs "as the base languages are extended with staging annotations to deliver MSP functionality" [1, p. 113:31]

Metalanguage Different from the Object Language [1, 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" [1, 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 [1, p. 113:30]
- Used by some AOP and generative metaprogramming systems [1, 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" [2, 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)]" [2, p. 327]. "DSLs come in a wide variety of forms, ... [including] textual ... [and] diagrammatic" [2, p. 328].

"System family engineering distinguishes between at least two kinds of development processes: *domain engineering* and *application engineering*" [2, p. 328]. "Domain engineering ... is concerned with the development of reusable assets such as components, generators, DSLs, analysis and design models, user documentation, etc." [2, p. 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" [2, p. 329]. Application engineering includes "requirements elicitation, analysis, and specification" and "the manual or automated construction of the system from the reusable assets" [2, 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 [2, 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" [2, 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" [2, 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" [2, 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" [2, 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" [2, p. 331]
  - Transformational view: "a problem space is represented by a ... [DSL], whereas the solution space is represented by an implementation language" [2, p. 331]. "A program in a ... [DSL]" is transformed into "its implementation in the implementation language" [2, p. 331]
- Problem space: "a set of domain-specific abstractions that can be used to specify the desired system-family member" [2, p. 330]
- Solution space: "consists of implementation-oriented abstractions, which can be instantiated to create implementations of the [desired] specifications" [2, 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" [2, p. 332-333]
- Feature modeling: "a method and notation to elicit and represent common and variable features of the systems in a system family" [2, p. 333]. Can be used during domain analysis as "the starting point in the development of both system-family architecture and DSLs" [2, 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" [2, p. 336]. Often uses DSLs and sometimes deals with system families, making it related to generative software development [2, p. 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 [3, p. 1]

Can I rearrange quotes like this?

- "Generators are often the technique of last resort . . . to automate, elevate, modularize or otherwise facilitate program development" [3, p. 2]
- Why is it beneficial "to statically check the generator and be sure that no type error arises during its *run time*" [3, 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" [3, 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" [3, p. 6]

3.3.1 Techniques for Program Generation [3, p. 3-5]

1. Generation as text: "producing character strings containing the text of a program, which is subsequently interpreted or compiled" [3, 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" [3, p. 4]
4. Macros: "reusable code templates with pre-set rules for parameterizing them" [3, p. 4]
5. Generics: Mechanisms with "the ability to parameterize a code template with different static types" [3, p. 5]
6. Specialized languages: Languages with specific features for program generators, such as AOP and *inter-type declarations* [3, p. 5]

3.3.2 Kinds of Generator Safety [3, p. 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" [3, 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" [3, p. 8])

3.3.3 Methods for Guaranteeing Fully Structured Generation [3, p. 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* [3, p. 9]
2. Class Morphing: similar to MetaObject Protocols (MOPs)?
3. Reflection: (e.g., SafeGen [3, 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) [3, p. 16]
5. Macro systems, although "safety guarantees carry the cost of some manual verification effort by the programmer" [3, p. 19]

3.4 Software Metrics

- The following branches of testing started as parts of quality testing:
  - Reliability testing [4, p. 18, ch. 10]
  - Performance testing [4, 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" [4, p. 18]
- The US Software Engineering Institute has a checklist for determining which types of lines of code are included when counting [4, 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) [4, p. 36]
  - These attributes must be defined before they can be measured [4, p. 38]

3.5 Software Testing

3.5.1 General Testing Notes

- Simple, normal test cases (test-to-pass) should always be developed and run before more complicated, unusual test cases (test-to-fail) [5, p. 66]

3.5.2 Types of Testing

Static Black-Box (Specification) Testing [5, p. 56-62]

Most of this section is irrelevant to generating test cases, as they require human involvement (e.g., Pretend to Be the Customer [5, p. 57-58], Research Existing Standards and Guidelines [5, p. 58-59]). However, it provides a "Specification Terminology Checklist" [5, 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")

Dynamic Black-Box (Behavioural) Testing [5, p. 64-65]

This is the process of "entering inputs, receiving outputs, and checking the results" [5, p. 64]. Note that while black-box testing is usually done at a higher (e.g., system) level, unit testing can also be black-box [6, p. 1].

Requirements

- Requirements documentation (definition of what the software does) [5, p. 64]; relevant information could be:
  - Requirements: Input-Values and Output-Values
  - Input/output data constraints

Exploratory Testing [5, p. 65]

An alternative to dynamic black-box testing when a specification is not available [5, p. 65]. The software is explored to determine its features, and these features are then tested [5, p. 65]. Finding any bugs using this method is a positive thing [5, 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 [7].

Equivalence Partitioning/Classing [5, p. 67-69]

The process of dividing the infinite set of test cases into a finite set that is just as effective (i.e., by revealing the same bugs) [5, p. 67].

Requirements

- Ranges of possible values [5, p. 67]; could be obtained through:
  - Input/output data constraints
  - Case statements

Data Testing [5, p. 70-79]

The process of "checking that information the user inputs [and] results", both final and intermediate, "are handled correctly" [5, p. 70].

