ABSTRACT

Title of Thesis: Model-free Stateful Random Testing
Ethan Khan Chou
Master of Science, 2025

Thesis Directed by: Professor Leonidas Lampropoulos
Department of Computer Science

Property-based testing (PBT) has enabled greater confidence in the correctness of programs and facilitated formal verification by exposing false theorems earlier in the proving process. However, most PBT libraries are designed for pure programs, even though much critical software such as systems software are stateful. Current stateful PBT libraries generally require the programmer to implement a model of the program under test, increasing developer overhead and introducing another source of error.

In this thesis, I aim to significantly reduce developer overhead when testing stateful programs. I introduce `seq-test`, a tool for testing stateful C programs that eliminates the need for models. `seq-test` is embedded in CN, a specification language for formally verifying C programs. Test generation uses a generation-by-execution strategy, where CN preconditions are used to discard illegal call sequences, and CN postconditions are used to detect bugs. We also implement two-phase shrinker to increase the usability of generated falsifying examples in debugging.

Finally, I evaluate the bug-finding ability of `seq-test` against AFL++, a popular fuzzer, and Bennet, a PBT tool embedded in CN, on a variety of case studies.

Model-free Stateful Random Testing

by

Ethan Khan Chou

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2025

Advisory Committee:
Professor Leonidas Lampropoulos, Chair/Advisor
Professor Milijana Surbatovich
Professor David Van Horn

Dedication

To Rance and Katya.

Acknowledgments

I would like to first thank my advisor, Leonidas (or Leo as he is more affectionately known in our lab) Lampropoulos, for taking me as his student on such short notice after the unexpected loss of Rance, and for his support and help over the past nearly two years.

I also want to express my gratitude to Milijana Surbatovich for her help in devising evaluation strategies and for being on my committee. To David Van Horn, for joining my committee on short notice and making my defense possible. To Zain Aamer for providing much of the evaluation tooling. To Rini Banerjee for making our requested changes to Fulminate that sped up our tool. To José Manuel Calderón Trilla, for inspiring my love for programming languages, for showing me the kind of educator I aspire to be, and for many handmade lattes from his espresso machine. To Alperen Keleş, for sitting in on my many meetings with Leo brainstorming next steps and new features, and for being a wonderful research partner on the D4 paper. To Justine Frank, for sharing random tidbits of programming language knowledge and going on an insane 72-hour train trip across America with me and another 80-hour train trip soon. To Pierce Darragh, for late night conversations (please fix your sleep schedule), Utah information, and post-PLUM Jam coffee runs. To Cliff Bakalian, for taking me as a CMSC330 TA and allowing me to teach discussions, lectures, and exam reviews. To the CMSC330 TAs for being the best coworkers one could hope to have, many of whom I am now honored to call friends. To all of PLUM, for their feedback on my talks and for sharing their knowledge with me at Jam every week. To the Gamer Symphony Orchestra (especially Minsi and Isaac) for providing a community to de-stress with every Thursday and Sunday evening. And to my friends Arnav, Dan, Hanock, Jenna, Jordan,

Rafa, Nathan (the foodie variety), Nathan (the NYC-but-actually-Hoboken variety), Ceren, Tim, Mili, Stephanie, Jey, Sarah, and Danesh for the memories we've made together and their support through tough times.

Last, but never least, I want to thank my family. To my aunt and uncle, for being my home away from home. To my cousin for San Francisco shenanigans and random philosophical moments. And to my parents, for their endless love, for financing my education, and for reminding me to eat once in a while. I love you both forever and always.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

1.1 Motivation

Writing unit tests is a time consuming and tedious process for the programmer, as they must devise many interesting inputs for the program under test. In doing so, they are ultimately thinking in terms of a specification or property, and the unit test input is just an instantiation of that specification. Thankfully, automatic testing tools such as QuickCheck and its many variants [6] have enabled testing of software against a specification by generating a large number of random inputs, also known as property based testing (PBT). For an example of a property, consider the following written in Haskell syntax: given a list `xs`, reversing `xs` then reversing `xs` again should yield the original list `xs`. This is also known as a "There and back again" property [26].

```
reverse (reverse xs) = xs
```

Figure 1.1: "There and back again" property on `reverse`

To test this property, a PBT library first generates a list `xs` then executes the property. If the result is true, it tries again until it reaches a specified number of tests. If not, a (usually minimal) counterexample is reported to the user.

If we view software verification strategies as a spectrum with unit testing at one end, then formal verification is at the opposite end, providing the strongest guarantees but requiring much

more time, effort, and expertise. It is far too common to spend hours, days, even weeks on a proof only to discover that a theorem is not actually true. PBT sits in the middle of the spectrum: much more confidence in the correctness of code but not significantly more developer time compared to unit testing. Therefore, it is no surprise that PBT has also been applied to proof assistants like Rocq (formerly Coq) to aid in formal verification [17]. We can encode the theorem as a specification in a PBT library and if there is a falsifying input, either the theorem is not valid or we found a bug in our program. For a simple example, consider the following theorem written in Haskell syntax, where `(++)` means list concatenation and `xs` and `ys` are both lists.

```
reverse (xs ++ ys) = reverse xs ++ reverse ys
```

Figure 1.2: False property on `reverse`

If we try to prove this, we soon get stuck, because this theorem is not actually true! The correct theorem is:

```
reverse (xs ++ ys) = reverse ys ++ reverse xs
```

Figure 1.3: Corrected property on `reverse`

Running QuickCheck on the false theorem yields the following falsifying input:

```
*** Failed! Falsified (after 3 tests and 5 shrinks):
[0]
[1]
```

Figure 1.4: Example QuickCheck output

Observe that the counterexamples are minimal in terms of length and that the numbers in the list are small. Since PBT libraries work by generating random data, the first falsifying inputs found usually contain a lot of noise (e.g. extraneous list elements) that make it difficult for a

programmer to use the falsifying input for debugging. Therefore, PBT libraries apply shrinking: trying progressively smaller counterexamples until it can no longer reproduce the failure. In the case of shrinking a list of integers, a PBT library might try to remove items from a list and make the absolute value of the integers in the list smaller.

It is no accident that the examples so far only involve pure functional programs. PBT libraries generally focus on testing such programs because they are easier to automatically test than stateful programs, as it need not be concerned about the states between function calls and before and after execution. Furthermore, the specifications we test in a pure program can be fine grained (e.g. involving a single function) while still revealing interesting behaviors. When programs involve state, reasoning about the properties of the program becomes more difficult. Goldstein et al. [9] interviewed 31 software engineers at Jane Street about their usage of PBT libraries and found that state made it "harder to think about what are the right laws" and that "dealing with the interaction of very large and complicated systems" makes "meaningfully test[ing] with PBT" more difficult.

In stateful programs, bugs live in the modification of state, which generally involves multiple stateful functions executed in a specific order. Therefore, automatic testing of stateful programs requires a generator for states and for inputs, and we quickly run into the state explosion problem. PBT libraries come with generators for basic types like integers and characters out of the box, and some can automatically derive generators from predicates [5]. However, complex generators may still have to be hand-written, and hand-tuned generators can be much more effective at finding bugs by virtue of fine control over the distribution of test data and the programmer's domain-specific knowledge, at the expense of greater development time.

To avoid the need for state generators, stateful PBT libraries have been developed, such as

Hedgehog [10] and quickcheck-state-machine [2]. Rather than generating function calls directly, these libraries generate abstract commands that are later mapped to the corresponding functions under test at test execution. To make test generation possible, stateful PBT libraries require the programmer to separately define an abstract model of the program under test, transition functions for the state machine, preconditions and postconditions for each transition, and a way to relate the abstract model with the actual implementation when executing the test. This introduces more development and maintenance overhead, more boilerplate code, and the need to ensure that the abstract model actually corresponds to the system under test. However, this does not mean that the stateful generation paradigm is less useful. The idea of generating sequences of commands or calls to an API achieves two goals. It provides the opportunity to interleave states to more effectively discover bugs, and it reflects how a stateful API is actually used: building up the state from an initial state using calls to the API rather than generating the state directly.

```
Queue q;
gen_queue(&q);
int v_0 = pop(&q);
```

(a) PBT test

```
Queue q;
init_queue(&q);
push(&q, 5);
push(&q, 7);
int v_0 = pop(&q);
```

(b) Stateful PBT test

Figure 1.5: Stateless PBT vs. Stateful PBT

The C-like code above illustrates the difference between a PBT library that might automatically derive or require the programmer to define a generator for a random stack to test a function, whereas a stateful PBT library might generate the stack by generating calls to `init_queue` and `push`, both of which are part of the implementation of the API rather than testing-specific code. Observe that the sequencing of multiple parts of the API and how they interact with each other is

closer to an end-to-end test than a unit test. Although it is not easy to define a property analogous to the reverse example over a sequence, each postcondition of each state transition function is a property. Preconditions and postconditions provide a lower barrier of entry to reasoning about the properties of code, as the programmer only has to think on a per function basis.

