System Verification and Validation Plan for Diagnose

Andrea Clemeno

December 14, 2020

1 Revision History

Date	Version	Notes
10/29/2020	1.0	First Draft of VnV plan
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
12/12/2020	1.4	Updating to Dr. Smith's comments
12/14/2020	1.5	Second Draft of VnV plan

Contents

List of Tables

2 Symbols, Abbreviations and Acronyms

The following table identify the symbols, abbreviations and acronyms used throughout this document.

2.1 Table of Symbols, Abbreviations and Acronyms

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 (**?**) apply to this document.

symbol	**description**
N/A	Not Applicable
OS	Operating System
SRS	Software Requirements Specification
T	Test
TC	Test Case
VnV	Verification and Validation

Table 1: Table of Symbols, Abbreviations and Acronyms

This document outlines a higher level description of verification and validation plans for the documentation of the Diagnose program, including the SRS, design, implementation and software validation. Following this, the system tests for functional and non-functional requirements are described in section **??**. Moreover, how test cases meet requirements will be identified. After describing the system testing, the unit testing for necessary modules will be explained as well as the traceability from the test cases to the modules in section **??**.

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. Additionally, 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 (**?**), the following VnV Plan, Drasil documentation (**?**) and VnV Report (**?**).

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 project developer, 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 (**?**). The feedback is provided using issues on GitHub repository: CAS741-Diagnose.

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 (**?**). The results will be summarized in the VnV report (**?**). 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 project developer, Andrea Clemeno.

4.3 Design Verification Plan

The specifications and implementation of the software will be documented extensively with several reports. 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 techniques involving manual or automated interactions. The software will be generated using the Drasil Framework to generate some of the software artifacts, specifically the SRS and python code (**?**). The generated code implements python to achieve goals and fulfill the requirements outlined 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 code evaluation will involve unit and system testing, static and dynamic code analysis, linting and continuous integration. The tools for the evaluations will be mentioned in section **??**. Furthermore, the test cases used in system and unit testing will be stated in sections **??** and **??** respectively.

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, a python profiler called cProfile will be used to profile speed dynamically (**?**).

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, some modules will be tested with the unittest python library within Spyder. Respectively, the systems test will be completed through black box testing with Pytest.

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 (**?**).

Test	**Verification Tool**
Static Analysis	Spyder
Linting	Spyder
Dynamic Analysis	cProfile
System Test	Pytest
Unit Test	unittest
Continuous Integration	Travis CI

Table 2: 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 Modeling plasma virus concentration during primary HIV infection seen in The Journal of Theoretical Biology (**?**).

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 Input Verification

The inputs will undergo tests to satisfy R1 and R2 from Diagnose SRS. Specifically ensuring values are input and align with the input constraints.

Table **??** displays the inputs and outputs of test cases for the input constraints tests.

Test Cases	N_o ($\frac{mol}{mL}$)	N_t ($\frac{mol}{mL}$)	t_t **(d)**	t_p **(d)**	**Output**
TC-1-1	12000000	5000000	1	30	-
TC-2-1	0	5000000	1	30	Exception: InputError
TC-2-2	-12000000	5000000	1	30	Exception: InputError
TC-3-1	12000000	0	1	30	Exception: InputError
TC-3-2	12000000	-10000000	1	30	Exception: InputError
TC-4-1	5000000	12000000	1	30	Exception: InputError
TC-5-1	12000000	5000000	0	30	Exception: InputError
TC-5-2	12000000	5000000	-1	30	Exception: InputError
TC-6-1	12000000	5000000	1	0	Exception: InputError
TC-6-2	12000000	5000000	1	-1	Exception: InputError
TC-7-1	12000000	5000000	30	1	Exception: InputError

Table 3: Test Cases used for Verification of Diagnose

Input Constraints Test

1. T-1:

   Control: Automatic

   Initial State: N/A

   Input: N_o = 12000000 $\frac{mol}{mL}$; N_t= 5000000 $\frac{mol}{mL}$; t_t = 1 d; t_p = 30 d (TC-1-1)

   Output: k = 0.8755; N_p = 4.708e-05

   Test Case Derivation: Inputs within expected input range are used in the previous test to ensure successful input.

   How test will be performed: Pytest will be implemented.

2. T-2:

   Control: Automatic

   Initial State: N/A

   Input: TC-2-1, TC-2-2

   Output: Exception: InputError

   Test Case Derivation: N_o input is out of bounds and not an expected input value. TC-2-1 and TC-2-2 have zero and negative inputs for N_o respectively.

   How test will be performed: Pytest will be implemented.

