



Why3-py: A Tool for Formal Verification of Hypothesis Testing and Meta-Analysis in Python[★]

Akira Tanaka¹[0000–0002–0820–0766] and Yusuke Kawamoto¹[0000–0002–2151–9560]

National Institute of Advanced Industrial Science and Technology (AIST), Tokyo,
Japan

Abstract. The reproducibility crisis in scientific research has received widespread recognition, thereby increasing the importance of meta-analyses that integrate statistical analyses from multiple studies. However, statistical methods often have ambiguous and implicit underlying assumptions, which can lead to their erroneous applications and interpretations. To address this issue, we propose a formal verification framework for statistical Python programs. Specifically, we present **Why3-py**, a Python front-end for the **Why3** verification platform that transforms Python code into verification-oriented **WhyML** representations, addressing the challenges arising from Python’s dynamic typing and runtime polymorphism. Furthermore, we extend the **StatWhy** tool to support the verification of meta-analysis methods. These tools enable meta-analysts to identify overlooked assumptions and misuse of analyses, and to verify the correct use of hypothesis testing and meta-analysis methods in Python code.

Keywords: Formal method · automated verification · program verification · **Why3** · Python · statistical hypothesis testing · meta-analysis.

1 Introduction

The reproducibility crisis in scientific research has drawn increasing attention to the trustworthiness of statistical analyses, particularly meta-analyses that aggregate evidence from multiple studies. However, the correctness of statistical analyses is fundamentally different from that of conventional programs. Statistical methods crucially rely on assumptions about data-generating processes, such as distributional properties of unobservable true populations, which cannot be verified solely from the program and observed data. As a result, statistical programs may produce seemingly plausible outputs while being based on inappropriate or missing assumptions, leading to incorrect scientific conclusions.

This issue is exacerbated in practice by the widespread use of Python and its scientific libraries, which make it easy to apply statistical methods without explicitly stating their assumptions. However, existing program verification techniques focus on functional correctness and do not capture the assumption-dependent nature of statistical reasoning or the usage of external libraries.

[★] The artifact of the paper is available at <https://zenodo.org/records/21694782> and <https://github.com/fm4stats/why3-py>.

To address this gap, we propose a formal verification framework for Python statistical code, based on explicit specifications of assumptions and interpretations. Specifically, we present **Why3-py**, a Python front end for the **Why3** platform that transforms annotated Python code into **WhyML**, and an extension of **StatWhy** [11] with meta-analysis specifications. Importantly, our goal is not to verify implementations, but to ensure the correct use of statistical methods, e.g., detecting incorrect integration of p -values in meta-analysis. For this purpose, we redesigned **Why3**’s Python plugin to support real arithmetic and embed **WhyML** specifications in Python code by developing a parser for mixed Python–**WhyML** syntax and incorporating static type information using **mypy** [16].

Contributions. Our main contributions are summarized as follows:

- We present **Why3-py**, a formal verification tool for Python statistical code.
- We extend **StatWhy** with specifications for meta-analysis methods and show how our approach detects missing assumptions and misuse of meta-analyses.

These tools are available with a range of examples and documentation [12, 21]. To the best of our knowledge, this is the first approach that enables meta-analysts to formally verify Python statistical code. We consider this work the first step toward developing a framework to verify the integrity of scientific conclusions.

Related Work. *Logic for Statistics.* Modal logic has been employed to express statistical properties [8, 14]. The work on statistical epistemic logic [8–10] is the first attempt to define a modal logic to describe properties of statistical methods. *Belief Hoare logic (BHL)* [13, 15] is a program logic equipped with a modal operator for statistical belief. Based on BHL, the **StatWhy** tool [11] automatically verifies the correct use of statistical methods in OCaml programs.

Data Validation. A few works [22, 23] have proposed methods for checking whether data satisfy statistical assumptions. In contrast, **StatWhy** with **Why3-py** focuses on formally proving the correct use of statistical methods in programs.

Program Verification Tools. Deductive program verification aims to check a program’s correctness by proving that the program satisfies its formal specification. Various tools have been developed to statically verify the correctness of programs; e.g., **Why3** [7] for OCaml, **Dafny** [17] for imperative programs, **Frama-C** [5] for C programs, and **KeY** [1] for Java programs.

