

# Practical Aspects of Storage Verification: A Deep Dive into SharedStore and Property-Based Testing <sup>★</sup>

Cuiyang Wang<sup>[21049031]</sup>, Felix Jing<sup>[21012858]</sup>, and Jiaqi Zhang<sup>[21009547]</sup>

University of Waterloo, Waterloo ON, Canada

c222wang@uwaterloo.ca

f2jing@uwaterloo.ca

j225zhan@uwaterloo.ca

95



**Abstract.** This project report explores the practicalities of software verification, with a focus on property-based testing, and provides a detailed analysis of SharedStore, a distributed storage system. It also illustrates storage verification, property-based testing(PBT) theory, frameworks, and shared storage in detail. We also implemented the PBT on pickleDB in Python3 with the Hypothesis library. Moreover, we show our own understanding by evaluating SharedStore’s bug detection and validation processes, discussing its limitations and potential future improvements. It integrates these insights to offer a comprehensive understanding of software verification in real-world applications, emphasizing the need for robust testing methodologies in software systems.

**Keywords:** Software Verification · Property Based Testing · Shared-Store.

## 1 Introduction

In this study, we focus on the practical aspects of software verification, an essential topic covered in the ECE653 course. Additionally, we broaden our exploration to include property-based testing, a valuable technique that isn’t covered in ECE653. We aim to explore its broader applications, understand the reasons behind its underutilization in some projects, and identify the circumstances that warrant its use. To achieve this, we have selected papers, specifically references [9] and [2]. The papers have theoretical and practical sides, we emphasize more on the usefulness and dive more into the property-based testing.

One of our key focus areas is property-based testing, an important aspect of software verification. Despite not being covered in the course slides, Professor Arie Gurfinkel said that property-based testing holds significant potential in enhancing the robustness of software systems and is worth the effort for us to study. Through an independent study, we aim to shed light on its benefits and practical applications.

---

<sup>★</sup> Supported by Professor Arie Gurfinkel

Furthermore, we will critically review the validation process of SharedStore, a distributed storage system. This will involve an in-depth analysis of their bug detection and validation efforts, early detection strategies, and the role of formal methods experts. We will also examine the limitations of their current approach and discuss potential future directions.

By integrating these elements, we hope to provide a comprehensive understanding of software verification and its practical implications, thereby enriching our knowledge and application of the concepts learned in the ECE653 course.

## 2 Storage verification

### 2.1 Introduction to storage verification

Software verification promises a better construction of critical systems, as we have learned in class as well as numerous research has been done. It can provide mathematical guarantees to the functional correctness and improve reliability of software at compile, this is what can be said after taking ECE653. In this report, we will be focusing on distributed storage. The distributed storage system can be viewed as a general instance of a system that interacts with an asynchronous environment[1/a]. In [9], the authors argue that distributed storage systems can fit naturally into the same category as a failure-prove, asynchronous environment.

### 2.2 Automated verification for large-scale software development

To make automated verification practical in large-scale software development, the important thing is to balance exploiting automation and controlling automation [9]. Ideally, one wants to know whether the code and proofs are correct or not, or decisive. This keeps the developer teams engaged and efficient. The challenge arises when the verification is undecided, which usually occurs for general-purpose verification. Then most automation tools will “time out”. Thus, the outcome takes longer cannot provide meaningful direction, and leads to productivity decreases. Authors in [9], proposed a tight verification development cycle discipline to make sure developers spend most of the time squashing time-out-prone code.

### 2.3 Assumptions

Before we discuss more about verifications, we need to recognize a few assumptions we have been making. I think these assumptions are useful to isolate the system from more dramatic and unpredictable factors that could lead to system failure and also make our discussion more complete and logical. Firstly, users must trust the top-level application-facing specification of the contract produced by the system. We assume the high-level design is made correctly, and there shouldn't be any design specification flaws, as the verification goal is to verify and guarantee the build system matches the given specification.

Secondly, we assume the interface to the outside environment (i.e., the rest of the world) and how they behave. In other words, if the downstream or any hardware could affect the input and output, it is not the focus of the software verification.

