Medical Diagnosis Prediction Tool: System Verification and Validation Plan for Diagnose

Andrea Clemeno

December 12, 2020

1 Revision History

Date	Version	Notes
10/29/2020	1.0	First Draft of VnV
10/31/2020	1.1	Updating first draft
12/05/2020	1.2	Updating with test cases
12/10/2020	1.3	Updating to address comments

Contents

List of Tables

2 Symbols, Abbreviations and Acronyms

The following tables identify the symbols, abbreviations and acronyms use throughout this document.

2.1 Table of Symbols

The table that follows summarizes the symbols used in this document along with their descriptions. The symbols are listed in alphabetical order. In addition, all symbols, abbreviations, and acronyms recorded in the SRS for Diagnose (Clemeno, 2020) apply to this document.

symbol	**description**
API	Application Program Interface
SRS	Software Requirements Specification
T	Test
VnV	Verification and Validation

Table 1: Table of Symbols

2.2 Abbreviations and Acronyms

The table that follows summarizes the symbols used in this document that allude to different sections of the Software Requirements Specification. The symbols are listed in alphabetical order.

symbol	**description**
A	Assumption
DD	Data Definition
GD	General Definition
GS	Goal Statement
IM	Instance Model
LC	Likely Change
PS	Physical System Description
R	Requirement
SRS	Software Requirements Specification
Diagnose	Medical Diagnosis Prediction Tool for Acquired immunodeficiency syndrome (AIDS)
T	Theoretical Model
ULC	Unlikely Change

Table 2: Table of Abbreviations and Acronyms

This document outlines the verification and validation plans for components significant for the implementation of the Diagnose program, including the SRS, design, implementation and software validation. Following this, the system tests for functional and non-functional requirements are indicated. Moreover, how test cases meet requirements will be identified. After describing the system testing, the unit testing for all requirements using different modules will be explained as well as the traceability from the test cases to the modules.

3 General Information

3.1 Summary

The software under tested works with viral load concentrations from patients infected with the HIV-1 virus to determine the efficiency of their immune system. The efficiency can be described by the elimination rate of the virus in the patient's body. This rate of change will help predict the viral load concentration of the patients after a user-defined number of days.

3.2 Objectives

The objective of Diagnose is to provide predictions that are tested and verified. The various intended users of the software will need to view the results easily. Lastly, the software should be optimized and provided timely. The following VnV plan will build confidence in the software correctness, demonstrate adequate portability for the intended users, and ensure efficient software performance.

3.3 Relevant Documentation

The Diagnose program will use different documentation to identify its purpose and the development methods used. The documentation includes the SRS (Clemeno (2020)), the following VnV Plan, Drasil documentation (Clemeno (2020b)) and VnV Report (Clemeno (2020a)).

4 Plan

4.1 Verification and Validation Team

Verification and validation are used to build confidence in the correctness of Diagnose. The following document will be reviewed by: the primary reviewer, Andrea Clemeno, the domain expert reviewer, Elizabeth Hofer, and the secondary reviewer, John Ernsthausen. In addition, Dr. Smith, the CAS 741 course instructor, will review the VnV plan. The reviewers will ensure the document is in accordance with the VnV plan checklist (Smith (2019)).

4.2 SRS Verification Plan

The verification of the SRS will be done to ensure that the requirements specified are in alignment with the outlined objective of the Diagnose program. The SRS verification plan will involve reviewing the document against the SRS checklist and providing feedback using issues on GitHub repository (Smith (2020)). The reviewers that will verify the SRS document include: the class instructor, Dr. Smith; the domain expert reviewer, Elizabeth Hofer; the secondary reviewer, Tiago de Moraes Machado; the primary reviewer, Andrea Clemeno.

4.3 Design Verification Plan

The specifications and implementation of the software will be documented extensively with several reports including the SRS, VnV plan and VnV report. However, the design will not be documented as the modules of the code will be generated by the Drasil Framework. The generation of the design will be modular, correctness is assumed and verification is not required.

