

Property-Based Testing of Separation Logic Specifications

Zain K Aamer[✉]

University of Pennsylvania
Philadelphia, USA
zaamer@seas.upenn.edu

Abstract

Separation logic (SL) has been a tried and tested method for specifying the behavior of heap-manipulating programs. We apply property-based testing (PBT) to these specifications for C programs, generating inputs that satisfy preconditions and checking that postconditions hold. Besides providing a specification language for testing, using SL means that PBT can act as a stepping stone towards full verification. We have previously developed tools that explore three generation techniques for SL specifications in the fragment supported by the CN verifier. Ongoing and future work develops AustenTest, a standalone tool that uses a separation-logic IR covering a larger SL fragment. AustenTest will be a publicly available, supported tool for other separation-logic verifiers to be able to use. Simultaneously, we are working on applying grammar-based fuzzing as a fourth approach to generation.

CCS Concepts: • Software and its engineering → Software testing and debugging; • Theory of computation → Program specifications; Separation logic.

Keywords: Random Testing, Property-Based Testing, Separation Logic, C

ACM Reference Format:

Zain K Aamer. 2026. Property-Based Testing of Separation Logic Specifications. In *Companion Proceedings of the 2026 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '26)*, October 4–9, 2026, Oakland, CA, USA. ACM, New York, NY, USA, 3 pages. <https://doi.org/10.1145/3837729.3842144>

1 Motivation

Systems software is what everything runs upon, and so should be correct. Many techniques have been applied to gaining assurances about the correctness of software in low-level languages like C, with one approach being formal verification. There have been many tools built for formal verification of C programs, but on a per-function basis, formal

verification is typically all-or-nothing, meaning either the proof succeeds or fails. Fulminate [4] takes specifications for CN [9] and checks that they hold at runtime, providing a potential test oracle. As such, we will largely focus on the CN fragment of separation logic.

A common lightweight method of gaining assurance is testing. Unit tests are cheap to write, but only check specific fixed inputs. Fuzz testing explores many inputs, but requires a harness which serves as the entrypoint. While there has been work on library fuzzing, these tools run the risk of false positives. We would instead like to apply property-based testing (PBT) [5], which checks executable properties across many randomly generated inputs, on a per-function basis.

While PBT has been applied in low-level languages like C before, the user is required to write random input generators by hand due to the lack of information provided by the weak C type system. We would like to take CN preconditions for a given function (which can constrain heap shapes and values), synthesize a random generator of precondition-satisfying inputs, and see if Fulminate detects an error during its execution. If it does, we've discovered a mismatch between specification and implementation, with the user deciding which is wrong. Due to the testing being performed at a per-function level, the failures are more local. Since this same specification can also be used for verification with CN, property-based testing provides a direct stepping stone toward greater assurance before continuing on to full proof.

2 Problem

Core synthesis problem. Let $f(\bar{x})$ have separation-logic contract $\{\text{Pre}\} f(\bar{x}) \{\text{Post}\}$. The core synthesis problem is to generate concrete arguments $\bar{v}$ and an initial heap H such that for a store σ binding $\bar{x}$ to $\bar{v}$,

$$(\sigma, H) \models \text{Pre}.$$

We must generate the arguments, allocate and place values in the heap, and then invoke f . The difficulty is not to find one model of Pre, but to generate many diverse satisfying inputs.

Costs of input generation. Multiple factors interact when determining an effective generation technique. A campaign pays an initial synthesis cost T_{synth} to translate, analyze, and compile a specification. Each test then incurs recurring costs



This work is licensed under a Creative Commons Attribution 4.0 International License.

SPLASH Companion '26, Oakland, CA, USA

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2910-2/2026/10

<https://doi.org/10.1145/3837729.3842144>

T_{sample} , T_{FUT} , and T_{oracle} for input generation, function execution, and oracle evaluation, respectively. T_{synth} determines how quickly testing can begin, while the recurring costs determine how many useful executions fit into the available time. Input quality matters too: a generator is useful when it reaches diverse behavior, not when it generates many similar inputs. While developers are changing specs, they may run testing for less than a minute, like traditional PBT [6]. A more expensive generator synthesis technique must make up its cost through better or faster inputs.

