Student Projects on `rse_code_annotations`

Seminar papers, Bachelor's and Master's theses

Julian Dehne

2026-07-29

Repository: github.com/juliandehne/rse_code_annotations
Contact: julian.dehne@gi.de

1 Context

Research software is a standard component of the research process (Gruenpeter et al. 2021; Felderer et al. 2025), and a growing share of it is written by a machine. The author who publishes on top of that code still answers for whether the result is valid.

The established quality criteria of the field — FAIR4RS, reproducibility, reusability (Barker et al. 2022) — are criteria of *reliability*: the same code produces the same output twice. They say nothing about *validity*, that is, whether the code measures what it is supposed to measure (Schnell et al. 1999).

`rse_code_annotations` addresses that gap under a specific assumption: not all research code carries the scientific claim. Most of it is glue — argument parsing, plotting, moving files. Validity rests on a smaller subset: the mathematics, and the points where data enters and leaves the program. The library marks that subset so the review budget can be spent on it. It contains no LLM and makes no network calls.

2 The tool

2.1 Four annotations

Four decorators attach reflection metadata to a function without changing its runtime behaviour.

Annotation	Marks	Tool support
`@functional`	pure mathematical function	inferred formula for inspection; determinism and expected-value stubs
`@mapping`	transforms one format or object into another	shape-transform stub
`@data_input`	boundary where data enters (reads a file)	`tmp_path` read test
`@data_output`	boundary where data leaves (writes a file)	`tmp_path` write test

```python
from rse_annotations import functional, data_input

@data_input(fields={"path": "codings as CSV"})
def load_codes(path):
    """:returns rows: one row per coding."""
    return list(csv.DictReader(open(path, newline="")))

@functional
def normalize(value, lo, hi):
    """:returns: (value - lo) / (hi - lo)."""
    return (value - lo) / (hi - lo)
```

2.2 Current state

The command-line tool has three modes.

Inspect (`--inspect`) derives the mathematical formula of each `@functional` function from its source and presents it for a verdict; verdicts are written to `inspection.yaml`. Three backends run in parallel: AST rendering (no dependencies), symbolic execution via SymPy, and `latexify`. The formula is rendered for a human to judge, never proved. Where no closed form is available — Krippendorff's α is the standard case — a generic differential check compares the function against a reference implementation on the same seeded random inputs.

Stubs (`--stubs`) emits one `pytest` scaffold per annotation, following a fixed per-kind pattern. Every stub body calls `skip()`, so nothing passes silently.

Coverage (`--coverage`) reports how much of a codebase is annotated and which functions are the next candidates, inferring the appropriate kind from the shape of each function body. It is purely static: every `*.py` file is parsed with `ast`, nothing is imported and nothing is executed.

A running annotation study (`lni_study`, roughly 250 functions) serves as the test bed for all topics below.

3 Topics

Each topic is a bounded extension of the existing library plus an evaluation against the test bed. Design science research is the fitting methodological frame (Vom Brocke, Hevner, and Maedche 2020).

3.1 Metamorphic relations as a fourth mode

Scope: Bachelor's thesis

Background. The generated stubs require an expected value. For research code that value often does not exist — nobody knows the correct α in advance. Metamorphic testing addresses this test-oracle problem by checking a relation between several runs rather than a single result (Chen, Cheung, and Yiu 1998): a permutation of the coders must not change α; an additional unanimous coder must not lower it.

Task. Catalogue metamorphic relations per annotation kind and generate them. For `@functional`: invariance under permutation, scaling and additivity behaviour. For `@mapping`: preservation relations such as cardinality and key set. For `@data_input`/`@data_output`: round-trip relations (read $\circ$ write = id). Implement the catalogue as a fourth mode of the CLI, alongside inspect, stubs and coverage.

Deliverables. (a) the catalogue with its rationale; (b) a generator emitting runnable `pytest` cases from an annotation; (c) an evaluation on the test bed reporting how many functions receive an oracle-free test, and what share of injected faults (mutations) those tests catch.

Prerequisites. Python, `pytest`, some AST work. No prior testing theory required.

3.2 Contracts instead of stubs, decided by an SMT solver

Scope: Bachelor's thesis

Background. A stub is a request to a human. A contract is a statement about the function that a machine can decide. For the small arithmetic cores marked by `@functional`, the second is within reach.

Task. Extend `@functional` with optional pre- and postconditions (`@functional(requires="lo < hi", ensures="0 <= result <= 1")`), translate them to SMT-LIB and let Z3 decide them — yielding a proof over the covered fragment, or a counterexample that serves directly as a test case. Determine which fragment of Python remains translatable: integers, real arithmetic, bounded loops, lists. Document where the translation breaks down.

Deliverables. (a) the extended annotation and its translator; (b) a classification of translatable constructs with evidence; (c) an evaluation on the test bed: for how many real `@functional` functions does the solver return a verdict rather than a timeout?

Prerequisites. Python; willingness to learn Z3 and SMT-LIB. Formal logic at the level of an introductory course.

Delimitation. Not a full verifier. The target is the cheap subset.

3.3 Deductive verification of selected `@functional` cores

Scope: Master's thesis

Background. The formula is currently rendered and checked differentially against a reference implementation; neither step proves anything. Deductive program verification (Dafny, Why3, Frama-C) checks a function against pre- and postconditions; proof assistants (Coq, Lean) address the mathematics itself. Machine-checked mathematics at scale is long established (Robertson et al. 1996); the open question is what it costs at the scale of a single research project.

