System Verification and Validation Plan for Software Engineering

Team 8 – Rhythm Rangers

Ansel Chen
Muhammad Jawad
Mohamad-Hassan Bahsoun
Matthew Baleanu
Ahmed Al-Hayali

November 4, 2024

Revision History

Date	Version	Notes
Date 1	1.0	Notes
Date 2	1.1	Notes

[The intention of the VnV plan is to increase confidence in the software. However, this does not mean listing every verification and validation technique that has ever been devised. The VnV plan should also be a **feasible** plan. Execution of the plan should be possible with the time and team available. If the full plan cannot be completed during the time available, it can either be modified to "fake it", or a better solution is to add a section describing what work has been completed and what work is still planned for the future. —SS]

[The VnV plan is typically started after the requirements stage, but before the design stage. This means that the sections related to unit testing cannot initially be completed. The sections will be filled in after the design stage is complete. the final version of the VnV plan should have all sections filled in. —SS]

Contents

List of Tables

[Remove this section if it isn't needed —SS]

List of Figures

[Remove this section if it isn't needed —SS]

1 Symbols, Abbreviations, and Acronyms

symbol	description
T	Test

[symbols, abbreviations, or acronyms — you can simply reference the SRS (Author, 2019) tables, if appropriate —SS]

[Remove this section if it isn't needed —SS]

This document ... [provide an introductory blurb and roadmap of the Verification and Validation plan —SS]

2 General Information

2.1 Summary

[Say what software is being tested. Give its name and a brief overview of its general functions. —SS]

2.2 Objectives

[State what is intended to be accomplished. The objective will be around the qualities that are most important for your project. You might have something like: "build confidence in the software correctness," "demonstrate adequate usability." etc. You won't list all of the qualities, just those that are most important. —SS]

[You should also list the objectives that are out of scope. You don't have the resources to do everything, so what will you be leaving out. For instance, if you are not going to verify the quality of usability, state this. It is also worthwhile to justify why the objectives are left out. —SS]

[The objectives are important because they highlight that you are aware of limitations in your resources for verification and validation. You can't do everything, so what are you going to prioritize? As an example, if your system depends on an external library, you can explicitly state that you will assume that external library has already been verified by its implementation team. —SS]

2.3 Challenge Level and Extras

[State the challenge level (advanced, general, basic) for your project. Your challenge level should exactly match what is included in your problem statement. This should be the challenge level agreed on between you and the course instructor. You can use a pull request to update your challenge level (in TeamComposition.csv or Repos.csv) if your plan changes as a result of the VnV planning exercise. —SS]

[Summarize the extras (if any) that were tackled by this project. Extras can include usability testing, code walkthroughs, user documentation, formal proof, GenderMag personas, Design Thinking, etc. Extras should have already been approved by the course instructor as included in your problem statement. You can use a pull request to update your extras (in TeamComposition.csv or Repos.csv) if your plan changes as a result of the VnV planning exercise. —SS]

2.4 Relevant Documentation

[Reference relevant documentation. This will definitely include your SRS and your other project documents (design documents, like MG, MIS, etc). You can include these even before they are written, since by the time the project is done, they will be written. You can create BibTeX entries for your documents and within those entries include a hyperlink to the documents. —SS]

Author (2019)

[Don't just list the other documents. You should explain why they are relevant and how they relate to your VnV efforts. —SS]

3 Plan

[Introduce this section. You can provide a roadmap of the sections to come. —SS]

3.1 Verification and Validation Team

[Your teammates. Maybe your supervisor. You should do more than list names. You should say what each person's role is for the project's verification. A table is a good way to summarize this information. —SS]

3.2 SRS Verification Plan

The following approaches will be used for SRS verification:

- Formal reviews with the supervisor

- A checklist that will be given to the supervisor and any peer reviewers. It will also serve as a guide for the developers of the system
- Using feedback from grading to create new checklists and update existing checklists
- Ad-hoc reviews from peers and other teams in the course

This is the initial SRS checklist that reviewers will use. It will be updated as reviews are performed:

☐ Does each functional requirement have a detailed and accurate description, rationale and fit criteria?

☐ Is each requirement (both functional and non-functional) relevant and necessary?

☐ Are all functional requirements traceable to at least one use case?

☐ Are all fit critera unambiguous and verifiable?

☐ Have all issues opened by reviewers been closed?

