Todo list

Contents

List of Figures

List of Tables

List of Source Codes

Chapter 1

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
MDD	Model-Driven Development
MOP	MetaObject Protocol
MR	Metamorphic Relation
MSL	MultiStage Language
MSP	MultiStage Programming
MT	Metamorphic Testing
PPTMP	PreProcessing-Time MetaProgramming
RTMP	RunTime MetaProgramming
SST	Skeleton Syntax Tree
C-use	Computational Use
P-use	Predicate Use
Projectile	Projectile
V&V	Verification and Validation

Chapter 2

Notes

2.1 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]

2.1.1 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]

2.1.2 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]
- Means that the system can “observe and possibly modify its structure and behaviour” [2, p. 22]; these processes are called “introspection” and “intercession”, respectively [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*:

    Investigate
  - *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]
- Includes *feature-oriented programming* approaches

  Investigate?

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]

2.1.3 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]

2.1.4 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]

2.1.5 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]

- Allows for reuse of "the original language['s] …well-known features 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]

2.2 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” [3, 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)]” [3, p. 327]. “DSLs come in a wide variety of forms, …[including] textual …[and] diagrammatic” [3, p. 328].

“System family engineering distinguishes between at least two kinds of development processes: *domain engineering* and *application engineering*” [3, p. 328]. “Domain engineering …is concerned with the development of reusable assets such as components, generators, DSLs, analysis and design models, user documentation, etc.” [3, 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” [3, p. 329]. Application engineering includes “requirements elicitation, analysis, and specification” and “the manual or automated construction of the system from the reusable assets” [3, 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 [3, 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” [3, 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” [3, p. 338].

2.2.1 Definitions

- Generative domain model: “a mapping between *problem space* and *solution space*” which “takes a specification and returns the corresponding implementation” [3, 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” [3, 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” [3, p. 331]
  - Transformational view: “a problem space is represented by a …[DSL], whereas the solution space is represented by an implementation language” [3, p. 331]. “A program in a …[DSL]” is transformed into “its implementation in the implementation language” [3, p. 331]

- Problem space: “a set of domain-specific abstractions that can be used to specify the desired system-family member” [3, p. 330]

- Solution space: “consists of implementation-oriented abstractions, which can be instantiated to create implementations of the [desired] specifications” [3, 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” [3, pp. 332-333]

- Feature modeling: “a method and notation to elicit and represent common and variable features of the systems in a system family” [3, p. 333]. Can be used during domain analysis as “the starting point in the development of both system-family architecture and DSLs” [3, 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” [3, p. 336]. Often uses DSLs and sometimes deals with system families, making it related to generative software development [3, pp. 336-337]

2.3 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 [4, p. 1]

- Generators are used “to automate, elevate, modularize or otherwise facilitate program development” [4, p. 2]

- Why is it beneficial “to statically check the generator and be sure that no type error arises during its *run time*” [4, 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" [4, 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" [4, p. 6]

2.3.1 Techniques for Program Generation [4, pp. 3-5]

1. Generation as text: "producing character strings containing the text of a program, which is subsequently interpreted or compiled" [4, 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" [4, p. 4]

4. Macros: "reusable code templates with pre-set rules for parameterizing them" [4, p. 4]

5. Generics: Mechanisms with "the ability to parameterize a code template with different static types" [4, p. 5]

6. Specialized languages: Languages with specific features for program generators, such as AOP and *inter-type declarations* [4, p. 5]

2.3.2 Kinds of Generator Safety [4, 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" [4, 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" [4, p. 8])

2.3.3 Methods for Guaranteeing Fully Structured Generation [4, 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* [4, p. 9]

2. Class Morphing: similar to MetaObject Protocols (MOPs)?
3. Reflection: (e.g., SafeGen [4, 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) [4, p. 16]
5. Macro systems, although "safety guarantees carry the cost of some manual verification effort by the programmer" [4, p. 19]

2.4 Taxonomy of Fundamental Concepts of Meta-Programming

2.4.1 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" [2, p. 18]
  - It may or may not preserve the program's semantics [2, p. 18]
  - In metaprogramming, "the transformation algorithm describes generation of a particular instance depending upon values of the generic parameters" [2, 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" [2, 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" [2, 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" [2, p. 19]
- Generative component: a generic component that has "explicitly added generative technology" [2, p. 24]
- Separation of concerns: "the process of breaking a design problem into distinct tasks that are orthogonal and can be implemented separately" [2, p. 21]

2.4.2 Other Notes

should I change this if it's British English?