Existing property-based testing tools offer push-button testing (i.e. being able to run tests with minimal additional code besides the property) of single functions. Existing stateful property-based testing tools generate sequences of function calls, using interactions between stateful functions to discover bugs. However, they are decidedly not push-button, requiring extensive test-only code such as a programmer-defined state machine or model as well as preconditions, postconditions, and invariants. We believe the programmer should focus as much as possible on the specification of their program and less on boilerplate test code. Thus, we aim to bridge the gap with our tool, `seq-test`, which uses programmer-specified preconditions and postconditions for push-button testing of stateful C APIs. We compare `seq-test` against two random testing tools, Bennet and AFL++, which leverage programmer-specified preconditions and postconditions to test single functions. While `seq-test` is not as effective at finding bugs in some selected programs, for bugs involving the interaction of states between functions, `seq-test` detects them while Bennet and AFL++ cannot.

1.2 Contributions and thesis structure

1. **CN and Fulminate:** We describe the features of CN, a separation-logic refinement type system for verifying C code developed at the University of Cambridge and the University of Pennsylvania [21], and Fulminate [4], the executable form of CN. In the process of

verifying C code, the programmer writes CN preconditions, postconditions, predicates, and other annotations as comments in the source code of a C program. This section largely introduces the features by implementing a verified queue.

2. **Tool design**: We introduce our testing tool, `seq-test`, and how it generates random call sequences. `seq-test` uses CN annotations and the types of the functions under test to guide its generation. We also explain how `seq-test` chooses functions during generation to create "interesting" sequences that effectively find bugs and how falsifying sequences are shrunken. This chapter also acts as a tutorial of `seq-test` and an explanation of its features. We demonstrate `seq-test` on buggy version of the queue implemented in the previous chapter to demonstrate a workflow using the testing tool and to concretize the tool design with visualizations of the internal state of the tool while generating and shrinking call sequences. We also identify advantages and drawbacks of our approach.

3. **Evaluation**: We run `seq-test` on a modified version of the evaluation for Bennet [1], a testing tool that derives generators from CN predicates. The evaluation comprises case studies from the CN tutorial [22], a demonstration of Fulminate (a tool that inserts runtime checks for CN specifications) [14], and a few other sources. We then compare `seq-test` to Bennet and AFL++ [7], a popular fuzzing tool, and assess its bug-finding ability with Etna [25], a tool for evaluating PBT frameworks. Etna allows us to inject bugs, run multiple trials, and summarize the results.

4. **Conclusion**: Finally, we conclude with related work and directions for future work.

Chapter 2: CN and Fulminate

The operational semantics of C and the minutiae of separation logic are beyond the scope of this thesis, but we must first understand the mechanism underlying the preconditions and postconditions that make generation possible. To this end, we largely introduce CN by example, implementing a verified queue, and give a sample of the expressive power of CN.

2.1 Motivation

Errors in systems software can compromise the correctness of the software running above it, which motivates significant efforts to ensure correctness in systems software. However, there is a gap between verified software and systems software in that verified software is generally written with verification in mind from the start. However, verification-driven development requires specialized knowledge and thus impedes the wider adoption of verification techinques in development. Ideally, systems software teams could write systems code as they usually would (e.g. in C) and verify that code. Unfortunately, C is a far from an ideal verification target, as it contains manipulation of pointer representations, complex ownership patterns, function pointers, undefined behaviors, etc. Real systems software uses many such features that are difficult to verify, but restricting their use to ease verification would make implementing high-performance production systems difficult, if not impossible. Furthermore, it is not enough to just to develop a tool that

makes verification possible, but usable: the developer should be able to understand, write, and maintain some of the assertions used in verification. Verification should cleanly return an accept or failure, and failures should be explained with counterexamples. CN was designed to reconcile these two aspects of two main challenges in verifying systems software: handling real software and making verification usable.

2.2 Technical Details

CN is built upon Cerberus [18], an explicit semantics for a large portion of ISO C. Cerberus is elaborated to a much simpler language named Core, which is executable as a test oracle. CN does not directly verify a C program but instead uses its Core elaboration, because it is easier to design a refinement type system for the more straightforward semantics of Core than for C. Most of Cerberus is implemented in Lem, a language based on OCaml for developing executable formal models, which is compiled to OCaml. The parser and other tooling are written in OCaml.

The semantics cover enough of ISO C to allow CN to verify conventional systems code, including the verification target of the original paper, the pKVM hypervisor developed by Google. Since Cerberus is parameterized by its memory model, the authors chose a concrete memory model (i.e. pointers are integers), but this does not fully handle integer-pointer casts.

The type system is a separation logic refinement type system with an SMT backend for refinement subtyping and pointer-equality reasoning. Refinement types are types endowed with a predicate, also called a refinement [24]. Separation logic [23] allows CN to reason about mutable data structures where fields can be referenced from more than one point. For verification to be usable, it must at least fulfill the following two requirements: the verification must compose with

the code structure, so the specification matches the intuition of the developer and scales better, and the verification must be predictable; every input program must either be accepted or rejected. Thus, they choose a refinement type system with linear resource types, leveraging prior work on decidable type inference for refinement types. Linear resource types (i.e. every resource must be used exactly once) make it possible to write function specifications in terms of their value-dependent memory footprint (e.g. a queue is a pointer to a memory block iff it is non-null).

2.3 Types

2.3.1 Refinement types

The examples in this subsection and in the remaining subsections of this chapter are adapted from the CN tutorial [22] and CN paper [21].

In CN, refinement types are used to place constraints on the argument and return types of a function. Consider the `increment` function below on the left, which has the undefined behavior of integer overflow and thus causes a type error:

```
int increment (int n) {
  return n + 1;
}
```

(a) Original `increment` implementation

```
int increment (int n) {
/*@ requires n < power(2,32) - 1 @*/
/*@ ensures return == n + 1 @*/
  return n + 1;
}
```

(b) Correctly typed `increment`

Figure 2.1: unverified and verified `increment`

The `requires` keyword establishes the precondition of the function and the `ensures` keyword establishes the postcondition. In this case, the precondition requires that `n` is sufficiently

small to prevent overflow, while the postcondition checks the functional correctness. The CN type of `increment` in a more standard refinement type notation:

$$\forall n : \texttt{int}.(n < 2^{32} - 1) \Rightarrow \exists r : \texttt{int}.(r = n + 1) \ \wedge \ \texttt{emp}$$

$(n < 2^{32} - 1)$ specifies a refinement on the base type of n, which is an integer. `emp` comes from separation logic and represents a heap where all addresses contain undefined memory. In the CN-annotated C code, the base types are written as regular C types, while refinements on those types are written in the annotations. Refinements are boolean typed expressions, but CN supports far more expressions than in this simple example. These include boolean operations, the ternary operator, pointer offsets, pointer to integer casts, and struct member access and update.

2.3.2 Resource types

To reason about memory safety, CN uses separation logic resources and ownership, where a resource represents the permission to access a memory region [22]. Consider the simple example of a function that returns the integer pointed to by an integer pointer:

```
unsigned int read (unsigned int *p) {
    return *p;
}
```

Figure 2.2: `read` function

For the read to be safe, `p` must be initialized and the function must have permission to access the memory `p` points to, so the function fails to type check in its current form. Therefore, we must assert ownership over the memory region. Given a type `T` and pointer `p`, `RW<T>(p)` asserts ownership over an initialized memory region the size of type `T` starting at location `p`. We can now verify that `read` is safe by adding a precondition that requires ownership of `p`, and

since the function only reads the pointer, we can also add a postcondition returning ownership of `p`. Finally, we can also reason about the actual value being pointed to using `take P = ...`, which binds the pointee value to the name `P`. Combining these features, we have the following verified `read` function:

```
unsigned int read (unsigned int *p)
/*@ requires take P = RW<unsigned int>(p);
    ensures take P_post = RW<unsigned int>(p);
            return == P;
            P_post == P;
@*/
{
    return *p;
}
```

Figure 2.3: Verified `read` function

Using `take P = ...` allows us to bind the initial pointee value of `p` to the name `P`. We can then use `P` to specify that the initial and returned pointee values are the same, as one would expect from a read function. `return == P` and `P_post == P` are not strictly required for the function to pass verification; they only specify the functional correctness of `read`. However, `take P_post = RW<unsigned int>(p)` is required. Since CN uses linear types, if a function takes ownership of memory, it must either return it to the caller or deallocate it, so leaving that postcondition out is equivalent to leaking memory from the perspective of the verifier.

2.4 Case study on imperative queues

We will now use an imperative queue with init, push, and pop operations as a case study to introduce more advanced features of CN and to show the features previously introduced in a single context. We reuse the case study in explanation of `seq-test`. In the rest of this chapter,

we work through a short proof of correctness for `push_queue`.

