System Verification and Validation Plan for BeamBending

Team Drasil
Jason Balaci

February 15, 2023

1 Revision History

Date	Version	Notes
Feb 12	0.0	Format template.
Feb 13	0.1	Preliminary work, read through and filling in easy spots.
Feb 14	0.2	Preliminary copy of "general information" section.
Feb 14	0.3	Preliminary copy of "plan" section.

Contents

List of Tables

2 Symbols, Abbreviations, and Acronyms

The Symbols, Abbreviations, and Acronyms in this document builds upon those from BeamBending's related SRS document [1].

Symbol	Description
CAS	Computing and Software department (McMaster University)
SRS	Software Requirements Specification
T	Test
VnV	Verification and Validation

3 General Information

This document describes the plan of action related to the Verification and Validation (VnV) of the Beam Bending analysis program (BeamBending). This VnV plan will describe a plan of action for *validating* that the Software Requirements Specification (SRS) [1] for BeamBending satisfies stakeholders, and *verifying* that a supposedly conforming software does indeed accurately satisfy the requirements.

3.1 Summary

The BeamBending Software Requirements Specification (SRS) [1] derives the software requirements of a hypothetical program that solves deflection analysis problems for simply-supported beams. The focus of the program is to predict how simply-supported beams handle imposed, distributed loads.

3.2 Objectives

The objective of this document is to outline a plan of action for:

1. *auditing* a continuously developed SRS document [2] for logical consistency,
2. *validating* said SRS satisfies stakeholder requirements, and
3. *verifying* that a produced software artifact conforms everything laid out in the SRS document (including, but not limited to, the functional and nonfunctional requirements), through both transparent and opaque testing.

In doing this, we hope to build confidence in the coherence of software requirements, and correctness and conformance of a software to said specifications.

3.3 Relevant Documentation

As we intend to build (generate) the software with Drasil, the only relevant documentation is that which is originally manually built, including:

1. the SRS document [1], and

2. this VnV plan.

When the BeamBending program is re-created in Drasil, we may think of that as a sort of "documentation" that we can to the above list.

4 Plan

The "whole" Verification and Validation plan for BeamBending consists of multiple sub-plans. Notably, it has a designated team (with sub-teams) who will be executing the related sub-plans stipulated in this document. Team members will take responsibility for various aspects of verification and validation.

4.1 Verification and Validation Team

Roles (Table 1) are assigned to each team member (Table 2), dictating the minimum responsibilities each member has for each related project.

Table 1: Table of VnV Roles

Role	Description/responsibilities
Authoritative Expert	Manager of all review committees, and distinguished reviewer and domain expert.
Domain Expert	Reviewer with considerable knowledge on underlying domains.
Author	Writer.
Reviewer	Ensures documents are logically coherent and well-formed.
Verifier	Assures Drasil encoding of SRS accurately re-creates the manually created SRS.
Validator	Assures SRS satisfies stakeholder requirements.
VnV-er	Verifier ∪ Validator.

JB: Odd...

Table 2: Table of VnV Teammates

Assignee	Project	Role(s)
Dr. Smith	∗[1]	Authoritative Expert[2].

[1] ∗: match all.
[2] Pretend we're not in CAS 741.

Jason Balaci	*	Author.
Sam Crawford	*	Domain Expert, Reviewer, and VnV-er.
Mina Mahdipour	SRS	Reviewer.
Deesha Patel	VnV	Reviewer.
Maryam Valian	Drasil	Reviewer & VnV-er.
Class of CAS 741	*	*

4.2 SRS Verification Plan

In addition to checking that BeamBending's Software Requirements Specification (SRS) conforms to Dr. Smith's provided SRS checklist, we will have:

1. a designated reviewing committee with a governing authoritative expert,
2. a public presentation with a reviewing audience,
3. built the project in Drasil, where we can build automated consistency checks and generate certain aspects of the document to avoid error,
4. at least one external reviewer (Dr. Jacques Carette of the Drasil project) when the whole BeamBending project is sent to the main Drasil repository for merging, and finally,
5. regular updates and sporadic reviews by current and future Drasil team members and onlookers (assuming the project is merged as an official case study of Drasil).

4.3 Design Verification Plan

The software design does not need verification as the design of the software will be based on Drasil's existing software family generator [3]. The onus of Drasil's validation is up to the Drasil team[3].

JB: Well, I am extending Drasil a bit.

[3]Including, but not limited to, Dr. Spencer Smith, Sam Crawford, and Jason Balaci.

4.4 Verification and Validation Plan Verification Plan

To assure that the Verification and Validation Plan adequately tests both the SRS document and the relevant software, we will largely assume the "many eyes" hypothesis [4] with many "eyes" of different skill-sets and academic backgrounds (see Table 2). Each team member should test that this document conforms to the general VnV Checklist document [5].