Lastly, we assume the correctness of the verification tools (i.e., Dafny).

## 2.4 Verifying storage system

In [figure1](#), the authors showed an approach that can be applied to multi-node, multi-disk storage systems and we will dive into more detail. There are 3 state machines: program statement, environment, and IO system. An environment contains a disk in which crashes may occur. This represents the Storage IO System.

Since the focus of our project is the practical useful side not on VeriBetrKV itself, we won't go into detail about the framework, but the details can be found on [\[9\]](#) and we believe it is a good material.

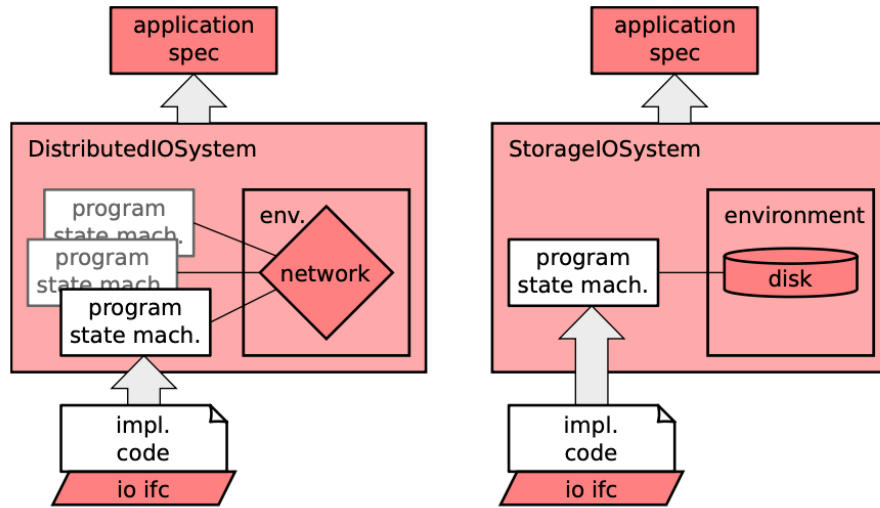


Fig. 1. Simplified Storage IO System

## 2.5 Disciplined automation


The time cost introduced after using a verification major comes from the amount of typing annotation required and the amount of time the verifier takes to do its proof work.

Some of the functional verification can take a long time or even be undecidable. Users and developers have introduced time limits [\[16\]](#) and manual

guidance. In interactive theorem prover, such as [14], software developers need to tell the next step to take.

From assignment 3 and in the lecture examples, we have reasonable ground to say for a large system to be verified, there must be a large number of definitions, such as invariants and state-machines transition relations. Thus, the need for automated verification is much needed, otherwise, developers would spend a significant amount of time in verification rather than development, which is not helpful to productivity. On the other side, allowing the verifier to expose all the definitions, gives the theorem prover a large search space [9]. This can take a long time to explore. The balance between exploiting automation and controlling it can contribute significantly to a framework's success. If the automation is not strong enough, developers need to invest much time tediously typing the missing proof guidance. However, if the automation is too aggressive, the developer needs to wait longer to get the reply back. To have the system in the perfect balance is very difficult, in our opinion, it is good to let the system be a bit stronger than it should be, although it does take a longer time, the developers can use this time to do the development and other work while waiting.

As the software system grows larger, the possibility of time out increases considerably. From [9], the practice they have introduced is whenever there is a 20-second response time, developers have been asked to resolve the timeout. This response time reflects how much information the prover has been trying to prove or verify. By monitoring the response time, their team essentially controls the amount of information passed into the prover. As with the different computing power, the response time should be adjusted, rather fixated on 20s.

There is also another technique that developers can intentionally hide certain definitions and only expose them when needed. Take Dafny as an example, `opaque`, is used to hide the definitions whereas, `reveal`, is used to explicitly reveal the definitions. 


### 3 Property-based testing theory

Now we transitioning the topic into testing. While testing is essential for creating robust software, many developers often find it repetitive and dull. However, the development of property-based testing is emerging as a unique method for simpler and more effective testing.