2.4.1 Datatypes

CN has algebraic datatypes, which are useful for reflecting heap structures defined in C into CN to write specifications for C functions that manipulate heap structures like linked lists. The CN datatype is essentially a model of the C structure for the verifier.

For example, here is a C definition for singly-linked lists and its CN counterpart:

```
struct queue_cell {
  int first;
  struct queue_cell* next;
};
```

(a) C list definition

```
datatype List {
  Nil {},
  Cons {i32 Head, datatype List Tail}
}
```

(b) CN list definition

Figure 2.4: C vs CN struct definition

Note that the `queue_cell` struct is the same as a singly-linked list, just with different field names, so we can use the CN list datatype to write CN predicates, functions, and lemmas over CN linked lists to verify a C implementation of a queue, and use the datatype constructors directly in CN preconditions and postconditions. For reference, the queue API function signatures and the `queue` struct definition are provided:

```
struct queue {
  struct queue_cell* front;
  struct queue_cell* back;
};
```

```
struct queue* empty_queue ()
void push_queue (int x, struct queue *q)
int pop_queue (struct queue *q)
```

Figure 2.5: Queue struct definition and API

2.4.2 Predicates, functions, and lemmas

First, let us use the CN list to define predicates to check a pointer `q` points to a valid queue.

```
predicate (datatype List) QueuePtr_At (pointer q) {
  take Q = RW<struct queue>(q);
  assert (   (is_null(Q.front)  && is_null(Q.back))
          || (!is_null(Q.front) && !is_null(Q.back)));
  take L = QueueFB(Q.front, Q.back);
  return L;
}
predicate [rec] (datatype List) QueueAux (pointer f, pointer b) {
  if (ptr_eq(f,b)) {
    return Nil{};
  } else {
    take F = RW<struct queue_cell>(f);
    assert (!is_null(F.next));
    assert (ptr_eq(F.next, b) || !addr_eq(F.next, b));
    take B = QueueAux(F.next, b);
    return Cons{Head: F.first, Tail: B};
  }
}

predicate (datatype List) QueueFB (pointer front, pointer back) {
  if (is_null(front)) {
    return Nil{};
  } else {
    take B = RW<struct queue_cell>(back);
    assert (is_null(B.next));
    assert (ptr_eq(front, back) || !addr_eq(front, back));
    take L = QueueAux (front, back);
    return Snoc(L, B.first);
  }
}
```

Figure 2.6: Predicates for verifying `push_queue`

In plain English, the top level `QueuePtr_At` takes ownership of a queue pointer and asserts that a queue is either empty or nonempty, then returns the C queue as a CN list. The helper predicate `QueueFB` is required since conditional expressions are not allowed after a `take`. It

returns an empty queue if the `front` pointer is null. Otherwise, we must walk through the queue in the C heap and extract an initialized and owned version of the C queue as a CN list. However, we must treat the final cell in the list separately for two reasons. First, pushing to the queue follows the `back` pointer directly rather than recursing from the `front` pointer. Therefore, we `take` the `back` pointer immediately. Second, there will be two pointers to the final cell: one is the `back` pointer and the other is the `next` pointer of the second to last `queue_cell`. The `QueueAux` predicate walks through the queue and returns the corresponding CN list except for the final cell. To construct the complete queue, we add the final cell to the list returned by `QueueAux` to the end with the function `Snoc`:

```
function [rec] (datatype List) Snoc(datatype List Xs,
   i32 Y) {
  match Xs {
    Nil {} => {
      Cons {Head: Y, Tail: Nil{}}
    }
    Cons {Head: X, Tail: Zs}  => {
      Cons{Head: X, Tail: Snoc (Zs, Y)}
    }
  }
}
```

Figure 2.7: `Snoc` definition in CN

We now have almost everything we need to prove `push_queue` correct, but we need to provide the SMT solver with some facts about pushing to a queue using the lemma on the following page.

This lemma states that we have two ways to read out the values in some sequence of `queue_cells` of length at least two starting from the cell `front`. Let `c` be the cell following some given cell `p`. We can either gather the cells from `front` to `c`, or we can gather just the

```
/*@
lemma push_lemma (pointer front, pointer p)
  requires
      ptr_eq(front, p) || !addr_eq(front, p);
      take Q = QueueAux(front, p);
      take P = RW<struct queue_cell>(p);
  ensures
      ptr_eq(front, P.next) || !addr_eq(front, P.next);
      take Q_post = QueueAux(front, P.next);
      Q_post == Snoc(Q, P.first);
@*/
```

Figure 2.8: Lemma for verifying `push_queue`

cells from front to `p` and then `Snoc` the single value from `c`.

When we apply the lemma, `p` will be the original `back` cell and `c` will be the new cell. To prove the lemma by induction, let `p` to be any internal cell in the list starting at front and let `c` be cell following `p`. If we want our precondition and postcondition to use `QueuePtr_At`, we need this lemma because adding a new cell at the end of the queue requires reassigning ownership of the original `back` cell. Recall that in `QueuePtr_At`, we take ownership of the `back` cell separately from the rest of the queue. In the postcondition, the original `back` cell needs to be treated as part of the rest of the queue, so the new `back` cell can be treated separately.

Bringing everything together, we now have the verified `push_queue` function.

2.5 Fulminate

Fulminate [4] is the executable form of CN specifications. However, evaluating separation logic predicates in general may require backtracking, searching for concrete values to instantiate existentials, and performing worst-case exponential runtime search for heap splittings. Fortunately, the restrictive syntax of CN that forces programmers to write specifications that compile only to

```
void push_queue (int x, struct queue *q)
/*@ requires take Q = QueuePtr_At(q);
    ensures take Q_post = QueuePtr_At(q);
            Q_post == Snoc (Q, x); @*/
{
  struct queue_cell *c = cn_malloc(sizeof(struct queue_cell));
  c->first = x;
  c->next = 0;
  if (q->back == 0) {
    q->front = c;
    q->back = c;
    return;
  } else {
    struct queue_cell *oldback = q->back;
    q->back->next = c;
    q->back = c;
    /*@ apply push_lemma (q->front, oldback); @*/
    return;
  }
}
```

Figure 2.9: Verified `push_queue`

decidable SMT queries also results in predicates that can be executed in a reasonable amount of time over concrete heap states using a reified ghost state to record ownership information at runtime. From the predicates, functions, and lemmas above, one might notice that they look like they could be executed like a function in any programming language. Indeed, this simple observation underlies the reasonable runtime overhead of Fulminate.

The ghost state maps each byte of memory to the stack depth of the C function holding ownership of that byte, if it exists. At each ownership creation point, such as function parameters, instrumentation is added to map the section of memory holding the value of the variable to the current stack depth. At ownership destruction, such as at function exit, the section of memory is unmapped. Before memory is accessed, the ghost state is checked to ensure that it is mapped to the current stack depth. For each predicate in both the precondition and postcondition, the

predicate is evaluated as a boolean function. In the precondition, the accessed memory is checked to ensure that it is mapped to the caller's stack depth, and then it is updated to the current stack depth, while in the postcondition, the accessed memory is checked to ensure that it is mapped to the current stack depth, and then it is updated to the caller's stack depth. The postcondition instrumentation is inserted as an epilogue, so all return statements in the original code are replaced with an assignment of the return value to a return variable followed by a `goto` to the epilogue.

For demonstration, here is a function with a precondition requiring the argument to be 42:

```
void f(int x)
/*@ requires x == 42i32; @*/
{ }
```

Figure 2.10: Basic Fulminate precondition example

Fulminate adds an executable version of the `x==42` (where x is a 32-bit integer), which converts the argument to the Fulminate representation then checks the equality. The instrumented code, simplified for readability, is:

```
void f(int x)
{
    cn_bits_i32* x_cn = convert_to_cn_bits_i32(x);
    cn_assert(cn_bits_i32_equality(
        x_cn, convert_to_cn_bits_i32(42LL)), PRE
    );
}
```

Figure 2.11: Instrumented version of Figure 2.10

Violations of preconditions or postconditions cause the program to crash with an exit code corresponding to the type of violation along with an error message directing the programmer to the line in the source code containing the specification that was violated.