4.5 Implementation Verification Plan

A proof of concept should be built and manually tested. When the project is re-created in Drasil, the generated software artifacts should be similar to the proof of concept, up to code style and organization. The generated software artifacts should be tested against the manually created artifacts if non-trivial or significant differences exist. Additionally, as Drasil does not yet generate unit tests[4], the unit tests will be ported over to the generated software artifacts.

By re-writing the SRS with Drasil, the software implementation will be generated. We have faith in the Drasil work, and so, the "Implementation Verification Plan" is largely a "Solution Validation Plan" with an extra set of requirements for the configuration of Drasil's code generator. The solution proposed in the SRS is to be validated by peer review, code walkthrough, external audit, audit by assigned reviewers (see Table 1), and audit by the authoritative figure (Dr. Smith). The configuration requirements for Drasil's code generator are as follows:

1. generate code:
    (a) in Python,
    (b) with full code comment coverage[5], and
    (c) "full" modularity[6],

[4] But Sam might fix this for us!

[5] The ratio of the number of well-documented "code" components to the number of "code" components, where a code component is defined as any logical component of a codebase (such as functions, data types, classes, etc.).

[6] The generated software artifact should be broken up into multiple logically grouped software artifacts.

2. generate a Makefile with all common usage types (e.g., build, run, deps) as targets,

3. generate basic usage documentation, and

4. generate SRS artifacts from the same pool of knowledge used to build the previous two components.

4.6 Automated Testing and Verification Tools

The reference code implementation and final generated code artifacts will be tested (along with code coverage) using **pytest** [6] to automatically test the code against a series of unit tests (see Section 6). Continuous integration will be used to assure that changes to the SRS encoding in Drasil does not change against the well-tested artifacts[7]. The Python code will be aggressively formatted with Black [7].

4.7 Software Validation Plan

[TPLT — 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.]

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

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

[7] These are captured in the "stable" folder in Drasil's code repository, where "stable" artifacts remain manually tested.

5 System Test Description

5.1 Tests for Functional Requirements

[TPLT — 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.]

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

5.1.1 Area of Testing1

[TPLT — 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.]

Title for Test

1. T1

   Control: Automatic.

   Initial State: N/A.

   Input: $w(x) = 0$

   Output: $y_a(x) = 0$

   Test Case Derivation: WolframAlpha.

   How test will be performed: Observing $\forall x \in \mathbb{R} \; . \; y(x) \approx_\epsilon y_a(x)$ (with pytest using samples or symbolic equivalence, depending on solution).

2. T2

   Control: Automatic.

   Initial State: N/A.

   Input: $w(x) = 1$

   Output: $y_a(x) = \frac{x}{24}(x^3 - 20x^2 + 1000)$

Test Case Derivation: WolframAlpha.

How test will be performed: Observing $\forall x \in \mathbb{R} \; . \; y(x) \approx_\epsilon y_a(x)$ (with pytest using samples or symbolic equivalence, depending on solution).

3. T3

   Control: Automatic.

   Initial State: N/A.

   Input: $w(x) = -1$

   Output: $y_a(x) = -\frac{x}{24}(x^3 - 20x^2 + 1000)$

   Test Case Derivation: WolframAlpha.

   How test will be performed: Observing $\forall x \in \mathbb{R} \; . \; y(x) \approx_\epsilon y_a(x)$ (with pytest using samples or symbolic equivalence, depending on solution).

4. T4

   Control: Automatic.

   Initial State: N/A.

   Input: $w(x) = x$

   Output: $y_a(x) = \frac{x}{360}(3x^4 - 1000x^3 + 70000)$

   Test Case Derivation: WolframAlpha.

   How test will be performed: Observing $\forall x \in \mathbb{R} \; . \; y(x) \approx_\epsilon y_a(x)$ (with pytest using samples or symbolic equivalence, depending on solution).

5. T5

   Control: Automatic.

   Initial State: N/A.

   Input: $w(x) = -x$

   Output: $y_a(x) = -\frac{x}{360}(3x^4 - 1000x^3 + 70000)$

   Test Case Derivation: WolframAlpha.

   How test will be performed: Observing $\forall x \in \mathbb{R} \; . \; y(x) \approx_\epsilon y_a(x)$ (with pytest using samples or symbolic equivalence, depending on solution).

6. T6

   Control: Automatic.

   Initial State: N/A.

   Input: $w(x) = 80000x^3$

   Output: $y_a(x) = \frac{20000x}{21}(x^6 - 70000x^2 + 6000000)$

   Test Case Derivation: WolframAlpha.