Verification of Python Programs. Formal verification of Python programs remains relatively underexplored compared to statically typed languages, such as C, Java, and OCaml. **Nagini** [6] provides automated verification for a statically typed subset of Python, based on the **Viper** intermediate verification infrastructure. The **PyVeritas** tool translates Python code into C to apply bounded model checking [20]. **Why3** provides a front end for **micro-Python**, a minimal subset of Python for education purposes that does not include real numbers or strings. Several tools translate Python into **Dafny** [4, 18, 19]. However, these approaches require restrictive subsets of Python and precise semantic modeling. In contrast, our approach supports practical Python statistical code by combining type reconstruction and abstraction of external libraries, enabling scalable verification.

2 Background

Statistical Hypothesis Testing. *Hypothesis testing* [2] determines whether observed data provide sufficient evidence to support a claim called an *alternative hypothesis*. In a hypothesis test, the alternative hypothesis usually claims that there is an effect or relationship between variables, whereas the corresponding *null hypothesis* represents a claim that there is no effect or no relationship. The purpose of hypothesis testing is to assess whether the available evidence is sufficient to reject the null hypothesis in favor of the alternative hypothesis. This assessment uses a *p-value*, which quantifies how unlikely the observed data would have been generated under the null hypothesis. The null hypothesis is rejected if the *p-value* is smaller than a threshold, e.g., 0.05.

The correctness of a statistical program crucially depends on assumptions about the data-generating process; e.g., many hypothesis testing methods assume that the underlying population follows a normal distribution. However, such assumptions are inherently unverifiable from the program and observed data, as they concern properties of an unknown true population which we can only partially and empirically learn from data. Consequently, the correctness of statistical programs cannot be derived only from formally provable assertions.

Belief Hoare Logic (BHL). To address this limitation, *belief Hoare logic* (BHL) [13, 15] is designed as a program logic equipped with epistemic modal operators for the *statistical beliefs* acquired by hypothesis testing. The *epistemic logic* used in BHL has two kinds of modal operators: $\mathbf{K}$ and $\mathbf{K}_{y,A}^{\leq \epsilon}$. Intuitively, $\mathbf{K}\varphi$ represents that we know φ . The *knowledge modality* $\mathbf{K}$ is defined as the box operator in the S5 modal logic system. Since the result of the hypothesis test may be wrong, we use the *statistical belief modality* $\mathbf{K}_{y,A}^{\leq 0.05}$ instead of the knowledge modality $\mathbf{K}$. $\mathbf{K}_{y,A}^{\leq \epsilon}\varphi$ represents that by a hypothesis test A on a dataset y , we believe φ with a *p-value* $\alpha \leq \epsilon$. The *satisfaction* of a formula φ in a possible world w is denoted by $w \models \varphi$ and is defined straightforwardly in a Kripke model.

In BHL, we describe a procedure for hypothesis testing as a program C . We then describe the requirements for the hypothesis tests as a *precondition* formula ψ_{pre} , e.g., representing that a dataset has been sampled from a normal distribution. We also specify the statistical belief acquired by the hypothesis test as a *postcondition* formula φ_{post} ; e.g., $\mathbf{K}_{y,A}^{\leq 0.05}\varphi$ denotes that by a hypothesis test A on a dataset y , we believe an alternative hypothesis φ with a *p-value* $\alpha \leq 0.05$.

Finally, we describe the whole inference as a *judgment* $\Gamma \vdash \{\psi_{\text{pre}}\} C \{\varphi_{\text{post}}\}$, representing that whenever the precondition ψ_{pre} is satisfied, the execution of the program C results in the satisfaction of the postcondition φ_{post} . By deriving this judgment using derivation rules in BHL, we conclude that the program C results in the statistical belief φ_{post} whenever the requirement ψ_{pre} is satisfied.

In this framework, specifications may include both verifiable properties and assumption-based statements. This allows for a clear separation between (i) properties that can be formally verified through deductive reasoning and (ii) assumptions that must be justified empirically or externally.

