
Synthesizing Test Cases of Python Programs Via LLM and Soufflé¹

Zheng Wangyuan
Independent Researcher

With
Apart Research

Abstract

Spec-Driven Development easily becomes leaky abstractions that cannot keep pace with real-world fast-prototyping. Test-Driven Development is lacking in formal guarantees and its epitome, automated testing, is difficult to implement. We present a combined framework to turn a Python Program into both unit tests and property-based tests with the help of LLM, logic programming Soufflé, static analysis, and various testing techniques. We evaluated our testing framework on three case studies: CutePetsBoston, HuggingFace’s Transformers, and Dacite. We found our framework useful for automatically deriving test cases directly from programs with several limitations. We conclude that the future of formal verification should be a combination of formal and informal methods and the success of it in everyday, real world applications depends upon finding the best use cases for both.

¹ Research conducted at the [The Secure Program Synthesis Hackathon](#), May 2026

1. Introduction

This project began with several practical problems that we encountered in everyday programming. We have worked with workflows that are almost entirely Spec-Driven Development or Test-Driven Development. In the former case, it is both hard to specify the exact security policies for everyday programs and scale the specifications. TLA is one example where it shines when there is a specific security policy to enforce but becomes inadequate when everyday applications require multiple if not hundreds of small, heterogeneous checks here and there. And this is not just indicative of TLA's flaw alone but Spec-Driven Development in general: in real world programming developers work fast and iteratively to gradually perfect their software instead of coming up with specifications. Test-Driven Development is what most developers do in their daily job, but rarely do developers write good or even complete tests. Testing is treated as a hassle to be done once and for all. The end result of this attitude is that most tests are trivial and do not really surface problems in design or implementation. Property-based testing is a good balance between the formal specification and informal testing, but to get property-based tests we return to the old problems of coming up with good properties again. Our project envisions a kind of automated testing where initial properties about the program are extracted, composed, and finally turned into both unit and property-based tests for validation. We strive to provide formal reasoning via logic programming and static analysis to testing in an automated manner so that developers can focus on the design and implementation of their software and have a convenient way to reflect on what they have produced so far without the hassle of manually writing tests or properties/specifications for more formal guarantees.

Our main contributions are:

1. We designed and implemented a complete Python-to-Datalog-to-Tests pipeline.
2. We devised a scheme to divide the properties we acquired into three categories with the most restrictive ones as candidates for unit tests, the less restrictive ones for Property-based tests, and the most open ones as part of manual review.
3. We created an evaluation strategy including relation quality, coverage, and mutation analysis and evaluated our testing framework with three case studies of varying degrees of complexity: Dacite, CutePetsBoston, and HuggingFace's Transformers.

2. Related Work

None of the methods or tools in this testing framework is new to the formal methods or software engineering community. Soufflé has been used extensively for static analysis (Herbert Jordan, Bernhard Scholz, Pavle Subotic. *Soufflé: On Synthesis of Program Analyzers*. CAV 2016 and Bravenboer, Smaragdakis. *Strictly Declarative Specification of Sophisticated Points-to Analyses*. OOPSLA 2009). Property-based Testing is also a mature field on its own with Haskell's QuickCheck as a prominent example (Koen Claessen, John Hughes. *QuickCheck: A*

Lightweight Tool for Random Testing of Haskell Programs. ICFP 2000). There have been countless publications on LLM-assisted testing, various static analysis techniques, spec extraction, and knowledge graphs. We won't list any to keep this part short.

3. Methods

We first extract facts about our program based on Python AST. The facts are mostly common relations we can find in a python file. On a higher level, we have modules, file paths, imports, classes, inheritance, functions, and methods. We also focused on extracting facts about dataclasses because DataClasses are natural candidates for data modeling for a Python program because they make the relations between data explicit. Other common programming language features are also included: types, signatures, effects (e.g. read and write), and control flows. Some minor semantic properties are also extracted at this mostly syntactic stage: string/boolean/numeric literals, the length function, slicing functions, and so on.