3.3 Design Verification Plan

[Plans for design verification —SS]
[The review will include reviews by your classmates —SS]
[Create a checklists? —SS]

3.4 Verification and Validation Plan Verification Plan

[The verification and validation plan is an artifact that should also be verified. Techniques for this include review and mutation testing. —SS]
[The review will include reviews by your classmates —SS]
[Create a checklists? —SS]

3.5 Implementation Verification Plan

3.5.1 Static Analysis

Static analysis is the analysis of program content without execution. Below is a description of static analysis techniques we plan to implement as part of our testing process.

Linting Linters like `Pylint`, `Ruff`, or `Flake8` check for errors, enforce a coding standard, identify code smells, and can make code refactoring suggestions.

Formatting Formatters like `Black` and `Ruff` (indeed, Ruff is also a formatter) standardize code appearance and adhere to style guides, allowing the code reader to focus on code content.

Type Checking Type checkers like `mypy` ensure correct use of variables and functions in code using type hints, as outlined in PEP 484. Type hinting can also serve as documentation when publishing an API reference or developer guide.

Security Checking Static security checkers like `Bandit` find common security vulnerabilities in code, e.g., framework misconfiguration (`B2XX`, e.g., exposing the Flask debugger in a production application, allowing remote code execution), blacklist calls (`B3XX`, e.g., loading serialized pickle files), blacklist imports (`B4XX`, e.g., importing `ftplib` for insecure file transfer), cryptography (`B5XX`, e.g., missing certificate validation), and injection (`B6XX`, e.g., testing for SQL injection).

Code Metrics Analysis A code metrics analysis tool like `radon` can provide insights on various aspects of the codebase:

> **Raw metrics** Number of lines of source code (SLOC), logic (LLOC), comments, and whitespace.
>
> **Cyclomatic complexity** Number of decisions (or linearly independent paths) in a code block.
>
> **Halstead metrics** Statically-generated program metrics.

Composite Analysis Techniques Tools like `Prospector` combine multiple analysis techniques into one, i.e., linting via `Pylint`, `Pyflakes`, or `Ruff`,

PEP 8 and PEP 257 formatting via `pycodestyle` and `pydocstyle`, code complexity analysis via `McCabe`, simple security checking via `Dodgy`, packaging quality checking via `Pyroma`, unused modules checking via `Vulture`, type checking via `Mypy` or `Pyright`, and security checking via `Bandit`.

Code Walkthroughs Checklist-driven walkthroughs of featurization algorithms with other teammates and optionally the supervisor. Refer to the appendix section 6.3 for a sample checklist.

Peer Desk Checks Changes to the codebase will require approval from at least one other group member via a pull request review before merging to the main/production branch.

3.5.2 Dynamic Testing

Dynamic testing is the analysis of program runtime responses during and after execution. Below is a description of dynamic testing techniques we plan to implement as part of our testing process. For further details about the automated components, refer to section 3.6.

System Testing Tests are orchestrated via Testing orchestration tools like `tox` can manage all testing, from atomic unit tests to end-to-end system tests. System tests outlined in section 4 will be run to ensure necessary requirements are met.

Unit Testing Unit testing frameworks like `pytest` or `unittest` can verify that the implementation matches designs described in other system documents.

User Interface Testing UI components can be described as a set of discrete interactions, i.e., a transition model can capture user interactions as events then test it. Tools like the `Selenium` Python API automate web-based interactions, i.e., can serve as a testing framework to automate interaction sequences using the UI model.

Integration Testing Testing frameworks like `pytest` can be combined with modular fixtures or factories via `factoryboy` to simulate databases or other complex objects for testing operability between various interfaces, i.e., integration testing.

Regression Testing Persistence of tests across iterations of the project facilitates ease of regression testing, with automation via GitHub actions.

3.6 Automated Testing and Verification Tools

[What tools are you using for automated testing. Likely a unit testing framework and maybe a profiling tool, like ValGrind. Other possible tools include a static analyzer, make, continuous integration tools, test coverage tools, etc. Explain your plans for summarizing code coverage metrics. Linters are another important class of tools. For the programming language you select, you should look at the available linters. There may also be tools that verify that coding standards have been respected, like flake9 for Python. —SS]

[If you have already done this in the development plan, you can point to that document. —SS]

[The details of this section will likely evolve as you get closer to the implementation. —SS]

3.7 Software Validation Plan

[If there is any external data that can be used for validation, you should point to it here. If there are no plans for validation, you should state that here. —SS]

[You might want to use review sessions with the stakeholder to check that the requirements document captures the right requirements. Maybe task based inspection? —SS]