For a more complex example involving ownership, recall the `read` function from the previous section, followed by the instrumented code simplified for readability:

```
unsigned int read (unsigned int *p)
/*@ requires take P = RW<unsigned int>(p);
    ensures take P_post = RW<unsigned int>(p);
            return == P;
            P_post == P;
@*/
{ return *p; }

unsigned int read (unsigned int *p)
{
    // begin precondition
    unsigned int __cn_ret;
    cn_pointer* p_cn = convert_to_cn_pointer(p);
    cn_bits_u32* P_cn = owned_unsigned_int(p_cn, PRE);
    // begin body
    c_add_to_ghost_state((&p), sizeof(unsigned int*),
       get_cn_stack_depth());
    { __cn_ret = CN_LOAD(*CN_LOAD(p)); goto __cn_epilogue; }
    // begin postcondition
    __cn_epilogue:
        c_remove_from_ghost_state((&p), sizeof(unsigned int*));
    {
        cn_bits_u32* return_cn = convert_to_cn_bits_u32(__cn_ret);
        cn_bits_u32* P_post_cn = owned_unsigned_int(p_cn, POST);
        cn_assert(cn_bits_u32_equality(P_post_cn, P_cn), POST);
        cn_assert(cn_bits_u32_equality(return_cn, P_cn), POST);
    }

    return __cn_ret;
}
```

Figure 2.12: Instrumented `read` function

Chapter 3: The `seq-test` Tool

In this chapter, we introduce `seq-test`, a stateful property-based testing tool for CN. We describe how `seq-test` uses CN, Fulminate, and function signatures to guide generation and discover bugs and the two-phase algorithm for shrinking falsifying examples. Since Cerberus is implemented in OCaml, to use the existing infrastructure for C functions and types, we implement `seq-test` in OCaml as a subcommand of the `cn` command. This chapter also serves as a tutorial for `seq-test`, enumerating its different options and their implementation details.

3.1 Overview

The examples used to illustrate various aspects of our implementation center around the imperative queue from the previous chapter. For reference, here is the C struct definition, CN datatype definition, and function signatures of the operations on the imperative queue:

```
struct queue {                  datatype List {
  struct queue_cell* front;         Nil {},
  struct queue_cell* back;          Cons {i32 Head, datatype List Tail}
};                              }

struct queue_cell {             struct queue* empty_queue ()
  int first;                    void push_queue (int x, struct queue *q)
  struct queue_cell* next;      int pop_queue (struct queue *q)
};
```

Figure 3.1: Queue struct definition, CN datatype definition, and queue operations

```
int pop_queue (struct queue *q)
/*@ requires take Q = QueuePtr_At(q);
             Q != Nil{};
    ensures  take Q_post = QueuePtr_At(q);
             Q_post == Tl(Q);
             return == Hd(Q);
@*/
```

Figure 3.2: `pop_queue` and its specification

Consider the `pop_queue` function with the CN specification above. Suppose the implementation has a bug which returns 0 rather than the front of the queeu when there is more than one element in the queue. Of course, this is a simple bug, but it is enough to demonstrate all the components of `seq-test`. The postcondition being violated is `return == Hd(Q)`, but to discover the violation, we need to generate a queue, then pop from it.

```
struct queue* x1;
gen_queue(x1);
int x2 = pop_queue(x0);
// assert something about x1
```

(a) Stateless PBT test

```
struct queue* x1 = empty_queue();
push_queue(5, x1);
push_queue(-12, x1);
int x2 = pop_queue(x1);
```

(b) Stateful PBT test

Figure 3.3: Stateless PBT vs. Stateful PBT

A stateless PBT library like QuickCheck [6] might require you to write a generator and a shrinker if it does not have them implemented for the input types by default, as in the example code. However, in the test generated by the stateful PBT library, observe that we already have a generator for queues "for free" in the form of initializing a queue with `empty_queue` and pushing to it with `push_queue`. It is the intuition that test input can be built up from calling a stateful API as though a user were calling it that motivates the generation strategy of this tool.

3.2 Generation

If we want to generate a call to `push_queue`, we need to be able to generate an `int` and a `struct queue*`. Primitive types such as `char`, `int`, `unsigned int`, `bool`, and `double` can be generated using the `Random` module in OCaml, but `seq-test` cannot generate heap memory for the pointer to point to. Thankfully, `seq-test` does not need to do this at all. In the queue example, it can call `empty_queue` and assign the return value to some pointer variable, then pass that variable as an argument to `push_queue`. However, the generator now needs to remember that it generated an empty queue.

3.2.1 Generation context

The generation context keeps track of previously generated calls and for each call, the variable to which the return value of the call is assigned, if it exists. This context is entirely symbolic: the variable names and types are known during generation, but the concrete value is not. The context has the following OCaml type, slightly altered for readability:

```
type call = string option * bool * ctype * string *
            (ctype * string) list
type context = call list
```

Figure 3.4: Call and generation context type

The elements of the `call` 5-tuple represent the following respectively:

1. The variable name to which the result of a call is assigned. If the return type of the function is void, then there is no variable name.
2. If the function is static. Fulminate requires static functions to have a special prefix, so this

field is used when converting a `call` to the C function call in the generated test.

3. The function return type. `ctype` represents the C types supported by Cerberus.

4. The function name.

5. The generated arguments to the function and their types.

For example, here is a generated sequence and the corresponding context:

```
struct queue* x1 = empty_queue();
push_queue(1298, x1)};
```

(a) The sequence

```
[(Some x1, false, struct queue*,
  empty_queue, []),
 (None, false, void, push_queue,
 [(int,1298),(struct queue*,x1)])]
```

(b) The context

Figure 3.5: Generated sequence and its context

3.2.2 Function callability

Generalizing the process of generating a `push_queue` call, we now have an intuitive definition of callability: a function is callable during generation if, for each of its arguments, the argument can be generated or the result of a previous call can be passed in. When `seq-test` generates a call, it only considers callable functions. To generate a function, for each argument, it first checks if the argument is a primitive type, which `seq_test` can generate. It also scans over the context looking for previously generated results of the same type[1] as the desired argument, accumulating them in a list. Finally, it chooses to select from the list or a generated argument (if it exists) with equal probability. However, this process says nothing about how a callable function is chosen to be called. Clearly, a uniform distribution of callable functions is not always optimal.

[1] If for some type `t`, the desired argument has type `**t` and there exists a variable `x` of type `*t`, the generator will pass in `&x`, so `x` would be included in the list.

In the queue example, we need at least double the number of `push_queue` calls to be generated as `pop_queue` calls, since the bug requires popping from a queue with at least two elements.

3.2.3 Generating interesting tests

In the generation context, the variable names, static markers, and variable types are strictly necessary for generating calls, since determining the callability of the function requires knowing which values of which type are available to reuse. The extra function name and function argument fields in the 5-tuple are used for tracking statistics about the generation. At a high level, a random generator is a probability distribution over values of some type. Therefore, it is useful for the programmer to see the proportion of each function in the test, and evaluate whether the distribution should be altered. For example, the programmer might want to generate more calls to functions that are more likely to cause bugs. After running `seq-test` on a correct queue, one might see the following output:

```
passed: 281, failed: 0, skipped: 0
Distribution of calls:
empty_queue: 18 calls (6.41% of calls)
push_queue: 182 calls (64.77% of calls)
pop_queue: 81 calls (28.83% of calls)
```

Figure 3.6: Example output of `cn seq-test queue.c`

To create the weighted distribution over callable functions, the generator calculates a score for each function based on the types of the function arguments and the number of arguments that can be filled in by passing in previously generated return values. The exact scores used are not particularly important, as the weight of each function is relative to the total weight of all callable functions. For the queue example, suppose we have the same context shown

on the previous page, where we have generated `struct queue* x1 = empty_queue()` followed by `push_queue(1298, x1)`. At this point in the generation, the score of a `push_queue` will be calculated by adding the base score (1), the score of an integer (15), the score of a pointer to a struct (50), and a bonus (25) for having an argument that uses a previously generated value, `x1` for the pointer to a queue in this case. The base score exists so all callable functions at any given point in the generation have a nonzero probability of being called. The bonus score for arguments that can reuse previously generated return values is intended to cause the generation to interleave previous states with the current one, thus increasing the likelihood of discovering bugs. Calculating all the scores for the queue functions, we have `empty_queue` with a score of 1, `push_queue` with a score of 91, and `pop_queue` with a score of 76, so a `push_queue` has a $\frac{91}{1+91+76} \approx 54\%$ chance of being called at the current point in the generation.

3.2.4 Discarding invalid calls

We now need to check if the generated call should be added to the sequence or discarded if the call violates a precondition. Consider `pop_queue`, which has the following precondition:

```
int pop_queue (struct queue *q)
/*@ requires take Q = QueuePtr_At(q);
                    Q != Nil{}; @*/
```

Figure 3.7: Precondition of `pop_queue`

Suppose we have generated a generated the following sequence:

```
struct queue* x1 = empty_queue();
signed int x2 = pop_queue(x1);
```

Figure 3.8: Illegal sequence of calls

This clearly violates the precondition that `Q != Nil`, so the call to `pop_queue` should be discarded. To check the validity of the sequence, the file under test is first instrumented with Fulminate to add the executable forms of the preconditions and postconditions, among other instrumentation. The generated test file is also instrumented to initialize the ghost state and add instrumentation to keep track of ownership of the return values, such as the empty queue assigned to `x1` in the example above. The instrumented source file and instrumented test file are then compiled and linked to produce an executable test file. The test file is executed, and if the exit code is 1 (the exit code assigned to a precondition violation by Fulminate), the newly generated call is discarded and another call is generated. Users may set a maximum number of calls to generate with the `--max-num-calls` flag. Note that this is the maximum number of calls generated, which includes calls that are discarded. As one might expect, a major bottleneck of the generate, compile/link, execute loop is the compilation/linking. For simplicity, we deem the process to add a call to a sequence an *iteration*, and the entire process of generating a sequence *generation by execution*.