4.4 Implementation Verification Plan

The implementation of the software will be verified using several static methods involving manual or automated interactions. The software will be generated using the Drasil Framework to generate all of the software artifacts (Jacques Carette, 2020). The generated python code

The implementation of the software will be verified with several static methods involving manual or automated interactions. The software will be developed using the Drasil Framework to generate all of the software artifacts.Jacques Carette (2020). The code developed uses python to achieve goals and fulfill requirements in the SRS. The design of the code will be evaluated by the project developer, domain expert, the secondary reviewer and the class instructor. The evaluation will involve code inspections where coding syntax, structure and standards are upheld. In addition, code walkthroughs will be performed where the evaluators will try to determine the output of the code using the code with little to no context. The code walkthrough will verify that the requirements and goals of the code are met. Moreover, automatic methods will verify the design by displaying success messages after certain checkpoints throughout the code. The mentioned tests will be explained in more detail in Section 5.

4.5 Automated Testing and Verification Tools

The Diagnose software will be tested and verified with several tools for unit and systems testing, static and dynamic analysis, linting and continuous integration. The static automatic testing will be completed in Spyder, a Python Integrated Development Environment, using several checkpoints with success and failure indicators. Additionally, the Spyder platform will analyze the code for potential errors in the process of linting. For testing performance optimization dynamically, a python profiler called cProfile will be used to profile speed (Sia (2020)).

Automated testing will be implemented for individual units as well as the integrated system to ensure that all the sections work separately and together seamlessly. In terms of unit testing, each unit will be tested with the unittest python library within Spyder. Respectively, the systems test will be completed through black box testing with the Python Black Box tool called pbbt.

Lastly, the implementation of the design will be verified with continuous integration through Travis CI that is used in conjunction with Drasil. Drasil implements Travis CI to integrate code into a Github repository each day to complete automated tests to verify the code (Jacques Carette (2020)).

Test	Verification Tool
Static Analysis	Spyder
Linting	Spyder
Dynamic Analysis	cProfile
System Test	Blackbox testing using pbbt
Unit Test	Unittest within Spyder
Continuous Integration	Travis CI

Table 3: Automated Testing and Verification Tools

4.6 Software Validation Plan

The software validation plan will be implemented at the end of the development process to determine if the real world problem is characterized correctly. The validation of Diagnose will be completed by comparing the outputs of the software to several cases from scientific study called Viral Dynamics of Acute HIV-1 Infection seen in The Journal of Experimental Medicine (Little (1990)).

5 System Test Description

5.1 Tests for Functional Requirements

This section will define the tests to ensure Diagnose meets the functional requirements seen in the SRS document for Diagnose. The subsections combine several requirements that are be separated based on common ideas.

5.1.1 Testing inputs

The user-defined inputs will undergo tests to ensure that numerical data was received and aligns with the input constraints. The tests will automatically display feedback if the conditions above are not met. The tests called id-added, input-received and input-verify are described in greater detail below:

Input Testing

Test	N_o (mol)	N_t (mol)	t_t (s)	t_p (s)	Output
Test 1-1	10000000	5000000	1	30	-
Test-2-1	0	5000000	1	30	Exception: InputError
Test-2-2	-10000000	5000000	1	30	Exception: InputError
Test-3-1	0	5000000	1	30	Exception: InputError
Test-3-2	-10000000	5000000	1	30	Exception: InputError
Test 4-1	5000000	10000000	1	30	Exception: InputError
Test-5-1	10000000	5000000	0	30	Exception: InputError
Test-5-2	10000000	5000000	-1	30	Exception: InputError
Test-6-1	10000000	5000000	0	30	Exception: InputError
Test-6-2	10000000	5000000	-1	30	Exception: InputError
Test 7-1	10000000	5000000	30	1	Exception: InputError

1. id-added

   Control: Automatic

   Initial State: Diagnose running

   Input:

Test input 1: $id = 123$
Test input 2: $id =$ " "

   Output:

   - Test output 1: "id-added: success"
   - Test output 3: "id-added: failure. Try again."