Deductive Program Verification for Statistical Code. Using BHL as a theoretical foundation, StatWhy provides a verification framework for statistical

```

def fail_meta_pvs_fisher(pv1 : real, pv2 : real, pv3 : real, observe_harm: formula) -> real :
  #@ requires \
  #@   let pvs = Cons pv1 (Cons pv2 (Cons pv3 Nil)) in \
  #@   pvalues pvs /\ intend_to_detect_excess_of_small_pv /\ \
  #@   forall w': world. (w' |= (Equiv indicate_risk (Impl disj_exps observe_harm))) \
  #@   ensures \
  #@   pvalue result /\ (World !st interp |= StatB (Eq result) indicate_risk)
  pvs = Cons(pv1, Cons(pv2, Cons(pv3, Nil)))
  return exec_combine_pvs_fisher(pvs, exps, disj_exps, observe_harm)

#@ execution
res = fail_meta_pvs_fisher(0.1, 0.2, 0.3, observe_harm)

```

Fig. 1: A Python program that uses Fisher’s method to compute a meta-analytic p -value `res` from three independent studies with p -values `pv1`, `pv2`, and `pv3`. `Cons` and `Nil` are used to construct a list of p -values corresponding to a WhyML list, because a Python list is translated into a WhyML array by `Why3-py`. The formula `disj_exps` is defined as the disjunction of the experimental situations (`sit exp1`), (`sit exp2`), and (`sit exp3`) outside the function.

code. Given an OCaml program annotated with a formal specification written in the Gospel language [3], `StatWhy` verifies whether a programmer has appropriately annotated the program with the specification, i.e., the requirements for hypothesis testing and the interpretation of the test results. `StatWhy`’s implementation is based on the `Why3` verification platform [7] to support compositional reasoning about programs with control flow and multiple statistical operations.

Importantly, this approach attempts not to prove the validity of statistical assumptions, but to ensure that such assumptions are explicitly annotated, hence preventing the misuse and misinterpretation of statistical methods. However, there remains a significant gap between the `Why3` platform and statistical programming practice, where Python and its rich libraries are widely used.

3 Motivating Example: Meta-Analysis in Python

Meta-Analysis of Harm Using Fisher’s Method. Typical users of the `Why3-py` tool are meta-analysts who have a basic knowledge of statistical methods but wish to avoid mistakes in their application and interpretation. To illustrate the main idea of `Why3-py`, consider a meta-analysis in which meta-analysts obtain multiple p -values, each corresponding to evidence from an independent study on the harm of a medicine, and then combine them into a single p -value that captures the accumulated evidence of harm using *Fisher’s method*. In this setting, the goal of formal verification is not to prove the correctness of the implementation of Fisher’s method, but rather to verify that the program uses Fisher’s method appropriately to detect the harm of the medicine.

Structure of the Code. Fig. 1 shows a Python program that calls the function `exec_combine_pvs_fisher`, which uses `SciPy` to compute a meta-analytic p -value `res` from three independent p -values `pv1`, `pv2`, and `pv3` using Fisher’s method. This code consists of two intertwined layers. The *executable layer* is a

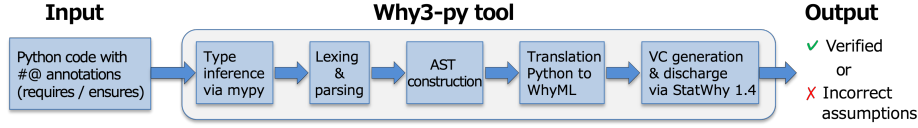


Fig. 2: Overview of the Why3-py verification workflow.

Python program that calls the function `exec_combine_pvs_fisher`. The *specification layer* is written in comments prefixed by `#@`. A programmer specifies the requirements and the interpretation of this function by annotating the code with the precondition in the `requires` clause and the postcondition in the `ensures` clause using Gospel. The Why3-py tool then parses these comments and generates VCs, which are discharged by Why3. Thus, the same code serves as both an executable Python program and a formally specified verification target. We remark that tool users need not develop formal specifications from scratch. Instead, they can adapt the preconditions and postconditions provided by StatWhy’s modules for individual statistical functions to their specific analyses.

Details of the Code. Although this example has a very simple program structure, its correctness depends on several assumptions, including non-trivial ones. In the precondition, the formula `pvalues pvs` checks that `pvs` is a list of p -values, i.e., real numbers in $[0, 1]$. `intend_to_detect_excess_of_small_pv` indicates that the meta-analyst intends to detect an excess of small p -values, which is assumed for Fisher’s method. The formula `indicate_risk` specifies the analyst’s decision rule, which stipulates that observing harm in one of the three experimental situations (denoted by `Impl disj_exps observe_harm`) leads to reporting a risk. Finally, the postcondition states that the output `result` is a p -value, and that by combining the three p -values using Fisher’s method, we obtain a statistical belief that the alternative hypothesis `indicate_risk` holds with the combined p -value `res` in the current world (`World !st interp`) equipped with a hypothesis testing record `!st` and an interpretation `interp` of private variables, where `!st` denotes the dereferenced value of the mutable reference `st`.