One caveat of this strategy is the idempotence of the test. For example, if the program under test writes to a file, then running the test multiple times may yield different results, since the file changes after each run. Model-based generation gets around this by requiring the programmer to define a model initialization function, so we allow the programmer to specify an initialization function with the `--init` option. Otherwise, we attempt to select the initialization function by callability, under the assumption that an initialization function is always callable. In the queue example, the `empty_queue` function is always selected first since it is the only callable function when starting from an empty generation context.

3.2.5 Generating multiple calls per iteration

Instead of generating a single call per iteration, `seq-test` can be configured to generate multiple calls per iteration with the `--num-tests` option, where a test refers to a sequence of calls. The generator maintains n generation contexts for n tests, and the contexts have no dependency on each other. All sequences exist within a single test file. In a single iteration, the generator attempts to add one call to each sequence and updates the contexts for each sequence accordingly. In effect, it is almost as if the user manually ran the `seq-test` command n times, each time generating a single sequence. The result differs from manually running the command in that the generation can short circuit: if any sequence results in a postcondition violation, the generation immediately stops, all nonfailing sequences are discarded, and shrinking begins on the first failing sequence.

Generating multiple sequences with separate contexts also increases the likelihood that a bug, if it exists, will be found within a single call to `seq-test`, which can be illustrated with the following simple example. Suppose the following sequence has been generated so far, and the user called `seq-test` with `--max-num-calls = 5`:

```
struct queue* x1 = empty_queue();
push_queue(45, x1);
signed int x2 = pop_queue(x1);
push_queue(209, x1);
```

Figure 3.9: Sequence generated so far by `cn seq-test --max-num-calls=5`

However, recall that the bug in the queue example is `pop_queue` on a queue with more than one element returns zero rather than the actual front element of the queue. Since we can only generate one more call before we reach the maximum number of calls, and the queue size is zero

at this point in the generation, there is no way the bug will ever be discovered. By generating multiple sequences at once, the likelihood is reduced that all sequences will end up in a state where the bug cannot be discovered upon generating the final call.

3.3 Shrinking

Counterexamples discovered through random testing are often large and contain noise that is irrelevant to the bug. The goal of shrinking is to minimize counterexamples to make them amenable to debugging by a human. Below is an example of shrinking on a sequence of calls:

```
struct queue* x1 = empty_queue();
push_queue(-102, x1);
push_queue(32, x1);
struct queue* x2 = empty_queue();
push_queue(22, x2);
signed int x3 = pop_queue(x2);
push_queue(x3, x1);
signed int x4 = pop_queue(x1);
```

(a) Before shrinking

```
struct queue* x1 = empty_queue();
push_queue(1, x1);
push_queue(0, x1);
signed int x2 = pop_queue(x1);
```

(b) After shrinking

Figure 3.10: Example of shrinking

Shrinking is essentially a greedy hill climbing algorithm that reaches a locally minimal counterexample. In general, a shrinking function `s : 'a -> 'a list` (to use OCaml notation) takes a value of type `'a` and produces a list of smaller values of the same type. Whoever writes the function decides what constitutes a smaller value, but all members of the returned list must be strictly smaller than the input. Given a property `P` and a value `x` such that `P x` is false, for each `x' ∈ s x`, a *shrinker* tries `P` on `x'` until it finds an `x'` that does not fail the property. There are various orders for the shrinker to go through `s x`, such as from smallest to greatest or by binary search. For example, to shrink an integer `n`, the shrinking function could generate the following

list: `[0, n/32, n/16, n/8, n/4, n/2]`. However, the sort of shrinking discussed so far concerns how stateless PBT libraries shrink: shrinking inputs to functions. A shrinker for stateful PBT needs to reduce the number of calls in addition to shrinking their inputs.

3.3.1 Phase 1: Eliminating extraneous calls

First, we define a minimal sequence as a sequence that cannot have any calls removed from it without failing to reproduce the postcondition violation. If multiple sequences are generated within a single test file, the first sequence that causes a postcondition violation is selected for further shrinking and all other sequences are discarded. Technically, this discard phase is the first phase of shrinking, but it is trivial to implement, so we do not consider it to be a phase.

In the first actual phase, we remove extraneous calls within a sequence. On one extreme, we could generate all possible smaller sequences (the powerset) to reach a globally minimal sequence. However, for a sequence of length n, we would have to generate 2^n sequences, which is intractably expensive. Instead, the shrinking functions returns multiple sequences (one with only the first call removed, only the second call, and so on), runs the new sequences, and checks if a postcondition violation still occurs. If the violation no longer occurs in any sequence, the shrinking ends. Otherwise, the shrinker chooses the first failing sequence to continue shrinking and the process repeats until the shrunken sequence is the same size as the sequence from the previous step (i.e. a fixpoint) or until the last call in the sequence is reached.

As mentioned in the generation section, running the updated test is the most expensive part, since it involves linking and compiling the test file. If followed the high-level description exactly, the shrinker would recompile and run the test file up to $n - 1$ times for a sequence of length n,

once for each call removed. Therefore, we use two methods to reduce the number of recompiles.

For the first method, we identify the calls that are absolutely required to reproduce a postcondition violation. We call this set of calls the minimal set. The shrinker uses a directed acyclic graph (DAG) (V, E) where each $v \in V$ is a call and each directed edge $(c_1, c_2) \in E$ means call c_1 uses the result of call c_2 in its arguments. We call (V, E) the dependency graph. However, observe that the type of our generation context implicitly encodes this DAG, where the variables used in the arguments of a call represent the directed edges to another call. The final call must be in the minimal set since it triggered the postcondition violation, so the shrinker performs depth-first search starting from the node representing the final call, and the visited nodes become the minimal set. Calls in the minimal set are never removed, saving recompiles.

```
struct queue* x1 = empty_queue();
push_queue(-102, x1);
push_queue(32, x1);
struct queue* x2 = empty_queue();
push_queue(22, x2);
signed int x3 = pop_queue(x2);
push_queue(x3, x1);
signed int x4 = pop_queue(x1);
```

Figure 3.11: Minimal set of a sequence

The highlighted calls in the example above represent the minimal set. Starting from the final call, the shrinker sees that `pop_queue` uses `x1` as an argument, so the call whose result is stored in `x1` is also added to the minimal set.

For the second method, if we remove a call that binds its result to some variable `x`, all calls after that use `x` as an argument are also removed. The shrinker constructs another directed acyclic graph (V', E'), where each $v' \in V'$ is a call and each directed edge $(c'_1, c'_2) \in E'$ means call c'_2 uses the result of call c'_1 in its arguments. For a call c that is selected for removal, the shrinker

performs depth-first search starting from the node representing c. The visited nodes are removed as a group, but if any visited nodes are also in the minimal set, then none of the nodes can be removed. Since each $(c'_1, c'_2) \in E'$ represents the reverse of the edges in the dependency graph, we call (V', E') the reverse dependency graph. The highlighted calls in the example are the calls

```
struct queue* x1 = empty_queue();
push_queue(-102, x1);
push_queue(32, x1);
struct queue* x2 = empty_queue();
push_queue(22, x2);
signed int x3 = pop_queue(x2);
push_queue(x3, x1);
signed int x4 = pop_queue(x1);
```

Figure 3.12: Calls removed when `x2` is selected for removal

that are removed when `x2` is removed. Since `x3` uses `x2` as an argument, `x3` is removed and the call to `push_queue` immediately after is also removed, since it uses `x3` as an argument.

Finally, the shrinking function `context -> context list` generates a list of sequences where each sequence is the result of removing one call and applying the two aforementioned optimizations. The sequences in the list are tried from least to most number of calls, and the first failing sequence is kept. The process repeats no calls can be removed.

It is also important to note that we used variable names to determine dependencies because removing a variable is guaranteed to cause a compilation failure if there is code using that variable, so attempting to compile would be wasteful. However, there are certainly other forms of dependencies in a stateful program that we cannot detect. For example, if a function writes to a file and another reads from the same file, removing the call to the write function would affect the result of the read.