Completed foundation. The completed work establishes three points in the design space. Bennet compiles CN specs to randomized backtracking C generators and uses blame-directed backtracking to regenerate failed choices [2]. Lucas uses runtime abstract interpretation to narrow choices before generation and refine regeneration after a failure [3]. Darcy chooses a random path through the precondition and solves the constraints on that path using MaxSMT for diversity [1]. Bennet and Lucas provide fast tests when constraints can be resolved locally, while Darcy can generate inputs for complex interconnected preconditions at a much higher per-input cost. The largest contrast is seen for Bennet’s ring queue case study, whose push function causes Darcy to stall on a recursive constraint, taking roughly three to three and a half hours to generate 100 inputs, versus about three seconds for Bennet and Lucas. At the other extreme, Darcy was applied to the production pKVM hyp allocator, whose complex preconditions Bennet and Lucas were unable to test. These cases show that no technique is universally best. All three nevertheless remain coupled to CN’s representation and generate heaps and arguments directly.

Specification formats. VeriFast [7] and VCDryad [8] use different specification languages than CN’s. A testing tool tied to one separation logic syntax cannot directly be applied to the exact same specs for a different verifier. We’d like a common separation-logic IR that verifiers can export, so that we can use a single testing backend.

The representation of generated inputs. The current systems make choices that directly build a heap and arguments. An alternative is to synthesize a small program that constructs the input. Grammar-based generators such as ISLa and FANDANGO combine structured generation with semantic constraints [10, 13], but here our constraints depend on the heap built by the generated program. We hope to derive grammar constraints from a separation-logic precondition and to restrict rules which would construct invalid heaps.

More empirical evidence. The Bennet, Lucas, and Darcy case studies let us measure AustenTest’s coverage and bug finding, but are rather limited. The VCDryad benchmarks would provide many more functions with specifications. Possibly selected SL-VC tasks could be adapted as well. Historical

Defects4C bugs would offer a real-world dataset, but a newly written specification can be wrong. The evaluation must therefore separate a genuine implementation failure from a specification-induced false positive.

Two questions guide the proposed work. How well does AustenTest find bugs and cover target code? When do the existing tools and grammar-based fuzzing each work best?

3 Approach

A shared separation-logic IR. The ongoing AustenTest implementation exposes a common separation-logic IR for testing. CN has been modified to export its specifications to this IR, and we are developing a VCDryad frontend that does the same. AustenTest can then consume the same representation to generate inputs for the target function. The exporter boundary keeps source-language differences out of the testing engines.

Synthesizing input-building programs. Instead of generating a heap and argument tuple directly, the new approach generates a program that allocates memory, assigns variables, sets fields, and calls functions. The separation-logic precondition supplies semantic constraints over the program grammar, so a derivation describes both how to build an input and the relationships that input must satisfy. We plan to restrict the grammar to prevent definitely incorrect programs, such as allocating too little memory for a node in a linked list. This approach also means that any failure-causing programs can be presented to the user as a readable output (as opposed to a dump of the heap).

Roadmap. The milestones are to stabilize the AustenTest IR, complete the VCDryad frontend, implement the precondition-constrained construction grammar, add always-invalid-program pruning, and compare all four generation techniques. If possible, we will also include tasks from SL-VC by converting their specifications to CN or VCDryad. We will record specification constructs that the IR cannot yet represent. If pruning provides no end-to-end benefit, the constrained grammar is still an interesting baseline, and the negative result will be informative.

4 Evaluation Methodology

Existing Bennet, Lucas, and Darcy case studies span a range of heap-manipulating workloads, including recursive and position-independent data structures, allocator implementations, and critical code examples. By implementing the VCDryad frontend and applying AustenTest to the published benchmark of 152 annotated C routines, we will greatly grow our evaluation. If feasible, we will also include tasks from SL-VC by converting their specifications to either CN or VCDryad [11]. Defects4C will provide a separate corpus of

real-world bugs [12], for which we will write new specifications. These large-scale experiments will address two questions.

RQ1: How well does AustenTest find bugs and cover code? On the Bennet, Lucas, and Darcy case studies, repeated fixed-budget trials will report code coverage, valid target executions, bugs found, and time to each failure, reusing the existing faults and mutants. The VCDryad (and maybe SL-VC) tasks will additionally show how many specifications fall within our supported fragment, which constructs are unsupported, and how input generation performs on the supported preconditions.

For eligible Defects4C functions, we will write separation-logic specifications and test the buggy versions. Each failure-producing input will be replayed on the corresponding fixed version. We will count a defect as found only when the failure occurs on the buggy revision and not on the fixed revision. We will assume that a failure that occurs after the fix is a specification-induced false positive.