Boundary Conditions [5, p. 70-74] "[S]ituations at the edge of the planned operational limits of the software" [5, p. 72]. Often affects types of data (e.g., numeric, speed, character, location, position, size, quantity [5, 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 [5, p. 72-73]). Data at these boundaries should be included in an equivalence partition, but so should data in between them [5, 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" [5, p. 73].

Requirements

- Ranges of possible values [5, 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 [5, p. 201-205] *Buffer overruns* are "the number one cause of software security issues" [5, 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 [5, 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" [5, p. 201]. Any unsafe versions of these functions that are used should be replaced with the corresponding safe versions [5, p. 203-204].

Sub-Boundary Conditions [5, p. 75-77] Boundary conditions "that are internal to the software [but] aren't necessarily apparent to an end user" [5, p. 75]. These include powers of two [5, p. 75-76] and ASCII and Unicode tables [5, p. 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 [7]. 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.

Requirements

- Increased knowledge of data type structures (e.g., monoids, rings, etc. [7]); 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 [5, p. 77-78] These should be their own equivalence class, since "the software usually handles them differently" than "the valid cases or . . . invalid cases" [5, 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 [7].

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 [5, p. 78-79] This is testing-to-fail [5, 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)

State Testing [5, p. 79-87]

The process of testing "a program's states and the transitions between them" [5, p. 79].

Logic Flow Testing [5, p. 80-84] This is 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 [5, p. 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 [5, p. 82-83]

For all of these tests, the values of the state variables should be verified [5, p. 83].

Requirements

- Knowledge of the different states of the program [5, p. 82]; could be obtained through:
  - The program's modules and/or functions
  - The program's exceptions
- Knowledge about the different state transitions [5, p. 82]; could be obtained through:
  - Testing the state transitions near the beginning of a workflow more?

Testing States to Fail [5, p. 84-87] The goal here is to try and put the program in a fail state by doing things that are out of the ordinary. These include:

- Race Conditions and Bad Timing [5, p. 85-86] (Is this relevant to our examples?)
- Repetition Testing: "doing the same operation over and over", potentially up to "thousands of attempts" [5, p. 86]
- Stress Testing: "running the software under less-than-ideal conditions" [5, p. 86]
- Load testing: running the software with as large of a load as possible (e.g., large inputs, many peripherals) [5, 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?
- 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 [5, p. 87-89]

- Act like an inexperienced user (likely cannot be generated by Drasil)
- Look for bugs where they've already been found (keep track of previous failed test cases?)
- Think like a hacker (is this out of scope?)
- Follow experience (this will implicitly be done just by using Drasil)

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

cite Dr. Smith

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

add constraints

(#3533), GamePhysics (#3609), and SSP (#3630), the output requirement was ready to be generated.

Bibliography

[1] Yannis Lilis and Anthony Savidis. “A Survey of Metaprogramming Languages”. In: *ACM Computing Surveys*. Vol. 52. Association for Computing Machinery, 2019-10, 113:1–40. DOI: https://doi.org/10.1145/3354584 (cit. on pp. 7–15).

[2] Krzysztof Czarnecki. “Overview of Generative Software Development”. In: *Unconventional Programming Paradigms*. Ed. by Jean-Pierre Banâtre, Pascal Fradet, Jean-Louis Giavitto, and Olivier Michel. Lecture Notes in Computer Science. Le Mont Saint Michel, France: Springer Berlin, Heidelberg, 2004-09, pp. 326–341. DOI: https://doi.org/10.1007/11527800 (cit. on pp. 15, 16).

[3] Yannis Smaragdakis, Aggelos Biboudis, and George Fourtounis. “Structured Program Generation Techniques”. In: *Grand Timely Topics in Software Engineering*. Ed. by Jácome Cunha, João P. Fernandes, Ralf Lämmel, João Saraiva, and Vadim Zaytsev. Cham: Springer International Publishing, 2017, pp. 154–178. ISBN: 978-3-319-60074-1 (cit. on pp. 16–18).

[4] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous & Practical Approach*. 2nd ed. Boston, MA, USA: PWS Publishing Company, 1997. ISBN: 0-534-95425-1 (cit. on p. 18).

[5] Ron Patton. *Software Testing*. 2nd ed. Indianapolis, IN, USA: Sams Publishing, 2006. ISBN: 0-672-32798-8 (cit. on pp. 18–23).

[6] Pramod Mathew Jacob and M. Prasanna. “A Comparative analysis on Black Box Testing Strategies”. In: *2016 International Conference on Information Science (ICIS)*. Manhattan, NY, USA: IEEE, 2016-08, pp. 1–6. DOI: 10.1109/INFOSCI.2016.7845290. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7845290 (visited on 2023-07-05) (cit. on p. 19).

[7] W. Spencer Smith and Jacques Carette. Private Communication. Hamilton, ON, Canada, 2023-07 (cit. on pp. 19, 21).

Appendix

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

```
# from
↪  https://stackoverflow.com/questions/54071312/how-to-pass-command-line-argu
## \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-dele
    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
```