3.3.2 Phase 2: Shrinking arguments

Since function calls generally have more than one argument and shrinking an argument requires one compilation, removing calls first then shrinking arguments often results in a net reduction in compilations compared to only shrinking arguments. The `seq-test` tool can only shrink integers, floats, and variables. Shrinking integers and floats occurs in the way described in the introduction of this section. Shrinking a variable of type t is straightforward: generate a value v of type t if possible and then call the shrinking function on v. Crucially, shrinking variables reduces unnecessary dependencies in the dependency graph and reverse dependency graph. To see this, let us revisit Figure 3.12.

If we removed `struct queue* x2 = empty_queue()`, we would have to remove all of its dependent calls, which includes `push_queue(x3, x1)`. However, we cannot remove `push_queue(x3, x1)`, because the bug only occurs when there is more than one element in the queue (`x1` in this case). Using `x3` as an argument creates an unnecessary dependency that forces the highlighted calls to remain in the sequence, even though any integer triggers the bug.

```
struct queue* x1 = empty_queue();
push_queue(1, x1);
struct queue* x2 = empty_queue();
push_queue(0, x2);
signed int x3 = pop_queue(x2);
push_queue(0, x1);
signed int x4 = pop_queue(x1);
```

(a) After shrinking arguments

```
struct queue* x1 = empty_queue();
push_queue(1, x1);
push_queue(0, x1);
signed int x4 = pop_queue(x1);
```

(b) Run first phase again

Figure 3.13: Result of running second phase of shrinking

After shrinking variables, we run the first phase again to remove all calls involving the queue `x2`.

Chapter 4: Evaluation

To evaluate the bug-finding ability of `seq-test`, we compared it against Bennet [1], a random testing tool for CN by Aamer and Pierce that synthesizes generators from CN specifications, and AFL++ [7], a popular mutation-based fuzzer that takes seed inputs and mutates them into new inputs. We discuss Bennet in more detail in Section 5.1.3. We used a modified version of the evaluation of Bennet for our evaluation because of its diverse set of case studies involving complex data structures and mission critical code. Overall, the `seq-test` falls short of the bug finding performance of Bennet, but exceeds AFL++ in some case studies in terms of lower runtime and fewer inputs generated.

4.1 Case studies

We used five of the six case studies from the Bennet evaluation and wrote one of our own. The unused case study is a real-world mission key management system (MKM) [8], which uses a finite state machine to specify the actions required to receive a secret key. We were unable to use this case study because the implementation of `seq-test` uses a newer version of Cerberus which contains a bug that prevents functions returning an `enum` type from being tested. Of the case studies we were able to use, we separate them into two categories: critical code and recursive data structures.

4.1.1 Critical code

The first three workloads are considered critical code due to their high cost of failure. The first is a circular queue from a demonstration of Fulminate [14]. Its mutants involve incorrect queue size calculations in the CN specification and C implementation, and allowing adding to a full queue. The second is a metro station simulation adapted from the CN tutorial [22], which contains many logical constraints on when trains can arrive or depart, how long a train can spend at the station, and so on. Its mutants involve altering the predicate used to determine valid station states by removing bounds or assertions on disjointness. These workloads have simpler implementations than the recursive data structures in the other workloads, but they involve complex preconditions. The correct versions of the circular queue and the metro station simulation have been formally verified with CN. The third workload is a simple file reader that is not formally verified with CN. The user can open a file named `test` with `open_file`, which contains the string `-123456`, and read the absolute value of the string as an integer with `read_file`. There are two mutants, and both involve reading the file. The first mutant has the read syscall reading one too many bytes, causing an ownership error when terminating the filled character buffer with NUL. The second mutant has `read_file` closing the file after reading and freeing the file pointer. For simplicity, we model a file pointer as a pointer to a struct containing the file descriptor as its only field.

4.1.2 Recursive data structures

The remaining three workloads are recursive data structures with invariants. Each data structure has multiple versions, where each version contains a different encoding of the invariant

as a recursive CN predicate. The two versions of the binary search tree (BST) and its mutants are based on *How to Specify It!* [13]. The first version encodes the BST ordering invariant by checking the ordering after the recursive call on the subtrees, while the second version requires that the minimum and maximum for keys within a subtree are passed in as arguments. The AVL tree is based on an example from VCDryad [20], and its two versions are the same as the BST. Finally, the three versions of the sorted list were implemented by the authors of Bennet themselves. The first version asserts after each recursive call that the value of current node is less than or equal to the head of the tail, if it exists. The second version checks if the value of the previous node is less than or equal to the value of the current node, wrapping the previous value in an option. The third version is the same as the second version, but passes the previous value as integer, using the minimum 32-bit signed integer when the previous value does not exist. Aamer and Pierce use different versions of predicates in their evaluation because the generator synthesis varied in difficulty for Bennet when the constraints depended on an argument versus the return value of a recursive generator. We chose to use different versions in our evaluation since programmers often express the same specification in different ways, and capturing those intentions better reflects the real world performance of `seq-test` than selecting a predicate that is faster for Fulminate to check.

4.2 Testing setup

We performed all benchmarks on a 2020 MacBook Air with an M1 processor and 8GB of RAM. To automatically inject manually described bugs and analyze the results, we used Etna [25], a benchmarking tool for evaluating PBT frameworks. We extended Etna to support testing

C programs via `seq-test`. To use the terminology of Etna, each *task* is a pair of a mutant (a buggy version of the code) and an executable specification that does not hold for the mutant. The tasks are grouped into *workloads*, which are the six case studies.

Since AFL++ feeds raw bytes into standard input, each task in the evaluation has a handwritten harness. The harness converts the bytes into valid input for the function under test, checks the precondition, discarding the input if it fails, and calls the function under test. We also gave AFL++ an empty seed input to start. We chose AFL++ as well because it uses random input generation like `seq-test`.

In all of workloads, we ran 20 trials for each task to reduce the impact of outliers. The timeout of each trial was 300 seconds. We ran `seq-test` with `--num-tests=20` and `--max-num-calls=200`. For all tools, we include all steps it would take an actual user to run the tool in our time data. This includes calling Fulminate to instrument the program and compiling the instrumented code. In the case of Bennet, we also include the time to synthesize the generators. The time column is the wall-clock time including the aforementioned steps and running the generated test. The input column is the number of times a function call is generated. In the case of Bennet and AFL++, they test a single function, and some tasks involve the same mutant but testing different functions each time to find the bug. If the tool found the bug in all 20 trials, we consider the task solved. If the tool found the bug in some trials, but not all of them, we consider the task to be partially solved. The results of this experiment are in Table 4.1.

Workload	Tasks	LoC	Tool	Solved (Partial)	Inputs	Time (s)
Metro Sim	6	293	seq-test	0 (0)*	N/A	N/A
			Bennet	5 (0)	27.58 (±5.83)	0.51 (±0.00)
			AFL++	6 (0)	18175.96 (±2814.22)	6.43 (±0.92)
Circular Queue	4	203	seq-test	4 (0)	419.90 (±44.65)	17.77 (±4.61)
			Bennet	4 (0)	3.65 (±0.65)	0.22 (±0.00)
			AFL++	4 (0)	5948.29 (±868.96)	19.15 (±3.40)
File Reader	2	58	seq-test	2 (0)	22.28 (±0.23)	0.84 (±0.01)
			Bennet	0 (0)**	N/A	N/A
			AFL++	0 (0)**	N/A	N/A
BST	32	602	seq-test	30 (2)	152.31 (±34.29)	3.25 (±0.89)
			Bennet	32 (0)	8.32 (±1.90)	0.38 (±0.02)
			AFL++	32 (0)	6621.45 (±1293.87)	3.32 (±0.59)
AVL	26	801	seq-test	22 (2)	311.34 (±37.17)	6.96 (±1.28)
			Bennet	26 (0)	4.41 (±1.20)	0.82 (±0.12)
			AFL++	25 (1)	20114.19 (±4563.32)	9.36 (±2.04)
Sorted List	6	119	seq-test	6 (0)	38.02 (±0.90)	0.34 (±0.01)
			Bennet	6 (0)	5.36 (±1.00)	0.21 (±0.01)
			AFL++	6 (0)	569.42 (±14.52)	0.58 (±0.01)

Table 4.1: Bug-finding results for each workload.

(a) Each entry shows the number of tasks solved (partially), and for inputs and time, the mean of means of each task within a workload with standard error. (*) See Section 4.3.2. (**) See Section 4.3.3.

4.3 Analysis

4.3.1 Overall results

Across the board, `seq-test` performed worse in terms of number of inputs, time to find the bug, and number of tasks solved compared to Bennet. Since Bennet synthesizes generators from the function precondition, the generated values almost always satisfy the precondition by construction. In this sense, Bennet represents a lower bound for the aforementioned metrics, so these results are not surprising. Bennet may have to backtrack to where a pointer was generated

on occasion if it did not allocate enough space before or if it assigned to memory that was assigned to before, but this is rare due to various generator optimizations.