[For those capstone teams with an external supervisor, the Rev 0 demo should be used as an opportunity to validate the requirements. You should plan on demonstrating your project to your supervisor shortly after the scheduled Rev 0 demo. The feedback from your supervisor will be very useful for improving your project. —SS]

[For teams without an external supervisor, user testing can serve the same purpose as a Rev 0 demo for the supervisor. —SS]

[This section might reference back to the SRS verification section. —SS]

4 System Tests

[There should be text between all headings, even if it is just a roadmap of the contents of the subsections. —SS]

4.1 Tests for Functional Requirements

[Subsets of the tests may be in related, so this section is divided into different areas. If there are no identifiable subsets for the tests, this level of document structure can be removed. —SS]

[Include a blurb here to explain why the subsections below cover the requirements. References to the SRS would be good here. —SS]

4.1.1 Area of Testing1

[It would be nice to have a blurb here to explain why the subsections below cover the requirements. References to the SRS would be good here. If a section covers tests for input constraints, you should reference the data constraints table in the SRS. —SS]

Title for Test

1. test-id1

   Control: Manual versus Automatic

   Initial State:

   Input:

   Output: [The expected result for the given inputs. Output is not how you are going to return the results of the test. The output is the expected result. —SS]

   Test Case Derivation: [Justify the expected value given in the Output field —SS]

   How test will be performed:

2. test-id2

Control: Manual versus Automatic

Initial State:

Input:

Output: [The expected result for the given inputs —SS]

Test Case Derivation: [Justify the expected value given in the Output field —SS]

How test will be performed:

4.1.2 Area of Testing2

...

4.2 Tests for Nonfunctional Requirements

[The nonfunctional requirements for accuracy will likely just reference the appropriate functional tests from above. The test cases should mention reporting the relative error for these tests. Not all projects will necessarily have nonfunctional requirements related to accuracy. —SS]

[For some nonfunctional tests, you won't be setting a target threshold for passing the test, but rather describing the experiment you will do to measure the quality for different inputs. For instance, you could measure speed versus the problem size. The output of the test isn't pass/fail, but rather a summary table or graph. —SS]

[Tests related to usability could include conducting a usability test and survey. The survey will be in the Appendix. —SS]

[Static tests, review, inspections, and walkthroughs, will not follow the format for the tests given below. —SS]

[If you introduce static tests in your plan, you need to provide details. How will they be done? In cases like code (or document) walkthroughs, who will be involved? Be specific. —SS]

4.2.1 Area of Testing1

Title for Test

1. test-id1

Type: Functional, Dynamic, Manual, Static etc.

Initial State:

Input/Condition:

Output/Result:

How test will be performed:

2. test-id2

   Type: Functional, Dynamic, Manual, Static etc.

   Initial State:

   Input:

   Output:

   How test will be performed:

4.2.2 Area of Testing2

...

4.3 Traceability Between Test Cases and Requirements

[Provide a table that shows which test cases are supporting which requirements. —SS]

5 Unit Test Description

[This section should not be filled in until after the MIS (detailed design document) has been completed. —SS]

[Reference your MIS (detailed design document) and explain your overall philosophy for test case selection. —SS]

[To save space and time, it may be an option to provide less detail in this section. For the unit tests you can potentially layout your testing strategy here. That is, you can explain how tests will be selected for each module. For instance, your test building approach could be test cases for each access program, including one test for normal behaviour and as many tests as needed for edge cases. Rather than create the details of the input and output here,

you could point to the unit testing code. For this to work, you code needs to be well-documented, with meaningful names for all of the tests. —SS]

5.1 Unit Testing Scope

[What modules are outside of the scope. If there are modules that are developed by someone else, then you would say here if you aren't planning on verifying them. There may also be modules that are part of your software, but have a lower priority for verification than others. If this is the case, explain your rationale for the ranking of module importance. —SS]

5.2 Tests for Functional Requirements

[Most of the verification will be through automated unit testing. If appropriate specific modules can be verified by a non-testing based technique. That can also be documented in this section. —SS]

5.2.1 Module 1

[Include a blurb here to explain why the subsections below cover the module. References to the MIS would be good. You will want tests from a black box perspective and from a white box perspective. Explain to the reader how the tests were selected. —SS]

1. test-id1

   Type: [Functional, Dynamic, Manual, Automatic, Static etc. Most will be automatic —SS]

   Initial State:

   Input:

   Output: [The expected result for the given inputs —SS]

   Test Case Derivation: [Justify the expected value given in the Output field —SS]

   How test will be performed:

2. test-id2

   Type: [Functional, Dynamic, Manual, Automatic, Static etc. Most will be automatic —SS]

   Initial State:

   Input:

   Output: [The expected result for the given inputs —SS]

   Test Case Derivation: [Justify the expected value given in the Output field —SS]

   How test will be performed:

3. ...