   Test Case Derivation:
   The expected result for the given inputs will be either "id-added: success" or "id-added: failure. Try again." When the user inputs an id, the inputs will be received successfully. In comparison, any nonsensical data inputs like a null input will cause this test to output a failure.

   How test will be performed:
   This automatic static test will be completed in Spyder using if-then-else loop conditions to display the previously mentioned output.

Input Testing

1. input-received

   Control: Automatic

   Initial State: Diagnose running

   Input:

Test input 3: $V_1 = 5 * 10^8$, $V_2 = 4 * 10^8$

Test input 4: $V_1 = 5 * 10^8$, $V_2 = 6 * 10^8$

Test input 5: $V_1 =$ abc123 , $V_2 = 6 * 10^8$

Test input 6: $V_1 =$ abc123 , $V_2 =$ 789xyz

   Output:

   - Test output 1: "input-received: success"
   - Test output 2: "input-received: success"
   - Test output 3: "input-received: failure. Try again with numerical values."
   - Test output 4: "input-received: failure. Try again with numerical values."

   Test Case Derivation:
   The expected result for the given inputs will be either "input-received: success." or "input-received: failure. Try again with numerical values." When the user-defined inputs are numbers, the inputs will be received successfully. In comparison, any nonsensical data inputs for one or more of the inputs will cause this test to output a failure.

   How test will be performed:
   This automatic static test will be completed in Spyder using if-then-else loop conditions to display the previously mentioned output.

2. input-verify

Control: Automatic

Initial State: Diagnose running and input-received is successful

Input:

Test input 7: $V_1 = 2 * 10^8$, $V_2 = 5 * 10^8$

Test input 8: $V_1 = 5 * 10^8$, $V_2 = 5 * 10^8$

Test input 9: $V_1 = 5 * 10^8$, $V_2 = 4 * 10^8$

Test input 10: $V_1 = 10 * 10^8$, $V_2 = 6 * 10^8$

Test input 11: $V_1 = 20 * 10^8$, $V_2 = 5 * 10^8$

Output:

- Test output 1: "input-verify: failure; V_1 should be greater than V_2."
- Test output 2: "input-verify: failure; V_1 should be greater than V_2."
- Test output 3: "input-verify: success"
- Test output 4: "input-verify: success"
- Test output 5: "input-verify success"

Test Case Derivation:
The expected result for the given inputs will be either "input-verify: success" or "input-verify: failure; V_1 should be greater than V_2." According to the input constraints specified in the SRS, the program can only determine the rate of clearance of the virus when the virus starts decreasing due the immune system affecting virus concentration. When the user-defined inputs include a greater V_2, the viral has yet to decrease and therefore, the input constraints are not met and the output is failure for test input 1.

How test will be performed:
This automatic static test will be completed in Spyder using if-then-else loop conditions to display the previously mentioned output.

5.1.2 Testing outputs

The produced output will undergo tests to ensure that data is produced, aligns with the output constraints and is displayed to the user. The output-produced and output-valid tests will automatically display feedback if the conditions above are not met. The output-displayed test will be manual in the form of a checked box that will be checked by the user if the outputs are displayed. The tests called output-produced, output-verify and output-displayed are described in greater detail below:

Output Testing

1. output-produced

   Control: Automatic

   Initial State: Diagnose analysis completed

   Input:

Test input 12: $\frac{dV}{dt} = 1.3$

Test input 13: $\frac{dV}{dt} = undefined$

Test input 14: $\frac{dV}{dt} = 0$

Test input 15: $\frac{dV}{dt} = -0.3$

   Output:

   - Test output 1: "output-produced: success"
   - Test output 2: "output-produced: failure"
   - Test output 2: "output-produced: success"
   - Test output 3: "output-produced: success"

   Test Case Derivation:
   The expected result for the given inputs will be either: "output-produced: success" or "output-produced: failure". When the program produces an output, whether negative, zero or positive, a numerical value will

have a successful result in this test. When an undefined result is produced, this test to output a failure.

How test will be performed:
This automatic static test will be completed in Spyder using if-then-else loop conditions to display the previously mentioned output.