When comparing `seq-test` to AFL++, which works similarly (generating an input randomly and discarding those that violate the precondition) `seq-test` uses far fewer inputs, with `seq-test` using 14.16x (Circular Queue), 43.47x (BST), 64.60x (AVL), and 14.97x (Sorted List) fewer inputs on average. The difference in runtime was much less pronounced, demonstrating the high runtime overhead of generating a single call in `seq-test` compared to AFL++. While `seq-test` took less time than AFL++ to find the bug on average, all workloads were within the margin of error except for Sorted List, where `seq-test` is 1.71x faster. However, the sheer number of inputs AFL++ is able to generate also means it is more likely to generate the exact shape of input required to solve a task, so it has more full solves than `seq-test`.

4.3.2 Partial solves

Workload	**Mutant**	**Detected (%)**
AVL1	insert_3_impl	0/20 (0.0%)
AVL1	rebalance_1	17/20 (85.0%)
AVL2	insert_3_impl	0/20 (0.0%)
AVL2	rebalance_1	17/20 (85.0%)
BST1	delete_empty_vs_singleton	19/20 (95.0%)
BST2	delete_empty_vs_singleton	17/20 (85.0%)

Table 4.2: Bug detection rates in partially solved and unsolved workloads.

Besides all of the Metro Simulation tasks, which `seq-test` was unable to solve, Table 4.2 shows the tasks `seq-test` partially solved or did not solve at all. The mutant `insert_3_impl` is a mismatch between the insertion specification and the implementation. A duplicate insertion to the tree should be a no-op according to the specification, but instead it is inserted into the

right subtree of the current node. In other words, discovering this bug requires inserting the same key into the same tree twice, with no deletions of the key in between. Looking at the generated sequences, we found three key issues. First, this bug requires generating the same integer x twice and passing x into `insert` twice, which is somewhat unlikely. Second, calls to `delete` and calls to `insert` are generated with near equal proportion, since they are assigned the same weight. Even when there were two calls to `insert` with the same key, there was often a call to `delete` that key in between the insertions. The fact that both Bennet and AFL++ only test a single function is an advantage in this scenario, since no other function interferes with the generation. Finally, the node passed in as the root into `insert` might not be same node that was first inserted into, since any previously generated node can be passed in with equal probability.

The mutant `rebalance_1` is a mismatch between the rebalancing specification and implementation on which rotation to perform when both could apply. An unbalanced tree with a child whose children are the same height triggers the bug, but since the generation suffers from the issues mentioned in the previous paragraph, the trees sometimes do not become large enough to trigger the bug, let alone be in the required shape. The average number of inputs to find this bug is 1344.58, significantly higher than the average over all AVL bugs, which demonstrates the difficulty for `seq-test` of finding this bug. We hypothesize that biasing the argument selection for more recently generated values will increase the solve rates for both mutants.

The mutant `delete_empty_vs_singleton` is an incorrect version of the deletion specification which replaces the deleted node with the root of the left subtree when the right subtree is empty or when the right subtree is a node with no children instead of only the former. Finding the bug requires deleting a node where the right subtree is a node with no children. This is somewhat difficult for `seq-test`, since it requires generating a key `x` once for the first insertion, then

inserting a key larger than x, and finally generating x again for the deletion, without any deletions of x between the insertions or any insertions that alter the required tree structure to trigger the bug. Therefore, it is unsurprising that detecting this bug required 559.73 inputs on average, over three times the overall BST average.

Finally, we discuss why `seq-test` was unable to find any bugs in the Metro Simulation. Recall that its mutants only involve altering the predicate used to determine valid station states by removing bounds or assertions on disjointness. Therefore, detecting the bugs requires generating a state consistent with the buggy specification, such that invoking the function on this state yields an output that violates the buggy specification. However, since the actual implementation is correct with respect to the correct predicate for valid states and the states accepted by the correct predicate is a strict subset of the states accepted by the buggy predicate, sequencing function calls to generate the state will never trigger the conditions to detect a bug.

```
struct State s = {
    ModeA = -9,
    ... // irrelevant fields omitted
};
tick(s);
```

Figure 4.1: Bennet generated counterexample

For example, Figure 4.1 shows a counterexample state `s` generated by Bennet where the arrival mode flag is set to -9, which triggers a postcondition violation in `tick`. This state is impossible to generate by sequencing functions in the metro simulation, because nowhere in the implementation is `ModeA` set to anything but 0 or 1, the valid values for the field.

4.3.3 File Reader results

Neither Bennet nor AFL++ were able to solve any task in the File Reader workload. In theory, the file pointer is easy for both tools to generate: allocate memory for the underlying struct, and generate an integer for the file descriptor field. However, the generated file descriptor is almost never valid, since the file is not actually open, so attempting a read syscall returns -1, and `read_file` immediately returns 0 after a failed read. Both bugs can only be detected after a successful read. Since `seq-test` first calls `open_file` to get a valid file pointer, only `seq-test` was able to reach the buggy code in `read_file`. Though the workload is simple, one can generalize the experiment results to many other contexts. In UNIX, a file is simply a sequence of bytes, so any I/O resource can be modeled as a file [16], such as a database connection. Of course, one might be able to work around the limitations of Bennet and AFL++ by writing wrappers around the functions under test to initialize the necessary external state, but this adds boilerplate and developer overhead for what is testing-only code.

4.3.4 Shrinking

Workload	**Average Test Length**	**Average Calls Removed**	**%Shrunken**
AVL	16.70	10.30	61.70%
BST	10.10	7.59	75.15%
Circular Queue	21.54	15.07	69.96%
Sorted List	2.05	0.05	2.44%
File Reader	2.01	0.01	0.50%

Table 4.3: Effect of shrinking on average remaining calls per failing test.

The average test length is the average length of the sequence that caused the postcondition violation after all other sequences have been discarded. We observe fairly consistent shrinking

performance across the first three case studies, which have a mix of shallow and deep bugs leading to longer counterexamples on average. Shrinking barely affects the Sorted List and File Reader, since the bugs are fairly shallow and it is not difficult to generate the minimal counterexample before any shrinking occurs.

4.4 Acknowledgments

Zain Aamer kindly provided the extension of Etna for Bennet and AFL++ and the AVL, BST, Metro Simulation (originally Airport Simulation), Circular Queue, and Sorted List for this evaluation. His responsiveness and familiarity with CN and Fulminate while writing this thesis was indispensable. Alperen Keleş helped troubleshoot issues with Etna, and when I was unable to apply mutations due to a bug, he quickly pushed out a hotfix.

Chapter 5: Conclusion

5.1 Related work

5.1.1 Model-based generation

Quviq developed a closed-source state machine library for the Erlang version of QuickCheck out of the need to test functions with side effects [12]. It generates sequences of function calls satisfying preconditions formulated in terms of a model of the system state, also known as model-based testing. The generated test case is represented symbolically as an Erlang term rather than a function that performs the test when called. Therefore during test generation, values returned by commands are not known. To allow later generated commands to access the results of earlier commands in the same test case, the symbolic test cases bind their results to a variable, and newly generated symbolic test cases can reuse variables, which we also implemented in our tool.

PropEr is an open-source implementation of a property based testing library for Erlang that incorporates model-based testing ideas from Quviq QuickCheck [3]. The stateful portion of the library consists of two modules: `proper_statem` and `proper_fsm`. `proper_statem` is more or less a clone of Quviq QuickCheck. `proper_fsm` offers a convenient way to test systems that can be modeled by a finite collection of states and transitions between them. The state machine defined in a `proper_statem` test allows any command fulfilling preconditions

to happen at any state, while finite state machine defined in a `proper_fsm` test limits commands to those that can be called in some test state.

Keles et al. [15] developed a random testing tool integrated in TursoDB, a full rewrite of SQLite in Rust, which I contributed to as well. The tool generates sequences of statements, such as `CREATE`s or `INSERT`s, termed interaction plans, and checks the result of executing the interaction plan against a property, such as "the database should not crash" or "`INSERT`ing a row r then `SELECT`ing r should return r." Unlike the tool described in this thesis, being constrained to generating SQL statements allows us to easily define a concrete model of the state of the database as a set of key-value pairs. This model prevents the generation of illegal queries such as `SELECT` from a nonexistent table. Similar to the tool described in this thesis, the shrinker removes one statement at a time in the order of generation, checking if the error is reproduced after removal. We also ran into the issue of path dependence causing non-minimal shrinking, as removing statement s_1 then s_2 might result in a smaller set of statements than removing statement s_2 then s_1, but it is not computationally feasible to try every order. Thus, improving shrinking heuristics is a promising avenue for future research.