Program Verification and Error Detection. Given this Python code, the Why3-py tool verifies whether the meta-analysis method is appropriately used by translating it into WhyML, generating *verification conditions* (VCs) using Why3 and StatWhy 1.4, and discharging them using SMT solvers, e.g., CVC5. Then Why3-py fails to verify this code owing to missing requirements in the annotation. However, it helps identify the missing requirements by analyzing undischarged VCs. For instance, analysts can easily identify the missing requirement that the three p -values should not be collected under *publication bias* (i.e., the phenomenon that studies reporting statistically significant results are more likely to be published). Importantly, such missing requirements do not manifest as runtime failures and are therefore difficult to detect in practice without verifying the specification annotated in the code.

4 Extending Why3 to Statistical Programs in Python

Challenges in Verifying Python Statistical Programs. Why3’s micro-Python plugin has limited expressiveness and lacks support for real arithmetic and calls to external scientific libraries, which are crucial for statistical programming. To address these limitations, we extend Why3 to support the verification of Python programs that rely on statistical libraries, such as SciPy and statsmodels. Our approach is based on the following key ideas. First, we redesign the Python plugin to support a richer subset of Python and to integrate it seamlessly with Why3’s verification engine. Second, we incorporate static type information obtained from a type checker to bridge the gap between Python’s dynamic typing and WhyML, which is a statically typed language. Third, we model statistical functions and external library calls using logical specifications, enabling the verification of their correct usage. Finally, we extend the StatWhy modules with meta-analysis specifications to integrate p -values from multiple studies. Through these extensions, Why3-py enables analysts to verify the correct use of hypothesis testing and meta-analysis methods in Python programs.

A Python Front End for Why3. The Why3-py tool (Fig. 2) provides a new Python front end based on Why3 1.8.0 that targets a subset of Python programs that can be statically analyzed using mypy [16], a static type checker for Python. Unlike the original micro-Python plugin, which relied on a plugin-specific, minimally expressive specification language, our tool supports writing logical specifications in the full WhyML language within Python code. This is achieved by embedding WhyML expressions in comments of the Python code. In addition, our Python front end supports real number arithmetic, which is essential for statistical programs. It also translates Python-specific constructs, such as keyword arguments, into the appropriate WhyML representations. These design decisions are motivated by the need to support expressive constructs for statistical reasoning. As a result, the new front end is not backward-compatible, but significantly improves the expressiveness of the specification for Python code.

Handling Dynamic Typing. Python’s dynamic typing poses a significant challenge when translating programs into WhyML, which requires explicit, static type information. Our approach addresses this issue by using mypy for static type checking and inference as a preprocessing step. Although mypy can infer the types of local variables from assignments, Why3-py still requires explicit type annotations for statistical functions in external libraries. In our implementation, the verification pipeline first applies mypy to the input Python code and uses the resulting type information to parse and construct the abstract syntax tree (AST). This type information is essential for resolving ambiguities in Python syntax. For instance, Python uses the same operator for both integer and real arithmetic, whereas WhyML distinguishes between them. Leveraging such type information, Why3-py translates Python operators into the appropriate WhyML operators, ensuring the correct handling of numeric computations. By restricting the input language to a subset of Python analyzable by mypy, Why3-py enables sound and tractable verification while still supporting practical Python code.

Parsing a Mixed Python–WhyML Language. To support the embedding of WhyML specifications within Python code, we design a custom parser that handles a mixed Python–WhyML language, where lines prefixed with `#@` are not ignored as Python comments, but are tokenized by lexers and parsed as WhyML specifications. Since Python and WhyML have different lexical rules, parsing their combined syntax requires coordinating the two lexers according to the syntactic context maintained by the parser. For this reason, `Why3-py` relies on Menhir’s incremental parsing API, which allows tokens to be supplied dynamically from the two lexers. Specifically, the system maintains both a Python lexer and a WhyML lexer, while using a single parser to process tokens from both lexers and managing input through a shared lexing buffer that stores the entire program in memory. This enables the parser to backtrack and switch between the two lexers: the parser first invokes the Python lexer to interpret the input as Python. If parsing fails, it resets the input position and invokes the WhyML lexer to reinterpret the same fragment as WhyML. This design allows seamless switching between the two languages within a single source file.

