Verification and Validation

Yiding Li
McMaster University

Revision History

Date	Version	Notes
2024-02-26	0.1	Initial draft
2024-03-16	0.2	Transcribe to edit
2024-04-18	1.0	Rework

Abbreviations and Acronyms

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

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:

- 1.1 Objectives discusses the definition of verification and validation and the purpose of this document
- 1.2 Characteristics of Intended Readers discusses characteristics required to understand this document
- 1.3 Relevant Documentation lists documents that are referenced in this document.

1.1 Objectives

VnV

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.

This document particularly focuses on VnV procedures that relate to requirements. While the project implements test suites for user-implemented operators, these tests are part of the requirements, as opposed to tests for requirements.

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

- MG.pdf: Module guide
- MIS.pdf: Module interface specification

2 Scope of Tests

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

1. Pseudo-Oracles lists programs that behave similarly to the system
2. Assumptions lists conditions that are assumed to be true by all test cases
3. Ignored Categories lists conditions that are not tested
4. Machinery lists machinery that are used to describe test cases

2.1 Pseudo-Oracles

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

2.2 Assumptions

Validation and verification procedures make the following assumptions:

- **Locality**. The library is always available and exists locally. Access to the internet is unavailable.
- **Direct access**. Test primitives can directly invoke the tested interface.
- **Resources are abundant**. Tests can use as much computational resource as necessary.

2.3 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.
- Describe performance: relative to a pseudo-oracle.

2.4 Machinery

The following subsections describe the machinery of testing.

- 2.4.1 Constants are immutable values.
- 2.4.2 Interfaces lists objects that are used in testing, and describes how to interact with these objects.
- 2.4.3 Primitives are functions that facilitate the testing process.
- 2.4.4 Participants lists people who participate in testing.

2.4.1 Constants

The following table describes testing constants in the following types:

- Table 1 are constants known before the fact. These constants describe the reality that is captured by the project.
- Table 2 are constants only used in test cases.
- Table 3 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

$\delta_F := 0.001$	Error margin for floating-point comparisons

2.4.2 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 $\to \mathbb{R}$

2.4.3 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

2.4.4 Participants

Participants of the project take one or more of the following roles:

- **Designers** design test cases.
- **Implementers** design test cases.
- **Problem domain experts** describe the problem and provide theoretical advise ... what does a problem domain expert do exactly?
- **Solution domain experts** perform code review
- **Usability testers** use the software then provide feedbacks about their experience
- **Document reviewers** review development documents, such as the ones listed in 1.3 Relevant Documentation.

2.5 Test Plan

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

- 2.5.1 Conditions: preconditions, postconditions, and invariants
- 2.5.2 Functional Requirements: functional requirements sourced from SRS.pdf
- 2.5.3 Nonfunctional Requirements: non-functional requirements sourced from SRS.pdf
- 2.5.4 Coverage: coverage criteria that should be met by test cases.
- 2.5.5 Traceability Information: connections from test cases to requirements

2.5.1 Conditions

The test plan does not consider conditions.

2.5.2 Functional Requirements

FR 1: Provides ways to implement operators

T 1: Test plan for FR 1

The system must ship with an implementation that optimizes. That implementation must pass satisfy the following test case.

Subject	System
Control	Manual
Initial State	P : Population is defined V : Variator is defined S: Selector is defined E : Evaluator is defined
Input	A population P
Output	prev_best $\leftarrow \arg\max_{g \in P} P$ $P' \leftarrow S\big(V(P)\big)$ next_best $\leftarrow \arg\max_{g \in P} P'$ **assert**(prev_best.fitness $<$ current_best.fitness)

FR 2,

FR 3,

FR 4,

FR 5: Correctly implemented variators, selectors, evaluators, and algorithm of the same genome type are interchange-able.

T 2: Test plan for FR 2, FR 3, FR 4, FR 5

This requirement is difficult to test, since it is not possible to guarantee that a user's implementation is correct. At minimum, all representations and operators that ship with the system must pass the test case in FR 1.

FR 6,

FR 7: Empirically, analyse then display the performance of an evolutionary operator or algorithm.

T 3: Test plan for FR 6 and FR 7

Select a representation and a set of operators that is implemented by both the system and Pyvolution. Analyse the algorithm in both frameworks, then compare the result.

FR 8,

FR 9,

FR 10,

FR 11,

FR 12: Provide operator suites for (a) OneMax, (b) floating point, (c) genetic programming, (d) linear programming, and (e) tangled program graph problem representations.

T 4: Test plan for FR 8, FR 9, FR 10, FR 11, and FR 12

Manual inspection. The corresponding modules must be implemented and all past test cases in FR 1.

FR 13: The system offers step-by-step instructions that lead to the success implementation of at least one operator suite

T 5: Test plan for FR 13

Consult an domain expert. Give the domain expert an instruction that permits the implementation of the OneMax module, then check if the expert's implementation behaves similarly to the version that is included in the system.

2.5.3 Nonfunctional Requirements

NFR 1: The interface is used and deemed intuitive by a domain expert

T 6: After T 5, consult the expert. Ask if the process of implementing the operator is intuitive, then record the response.

NFR 2: Every public object and public function is documented.

T 7: Manual inspection.

2.5.4 Coverage

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

2.5.5 Traceability Information

Table 4: Derivation of test cases from requirements

T	FR													NFR	
	1	2	3	4	5	6	7	8	9	10	11	12	13	1	2
1	✓														
2		✓	✓	✓	✓										
3						✓	✓								
4								✓	✓	✓	✓	✓			
5													✓		
6														✓	
7															✓

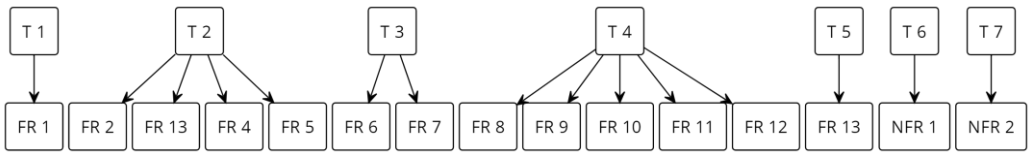

Figure 1: Traceability diagram of test cases from requirements