The next stage in our pipeline feeds the facts we extracted to a series of prepared rules written in Soufflé to derive composed relations. For example, if the dataclass AdoptionPost points to dataclass Pet and Pet points to dataclass Shelter, then, based on our transitive dependency rule, AdoptionPost also points to Shelter. Many kinds of rules are included: structural properties (required field), dataflow (input-to-output), dependency graphs (post-to-shelter), protocol obligations (no send after close), side-effects (env-read), abstract states (non null values), possible bugs (slicing boundaries), and so on.

After we derive the new relations, we turn the new relations combined with some old facts about our program into tests. We devise a three-tier system for creating tests. The properties that have a clear output without much variation become unit tests. The properties with a range of possible output become property-based tests (with the help of Hypothesis, a Python Property-based testing framework). The properties that are unclear and risky are directly written to the report for developers to review.

We also have an evaluation plan that first runs our tool on the targeted program and counts the ratio between derived relations and executable tests. We also measure what category of analysis the derived relations come from (e.g. whether it comes from a dataclass schema relation or a boundary relation or an effect relation). These statistics help us and developers see whether our tool is functioning at all. Then we do a tally of how many tests passed/failed/skipped/xfailed/xpassed with the xpassed representing expected-failure passing and xfailed expected-failure failing. We also measure coverage to see how much code our newly generated tests cover. Small-scale mutation testing is employed to judge whether our tests are actually effective in killing the mutant. Put simply, if we introduce a small bug (change smaller or

equal to smaller) that our analysis believes is important, do the generated tests catch the introduced bugs?

We conducted the entire pipeline including the evaluation on three case studies. CutePetsBoston is a personal open source project of mine that uses dataclass extensively so it is a natural candidate for developing and testing our entire framework. Dacite is used as a smaller but dataclass-heavy case study to see if it works for other similar projects. A bounded transformer program slice from HuggingFace is used to stress test our tool. When we first tested our tool with the Transformer case study, we realized that we over-fitted our tool for the CutePetsBoston project and therefore was not general enough. We then further generalized our tool for all Python projects that use dataclass extensively instead of just projects in the same shape of CutePetsBoston.

4. Results

The results from our three case studies show clear evidence that our tools can generate tests. Some tests are skipped due to missing/mistaken dependencies which can be fixed in later versions. We also have in our Github repo the lists of manual review candidates that our tool considers too risky or ambiguous to generate tests for (See generated_tests folder’s README.md). Though this initial result looks promising, it also exposes more problems that we plan to address in future versions. The lack of errors, failures, and other categories implies that our tool is deliberately generating safer tests instead of generating riskier tests that may surface more problems in program design or implementation. The case studies also show limited scope where we only use a small subset of Transformers codebase for experiment because it was taking too much time to just parsing the AST and doing static analysis. We also did not have a comparison with other LLM-assisted test generation framework due to time constraint.

Figure 1. Generated test validation results across targets

Target	Passed	Skipped	Errors	Failed	XPassed	XFailed
CutePetsBoston	73	0	0	0	0	0
Dacite	3	5	0	0	0	0
Transformers (Slice)	99	7	0	0	0	0

Validation results for the generated pytest suits on three targets.

5. Discussion and Limitations

This project aims to examine a different direction we can take in making our AI-generated code safer and more robust. To solve real world problems, we should create pipelines where we utilize the advantages of different methods and also have them complement each other. The difficulty and excitement then are about how to correctly combine different methods and when to switch from one method to another.

Limitations

The foremost limitation is the scope of our analysis tool because currently it can only handle Python projects with extensive use of DataClasses. Another big limitation of this project is that our current tool is still largely immature with rudimentary static analysis and mostly tests about the structure of the program. Another limitation is about things this project failed to demonstrate. Originally, I was envisioning a self-evolving testing framework where formal guarantees turn to informal tests and informal tests turn to formal guarantees with fresh changes in each iteration. This was too ambitious because as it turned out, it is already immensely difficult to extract properties from a program and turn them into good tests. The limitation of our case studies is already discussed above in the Result section.