RQ2: When does each generation technique work best? Across shared case studies, we will compare Bennet-style randomized backtracking, Lucas-style runtime abstract interpretation, Darcy-style constraint solving, and grammar-based fuzzing of input-construction programs. All configurations will receive the same wall-clock budget, cores, bounds, and random seeds. Measurements will distinguish T_{synth} , T_{opt} , T_{sample} , T_{FUT} , and T_{oracle} , and will report rejected constructions, valid executions, structural and value diversity, code coverage, bugs found, and time to each failure. Results will be grouped by relevant specification features, such as recursive functions, arrays, and pointer arithmetic. Due to fuzzing's speed, grammar-based fuzzing may not incur much overhead relative to Bennet and Lucas, unlike Darcy. Simultaneously, due to tools like FANDANGO's success at generating complicated constrained inputs, it may also be able to test the pKVM hyp allocator, which currently only Darcy can. Success would yield a comprehensive evaluation of the workloads, determining when each technique is effective.

Threats include the diversity of the benchmarks developed by others and errors in newly written specifications for Defects4C affecting the input distribution generated. We will publish the written Defects4C and converted SL-VC specifications.

Acknowledgments

This work is supported by the NSF under grant *SHF: Medium: Usable Property-Based Testing*, NSF #2402449.

References

- [1] Zain K Aamer, Rini Banerjee, Hiroyuki Katsura, David Kaloper-Meršinjak, Dimitrios J. Economou, Kayvan Memarian, Dhruv Makwana, Neel Krishnaswami, Benjamin C. Pierce, Christopher Pulte, and

- Peter Sewell. 2026. Code-Specify-Test-Debug-Prove: Flexibly Integrating Separation Logic Specification into Conventional Workflows. *Proc. ACM Program. Lang.* 10, PLDI, Article 200 (June 2026), 27 pages. doi:10.1145/3808278
- [2] Zain K Aamer and Benjamin C. Pierce. 2025. Bennet: Randomized Specification Testing for Heap-Manipulating Programs. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 413 (Oct. 2025), 30 pages. doi:10.1145/3764115
- [3] Zain K Aamer and Benjamin C. Pierce. 2026. Random Testing via Runtime Abstract Interpretation. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 381 (Oct. 2026), 29 pages. doi:10.1145/3839513
- [4] Rini Banerjee, Kayvan Memarian, Dhruv Makwana, Christopher Pulte, Neel Krishnaswami, and Peter Sewell. 2025. Fulminate: Testing CN Separation-Logic Specifications in C. *Proc. ACM Program. Lang.* 9, POPL, Article 43 (Jan. 2025), 33 pages. doi:10.1145/3704879
- [5] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 268–279. doi:10.1145/351240.351266
- [6] Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. 2024. Property-Based Testing in Practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/3597503.3639581
- [7] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods*. Springer, Berlin, Heidelberg, 41–55. doi:10.1007/978-3-642-20398-5_4
- [8] Edgar Pek, Xiaokang Qiu, and P. Madhusudan. 2014. Natural Proofs for Data Structure Manipulation in C Using Separation Logic. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 440–451. doi:10.1145/2594291.2594325
- [9] Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. 2023. CN: Verifying Systems C Code with Separation-Logic Refinement Types. *Proc. ACM Program. Lang.* 7, POPL, Article 1 (Jan. 2023), 32 pages. doi:10.1145/3571194
- [10] Dominic Steinhöfel and Andreas Zeller. 2022. Input Invariants. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 583–594. doi:10.1145/3540250.3549139
- [11] Hanyang Wang, Xiwei Wu, and Qinxia Cao. 2026. SL-VC: A Benchmark and Automated Framework for Separation Logic Verification Condition Proving. In *Forty-Third International Conference on Machine Learning*.
- [12] Jian Wang, Xiaofei Xie, Qiang Hu, Shangqing Liu, Jiongchi Yu, Jiaolong Kong, and Yi Li. 2025. Defects4C: Benchmarking Large Language Model Repair Capability with C/C++ Bugs. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 254–265. doi:10.1109/ASE63991.2025.00029
- [13] José Antonio Zamudio Amaya, Marius Smytzek, and Andreas Zeller. 2025. FANDANGO: Evolving Language-Based Testing. *Proc. ACM Softw. Eng.* 2, ISSTA (June 2025), ISSTA040:894–ISSTA040:916. doi:10.1145/3728915

Received 2026-07-04; accepted 2026-08-04