Task. Select five to eight `@functional` cores from the test bed on stated criteria and verify them in Dafny or Why3. Typical research mathematics rather than textbook algorithms: weighted means, agreement measures, normalisations, bootstrap statistics. Record the effort per core — lines of specification per line of code, number of proof hints, wall-clock time.

Deliverables. (a) the verified cores as a reusable artifact; (b) an effort log with its methodology; (c) a recommendation stating which classes of function justify a proof and which do not.

Prerequisites. Solid logic and semantics background. Dafny or Why3 can be learned during the thesis.

3.4 Empirical evaluation of the coverage heuristic

Scope: Master's thesis

Background. The coverage report proposes an annotation kind for every unannotated function based on the shape of its body: a file write → `@data_output`; pure arithmetic → `@functional`. Whether this heuristic identifies the functions that actually carry the scientific claim is untested.

Task. Construct a gold standard: several coders independently annotate a sample of real research projects for which functions carry the claim; report intercoder reliability; consolidate. Measure the heuristic against it. Report precision and recall per annotation kind, and analyse the false negatives — which claim-bearing functions does the heuristic miss, and why? The instruments are those of empirical social research (Schnell et al. 1999) applied to a software engineering object.

Deliverables. (a) an open gold-standard data set; (b) the performance measures with an error analysis; (c) revised heuristics, demonstrated against the data set.

Prerequisites. Empirical methods (sampling, intercoder reliability), Python for the analysis. Suited to measuring rather than building.

3.5 Extending stub generation

Scope: Master's thesis

Background. The current stubs follow a fixed pattern per annotation kind. They ignore what the code states about itself: signature, type annotations, docstring fields, value ranges, and the actual call sites in the project.

Task. Enrich generation so that a stub approaches a runnable test. Derive input generators from types and docstrings (connecting to property-based testing); harvest real arguments from call sites; incorporate the metamorphic relations from Topic 3.1; derive edge cases from the branches of the function body. Establish where automation should stop — a test that passes silently is worse than no test (Bourque and Fairley 2014).

Deliverables. (a) the extended generator; (b) a comparison on the test bed covering mutation score and manual effort against the current stubs; (c) an explicit discussion of the automation boundary.

Prerequisites. Good Python, AST work, `pytest` and `hypothesis`.

4 Supervision

The topics can be resized — larger for a Master's thesis, smaller for a seminar paper. Topic 3.1 and Topic 3.5 combine well if two students work in parallel. Own proposals in the same area are welcome. Related prior work on research software engineering in the natural sciences: Dehne et al. (2025).

Contact: `julian.dehne@gi.de`.

References

Barker, Michelle, Neil P. Chue Hong, Daniel S. Katz, Anna-Lena Lamprecht, Carlos Martinez-Ortiz, Fotis Psomopoulos, Jennifer Harrow, et al. 2022. "Introducing the FAIR Principles for Research Software." *Scientific Data* 9 (1): 622. https://doi.org/10.1038/s41597-022-01710-x.

Bourque, Pierre, and Richard E. Fairley, eds. 2014. *Guide to the Software Engineering Body of Knowledge (SWEBOK Guide): Version 3.0.* IEEE Computer Society. https://www.computer.org/education/bodies-of-knowledge/software-engineering/v3.

Chen, T. Y., S. C. Cheung, and S. M. Yiu. 1998. "Metamorphic Testing: A New Approach for Generating Next Test Cases." HKUST-CS98-01. Hong Kong: Department of Computer Science, The Hong Kong University of Science; Technology. https://arxiv.org/abs/2002.12543.

Dehne, Julian, Sebastian Christ, Fabian Goth, Jan-Niklas Grad, Magnus Hagdorn, Jan P. Thiele, Harald von Waldow, and Jonas Linxweiler. 2025. "Research Software Engineering for Natural Sciences." In *Softwaretechnik-Trends*, edited by Andrea Herrmann. Vol. 45. 3. Berlin: Gesellschaft für Informatik e.V.

Felderer, Michael, Michael Goedicke, Lars Grunske, Wilhelm Hasselbring, Anna-Lena Lamprecht, and Berhard Rumpe. 2025. "Investigating Research Software Engineering: Toward RSE Research." *Communications of the ACM* 68 (2): 20–23.

Gruenpeter, Morane, Daniel S. Katz, Anna-Lena Lamprecht, Tom Honeyman, Daniel Garijo, Alexander Struck, Anna Niehues, et al. 2021. "Defining Research Software: A Controversial Discussion." Zenodo. https://doi.org/10.5281/ZENODO.5504016.

Robertson, Neil, Daniel P. Sanders, Paul Seymour, and Robin Thomas. 1996. "A New Proof of the Four-Colour Theorem." *Electronic Research Announcements of the American Mathematical Society* 02 (01): 17–26. https://doi.org/10.1090/S1079-6762-96-00003-0.

Schnell, Rainer, Paul B Hill, Elke Esser, et al. 1999. *Methoden der empirischen Sozialforschung.* Vol. 6. R. Oldenbourg München ua.

Vom Brocke, Jan, Alan Hevner, and Alexander Maedche. 2020. "Introduction to Design Science Research." In *Design Science Research. Cases*, edited by Jan Vom Brocke, Alan Hevner, and Alexander Maedche, 1–13. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-030-46781-4_1.