2. output-verify

Control: Automatic

Initial State: Diagnose analysis completed; output-program is successful

Input:

Test input 16: $\frac{dV}{dt} = 1.3$

Test input 17: $\frac{dV}{dt} = 0$

Test input 18: $\frac{dV}{dt} = -0.3$

Test input 19: $\frac{dV}{dt} = -0.7$

Test input 20: $\frac{dV}{dt} = -0.9$

Output:

- Test output 1: "output-verify: failure"
- Test output 2: "output-verify: failure"
- Test output 3: "output-verify: success"
- Test output 4: "output-verify: success"
- Test output 5: "output-verify: success"

Test Case Derivation:
The expected result for the produced outputs will be either "output-verify: success" or "output-verify: failure". According to the output

constraints specified in the SRS, IM1 defines the rate of clearance of the virus is a negative value. When the outputproduced by the program ($\frac{dV}{dt}$) is greater than or equivalent to 0, the viral load has yet to decrease and therefore, the output constraints are not met and the test will fail.

How test will be performed:
This automatic static test will be completed in Spyder using if-then-else loop conditions to display the previously mentioned output.

3. output-displayed

Control: Manual

Initial State: Diagnose analysis completed; output-program , output-valid are successful

Input:

Test input 21: $\frac{dV}{dt} = -0.3$, user indicates "displayed".

Test input 22: $\frac{dV}{dt} = -0.9$, user indicates "displayed".

Test input 23: $\frac{dV}{dt} =$ "", user indicates "not displayed".

Output:

- Test output 1: "output-displayed: success"
- Test output 2: "output-displayed: success"
- Test output 3: "output-displayed: failure"

Test Case Derivation:
Possibilities of inputs and outputs of the test are identified above. In this test, the user will indicate if the output is displayed. The outputs of the test will be identified accordingly.

How test will be performed:
This manual static test will be completed in Spyder using user feedback to display the previously mentioned output.

5.2 Tests for Nonfunctional Requirements

This section will define the tests to ensure Diagnose fulfill the nonfunctional requirements seen in the SRS document of Diagnose. The subsections are be separated for different requirements.

5.2.1 Correctness

Testing the Correctness of Diagnose: The correctness of the software can be defined using the constraints of Diagnose. Correctness of software is dependent on the satisfaction of the requirements in the SRS. The testing for the correctness of Diagnose will involve referencing input-verify and output-verify tests from the Tests for Functional Requirements.

1. software-correctness

   Type: automatic

   Initial State: Diagnose software run complete

   Input:

   - Test input 1: input-verify = "success", output-verify = "success"
   - Test input 2: input-verify = "failure", output-verify = "success"
   - Test input 3: input-verify = "success", output-verify = "failure"

   Output:

   - Test output 1: "software-correctness: 100% correct"
   - Test output 2: "software-correctness: 50% correct"
   - Test output 3: "software-correctness: 50% correct"

   How test will be performed:
   The correctness of the Diagnose will be tested in Spyder using if-then-else loop conditions to display the previously mentioned output.

5.2.2 Reliability

Testing the Reliability of Diagnose: The reliability of the software tests if the product aligns with it's purpose. Reliability of Diagnose will be examined by comparing the output to several subject profiles from a scientific journal using relative error.

1. software-reliability

   Type: automatic

   Initial State: Diagnose software run complete

   Condition:

   - $X = (theoreticalvalue - experimentalvalue) \,/\, experimentalvalue$

   Result:

   - The software is $X\%$ reliable.

   How test will be performed:
   The reliability of the Diagnose will be quantified in Spyder to display the previously mentioned output.

5.2.3 Usability

Testing the Usability of Diagnose: The usability is highly important in the Diagnose software as many users will be interacting with the software. Testing the usability will determine if the users have a efficient interaction with the software. The system will be tested against a usability checklist and a usability survey to quantify the versatility of the user interface.

1. software-usabilitychecklist

   Type: manual

   Initial State: Diagnose software run complete

   Input:

   - The design of the user interface and checklist.

Result:

- The software aligns with $X\%$ of the checklist.