Unlike fuzzing, which is a popular testing technique focused primarily on discovering unexpected vulnerabilities in software, property-based testing is more concerned with verifying the properties of the software.

In contrast to Test-Driven Development (TDD), which aims to shape code according to specific expectations and requirements, property-based testing encourages a deeper investigation into a program's behavior, exploring both its capabilities and limitations.

Property-based testing is an approach designed to meet the security and robustness requirements of programs. As a systematic testing method, it highlights the critical properties of the software using high-level specifications. In contrast,

trivial or partial properties are often overlooked to ensure the efficiency of the testing process. For example, good practice in UNIX programs using property-based testing involves addressing privileged and daemon processes that cross-protection domains [7]. These adopt a slightly different form of specification that not only verifies the expected behavior of the software but also identifies generic security flaws. 

### 3.1 Advantages v.s. Disadvantages

**Advantages** A property-based testing (PBT) system simplifies the process by automating tasks that would normally be manually intensive. It generates a variety of input scenarios and applies them to specific properties, determining whether they succeed or fail. When a test fails, the framework investigates to pinpoint the cause of the failure. For instance, in a financial application, it might simulate different monetary amounts in transactions to ensure the correctness of the outcomes in every case.

Another advantage of PBT is that it requires a more formal reasoning about the tested systems than traditional testing methods. The formulation of specifications, in turn, aids in documenting and building hierarchical models of the system's behavior [1].

In practice, PBT libraries take property-based test cases and execute them repeatedly in an attempt to find a counterexample to the specification. Inputs are automatically selected from a large corpus, which may be dynamically generated and possibly selected at random.

**Disadvantages** Property-based tests inherently take longer than unit tests, which is intentional. In property-based testing, each property undergoes numerous tests as opposed to just one instance in unit testing.

Crafting effective properties is a challenging task. While writing a unit test for a specific scenario might be straightforward, developing a property that encompasses the same scenario broadly, allowing for testing with numerous instances, is significantly more complex.

Determining the sufficient number of properties is difficult: Similar to the challenge in unit testing, where ensuring complete coverage of a module, component, or system's functionality is often uncertain, ascertaining the adequacy of properties in property-based testing is equally challenging. However, caution is advised against overloading with properties, as each one adds to the total testing time, potentially leading to excessively long test suites.

### 3.2 Property

Properties are broad principles that define how a program behaves. A single property should logically apply to various inputs and outputs of a specific code segment. In contrast, conventional test suites consist of numerous specific instances which, ideally, represent the overall desired behavior of the program.

Properties operate at a more abstract level, outlining the expected results of an operation in general terms.

The effectiveness of these tests depends heavily on the test creator’s skill in considering various scenarios, influenced by their experience, meticulousness, and comprehensive understanding. Their reliability is as variable as the unknown elements they may encompass. The essence of these tests lies in demonstrating the existence, rather than the absence, of defects, as noted by E.W. Dijkstra [4].

In property-based testing, the goal is to establish a rule like ‘for any given list, the function should return the largest number possible.’ Subsequently, numerous lists are tested against this rule. The more inventive the list creation, the more robust the test is expected to be. The premise is that such dynamic probing can unearth ‘unknown unknowns.’ Although the test primarily reveals bugs rather than their absence, it aims to uncover a wider range of issues.

A challenge arises in cases like the one where the property—to return the largest number from a list—mirrors the code’s intended function. This overlap is a common issue in applying property-based testing for unit tests; the property is so straightforward that formulating it as a distinct, general rule separate from the existing code becomes challenging.

**Alternate wording of properties** Modeling is generally effective, but it comes with the limitation that it might involve rewriting the same program several times, including a version so straightforward that its accuracy is self-evident. However, this isn’t always feasible or practical. In such cases, turning to property-based testing is a viable alternative.

The initial challenge in this approach is identifying superior properties, which is often a more complex task than simply pinpointing any property. This complexity necessitates a deep understanding of the problem at hand. A useful method for finding a property is to rephrase the intended function of the code segment.