Translation to WhyML. After parsing, the constructed Python AST is translated into a WhyML AST. In this translation, Python function calls are mapped to corresponding WhyML function calls, and keyword arguments are translated into record constructions. Python lists are translated into WhyML arrays, whereas lists constructed using `Cons` and `Nil` are translated into WhyML lists. Logical expressions embedded in Python are represented as WhyML terms.

Verification with External Libraries. External libraries, such as SciPy, cannot be directly verified using Why3. To address this limitation, we adopt a specification-based abstraction approach in `Why3-py`. Instead of modeling the implementations of external functions, we associate them with logical contracts that formalize their behavior in our StatWhy 1.4 modules. Verification is then performed under the assumption that these contracts are satisfied. This approach allows us to reason about the correct use of library functions, rather than their internal implementation details. As a result, `Why3-py` supports modular verification of Python statistical code that depends on external scientific libraries.

Scope of Supported Python Code. `Why3-py` targets a subset of Python that can be statically analyzed using `mypy`. This includes numerical computations, conditional branching, loops, and calls to external libraries equipped with logical specifications. However, programs relying heavily on dynamic features such as dynamically constructed data structures are currently outside the scope.

5 Case Studies and Performance Evaluation

Detecting Incorrect Requirements for Meta-Analyses. Our StatWhy 1.4 supports the WhyML specification of various meta-analysis methods, including *Fisher’s method*, *Stouffer’s method*, and *Mantel-Haenszel’s method*. Relying on them, `Why3-py` can verify Python meta-analysis code including a specification (around 20 to 40 AST nodes) and detect various incorrect requirements. One of the important requirements is the absence of *publication bias*, as discussed in

Table 1: Verification times (in seconds) for popular multiple comparison methods covered in standard textbooks. #groups (resp. #comparisons) denotes the practical number of groups (resp. combinations of groups) compared in the tests.

Test methods	Metric	#groups					
		2	3	4	5	6	7
Tukey’s HSD test	Time (sec)	10.90	20.73	31.95	47.65	65.97	87.86
	#comparisons	1	3	6	10	15	21
Dunnett’s test	Time (sec)	10.03	18.71	22.51	26.71	32.90	35.39
	#comparisons	1	2	3	4	5	6
Steel-Dwass’ test	Time (sec)	10.61	20.30	31.32	46.05	65.60	93.72
	#comparisons	1	3	6	10	15	21

Sect. 3. If meta-analysts do not annotate the code with the requirement `sampled uniform_pv`, then `Why3-py` detects that this requirement is missing, indicating that meta-analysts cannot reliably apply Fisher’s method under publication bias.

Detecting *P*-hacking in Multiple Comparisons. `Why3-py` can also be applied to Python hypothesis testing code, including multiple comparisons within a single study. Such settings may involve *p-hacking*, where multiple hypotheses are tested simultaneously and *p*-values are selectively reported or manipulated, which can lead to incorrect scientific conclusions. By annotating Python code with its specification (around 30 AST nodes) and verifying it using `Why3-py`, analysts can detect incorrect requirements and interpretations of *p*-values.

Scalability of `Why3-py`. We evaluated the verification performance of Python programs of approximately 40 lines of code each using `Why3-py` and `StatWhy` 1.4, varying (i) the number of *p*-values integrated in meta-analyses and (ii) the number of groups compared in multiple comparisons. Our experimental evaluation uses the SMT solver `cvc5` 1.1.2 on a ThinkPad 480s with Debian 13, an Intel Core i7-8650U, and 24 GB of memory. The verification times for Python programs computing *p*-values using each meta-analysis method are within 8 seconds for practical numbers (e.g., 2 to 30) of studies. The verification times for the Python code for multiple comparisons are shown in Table 1. Although the number of comparisons between groups grows rapidly with the number of groups, the verification of these programs is efficient for practical numbers of groups.

6 Conclusion

We proposed a formal verification framework for Python statistical code. Specifically, we developed the verification tool `Why3-py` for Python statistical code and extended `StatWhy` to specify meta-analyses. To the best of our knowledge, this is the first approach to formally verifying Python statistical code.