- 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 [2, 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" [2, pp. 24-25]

2.5 Roadblocks to Meta-Programming

- "Generators are often the technique of last resort" [4, 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" [2, p. 26]

British

- 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" [2, 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") [2, 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 [2, p. 27]
- "Modification of the generated code usually removes the program from the scope of the meta-programming system" [2, p. 27]

2.6 Software Metrics

- The following branches of testing started as parts of quality testing:
  - Reliability testing [5, p. 18, ch. 10]
  - Performance testing [5, 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" [5, p. 18]
- The US Software Engineering Institute has a checklist for determining which types of lines of code are included when counting [5, 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) [5, p. 36]
  - These attributes must be defined before they can be measured [5, p. 38]

2.7 Software Testing

While [6] groups testing into static/dynamic and white-/black-box, [7, pp. 398-399] gives a classification "according to the criterion used to measure the adequacy of a set of test cases": coverage-based, fault-based (includes Mutation Testing [7, pp. 428-429]), and error-based ("based on knowledge of the typical errors that people make"). Testing approaches can also be grouped based on where test data comes from and can be specification, implementation, or error oriented [8, p. 440].

2.7.1 Information Required for Different Types of Testing

The information contained in Table 2.1 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).

Should I be including citations in this table?

Table 2.1: 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?

2.7.2 Definitions

- Software testing: "the process of executing a program with the intent of finding errors" [8, p. 438]

Find original source: Myers 1976

- Error: "a human action that produces an incorrect result" [7, p. 399]

- Fault: “the manifestation of an error” in the software itself [7, 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” [7, 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” [7, 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” [7, p. 400]
- Test Suite Reduction: the process of reducing the size of a test suite while maintaining the same coverage [9, p. 519]; can be accomplished through Mutation Testing [7, 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” [9, p. 519]
- Probe: “a statement inserted into a program” for the purpose of dynamic testing [8, p. 438]

Documentation

- Verification and Validation (V&V) Plan: a document for the “planning of test activities” described by IEEE Standard 1012 [7, 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 [7, pp. 412-413]; should also outline entry and exit conditions for the testing activities as well as any risk sources and levels [8, p. 445]
- Test Design documentation: “specifies …the details of the test approach and identifies the associated tests” [7, p. 413]
- Test Case documentation: “specifies inputs, predicted outputs and execution conditions for each test item” [7, p. 413]
- Test Procedure documentation: “specifies the sequence of actions for the execution of each test” [7, p. 413]
- Test Report documentation: “provides information on the results of testing tasks”, addressing software verification and validation reporting [7, p. 413]

2.7.3 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” [7, p. 398]

- If faults exist in programs, they “must be considered faulty, even if we cannot devise test cases that reveal the faults” [7, 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” [6, p. 113]

- Simple, normal test cases (test-to-pass) should always be developed and run before more complicated, unusual test cases (test-to-fail) [6, p. 66]

- Since “there is no uniform best test technique”, it is advised to use many techniques when testing [7, 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) [7, p. 432]; this relation only “compares the thoroughness of test techniques, not their ability to detect faults” [7, p. 434]

This should probably be explained after “test adequacy criterion” is defined

Steps to Testing [8, 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/module testing: “testing the individual modules [of a program]” [7, p. 438]; also called “component testing” [8, p. 444]

- Integration testing: “testing the composition of modules”; done incrementally using *bottom-up* and/or *top-down* testing [7, pp. 438-439], although other paradigms for design, such as *big bang* and *sandwich* exist [8, p. 489]

  – Bottom-up testing: uses *test drivers*: “tool[s] that generate[] the test environment for a component to be tested” [7, p. 410] by “sending test-case data to the modules under test, read[ing] back the results, and verify[ing] that they’re correct” [6, p. 109]

  - Top-down testing: uses *test stubs*: tools that "simulate[] the function of a component not yet available" [7, p. 410] by providing "fake" values to a given module to be tested [6, p. 110]
  - Big bang testing: the process of "integrat[ing] all modules in a single step and test[ing] the resulting system[]" [8, p. 489]. *Although this is "quite challenging and risky" [8, 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" [8, p. 490]*

Q #1: Bring up!

