System Verification and Validation Plan for BeamBending

Team Drasil
Jason Balaci

April 19, 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 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.

4 Plan

The Verification and Validation Plan for BeamBending consists of multiple smaller plans. Notably, it has a designated team (with subteams) who will be executing the related 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. Note that Sam Crawford is also the "primary" reviewer (at least for as long as CAS 741 runs).

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 SRS conforms to Dr. Smith's provided SRS checklist[2], we will have:

1. a designated reviewing committee with a supervisor and a primary reviewer,

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).

A further external audit may be needed if the software is to be used in non-educational applications[3]. As this is out of scope, we will provide no instruction.

[2]Which all reviewers should do loosely, but that my primary reviewer (Sam Crawford) is in charge of.

[3]Note that this software, documentation, and the likes is purely for educational purposes and hence comes with no warranty and no liability by the authors.

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[4]. The onus of Drasil's validation is up to the Drasil team[5].

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][6] with many "eyes" of different skill-sets[7] and academic backgrounds (see Table 2 and the general Drasil research team present and future). Each team member should test that this document conforms to the general VnV Checklist document [6].

4.5 Implementation Verification Plan

A proof of concept should be built and manually tested. When the project is recreated 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 to see if there are nontrivial or significant differences (to assure there are none[8]). Additionally, as Drasil does not yet generate unit tests[9], the unit tests will be performed on the generated software artifacts with a bit of manual effort to ensure compatibility.

By re-writing the SRS with Drasil, the software implementation will be generated. We have faith in the Drasil work[10], and so, the "Implementation

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

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

[6] It might help to think of Linus's Law [5] here.

[7] It might help to think of Linus's Law [5] here.

[8] Unless the manually created artifacts had problems, of course.

[9] But Sam might fix this for us!

[10] The faith is okay for now as Drasil has already been audited by enough reviewers. However, we should hopefully one day have a more concrete SRS document for each related aspect of Drasil that we can use to audit Drasil itself, to ultimately assure programs are

Verification Plan" is largely a "Solution Validation Plan" with an extra set of requirements for the inputs and configuration of Drasil's code generator to also be validated.

Disclaimer: the scheme for auditing the Drasil-encoding of the SRS, and the Drasil-generated solutions is fairly conventional (or so I believe), and so is reiterated here for educational purposes.

After the SRS has been verified (Section 4.2), the "Drasil" aspect of the project may be similarly verified by peer review, code walkthrough, external audit, audit by assigned reviewers (see Table 1), and audit by the supervisor (Dr. Smith). However, the focus of this will need to shift towards assuring that the encoding of the SRS is of nontrivial depth and breadth[11] and accurately represents the original SRS. The code walkthrough in particular should be done with fellow Drasil researchers to further assure that the capture of knowledge is indeed accurate and of sufficient depth and breadth.

By auditing the encoding of the SRS, we are effectively auditing the "input" given to Drasil. The configuration requirements for Drasil's code generator are as follows:

1. generate code:
    (a) in Python,
    (b) with full code comment coverage[12], and
    (c) with "full" modularity[13],
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.

---

indeed correct by construction.

[11] Whereby nontrivial structure of the knowledge is captured by the encoding. See Chapter 2 of [7] for a "deeper" explanation.

[12] 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.).

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

The configuration requirements should be verified similarly by all reviewers.

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* [8] 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[14]. The Python code will be "aggressively formatted" with Black [9].

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 [10]), 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 Jason Balaci (the author of this work). "Input," "output," and "theory"-focused inspection will primarily be done to ensure that that information contained in the SRS and the software satisfies stakeholders.

[14]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 Inputs Are Outputted Accurately

When conducting all other functional tests on the program, each test should additionally check to ensure that the inputs are re-iterated by the program before any sort of calculation. This may be done automatically by observing the outputs of the program, or manually in a similar fashion.

5.1.2 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). Please see the SRS document for reasonable values for E_B and I_B, obeying the constraints set forth in the SRS. The expected outputs should be confirmed using WolframAlpha [11].

Once all tests with trivial inputs are completed, all of the tests from Table 3 should be performed again[15] with nontrivial E_B and I_B (e.g., not 1). Since numeric scaling isn't very consequential to the output, we will omit for brevity. Testing with randomized inputs is a good strategy, similar to how testing is done via QuickCheck, and should be used with a reasonable

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

range of values (see the Table of Software and Physical constraints in the related SRS document). WolframAlpha may be similarly used as a control, but having a trusted solver locally may be beneficial.

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

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[16],

T_{NFR2} **Usability** is strongly tied to Drasil's ability to generate code that can output data[17],

T_{NFR3} **Maintainability** is satisfied through being constructed in Drasil, where changes in information have rippling effects and regeneration allows us to update everything to accommodate 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 three (3) major personal operating systems.

As Drasil is used to generate the code and is already fairly mature, we need no further work to assure these non-functional requirements are satisfied. All future improvements to Drasil will likely further improve the generated artifacts, and the qualities thereof. So as long as Drasil is sufficiently tested and the encoding of the SRS documents is audited, then the non-functional requirements should all be satisfied by construction[18].

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[19].

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

[17] 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.

[18] A kind of "correct by construction"-style programming is created through Drasil.

[19] 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	R4	NFR1	NFR2	NFR3	NFR4
$T{*}_{\text{BVP}}$	X	X						
$T{*}$	X	X	X	X				
T_{NFR1}				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 (by setting up a testing environment around the software artifacts). This section will be filled in once we have Drasil generating code[20].

[20]Or generated, if Sam finishes his work first!

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, 8, 15).

[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. 5).

[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] Wikipedia contributors. *Linus’s law — Wikipedia, The Free Encyclopedia*. [Online; accessed 19-April-2023]. 2022. URL: https://en.wikipedia.org/w/index.php?title=Linus%27s_law&oldid=1080337490 (cit. on p. 5).

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

[7] Jason Balaci. “Adding Types and Theory Kinds to Drasil”. MA thesis. McMaster University, 2022 (cit. on p. 6).

[8] Holger Krekel et al. *PyTest*. 2004. URL: https://docs.pytest.org/en/7.2.x/ (cit. on p. 7).

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

[10] Ferdinand P. Beer and E. Russell Johnston Jr. *Mechanics of Materials*. McGraw-Hill Ryerson, 1981 (cit. on p. 7).

[11] Wolfram Research Inc. *Wolfram Alpha*. Accessed on Feb. 17th, 2023. 2023. URL: https://www.wolframalpha.com (cit. on p. 8).

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.