However, there comes a point when the functions under test are so fundamental that devising properties for them becomes exceedingly difficult. These functions might be considered axiomatic, meaning they are inherently trusted to be accurate and serve as the system’s basic building blocks. In some situations, supplementing property-based testing with traditional unit testing is a perfectly valid approach. It’s important to remember that property-based testing is not a complete substitute for all forms of testing, but rather an additional tool that can often enhance testing outcomes.

**Symmetric properties** Another category of properties to consider is symmetric properties. These are particularly effective when dealing with two code segments that perform inverse functions, such as an encoder and a decoder. The key is to test these functions in tandem. If one function reverses the action of the other, then executing both functions in sequence should result in the original input becoming the final output. It’s worth noting that these operation sequences can extend beyond just a pair of functions.

**Stateless Properties** Stateless properties form the core element of all property-based testing tools and are universally supported by various versions of QuickCheck derivatives found in practical applications [13].

Gaining knowledge and understanding of these properties provides the essential basics needed for utilizing various property-based testing frameworks. More crucially, understanding stateless properties is the initial step in shifting one's mindset from thinking in terms of specific examples to thinking in terms of properties.

### 3.3 Modeling

Instead of relying solely on conventional testing methods, employing modeling can be highly effective. This process involves the developer creating a basic and straightforward model, which may be algorithmically less efficient, to validate the property. The model should be so uncomplicated that its correctness is apparent. The actual code implementation, typically aimed at efficiency, is then compared against this model. If both the model and the actual implementation exhibit consistent behavior, it's likely they are functionally equivalent.

Modeling proves especially beneficial in developing data structures. Most data structures are advanced versions of simpler, less efficient variants designed for specific use cases. For example, dictionaries or maps can often be equated to basic lists of keys and values. This comparison can extend to more complex operations common in trees and heaps, such as identifying the next smallest element. This method facilitates the gradual development of a range of increasingly complex data structures while keeping their test cases straightforward.

Modeling is also advantageous in integration testing of state-heavy systems with numerous side-effects or dependencies. These systems may operate intricately, but the user's experience remains relatively simple. Many advanced systems, like databases, execute complex tasks to maintain transaction integrity, prevent data loss, and optimize performance, yet their operations can often be simulated using simple in-memory data structures.

Finally, there is a unique and valuable modeling approach known as 'the oracle.' An oracle is essentially a reference implementation, possibly derived from another programming language or software system, serving as a ready-made model. The developer's responsibility is to ensure their implementation aligns with the oracle, guaranteeing identical outcomes.

### 3.4 Generators

Generators in property-based testing (PBT) are core tools that create diverse sets of input data for tests. In PBT, instead of writing specific test cases with fixed inputs, developers define properties that the code should always satisfy. Generators then automatically produce a wide variety of inputs to test these properties. They can generate simple data like numbers and strings or more complex structures like lists and custom objects. The main idea is that these

generators produce inputs that might not be anticipated by a developer, uncovering edge cases and potential bugs that traditional example-based testing might miss. This makes PBT a powerful method for thorough software testing.

Frameworks for property-based testing typically include a selection of basic generators as well as the capability to create custom generators.

Simple stateless properties, utilizing only the fundamental data types available in a property-based testing framework, can address a broad range of scenarios. However, as most programs eventually incorporate more complex data types, the creation of custom generators becomes essential. This section will introduce techniques for crafting custom generators and discuss how to assess their effectiveness and relevance for specific test scenarios.

Research papers referencing QuickCheck typically adopt its unique approach of generating test cases randomly. However, various other methods have also been explored. Notably, enumerative Property-Based Testing (PBT) has gained popularity within the functional programming community, as evidenced by works like those of Braquehais [12] and Runciman et al [15]. Additionally, there's growing interest in feedback-based PBT, as seen in studies by Lampropoulos et al. [10] and Sagonas [11]. Each of these methodologies offers its own set of advantages and compromises. The choice between these different approaches can significantly impact the efficacy of the testing process.