Acknowledgments. The authors were supported by JSPS KAKENHI Grant Number JP24K02924, Japan. Yusuke Kawamoto was supported by JST FOREST Grant Number JPMJFR242M, Japan.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M.: *Deductive Software Verification - The KeY Book - From Theory to Practice* (2016)
2. Arbuthnot, J.: An argument for divine providence, taken from the constant regularity observ'd in the births of both sexes. *Philosophical Transactions of the Royal Society of London* **27**(328), 186–190 (1710). <https://doi.org/10.1098/rstl.1710.0011>
3. Charguéraud, A., Filliâtre, J., Lourenço, C., Pereira, M.: GOSPEL - providing OCaml with a formal specification language. In: *FM'19*. pp. 484–501. Springer (2019)
4. Choudhury, A.: dafny-of-python: A tool for deductive verification of Python programs. GitHub repository (2020), <https://github.com/arsalan0c/dafny-of-python>
5. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Framas-C - A software analysis perspective. In: *SEFM'12*. pp. 233–247 (2012)
6. Eilers, M., Müller, P.: Nagini: A static verifier for Python. In: *Proc. CAV 2018, Part I*. LNCS, vol. 10981, pp. 596–603. Springer (2018)
7. Filliâtre, J., Paskevich, A.: Why3 - where programs meet provers. In: *ESOP'13*. LNCS, vol. 7792, pp. 125–128. Springer (2013)
8. Kawamoto, Y.: Statistical epistemic logic. In: *The Art of Modelling Computational Systems*. LNCS, vol. 11760, pp. 344–362. Springer (2019)
9. Kawamoto, Y.: Towards logical specification of statistical machine learning. In: *Proc. SEFM 2019*. pp. 293–311 (2019)
10. Kawamoto, Y.: An epistemic approach to the formal specification of statistical machine learning. *Software and Systems Modeling* **20**(2), 293–310 (2020)
11. Kawamoto, Y., Kobayashi, K., Suenaga, K.: StatWhy: Formal verification tool for statistical hypothesis testing programs. In: *Proc. CAV 2025, Part II*. LNCS, vol. 15932, pp. 216–230. Springer (2025)
12. Kawamoto, Y., Kobayashi, K., Suenaga, K.: User Documentation for StatWhy v1.4 (2026), available at <https://github.com/fm4stats/statwhy>
13. Kawamoto, Y., Sato, T., Suenaga, K.: Formalizing statistical beliefs in hypothesis testing using program logic. In: *Proc. KR'21*. pp. 411–421 (2021)
14. Kawamoto, Y., Sato, T., Suenaga, K.: Formalizing statistical causality via modal logic. In: *Proc. JELIA'23*. LNCS, vol. 14281, pp. 681–696. Springer (2023)
15. Kawamoto, Y., Sato, T., Suenaga, K.: Sound and relatively complete belief Hoare logic for statistical hypothesis testing programs. *Artif. Intell.* **326**, 104045 (2024)
16. Lehtosalo, J., van Rossum, G., Levkivskyi, I., Sullivan, M.J.: mypy: Static typing for Python (2026), <https://github.com/python/mypy>
17. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: *LPAR'10*. LNCS, vol. 6355, pp. 348–370. Springer (2010)
18. Leino, K.R.M.: Developing verified programs with Dafny. In: *ICSE '13*. pp. 1488–1490. IEEE Computer Society (2013)
19. Li, Y.C., Zetsche, S., Somayyajula, S.: Dafny as verification-aware intermediate language for code generation. *arXiv preprint arXiv:2501.06283* (2025)
20. Orvalho, P., Kwiatkowska, M.: PyVeritas: On verifying Python via LLM-based transpilation and bounded model checking for C. *CoRR* **abs/2508.08171** (2025)
21. Tanaka, A., Kawamoto, Y.: User Documentation for Why3-py v1.0 (2026), available at <https://github.com/fm4stats/why3-py>
22. Turcotte, A., Mehta, N.N.: The fault in our stats. In: *Proc. ASE 2025*. pp. 2491–2503. IEEE (2025). <https://doi.org/10.1109/ASE63991.2025.00205>
23. Turcotte, A., Wu, Z.: Expressing and checking statistical assumptions. *Proc. ACM Softw. Eng.* **2**(FSE), 2735–2758 (2025). <https://doi.org/10.1145/3729391>