Future Work

Our testing framework would benefit a lot from simply better static analysis and processing of the riskier review candidates which I believe can lead to some real insights about our program. We would also like to conduct more experiments on more Python projects for more real world feedback. The interoperation between LLM and our symbolic analysis is also weak and we would like more interaction between them. The current LLM-assisted rule-layer failed to have an impact on the project but we plan to better incorporate it into our pipeline because we do believe in LLM's ability to extract more semantic facts/relations about the test target.

6. Conclusion

SPS-VeriSpec, a complete pipeline from Python program to Datalog to concrete tests, shows potential for better automated testing by combining methods from static analysis, logic programming, and software engineering. We showed an interesting use case of Soufflé, a Datalog variant, beyond program analysis and towards test generation. In short, this project is a promising test-specification workflow to make program behaviors explicit, checkable, and reviewable.

Code and Data

- Code repository: <https://github.com/patrickZWY/SPS-VeriSpec>

- **Data/Datasets:** <https://github.com/codeforboston/CutePetsBoston>, <https://github.com/konradhalas/dacite>, <https://github.com/huggingface/transformers>

References

Use a consistent citation format. Include: Author(s), Year, Title, Venue/Publisher, and URL or DOI where available.

1. Soufflé Project. n.d. *Examples*. Soufflé website. <https://souffle-lang.github.io/examples>
2. plast-lab. n.d. *Doop*. GitHub repository. <https://github.com/plast-lab/doop>
3. Reps, Thomas; Horwitz, Susan; Sagiv, Mooly. 1995. *Precise Interprocedural Dataflow Analysis via Graph Reachability*. Proceedings of POPL 1995. <https://research.cs.wisc.edu/wpis/papers/popl95.pdf>
4. Horwitz, Susan; Reps, Thomas; Binkley, David. 1990. *Interprocedural Slicing Using Dependence Graphs*. ACM Transactions on Programming Languages and Systems (TOPLAS), 12(1). <https://research.cs.wisc.edu/wpis/papers/toplas90.pdf>
5. Claessen, Koen; Hughes, John. 2000. *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs*. Proceedings of ICFP 2000. <https://research.chalmers.se/en/publication/237427>
6. Hypothesis Developers. n.d. *Hypothesis Documentation*. Project documentation. <https://hypothesis.readthedocs.io/>
7. Keles, Alperen. n.d. *Property-Based Testing Materials*. Personal website / slides. <https://alperenkeles.com/>
8. Teleport. 2021. *Using Datalog to Test for Access*. Teleport blog. <https://goteleport.com/blog/testing-access-datalog/>
9. Alberi, Fabien. 2022. *An Introduction to Datalog*. Michelin blog. <https://blogit.michelin.io/an-introduction-to-datalog/>
10. Wojtowicz, Norbert. n.d. *Domain Modeling With Datalog*. Conference talk, RubyEvents. <https://www.rubyevents.org/talks/domain-modeling-with-datalog>
11. Lamport, Leslie. n.d. *TLA+ Home Page*. Official project website. <https://lamport.org/tla/tla.html>
12. Jordan, Herbert; Scholz, Bernhard; Subotic, Pavle. 2016. *Soufflé: On Synthesis of Program Analyzers*. Proceedings of CAV 2016. <https://souffle-lang.github.io/cav-paper>
13. Bravenboer, Martin; Smaragdakis, Yannis. 2009. *Strictly Declarative Specification of Sophisticated Points-to Analyses*. Proceedings of OOPSLA 2009. <https://dl.acm.org/doi/10.1145/1640089.1640106>
14. Cousot, Patrick; Cousot, Radhia. 1977. *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of*

Fixpoints. Proceedings of POPL 1977.

<https://cs.nyu.edu/~pcousot/COUSOTpapers/POPL77.shtml>

LLM Usage Statement

We used both Claude and Codex CLI for coding the implementation and brainstorming approaches in constructing the pipeline. All results and claims were independently verified.