Verification and Validation

Yiding Li
McMaster University

Abbreviations and Acronyms

Symbol	Description
SRS	Software requirements specification
VnV	Verification and validation
MG	Module guide
MIS	Module interface specification

Mathematical Notations

Symbol	Description
μ	Type of everything
τ	Type of types

1 Introduction

Verification and validation (**VnV**) ensures that the project sufficiently satisfies its requirements. If the project fails to satisfy its requirements, then VnV discusses and documents the extent of this failure.

VnV splits into **verification** and **validation**. Verification verifies the software against a set of requirements; validation considers the necessity and feasibility of requirements.

This section divides into the following subsections:

1. **Objectives** discusses the definition of verification and validation and the purpose of this document
2. **Characteristics of Intended Readers** discusses characteristics required to understand this document
3. **Relevant Documentation** lists documents that are referenced in this document.

1.1 Objectives

This document is the **verification and validation plan** (**VnV plan**), which documents the planning and instructs the implementation of VnV procedures. The VnV plan starts after the requirements stage and before the design stage, then undergoes iterative refinements as development continues.

1.1.1 Limitations of Testing

Testing cannot ensure correctness, only that the program performs correctly for the tested test cases. Testing only builds confidence that the software is correct. Techniques (such as unit testing) and metrics (such as coverage metrics) improve the effectiveness of tests at building performance.

1.2 Characteristics of Intended Readers

This document is designed for readers with the following roles and characteristics. Readers with these characteristics should be able to understand this document.

Developers design and implement Agolearn test cases following this document. Developers have the following characteristics:

- Understanding of modular testing
- Familiarity with evolutionary learning comparable to a graduate course on evolutionary learning

Users consult test cases to understand Agolearn and its capabilities. Users have the following characteristics:

- Understanding of modular software design
- Familiarity with evolutionary learning comparable to a graduate course on evolutionary learning

1.3 Relevant Documentation

This document references the following documents:

- SRS.pdf: Requirement specifications

2 Scope of Tests

This section discusses what tests do and do not cover. This section includes the following subsections:

1. **Assumptions** lists conditions that are assumed to be true by all test cases
2. **Ignored Categories** lists conditions that are not tested
3. **Machinery** lists machinery that are used to describe test cases

2.1 Assumptions

Validation and verification procedures make the following assumptions:

- **Locality**. The library is always available and exists locally. Access to the internet is unavailable.
- **The environment is abstract**. System-dependent resources, such as environment variables and system calls, are not explicitly available.
- **Direct access**. Test primitives can directly invoke the tested interface.

- **Resources are abundant**. Tests can use as much computational resource as necessary.

2.2 Ignored Categories

The following properties are either infeasible or impossible to test. Measure these properties, but do not write test cases for these measurements.

- Performance. The software is not performance sensitive.

2.3 Machinery

The following subsections describe the machinery of testing.

1. **Constants** are immutable values.
2. **Pseudo-Oracles** are reference implementations that behave similarly to Agolearn.
3. **Interfaces** lists objects that are used in testing, and describes how to interact with these objects.
4. **Primitives** are functions that facilitate the testing process.

2.3.1 Constants

The following table describes testing constants in the following types:

- **Ground truths** are constants known before the fact. These constants describe the reality that is captured by the project.
- **Test Constants** are constants only used in test cases.

- **Interfaces** describe interactions with the tested objects.
- **Error Margins** are special test constants that describe the leniency of test cases. A test case passes if the difference is within the given error margin.

TABLE 1: GROUND TRUTHS

None yet

TABLE 2: TEST CONSTANTS

None yet

TABLE 3: MARGINS OF ERROR

δ_F	Error margin for floating-point comparisons

2.3.2 Pseudo-Oracles

- **Pyvolution** - a modular evolutionary learning framework

2.3.3 Interfaces

Interface 1: Genome

Name	Genome	
Type	Data Structure	
Attributes	fitness: : $\mathbb{R}$	Quality of the genome as a solution

Interface 2: Population factory

Name	population_factory
Type	Set[Genome] $\rightarrow$ Population

Interface 3: Population

Name	Population	
Type	Data Structure	
Attributes	size : $\mathbb{Z}_{\geq 0}$	Size of the population

Interface 4: Variator

Name	Variator	
Type	Population $\rightarrow$ Population	
Attributes	arity : $\mathbb{Z}_{\geq 0}$	Size of the input parent group
	coarity : $\mathbb{Z}_{\geq 0}$	Size of the output population per input group
	residual : $\mathbb{Z}_{\geq 0}$	Number of additional members in the output population