   How test will be performed: Observing $\forall x \in \mathbb{R} \; . \; y(x) \approx_\epsilon y_a(x)$ (with pytest using samples or symbolic equivalence, depending on solution).

7. T7

   Control: Automatic.

   Initial State: N/A.

   Input: $w(x) = 80000x^2$

   Output: $y_a(x) = \frac{2000x}{9}(x^5 - 5000x^4 + 400000)$

   Test Case Derivation: WolframAlpha.

   How test will be performed: Observing $\forall x \in \mathbb{R} \; . \; y(x) \approx_\epsilon y_a(x)$ (with pytest using samples or symbolic equivalence, depending on solution).

8. T8

   Control: Automatic.

   Initial State: N/A.

   Input: $w(x) = 800000 \sin\left(\frac{x\pi}{L}\right)$

   Output: $y_a(x) = \frac{1}{3\pi^4}(40000L^2x(\pi^2(x^2-100)-6L^2)\sin\left(\frac{10\pi}{L}\right)+60L^2\sin\left(\frac{x\pi}{L}\right))$

   Test Case Derivation: WolframAlpha.

   How test will be performed: Observing $\forall x \in \mathbb{R} \; . \; y(x) \approx_\epsilon y_a(x)$ (with pytest using samples or symbolic equivalence, depending on solution).

9. T9

Control: Automatic.

Initial State: N/A.

Input: $w(x) = 800000 \sin\left(\frac{2x\pi}{L}\right)$

Output: $y_a(x) = \frac{1}{3\pi^4}(5000L^2x(2\pi^2(x^2-100)-3L^2)\sin\left(\frac{20\pi}{L}\right)+30L^2\sin\left(\frac{2x\pi}{L}\right))$

Test Case Derivation: WolframAlpha.

How test will be performed: Observing $\forall x \in \mathbb{R} \; . \; y(x) \approx_\epsilon y_a(x)$ (with pytest using samples or symbolic equivalence, depending on solution).

5.1.2 Area of Testing2

...

5.2 Tests for Nonfunctional Requirements

[TPLT — 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]

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

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

5.2.1 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:

5.2.2 Area of Testing2

...

5.3 Traceability Between Test Cases and Requirements

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

6 Unit Test Description

[TPLT — Reference your MIS (detailed design document) and explain your overall philosophy for test case selection.] [TPLT — This section should not be filled in until after the MIS (detailed design document) has been completed.]

6.1 Unit Testing Scope

[TPLT — 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.]

6.2 Tests for Functional Requirements

[TPLT — 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.]

6.2.1 Module 1

[TPLT — 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.]

1. test-id1

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

   Initial State:

   Input:

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

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

How test will be performed:

2. test-id2

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

   Initial State:

   Input:

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

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

   How test will be performed:

3. ...

6.2.2 Module 2

...

6.3 Tests for Nonfunctional Requirements

[TPLT — 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.]

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

6.3.1 Module ?

1. test-id1

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

   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:

6.3.2 Module ?

...

6.4 Traceability Between Test Cases and Modules

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

References

[1] Jason Balaci. "Beam Bending: examining a beam bending under load". In: *CAS 741 (Winter 2023)* (2023). Ed. by Sam. Crawford, Dr. Spencer Smith, and Class of CAS 741 (Winter 2023) (cit. on pp. v, 1, 16).

[2] David L. Parnas and P.C. Clements. "A Rational Design Process: How and Why to Fake it". In: *IEEE Transactions on Software Engineering* 12.2 (February 1986), pp. 251–257 (cit. on p. 1).

[3] The Drasil Team. *Drasil*. 2023-01. URL: https://github.com/JacquesCarette/Drasil (cit. on p. 4).

[4] Thomas Caraco, Steven Martindale, and H Ronald Pulliam. "Avian time budgets and distance to cover". In: *The Auk* 97.4 (1980), pp. 872–875 (cit. on p. 5).

[5] Spencer Smith. *capTemplate*. As at git blob #92517. 2023. URL: https://github.com/smiths/capTemplate/ (cit. on p. 5).

[6] Holger Krekel et al. *PyTest*. 2004. URL: https://docs.pytest.org/en/7.2.x/ (cit. on p. 6).

[7] Łukasz Langa et al. *Black: The uncompromising code formatter*. 2018. URL: https://black.readthedocs.io/en/stable/index.html (cit. on p. 6).

7 Appendix

7.1 Symbolic Parameters

In addition to the symbolic parameters from the SRS document [1], we will add ϵ, where $\epsilon = 10^{-3}$ (m), for usage as a tolerance for equivalence.

7.2 Usability Survey Questions

As the project will rely on Drasil to build the software from the requirement description, any and all "usability" and/or "accessibility" concerns should be directed towards the Drasil team as BeamBending will only use their basic (stable) public-facing tooling.