  - Sandwich testing: "combines the ideas of bottom-up and top-down testing by defining a certain target layer in the hierarchy of the modules" and working towards it from either end using the relevant testing approach [8, p. 491]

- System testing: "test[ing] the whole system against the user documentation and requirements specification after integration testing has finished" [7, p. 439] ([6, 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 [7, 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 [7, p. 439] and the needs of the customer(s) [8, 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" [7, p. 439]; not one of the four levels of testing identified by the IEEE standard [8, 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" [9, p. 509]. They can either be "deterministic" (returning a Boolean value) or "probabilistic" (returning "a real number in the closed interval [0, 1]") [9, p. 509]. Probabilistic test oracles can be used to reduce the computation cost (since test oracles are "typically computationally expensive") [9, 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" [9, 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" [9, p. 510]

British

- 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) [9, p. 510], and invariant detection (either known in advance or "learned from the program") [9, 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.) [9, 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.) [9, p. 510]
- "Lack of an automated test oracle": for example; a human oracle generating sample data that is "realistic" and "valid", [9, pp. 510-511], or crowdsourcing [9, 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" [7, p. 402]

Investigate

- "Test data generators" are mentioned on [7, p. 410] but not described

2.7.4 Static Black-Box (Specification) Testing [6, 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 [6, pp. 57-58], Research Existing Standards and Guidelines [6, pp. 58-59]). However, it provides a "Specification Terminology Checklist" [6, 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 [7, p. 408]

Coverage-Based Testing of Specification [7, 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 [7, 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 [7, p. 426].

2.7.5 Dynamic Black-Box (Behavioural) Testing [6, pp. 64-65]

This is the process of "entering inputs, receiving outputs, and checking the results" [6, p. 64]. [7] also calls this "functional testing".

Requirements

- Requirements documentation (definition of what the software does) [6, p. 64]; relevant information could be:
  - Requirements: Input-Values and Output-Values
  - Input/output data constraints

Exploratory Testing [6, p. 65]

An alternative to dynamic black-box testing when a specification is not available [6, p. 65]. The software is explored to determine its features, and these features are then tested [6, p. 65]. Finding any bugs using this method is a positive thing [6, 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 [10].

Equivalence Partitioning/Classing [6, 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) [6, p. 67]. The opposite of this, testing every combination of inputs, is called "exhaustive testing" and is "probably not feasible" [8, p. 461].

Requirements

- Ranges of possible values [6, p. 67]; could be obtained through:
  - Input/output data constraints
  - Case statements

Data Testing [6, pp. 70-79]

The process of "checking that information the user inputs [and] results", both final and intermediate, "are handled correctly" [6, p. 70]. This type of testing can also occur at the white-box level, such as the implementation of boundaries [7, p. 431] or intermediate values within components.

Boundary Conditions [6, pp. 70-74] "[S]ituations at the edge of the planned operational limits of the software" [6, p. 72]. Often affects types of data (e.g., numeric, speed, character, location, position, size, quantity [6, 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 [6, pp. 72-73]). Data at these boundaries should be included in an equivalence partition, but so should data in between them [6, 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" [6, 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 [8, p. 460].

Requirements

- Ranges of possible values [6, 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 [6, pp. 201-205] *Buffer overruns* are "the number one cause of software security issues" [6, 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 [6, 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" [6, p. 201]. Any unsafe versions of these functions that are used should be replaced with the corresponding safe versions [6, pp. 203-204].

Sub-Boundary Conditions [6, pp. 75-77] Boundary conditions "that are internal to the software [but] aren't necessarily apparent to an end user" [6, p. 75]. These include powers of two [6, pp. 75-76] and ASCII and Unicode tables [6, 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 [10]. 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 [7, p. 430]. These “error prone” points around these boundaries should be tested [7, p. 430] as before: “the valid data just inside the boundary, …the last possible valid data, and …the invalid data just outside the boundary” [6, 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 [7, p. 430].

Requirements

- Increased knowledge of data type structures (e.g., monoids, rings, etc. [10]); 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 [6, pp. 77-78] These should be their own equivalence class, since “the software usually handles them differently” than “the valid cases or …invalid cases” [6, 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 [10].

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 [6, pp. 78-79] This is testing-to-fail [6, 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 [6, pp. 79-87]

The process of testing “a program’s states and the transitions between them” [6, p. 79].