Interface 5: Selector

Name	Selector	
Type	Population $\to$ Population	
Attributes	outsize : $\mathbb{Z}_{\geq 0}$	Size of the output population

Interface 6: Evaluator

Name	Evaluator
Type	Genome $\to \mathbb{R}$

2.3.4 Primitives

The assert_epsilon primitive compares floating-point numbers, accounting for floating-point errors.

assert_epsilon: $\mathbb{R} \times \mathbb{R} \times \mathbb{R} \to \{\top, \bot\}$
function assert_epsilon(a, b, ε)
  if ($|a - b| < \varepsilon$) **do**
    return $\top$
  else
    return $\bot$
  end if
end ¶

The type primitive returns the mathematical type of its argument.

type: $\mu \to \tau$
function type($o : \mu$)
  return the type of o
end function

The best_fit primitive returns a member of a population with the highest fitness

best_fit: Population → Genome
function best_fit(G : Set[Genome])
  return $\arg\max_{g \in G} g$.fitness
end function

2.4 Test Plan

2.4.1 Verification and Validation Team

The VnV team includes verifiers and validators.

Verifiers: The following people review the implementation of this project.

Name	**Role**
Yiding Li	Designer and implementor of test cases
Stephen Kelly	Domain expert and usability reviewer

Validators: The following people review design artefacts.

Name	Role
Yiding Li	Author of development artefacts
Spencer Smith	Course instructor, reviewer
Fasil Cheema	Domain expert and primary reviewer
Fatemah	SRS reviewer
Tanya Djavaherpour	VnV reviewer
Phil Du	MG + MIS reviewer

2.5 Test Plan

This section describes elements that should be tested, and how to test them. These elements are as follows:

1. **Conditions**: preconditions, postconditions, and invariants
2. **Functional Requirements**: functional requirements sourced from SRS.pdf
3. **Nonfunctional** Requirements: nonfunctional requirements sourced from SRS.pdf
4. **Coverage**: coverage criteria that should be met by test cases.
5. **Traceability Information**: Reference from test cases to requirements.

2.5.1 Conditions

C1: The size of a population should adhere to variators.

T1: Test plan for C1

Subject	A variator
Control	Manual
Initial State	A population P is initialized and available A variator V is initialized and available
Input	A population P
Output	A population P' whose size is $(P/(V.\text{arity})) \cdot V.\text{coarity} + V.\text{extra}$

C2: The size of a population should adhere to selectors.

T2: Test plan for T2

Subject	A selector
Control	Manual
Initial State	A population P is initialized and available A selector S is initialized and available
Input	A population P
Output	A population P' whose size is $\max(S.\text{outsize}, P.\text{size})$

2.5.2 Functional Requirements

FR1: Type check input values

T3: Test plan for FR1

Subject	population_factory
Control	Manual
Initial state	A set of genomes G is available
Input	A set of genomes G
Output	**assert**(type(G) is Set[$\mathbb{R} \to \mathbb{R}$] or type($G$) is Set[$(A \to B) \to \mathbb{R}$])

FR2: Interchangeability of evolutionary operators of the same kind

T4: Each individual selector, variator, and evaluator must pass all test cases for selector, variators, or evaluators respectively.

FR3: Each iteration produces a population

T5: Static validation: each iteration results in a population.

FR4: Subsequent genomes should not have lower fitness.

T6: Test plan for FR4

Subject	System
Control	Manual
Initial State	A population P available A variator V A selector S A evaluator E
Input	A population P
Output	$\text{prev\_best} \leftarrow \arg\max_{g \in P} P$ $P' \leftarrow S\big(V(P)\big)$ $\text{next\_best} \leftarrow \arg\max_{g \in P} P'$ **assert**(*prev_ best.fitness* < *current_ best.*fitness)

2.5.3 Nonfunctional Requirements

NFR1: Usability

T7: The interface should be inspected and approved by Kelly.

NFR 2: Understandability

T8: Design artefacts should be inspected and approved by members of the validation team.

NFR 3: Maintainability

T9: Static verification: All functions should use type hints. All arguments of all public documents should be documented.

NFR 4: Portability

T10: Test cases should pass on a version of Microsoft Windows 11.

2.5.4 Coverage

Test cases should achieve 100% function coverage. Record, then document statement coverage.

2.5.5 Traceability Information

I am making a program that generates traceability information based on label references. I defer this part to when I finish that program.