5.1.2 Generation-by-execution

Ernst and Pacheco [19] implemented Randoop, a feedback-directed unit test generator for Java. Randoop outputs contract-violating tests, which demonstrate scenarios where the code violates an API contract, and regression tests, which capture the behavior of the current implementation. The programmer defines a contract by declaring a class implementing a contract checking interface. An example of a contract for a set API is reflexivity of equality. To generate a unit test, a method

in the class under test is randomly selected and the arguments are filled in using previously generated calls. Sequences that exhibit contract violations are outputted as contract-violating tests, while sequences that exhibit illegal behavior (e.g. popping from an empty queue) are discarded, and sequences that exhibit neither behavior (in other words, normal behavior) are outputted as regression tests. Since only sequences exhibiting normal behavior are extended, this allows Randoop to generate much longer tests before crashing compared to undirected random generation.

Hritcu et al. [11] use QuickCheck to discover violations of noninterference in a simple stack machine with control flow extended with a secure information flow control enforcement mechanism. Formally, noninterference is defined as follows: given any two executions with starting states that are indistinguishable to a low observer and ending in halted states S_1 and S_2 (the precondition), the final states S_1 and S_2 are also indistinguishable (the postcondition). Informally, secret inputs should not affect public outputs. The encoding of the noninterference property in QuickCheck is guarded by the precondition. States are randomly generated and discarded if they do not fulfill the precondition. They incrementally add more heuristics to the generation strategy. Initially, the sequences of machine instructions are independently and uniformly generated, which leads to short sequences and poor bug-finding performance. To reduce the likelihood of popping from an empty stack, a weighted distribution favoring the push instruction was then implemented. Finally, they perform "generation by execution." Only generated instructions that do not cause the machine to crash (e.g. popping from an empty stack) are retained. The new sequence is then executed to reach a new state and the process repeats to generate more instructions. Similar to our evaluation, they find that generation by execution in conjunction with a weighted distribution over the commands is the most effective at finding

postcondition violations.

5.1.3 Leveraging CN specifications

Aamer and Pierce [1] demonstrated that the syntax of CN is amenable to deriving generators for random inputs satisfying separation logic preconditions. Their tool, Bennet, synthesizes generators from CN specifications. The generators perform backtracking and regenerate pieces of the generated input if it cannot satisfy a precondition. They formalize the semantics for a domain specific language (DSL) describing such generators and a transformation of CN specifications to the DSL. By compiling the DSL into C, they can generate test cases for CN programs and use Fulminate for runtime checking of specification violations. The evaluation portion of this thesis is largely based on the evaluation for Bennet.

5.2 Future work

Having observed `seq-test` was unable to find a number of bugs because the distribution of calls and arguments created by the scoring heuristic was not optimal for the scenario, allowing the programmer to customize the distribution based on their domain specific knowledge is a possible solution. Of course, more metrics, such as distributions of values by type, should be displayed to enable fine-tuning of the generators. For another argument generation heuristic, we could also bias for more recently generated values to increase the likelihood of inserting duplicates, for example. `seq-test` is also quite limited in the values it can generate on its own. For any pointer type, it has to rely on programmer-defined functions. However, generating pointers to values of a primitive type should not be too difficult, so extending the generator in this

way is an easy step towards increasing the usability of `seq-test`.

The high runtime relative to the number of inputs also leaves much to be desired. The main runtime bottleneck that we have control over is the number of recompilations. Rather than generating a single call per sequence per compilation, we could generate several calls at a time per sequence, determine the call that causes a precondition violation, if it exists, and discard all calls after and including that call. Another possibility is leveraging the CN specifications themselves while generating a call, instead of using Fulminate as a black box oracle for accepting and rejecting calls after generating them. Since the generator has access to the specifications as OCaml data, we could check simple preconditions like `n >= 0` when generating instead of going through the full recompilation and checking. As a first step, checking specifications on memory would likely have to be dispatched to Fulminate, since modeling C runtime state is a significant undertaking but ripe for future research. We could also develop a domain-specific language to allow the programmer to define state machines representing their code, which would be compiled to OCaml. The generator would use the state machine to determine the validity of a generated call and only recompile and use Fulminate to check as a last resort. Alternatively, we could derive the state machine from the preconditions as Bennet does for generators.

Finally, just as naive generation-by-execution can be too inefficient for complex preconditions, shrinking-by-execution can also have high runtime overhead. While we did not empirically measure the runtime of shrinking, it is theoretically possible for it to be greater than the runtime of generating the original sequence, since the phase of removing calls is run twice, and there is another phase for shrinking arguments. Attempting to derive shrinkers from predicates, similarly to how Bennet derives generators from predicates, is a possible step to reduce this shrinking overhead, and it is an interesting research problem in its own right.

Bibliography

[1] Zain K Aamer and Benjamin C. Pierce. Bennet: Randomized specification testing for heap-manipulating programs. *Proc. ACM Program. Lang.*, 9(OOPSLA2), October 2025.

[2] Stevan Andjelkovic. quickcheck-state-machine: Test monadic programs using state machine based models. `https://github.com/stevana/quickcheck-state-machine#readme`, 2025. GitHub repository.

[3] Eirini Arvaniti. Automated random model-based testing of stateful systems. Diploma thesis, National Technical University of Athens, School of Electrical and Computer Engineering, July 2011.

[4] Rini Banerjee, Kayvan Memarian, Dhruv Makwana, Christopher Pulte, Neel Krishnaswami, and Peter Sewell. Fulminate: Testing cn separation-logic specifications in c. *Proc. ACM Program. Lang.*, 9(POPL), January 2025.

[5] Jan Christiansen and Sebastian Fischer. Easycheck: test data for free. In *Proceedings of the 9th International Conference on Functional and Logic Programming*, FLOPS'08, page 322–336, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, page 268–279, New York, NY, USA, 2000. Association for Computing Machinery.

[7] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[8] Galois, Inc. VERSE-OpenSUT. `https://github.com/GaloisInc/VERSE-OpenSUT`, 2025. GitHub repository.

[9] Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. Property-based testing in practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[10] hedgehogqa. haskell-hedgehog: Release with confidence, state-of-the-art property testing for haskell. `https://github.com/hedgehogqa/haskell-hedgehog`, 2025. GitHub repository.

[11] Catalin Hritcu, Leonidas Lampropoulos, Antal Spector-Zabusky, Arthur Azevedo de Amorim, Maxime Dénès, John Hughes, Benjamin C. Pierce, and Dimitrios Vytiniotis. Testing noninterference, quickly. *CoRR*, abs/1409.0393, 2014.

[12] John Hughes. Quickcheck testing for fun and profit. In *Proceedings of the 9th International Conference on Practical Aspects of Declarative Languages*, PADL'07, page 1–32, Berlin, Heidelberg, 2007. Springer-Verlag.

[13] John Hughes. How to specify it! In William J. Bowman and Ronald Garcia, editors, *Trends in Functional Programming*, pages 58–83. Springer International Publishing, Cham, 2020.

[14] John Hughes, Rini Banerjee, and Benjamin C. Pierce. Adventures in specification-based testing, 2024. Invited talk at Isaac Newton Institute Workshop on Big Specification: Specification, Proof, and Testing at Scale.

[15] Alperen Keles, Ethan Chou, Harrison Goldstein, and Leonidas Lampropoulos. D4: Debugging databases during development. `https://alperenkeles.com/documents/d4.pdf`, 2025. Submitted paper.

[16] Brian W Kernighan. *UNIX: A History and a Memoir*. Kindle Direct Publishing Seattle, WA, 2020.

[17] Leonidas Lampropoulos and Benjamin C. Pierce. *QuickChick: Property-Based Testing in Coq*, volume 4 of *Software Foundations*. Electronic textbook, 2025. Version 1.3.3, http://softwarefoundations.cis.upenn.edu.

[18] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. Into the depths of c: elaborating the de facto standards. *SIGPLAN Not.*, 51(6):1–15, June 2016.

[19] Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, OOPSLA '07, page 815–816, New York, NY, USA, 2007. Association for Computing Machinery.

[20] Edgar Pek, Xiaokang Qiu, and P. Madhusudan. Natural proofs for data structure manipulation in c using separation logic. *SIGPLAN Not.*, 49(6):440–451, June 2014.

[21] Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. Cn: Verifying systems c code with separation-logic refinement types. *Proc. ACM Program. Lang.*, 7(POPL), January 2023.

[22] Christopher Pulte, Benjamin C. Pierce, Cole Schlesinger, and Elizabeth Austell. Cn tutorial. `https://rems-project.github.io/cn-tutorial/`, 2024. Online tutorial.

[23] J.C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

[24] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. *SIGPLAN Not.*, 43(6):159–169, June 2008.

[25] Jessica Shi, Alperen Keles, Harrison Goldstein, Benjamin C. Pierce, and Leonidas Lampropoulos. Etna: An evaluation platform for property-based testing (experience report). *Proc. ACM Program. Lang.*, 7(ICFP), August 2023.

[26] Scott Wlaschin. Choosing properties for property-based testing. `https://fsharpforfunandprofit.com/posts/property-based-testing-2/#there-and-back`, December 2014.