Logic Flow Testing [6, pp. 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 [6, 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 [6, pp. 82-83]

For all of these tests, the values of the state variables should be verified [6, p. 83].

Requirements

- Knowledge of the different states of the program [6, p. 82]; could be obtained through:
  - The program’s modules and/or functions
  - The program’s exceptions
- Knowledge about the different state transitions [6, p. 82]; could be obtained through:
  - Testing the state transitions near the beginning of a workflow more?

Performance Testing “The intent of this type of testing is to identify weak points of a software system and quantify its shortcomings” [8, p. 447].

Testing States to Fail [6, pp. 84-87] The goal here is to try and put the program in a fail state by doing things that are out of the ordinary. These include:

- Race Conditions and Bad Timing [6, pp. 85-86] (Is this relevant to our examples?)
- Repetition Testing: "doing the same operation over and over", potentially up to "thousands of attempts" [6, p. 86]
- Stress Testing: "running the software under less-than-ideal conditions" [6, p. 86]
- Load testing: running the software with as large of a load as possible (e.g., large inputs, many peripherals) [6, 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 [6, 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*)

2.7.6 Static White-Box Testing (Structural Analysis) [6, pp. 91-104]

White-box testing is also called "glass box testing" [8, p. 439]. [8, 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 #2: Is this true?

There are also some more specific categories of this, such as Scenario-Based Evaluation [7, pp. 417-418] and Stepwise Abstraction [7, 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" [6, 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" [6, p. 91]; often avoided because of "the misconception that it's too time-consuming, too costly, or not productive" [6, p. 92]
- Especially effective early on in the development process [6, 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" [6, p. 92]
- Largely "done by the language compiler" or by separate tools [7, pp. 413-414]

Reviews [6, pp. 92-95], [7, pp. 415-417], [8, pp. 482-485]

- "The process under which static white-box testing is performed" [6, 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. [6, 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" [6, pp. 93-94]
- Many forms:
  - Peer Review: The most informal and at the smallest scale [6, p. 94]. One variation is to have each person 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 [7, p. 414]

  - Walkthrough: The author of the code presents it line by line to a small group that “question anything that looks suspicious” [6, p. 95]; this is done by using test data to “walk through” the execution of the program [7, p. 416]. A more structured walkthrough may have specific roles (presenter, coordinator, secretary, maintenance oracle, standards bearer, and user representative) [8, p. 484]
  - Inspection: Someone who is *not* the author of the code presents it to a small group of people [6, p. 95]; the author should be “a largely silent observer” who “may be consulted by the inspectors” [7, p. 415]. Each member has a role, which may be tied to a different perspective (e.g., designer, implementer, tester, [8, p. 439] user, or product support person) [6, p. 95]. Changes are made based on issues identified *after* the inspection [7, p. 415], and a reinspection may take place [6, p. 95]; one guideline is to reinspect *100%* of the code “[i]f more than 5% of the material inspected has been reworked” [8, p. 483].

- *Could be used to evaluate Drasil and/or generated code, but couldn’t be automated due to the human element*

Coding Standards and Guidelines [6, 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 [6, p. 96]
- These guidelines can range in strictness and formality, as long as they are agreed upon and followed [6, p. 96]
- This could be checked using linters

Generic Code Review Checklist [6, 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 [6, p. 99]
- Data declaration errors: bugs “caused by improperly declaring or using variables or constants” [6, p. 100]
- Computation errors: “essentially bad math”; e.g., type mismatches, over/underflow, zero division, out of meaningful range [6, p. 101]
- Comparison errors: “very susceptible to boundary condition problems”; e.g., correct inclusion, floating point comparisons [6, p. 101]
- Control flow errors: bugs caused by “loops and other control constructs in the language not behaving as expected” [6, p. 102]

- Subroutine parameter errors: bugs "due to incorrect passing of data to and from software subroutines" [6, p. 102] (could also be called "interface errors" [7, p. 416])
- Input/output errors: e.g., how are errors handled? [6, pp. 102-103]
- ASCII character handling, portability, compilation warnings [6, 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 [7, pp. 418-419]

Requires a formal specification [7, p. 418] and uses "highly formal methods of logic" [8, p. 438] to prove the existence of "an equivalence between the program and its specification" [8, p. 485]. It is not often used and its value is "sometimes disputed" [7, 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

2.7.7 Dynamic White-Box (Structural) Testing [6, 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" [6, p. 105].

Code Coverage [6, pp. 117-121] or Control-Flow Coverage [7, pp. 421-424]

"[T]est[ing] the program's states and the program's flow among them" [6, p. 117]; allows for redundant and/or missing test cases to be identified [6, 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 …" [7, 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]" [7, 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 [7, 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 [7, p. 421]).