### 3.5 Shrinking

Shrinking is a process used in property-based testing frameworks to simplify failure cases to the point where the minimal scenario causing the failure can be identified.

Often, the input that triggers a failure can be complex or large. Discovering this failing input might take numerous tries and include a lot of extraneous information. The modern property-based testing frameworks addresses this by 'shrinking' the data set. This is typically done by modifying the generators used in the test, aiming to reduce them to their most basic form.

Essentially, containers tend to reduce their contents, and other values aim to reach a more neutral state. A beneficial feature of this process is that when custom generators are created using other generators, they naturally 'inherit' this shrinking ability, potentially developing their own automatic shrinking process.

However, some data types lack a clear 'zero point.' For example, a vector with a fixed length or a tuple will retain their dimensions, and larger, user-defined recursive data structures may not have straightforward ways to shrink. This is particularly true in statically-typed languages influenced by Haskell's implementation, where actual type declarations serve as generators, unlike the function-based approach. In many statically-typed languages, where the foundational property-based testing framework may lack robust built-in options for generators, the default shrinking rules can be less effective, especially when the type system doesn't fully encapsulate the constraints of the actual data type.



In such cases, any information not encoded in the type system (like an ordered collection, a balanced tree, or a case-insensitive UUID represented as a string) must be explicitly defined in the shrinking rules.

In frameworks like PropEr and QuickCheck, the necessity to specify shrinking rules is reduced but not eliminated. Here, shrinking is more a strategic tool used to guide the testing framework in effectively minimizing the data set size while maintaining its relevance after a failure is detected.

### 3.6 Stateful Properties

Stateful property tests are highly effective in scenarios where the user’s perspective of what the code should accomplish is straightforward, but the internal implementation of the code is complex. This approach is particularly apt for a wide range of integration and system tests. In such contexts, stateful property-based testing emerges as a valuable tool in a developer’s toolkit, enabling extensive system testing with minimal code.

These tests represent an informal variant of model checking and are developed as a more advanced layer built upon stateless property-based tests. In QuickCheck-like frameworks, the essence of stateful testing lies in creating a mostly deterministic model of the system, generating a sequence of commands to mimic the operational flow, and then executing these commands on the actual system being tested. During this process, the system’s real-time responses are continually compared with the model to ensure they align with expected outcomes. A discrepancy between the model and the actual system results in a test failure.

**Components** There are three major components in stateful tests: the model, the creation of commands to simulate the execution process, and the verification of the system in comparison to the model.

The model itself consists of two aspects. The first aspect is the data structure that mirrors the anticipated state of the system. Secondly, the aspect involves a set of functions that alter this model state, reflecting potential changes in the system after certain operations are performed.

The process of generating commands is also twofold. It involves a collection of potential symbolic calls, each with generators to determine their parameters, and functions that assess whether a specific symbolic call is applicable, based on the current state of the model.

An interesting part of this process involves ‘preconditions,’ which are functions determining if a call is appropriate at a certain point in the test. These preconditions establish necessary conditions that must be met in the system for the current test scenario. For instance, an ATM might only allow deposits when a valid debit card is inserted, making the presence of a debit card a precondition for deposit operations.

The final component is the comparison of the actual system against the model. This is done using ‘postconditions,’ which are checks ensuring that certain

invariants are maintained after an operation. For example, if the model indicates a user has inserted a card but entered an incorrect password, the postcondition would confirm that the system correctly reports a login failure. This verification can involve checking overarching invariants expected to always hold or comparing the system’s actual output against the expected result based on the model state.

To summarize, stateful testing involves the following components: A model with a defined state and functions for modifying this state, generation of commands through symbolic calls, which are screened by precondition functions, and validation of the system’s actions against the model using postcondition functions.

### 3.7 Recap

To encapsulate, Property-Based Testing (PBT) is a comprehensive approach in software testing that is superior to traditional methods like fuzzing and TDD in exploring software behavior. Its main strengths lie in automating testing processes, generating diverse input scenarios, and uncovering complex bugs. PBT emphasizes verifying broader software properties, requiring a deep understanding to develop effective properties and modeling. Key elements of PBT include the use of generators for input data, shrinking for simplifying failing test cases, and stateful tests for complex systems. While it can be more time-consuming and complex than other methods, PBT’s thoroughness and formal reasoning framework significantly enhance software quality and reliability.

