System Verification and Validation Plan for BeamBending

Team Drasil
Jason Balaci

February 17, 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.
Feb 14	0.4	Preliminary dubious "system" (unit) tests.
Feb 16	0.5	I guess the system tests weren't as dubious as I thought! Cleaning up as per in-class feedback (E, I, zeroes)
Feb 17	0.6	Complete draft.

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] describes the requirements of a hypothetical program that analyzes beam deflection under imposed, distributed loads on a simply-supported beam.

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.

add.

You will also add the VnV Report

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
Supervisor	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 $\cup$ Validator.

Table 2: Table of VnV Teammates

Assignee	Project	Role(s)
Dr. Smith	*[1]	Supervisor.

[1] *: match all.

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 supervisor,
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]. However, in order to build the Boundary Value Problem (BVP) in Drasil, we will need to extend Drasil to generate BVP solving methods[2]. The onus of Drasil's validation is up to the Drasil team[3].

[2]This will be done and assumed as "trusted" when accepted into Drasil's main codebase [3].

[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 supervisor (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.

This is a good idea. The "many pairs of eyes" review only gets you so far. Something rigorous and systematic would be good. I think you should propose a code walkthrough for a critical piece of code and add it to one of our Drasil meetings. Please add details to your plan on how you would propose running the walkthrough. Include a citation to any relevant literature.

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

As the problem described in the SRS is similar to beam deflection problems commonly found in engineering textbooks (such as [8]), we will assume a potential stakeholder is a writer of one of said textbooks. Dr. Spencer Smith will also be an assumed stakeholder in the project as he suggested this project to the author. Input, output, and theory-based inspection will primarily be done to ensure that that information contained in the SRS and the software satisfies stakeholders.

[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

The tests for functional requirements may be split up into 3 categories, as follows:

1. testing that inputs match the understood inputs (R2 [1]),
2. testing that the BVP solver functions as expected, and
3. testing that the whole program accurately follows the instance models as described in the SRS (R3 [1]).

5.1.1 Testing Inputs Are Outputted Accurately

Throughout the next two categories of testing the functional requirements, we will have tests on the program, and each test should be additionally automatically checked that the re-iterated outputs match the intended inputs.

5.1.2 Testing BVP Solver

All of the tests for testing the BVP solver (Table 3) are done *automatically* with a trivially "empty" initial state (e.g., the program is not started and has been provided no inputs yet), and trivial inputs other than the BVPs themselves. The focus of this section is to test the BVP solver. The inputs should be provided as appropriate and the expected output should be printed. Each test will observe $\forall x : \mathbb{R} \ . \ x \in (0, L_B) \Rightarrow (y(x) \approx_\epsilon y_a(x))$ (with pytest using samples or symbolic equivalence, depending on solution). The expected outputs are confirmed using WolframAlpha [9].

clarify that y(x) is the calculated solution and ya(x) is the expected solution.