5.2.2 Module 2

...

5.3 Tests for Nonfunctional Requirements

[If there is a module that needs to be independently assessed for performance, those test cases can go here. In some projects, planning for nonfunctional tests of units will not be that relevant. —SS]

[These tests may involve collecting performance data from previously mentioned functional tests. —SS]

5.3.1 Module ?

1. test-id1

   Type: [Functional, Dynamic, Manual, Automatic, Static etc. Most will be automatic —SS]

   Initial State:

   Input/Condition:

   Output/Result:

   How test will be performed:

2. test-id2

   Type: Functional, Dynamic, Manual, Static etc.

   Initial State:

   Input:

   Output:

   How test will be performed:

5.3.2 Module ?

...

5.4 Traceability Between Test Cases and Modules

[Provide evidence that all of the modules have been considered. —SS]

References

Author Author. System requirements specification. https://github.com/..., 2019.

6 Appendix

This is where you can place additional information.

6.1 Symbolic Parameters

The definition of the test cases will call for SYMBOLIC_CONSTANTS. Their values are defined in this section for easy maintenance.

6.2 Usability Survey Questions?

[This is a section that would be appropriate for some projects. —SS]

6.3 Code Walkthrough Checklist

Here is a sample code walkthrough checklist. It will be updated as reviews are performed:

Functionality

- ☐ Does the code perform its intended task?
- ☐ Can this code be traced to a requirement?
- ☐ Is there redundant or unnecessary code?

Readability

- ☐ Do the functions and variables have meaningful names?
- ☐ Does the source code contain sufficient commenting?
- ☐ Does the code follow PEP8 style guidelines

Modularity

- ☐ Are there excessively long functions that can be broken down?
- ☐ Is the code organized?
- ☐ Is the code easy to change or expand upon?

Error Handling

- ☐ Are exceptions and errors handled gracefully?
- ☐ Do exceptions and erros have useful error messages?

Testing

- ☐ Are there enough unit tests to achieve 100% code coverage in tox?
- ☐ Does tox report that 100% of unit tests are passing?
- ☐ Are there enough system tests to cover all requirements?
- ☐ Are all system tests passing?

Reliability

- ☐ Is the code fault-tolerant? I.e., does the system continue to operate despite failures/faults?
- ☐ Does the code have effective exception-handling and error recovery mechanisms?

Efficiency

- ☐ How much memory or processor capacity does the program consume?
- ☐ Are algorithms optimized, avoiding unnecessary operations?

Reusability

- ☐ Can components be reused in other applications/other parts of the application?
- ☐ Does the program have a well-partitioned, modular design with *strong cohesion* and *loose coupling*?

Scalability

- ☐ Can the system grow to accomodate more users, servers, data or other components?
- ☐ Can it do so with acceptable performance and at acceptable cost?

Appendix — Reflection

[This section is not required for CAS 741 —SS]

The information in this section will be used to evaluate the team members on the graduate attribute of Lifelong Learning.

The purpose of reflection questions is to give you a chance to assess your own learning and that of your group as a whole, and to find ways to improve in the future. Reflection is an important part of the learning process. Reflection is also an essential component of a successful software development process.

Reflections are most interesting and useful when they're honest, even if the stories they tell are imperfect. You will be marked based on your depth of thought and analysis, and not based on the content of the reflections themselves. Thus, for full marks we encourage you to answer openly and honestly and to avoid simply writing "what you think the evaluator wants to hear."

Please answer the following questions. Some questions can be answered on the team level, but where appropriate, each team member should write their own response:

1. What went well while writing this deliverable?

2. What pain points did you experience during this deliverable, and how did you resolve them?

3. What knowledge and skills will the team collectively need to acquire to successfully complete the verification and validation of your project? Examples of possible knowledge and skills include dynamic testing knowledge, static testing knowledge, specific tool usage, Valgrind etc. You should look to identify at least one item for each team member.

4. For each of the knowledge areas and skills identified in the previous question, what are at least two approaches to acquiring the knowledge or mastering the skill? Of the identified approaches, which will each team member pursue, and why did they make this choice?