3. T-3:

   Control: Automatic

   Initial State: N/A

   Input: TC-3-1, TC-3-2

   Output: Exception: InputError

   Test Case Derivation: N_t input is out of bounds and not an expected input value. TC-3-1 and TC-3-2 have zero and negative inputs for N_t respectively.

   How test will be performed: Pytest will be implemented.

4. T-4:

   Control: Automatic

   Initial State: N/A

   Input: TC-4-1

   Output: Exception: InputError

   Test Case Derivation: Relationship between N_o and N_t is invalid and out of bounds and not an expected input values.

   How test will be performed: Pytest will be implemented.

5. T-5:

   Control: Automatic

   Initial State: N/A

   Input: TC-5-1, TC-5-2

   Output: Exception: InputError

   Test Case Derivation: t_t input is out of bounds and not an expected input value. TC-5-1 and TC-5-2 have zero and negative inputs for t_t respectively.

   How test will be performed: Pytest will be implemented.

6. T-6:

   Control: Automatic

   Initial State: N/A

   Input: TC-6-1, TC-6-2

   Output: Exception: InputError

   Test Case Derivation: t_p input is out of bounds and not an expected input value. TC-6-1 and TC-6-2 have zero and negative inputs for t_p respectively.

   How test will be performed: Pytest will be implemented.

7. T-7:

   Control: Automatic

   Initial State: N/A

Input: TC-7-1

Output: Exception: InputError

Test Case Derivation: Relationship between t_t and t_p is invalid and out of bounds and not an expected input values.

How test will be performed: Pytest will be implemented.

5.1.2 Output Verification

1. T-8:

   Control: Automatic

   Initial State: N/A

   Input: N_o = 12000000 $\frac{mol}{mL}$; N_t= 5000000 $\frac{mol}{mL}$; t_t = 1 d; t_p = 30 d (TC-1-1)

   Output: k = 0.8755; N_p = 4.708e-05

   Test Case Derivation: Inputs within expected input range are used in the previous test to ensure the correct values produce an output.

   How test will be performed: Pytest will be implemented.

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, including verifiability, understandable, reusable, maintainable, and portable.

5.2.1 Verifiability

Testing the verifiability of Diagnose:

1. T-9:

   Type: Manual

   Initial State: Functional requirements system testing completed.

   Condition: Software must output expected values and errors stated in section **??**.

Result: Functional requirements of SRS are verified using unit and system testing.

How test will be performed: The test will be implemented manually inspecting the completion on the tests outlined in sections **??** and **??**. A successful test is specified by the verification of each test for functional requirements.

2. T-10:

   Type: Manual

   Initial State: Non-functional requirements testing completed.

   Condition: Software must pass tests for non-functional requirements in section **??**.

   Result: Non-functional requirements of SRS are verified using system testing.

   How test will be performed: The test will be implemented manually inspecting the completion on the tests outlined in sections **??**, **??**, **??** and **??**. A successful test is specified by the verification of each test for non-functional requirements in section **??**.

5.2.2 Understandability

Testing the understandability of Diagnose:

1. T-11:

   Type: Automatic

   Initial State: Python code generated.

   Input: Generated modular python code and TC-1-1 from table **??**.

   Output: Convention, refactor, warning and error messages.

   How test will be performed: Test will be performed by linting using Spyder. A successful test is specified by verification of no error messages.

5.2.3 Reliability

Testing the reliability of Diagnose:

1. T-12:

   Type: Automatic

   Initial State: Diagnose software run completed.

   Input: The code generated by Drasil and cProfile command.

   Result: The total run time, wall time per function call, the cumulative time for a given function.

   How test will be performed:
   The test for 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. A successful test is specified by a total run time of less than 0.5 seconds.

5.2.4 Maintainability

Testing the maintainability of Diagnose:

1. T-13:

   Type: Manual

   Initial State: Completed documentation for Diagnose, including SRS, VnV plan and VnV report.

   Input/Condition: Traceability Matrices in completed documentation.

   Output/Result: Traceability to Requirements confirmed.

   How test will be performed: This test will be inspected to ensure that the tests can be traced to the requirements and the requirements can be traced to the goals of Diagnose. A successful test is specified by the verification of connection between each test case to a requirement ensuring all requirements are met.

5.2.5 Portability

Testing the portability of Diagnose:

1. T-14:

   Type: Manual

   Initial State: andreamclemeno/Drasil Github repository cloned in OS.

   Condition: Perform T-1 test from section **??** by running the make command in the Drasil/code directory in terminal.

   Result: Successful test implies portability of software.