- Statement/line coverage: attempting to "execute every statement in the program at least once" [6, p. 119]
  - Weaker than [7, p. 421] and "only about 50% as effective as branch coverage" [8, p. 481]
  - Requires 100% coverage to be effective [8, p. 481]
  - "[C]an be used at the module level with less than 5000 lines of code" [8, p. 481]
  - Doesn't guarantee correctness [7, p. 421]

Find original source: Miller et al., 1994

Find original source: Miller et al., 1994

Find original source: Miller et al., 1994

- Branch coverage: attempting to, "at each branching node in the control graph, …[choose] all possible branches …at least once" [7, p. 421]
  - Weaker than path coverage [7, p. 433]
  - Requires at least 85% coverage to be effective and is "most effective …at the module level" [8, p. 481]
  - Cyclomatic-number criterion: an adequacy criterion that requires that "all linearly-independent paths are covered" [7, p. 423]; results in complete branch coverage
  - Doesn't guarantee correctness [7, p. 421]

Find original source: Miller et al., 1994

- Path coverage: "[a]ttempting to cover all the paths in the software" [6, 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 [7, p. 420])

Q #3: 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 [8, pp. 471-473]; this is less feasible if a loop is present [8, pp. 473-476] since "a loop often results in an infinite number of possible paths" [7, p. 421]
  - If done completely, it "is equivalent to exhaustively testing the program" [7, p. 421]; however, this seems to overlook the effect of inputs on behaviour as pointed out in [8, pp. 466-467]

Q #4: Bring up!

  - Generally "not possible" to achieve completely due to the complexity of loops, branches, and potentially unreachable code [7, p. 421]; even infeasible paths must be checked for full path coverage [8, p. 439]!
  - Usually "limited to a few functions with life criticality features (medical systems, real-time controllers)" [8, 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) [6, p. 120]
  - "Also known as **extended branch coverage**" [7, p. 422]
  - Does not subsume and is not subsumed by path coverage [7, 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 [8, p. 464]

Data Coverage [6, pp. 114-116]

In addition to Data Flow Coverage [6, p. 114], [7, pp. 424-425], there are also some minor forms of data coverage:

- Sub-boundaries: mentioned previously in 2.7.5
- 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 [6, p. 116]

Data Flow Coverage [6, p. 114], [7, 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 [6, p. 114].

- "A variable is *defined* in a certain statement if it is assigned a (new) value because of the execution of that statement" [7, 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" [7, 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) [7, p. 424]
- Basic block: "[a] consecutive part[] of code that execute[s] together without any branching" [8, p. 477]
- Predicate Use (P-use): e.g., the use of a variable in a conditional [7, p. 424]
- Computational Use (C-use): e.g., the use of a variable in a computation or I/O statement [7, p. 424]
- All-use: either a P-use or a C-use [8, p. 478]
- DU-path: "a path from a variable definition to [one of] its use[s] that contains no redefinition of the variable" [8, pp. 478-479]
- The three possible actions on data are defining, killing, and using; "there are a number of anomalies associated with these actions" [8, pp. 478, 480] (see Data reference errors)

get original source: Beizer, 1990

Table 2.2 contains different types of data flow coverage criteria, approximately from weakest to strongest, as well as their requirements; all information is adapted from [7, pp. 424-425] .

Is this sufficient?

Q #5: How is All-DU-Paths coverage stronger than All-Uses coverage according to [7, p. 433]?

Fault Seeding [7, 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) [7, 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 [7, p. 427].

Table 2.2: 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 [7, 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) [7, 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" [7, pp. 428-429]. The goal is to maximize the number of mutants identified by a given test set [7, 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") [7, p. 429].

There is an unexpected byproduct of this form of testing. In some cases of one experiment , "the original program failed, while the modified program [mutant] yielded the right result" [7, 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!

add original source: KA85

2.7.8 Regression Testing

Repeating "tests previously executed ...at a later point in development and maintenance" [8, p. 446] "to make sure there are no unwanted changes [to the software's behaviour]" (although allowing "some unwanted differences to pass through" is sometimes desired, if tedious [8, p. 482]) [8, p. 481].