How test will be performed:
The usability of the Diagnose will be quantified manually by determining the percentage of items on the checklist that the software has.

2. software-usabilitysurvey

   Type: manual

   Initial State: Diagnose software run complete

   Input:

   - The design of the user interface and survey questions.

   Result:

   - A certain item should be discarded.
   - A certain item should be added.

   How test will be performed:
   The usability of the Diagnose will be quantified manually by determining possible changes from survey answers from the usability survey.

5.2.4 Performance

Testing the Performance of Diagnose: The adequate time and memory performance is essential for the Diagnose software. Testing the performance will ensure the software is useful for intended users.

1. software-profiling

   Type: automatic

   Initial State: Diagnose software run complete

   Input:

- The code generated by Drasil and cProfile library.

Result:

- The wall time per function call
- The cumulative time for a given function.

How test will be performed:
The test for deterministic performance will use the cProfile library atuomatically in Spyder to identify lines of the code that can possibly be optimized. The system will be profiled using ProfileC to determine the wall time per function call and cumulative time spent on a given function.

5.3 Traceability Between Test Cases and Requirements

The purpose of the traceability matrices is to provide easy references on what has to be additionally modified if a certain component is changed. Every time a component is changed, the items in the column of that component that are marked with an "X" may have to be modified as well. Table 4 shows which test cases are supporting which requirements.

	R-Inputs	R-Constraints	R-AIDsdiagnosis	R-VerifyOutput	R-Output
Tinput1	x				
Tinput2	x				
Tinput3	x				
Tinput4	x				
Tinput5	x				
Tinput6	x				
Tinput7		x			
Tinput8		x			
Tinput9		x			
Tinput10		x			
Tinput11		x			
Tinput12			x		
Tinput13			x		
Tinput14			x		
Tinput15			x		
Tinput16			x	x	
Tinput17			x	x	
Tinput18			x	x	
Tinput19			x	x	
Tinput20			x	x	
Tinput21					x
Tinput22					x
Tinput23					x

Table 4: Traceability Matrix Showing the Connections Between Requirements and test cases

References

A. Clemeno. Software requirements specification for diagnosis-aids: Medical diagnosis prediction tool for acquired immunodeficiency syndrome (aids). 2020.

Andrea Clemeno. Verification and validation report, December 2020a. URL https://github.com/andreamclemeno/CAS741-Diagnose/blob/master/docs/VnVReport/VnVReport.pdf.

Andrea Clemeno. Software requirements specification for diagnose.tex, December 2020b. URL https://github.com/andreamclemeno/Drasil/tree/master/code/stable/diagnose/SRS.

Spencer Smith Jacques Carette. Drasil: A framework for literate scientific software, 2020.

Susan J. Little. Viral dynamics of acute hiv-1 infection. *Journal of Experimental Medicine*, 1990.

Kim Sia. Python profilers: Learn the basics of a profiler for python in this blog, 2020. URL https://stackify.com/how-to-use-python-profilers-learn-the-basics/.

S. Smith. System verification and validation plan checklist. 2019.

S. Smith. Srs and ca checklist. 2020.

6 Appendix

The appendix presents the usability checklist and survey questions mentioned in section.

6.1 Usability Checklist

System Usability Checklist

- Grammar, spelling, presentation
  - No spelling mistakes
  - No grammar mistakes
  - All hyperlinks work
  - Symbolic names are used for quantities, rather than literal values
- Organization
  - Page title for every page
  - Step of process identified
  - Name of step of process
- User Interaction
  - Field names
  - Obvious field location
  - Important information highlighted
  - Help or Suggestion for each page

6.2 Usability Survey Questions

This is a section states the usability questions to ask for the manual test for the usability requirement.

1. What do you like most about the interface?
2. What would you like to change about the interface?

3. Did you face any challenge while using the site?
4. Were the buttons and fields easy to find and understand?
5. On a scale of 1-10, how easy was it to navigate through the interface?
6. Did you feel that the software took too long to load the website?
7. Did you feel that the software took too long to fetch your details on our website?
8. Do you have any suggestions or comments?