Table 3: Simple, Automatic, Tests

ID	**Inputs**			**Outputs**	**Control**
	$w_B(x)$	E_B	I_B	$y_a(x)$	
$\mathrm{T1}_{\mathrm{BVP}}$	0	1	1	0	WolframAlpha
$\mathrm{T2}_{\mathrm{BVP}}$	1	1	1	$\frac{x}{24}(x^3 - 20x^2 + 1\mathrm{E}^3)$	WolframAlpha
$\mathrm{T3}_{\mathrm{BVP}}$	-1	1	1	$-\frac{x}{24}(x^3 - 20x^2 + 1\mathrm{E}^3)$	WolframAlpha
$\mathrm{T4}_{\mathrm{BVP}}$	x	1	1	$\frac{x}{360}(3x^4 - 1\mathrm{E}^3x^3 + 7\mathrm{E}^4)$	WolframAlpha
$\mathrm{T5}_{\mathrm{BVP}}$	$-x$	1	1	$-\frac{x}{360}(3x^4 - 1\mathrm{E}^3x^3 + 7\mathrm{E}^4)$	WolframAlpha
$\mathrm{T6}_{\mathrm{BVP}}$	$8\mathrm{E}^4x^3$	1	1	$\frac{2\mathrm{E}^4x}{21}(x^6 - 7\mathrm{E}^4x^2 + 6\mathrm{E}^6)$	WolframAlpha
$\mathrm{T7}_{\mathrm{BVP}}$	$8\mathrm{E}^4x^2$	1	1	$\frac{2\mathrm{E}^3x}{9}(x^5 - 5\mathrm{E}^3x^4 + 4\mathrm{E}^5)$	WolframAlpha
∞ $\mathrm{T8}_{\mathrm{BVP}}$	$8\mathrm{E}^5\sin\left(\frac{x\pi}{L}\right)$	1	1	$\frac{1}{3\pi^4}(4\mathrm{E}^4L^2x(\pi^2(x^2-100) - 6L^2)\sin\left(\frac{10\pi}{L}\right) + 60L^2\sin\left(\frac{x\pi}{L}\right))$	WolframAlpha
$\mathrm{T9}_{\mathrm{BVP}}$	$8\mathrm{E}^5\sin\left(\frac{2x\pi}{L}\right)$	1	1	$\frac{1}{3\pi^4}(5\mathrm{E}^3L^2x(2\pi^2(x^2-100) - 3L^2)\sin\left(\frac{20\pi}{L}\right) + 30L^2\sin\left(\frac{2x\pi}{L}\right))$	WolframAlpha

It would be good to have a test with typical values for E, I, w(x), L

I see you were thinking the same thing.
Please be specific about the values for E & I.

5.1.3 Testing Outputs with Non-trivial Inputs

All of the tests from Table 3 should be performed again[8] with non-trivial E_Bs and I_Bs (e.g., not 1). Since numeric scaling isn't very consequential to the output, we will omit for brevity. WolframAlpha may be similarly used as a control.

5.2 Tests for Nonfunctional Requirements

The nonfunctional requirements are relatively uncomplicated to audit, mostly because of the usage of Drasil:

T_{NFR1} **Accuracy** is satisfied primarily through the tests of the functional requirements having a low tolerance[9],

T_{NFR2} **Usability** is strongly tied to Drasil's ability to generate code that can output data[10],

T_{NFR3} **Maintainability** is satisfied through being constructed in Drasil, where changes in information have rippling effects and re-generation allows us to update everything to accomodate changes,

T_{NFR4} **Portability** is satisfied because we aim to generate Python code, but also because all of Drasil's supported output languages are supported on the 3 major personal operating systems.

5.3 Traceability Between Test Cases and Requirements

The following table traces the test cases as shown in the earlier sections back to the functional and nonfunctional requirements[11].

[8]Note: referencing here will be done with the BVP subscript removed, $T*$.

[9]As this software is purely educational, accepting a higher tolerance is fine too.

[10]Unfortunately, list-like functionality remains limited, but will be improved. Also note that "usability" was defined in the SRS document. Specifically, we will not be testing for accessibility nor any other facet as this software is meant to be an intermediate program used for calculation, not visualization.

[11]Note that $*$ is used to quantify over each individual test case as it is redundant to have identical rows for the tests that are each intended to test the same concepts.

Table 4: Tracing Tests to Requirements

	R1	R2	R3	NFR1	NFR2	NFR3	NFR4
$T*_{\text{BVP}}$	X	X					
$T*$	X	X	X				
T_{NFR1}	X	X	X	X			
T_{NFR2}					X		
T_{NFR3}						X	
T_{NFR4}							X

6 Unit Test Description

As no software design documents will be constructed for Team Drasil's projects, we will bootstrap the Drasil-generated software artifacts for testing. This section will be filled in once we have Drasil generating code.

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. iv, 1, 7, 13).

[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 (1986-02), 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).

[8] Ferdinand P. Beer and E. Russell Johnston Jr. *Mechanics of Materials*. McGraw-Hill Ryerson, 1981 (cit. on p. 6).

[9] Wolfram Research Inc. *Wolfram Alpha*. Accessed on Feb. 17th, 2023. 2023. URL: https://www.wolframalpha.com (cit. on p. 7).

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.