- Should be done automatically [8, 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

British

Investigate!

expected output" and "the existing *oracles*" [9, p. 516]

- Its "effectiveness ...is expressed in terms of":
  1. difficulty of test suite construction and maintenance
  2. reliability of the testing system [8, pp. 481-482]
- Various levels:
  - Retest-all: "all tests are rerun"; "this may consume a lot of time and effort" [7, 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" [7, p. 411]

Investigate these

2.7.9 Metamorphic Testing (MT)

The use of Metamorphic Relations (MRs) "to determine whether a test case has passed or failed" [11, 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" [11, 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) [11, p. 68]. "MT is one of the most appropriate and cost-effective testing techniques for scientists and engineers" [11, p. 72].

Benefits of MT

- Easier for domain experts; not only do they understand the domain (and its relevant MRs) [11, p. 70], they also may not have an understanding of testing principles [11, p. 69]. *This majorly overlaps with Drasil!*
- Easy to implement via scripts [11, p. 69]. *Again, Drasil*
- Helps negate the test oracle [11, p. 69] and output validation [11, p. 70] problems from Roadblocks to Testing Scientific Software [11, 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) [11, pp. 70-72]
- Domain experts are sometimes unable to identify faults in a program based on its output [11, 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 [11, p. 67]

- For matrices, if $B = B_1 + B_2$, then $A \times B = A \times B_1 + A \times B_2$ [11, pp. 68-69]

- Symmetry of trigonometric functions; for example, $\sin(x) = \sin(-x)$ and $\sin(x) = \sin(x + 360°)$ [11, 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" [11, p. 70]

- Using machine learning to predict likely MRs to identify faults in mutated versions of a program (about 90% in this case) [11, p. 71]

2.8 Roadblocks to Testing

- Intractability: it is generally impossible to test a program exhaustively [7, p. 421], [8, pp. 439, 461]

- Undecidability: [8, p. 439]

Q #6: The textbook doesn't elaborate on this?

2.8.1 Roadblocks to Testing Scientific Software [11, p. 67]

- "Correct answers are often unknown": if the results were already known, there would be no need to develop software to model them [11, p. 67]; in other words, complete test oracles don't exist "in all but the most trivial cases" [9, p. 510]

- "Practically difficult to validate the computed output": complex calculations and outputs are difficult to verify [11, p. 67]

- "Inherent uncertainties": since scientific software models scenarios that occur in a chaotic and imperfect world, not every factor can be accounted for [11, p. 67]

- "Choosing suitable tolerances": difficult to decide what tolerance(s) to use when dealing with floating-point numbers [11, 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++ [11, 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)?

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. 1–9).

[2] 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*. London: Springer London, 2013, pp. 17–29. 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 (cit. on pp. 2, 12, 13).

[3] 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. 9, 10).

[4] 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. 10–13).

[5] 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 pp. 13, 14).

[6] Ron Patton. *Software Testing*. 2nd ed. Indianapolis, IN, USA: Sams Publishing, 2006. ISBN: 0-672-32798-8 (cit. on pp. 14, 16–29).

[7] Hans van Vliet. *Software Engineering: Principles and Practice*. 2nd ed. Chichester, England: John Wiley & Sons, Ltd., 2000. ISBN: 0-471-97508-7 (cit. on pp. 14–21, 23–32).

[8] 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 (cit. on pp. 14–17, 19, 20, 22–32).

[9] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. “The Oracle Problem in Software Testing: A Survey”. In: *IEEE Transactions on Software Engineering* 41.5 (2015), pp. 507–525. DOI: 10.1109/TSE.2014.2372785 (cit. on pp. 15, 17, 18, 31, 32).

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

[11] Upulee Kanewala and Tsong Yueh Chen. “Metamorphic Testing: A Simple Yet Effective Approach for Testing Scientific Software”. In: *Computing in Science & Engineering* 21.1 (2019), pp. 66–72. DOI: 10.1109/MCSE.2018.2875368 (cit. on pp. 31, 32).

Appendix

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

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


    assert not path.exists(output_filename)


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

    assert not path.exists(output_filename)
```

Source Code A.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
```