   How test will be performed: The test will be performed manual by executing the make command in terminal of different OS, including Linux, Windows, MacOS. A successful test is specified by the verification of each test for functional requirements. A successful test is specified by the GENERATED OUTPUT MATCHES STABLE VERSION indicator in terminal.

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 **??** shows which test cases are supporting which requirements.

	R1	R2	R3	R4	R5	NF1	NFR2	NFR3	NFR4	NFR5
T-??	X	X	X	X	X					
T-??	X	X								
T-??	X	X								
T-??	X	X								
T-??	X	X								
T-??	X	X								
T-??	X	X								
T-??	X	X	X	X	X					
T-??						X				
T-??						X				
T-??							X			
T-??								X		
T-??									X	
T-??										X
T-??			X							
T-??	X	X	X	X	X					

Table 4: Traceability Matrix Showing the Connections Between Requirements and Tests

6 Unit Test Description

The design of the generated python code into modules can be attributed to the Drasil framework. The documentation of the knowledge captured by Drasil to generate this code can be found in the Github repository: andreamclemeno/Drasil. The plans for unit testing the modules will be outlined in the following section.

6.1 Unit Testing Scope

The modules generated by Drasil for Diagnose include: Calculations.py, Control.py, InputConstraints.py, InputFormat.py, InputParameter.py and OutputFormat.py. Although the modules are generated, system or unit testing will be completed for all the modules to ensure verifiability. The testing type, verification tool and test reference can be found in Table **??**. Test reference names the test(s) where the testing plan can be found.

Module	Type	Verification Tool	Test Reference
Calculations.py	unit	unittest in Python	T-??
Control.py	unit	unittest in Python	T-??
InputConstraints.py	system	Pytest	T-??, T-??
InputFormat.py	system	Pytest	T-??, T-??
InputParameter.py	system	Pytest	T-??, T-??
OutputFormat.py	system	Pytest	T-??, T-??

Table 5: Diagnose Module Testing Type and Verification Tools

6.2 Tests for Functional Requirements

The verification for the functional requirements will be completed through automated testing using methods outlined in Table **??**. Unit testing processes are outlined for Calculations.py in section **??** and for Control.py in section **??**.

6.2.1 Calculations.py

1. T-15:

   Type: Automatic

   Initial State: Calculations.py generated from Drasil and test_Calculations.py created.

   Input: test_Calculations.py

   Output: Terminal output with OK indicator

   Test Case Derivation: Exceptions raised by Calculations.py are possible. If unittest fails because an exception is raised, Calculations.py is not behaving correctly. This test case was derived to determine behaviour irregularities.

   How test will be performed: Test is performed by implementing unittest in the terminal. In the directory of the test_Calculations.py code, the following command will be executed:

   ```
   python3 -m unittest test_Calculations.py
   ```

   The confirmation of OK indicator will specify a successful test.

6.2.2 Control.py

1. T-16:

   Type: Automatic

   Initial State: Control.py generated from Drasil and test_Control.py created.

   Input: test_Control.py

   Output: Terminal output with OK indicator

   Test Case Derivation: Exceptions raised by Control.py are possible. If unittest fails because an exception is raised, Control.py is not behaving correctly. This test case was derived to determine behaviour irregularities.

   How test will be performed: Test is performed by implementing unittest in the terminal. In the directory of the test_Control.py code, the following command will be executed:

   python3 -m unittest test_Control.py

   The confirmation of OK indicator will specify a successful test.

6.3 Tests for Nonfunctional Requirements

Planning for nonfunctional tests of units are not relevant for Diagnose.

6.4 Traceability Between Test Cases and Modules

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 **??** shows which test are supporting which modules.

	Calculations.py	Control.py	InputConstraints.py	InputFormat.py	InputParameter.py	OutputFormat.py
T-??	X	X	X	X	X	X
T-??						
T-??						
T-??						
T-??						
T-??						
T-??						
T-??	X	X	X	X	X	X
T-??						
T-??						
T-??						
T-??						
T-??						
T-??						
T-??			X			
T-??	X	X	X	X	X	X

Table 6: Traceability Matrix Showing the Connections Between Modules and Tests

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.

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.

MAX A. STAFFORD, LAWRENCE COREY, YUNZHEN CAO, ERIC S. DAAR, DAVID D. HO, and ALAN S. PERELSON. Modeling plasma virus concentration during primary hiv infection. *Journal of Theoretical Biology*, 203(3):285 – 301, 2000. ISSN 0022-5193. doi: https://doi.org/10.1006/jtbi.2000.1076. URL http://www.sciencedirect.com/science/article/pii/S0022519300910762.