## 4 Property-based Testing Frameworks

### 4.1 Hypothesis

**Overview** At its core, Hypothesis [18] is a Python library designed to create unit tests that are simpler to write and more powerful in execution. It enables tests to assert truths across a broader range of cases, not limited to preconceived scenarios. This approach, often termed property-based testing, was popularized by the Haskell library QuickCheck [3]. Hypothesis operates by generating arbitrary data that fits a specified criterion and then evaluates if certain guarantees or properties hold true under these varied conditions. This method shifts the paradigm from a traditional three-step unit test (setup, operation, assertion) to a more dynamic approach where for all data meeting a specification, operations are performed, and assertions about the results are made.

**Implementation** Hypothesis operates by allowing users to write tests based on properties, which are guarantees that the code should always uphold, regardless of the input. These properties could range from ensuring no exceptions are thrown, to maintaining visibility constraints after deletion, to verifying that serialization and deserialization yield the same value. The process generally involves three steps:

- Define data following a specific specification.
- Perform operations on this data.
- Assert something about the result.

In practical terms, Hypothesis employs strategies to generate test cases. For instance, using the `given` function along with the `strategies` module, it can generate a range of arguments for testing. This methodology is highly flexible, enabling the definition of valid input ranges or types and the exclusion of specific cases or values using functions like `assume()`.

By contrast, a typical unit test would involve setting up specific data, performing operations, and asserting results for that particular data set. Hypothesis expands this by considering a wide array of data sets that fit a given specification, thereby uncovering cases that might otherwise be overlooked.

**Advanced Features** Hypothesis stands out due to several advanced features:

**Shrinking:** Hypothesis not only identifies failures but also utilizes a feature known as shrinking. This refers to the process of reducing a failing example to a minimal form. It involves strategies to simplify the byte array involved in the test, adhering to rules such as shorter arrays being simpler and prioritizing lexicographically earlier arrays. This feature reduces a failing test case to its simplest form, making it easier to understand and debug. This process is essential for distilling complex test cases into manageable examples.

**Strategy Design and Data Generation:** Hypothesis allows for careful design of strategies and data generation, accommodating a variety of data types and structures. This flexibility is crucial for creating effective tests for complex data types. Each time the test runs, Hypothesis may generate different inputs, continually challenging the function with new scenarios. This continuous and varied testing approach leads to a more robust and reliable implementation over time.

**Example Database and Consistency:** Hypothesis maintains a database of examples, particularly failing ones, to ensure consistent test results over time. This database helps in quickly identifying regressions and persistent issues.

**A Simple Example** Suppose we have a Python function intended to calculate the square of a number but contains a mistake:

```
def faulty_square(x):
    return x * x if x > 0 else -x * x
```

In this function, the intention might be to return the square of 'x', but it incorrectly returns the negative of the square for non-positive values.

We write a Hypothesis test to check a basic property of squaring: the result should always be non-negative. Here's how it might look:

```
from hypothesis import given
from hypothesis.strategies import integers
@given(integers())
```

```
def test_square_non_negative(x):
    result = faulty_square(x)
    assert result >= 0
```

In this test: `@given` is a decorator provided by Hypothesis. It tells the testing framework to generate inputs for the test. Hypothesis generates a broad range of input values automatically. In this scenario, it includes positive numbers, negative numbers, and zero. `integers()` is a strategy in Hypothesis for generating integers.

When this test is run, Hypothesis generates a range of integers, including negative numbers and zero. When it tests any negative number, the assertion fails because the function incorrectly returns a negative result. Hypothesis then reports this failure, typically indicating the smallest (or simplest) example that led to the test failure. For instance, it might output:

```
Falsifying example: test_square_non_negative(x=-1)
AssertionError: assert -1 >= 0
```

This output shows that the test failed when `x` was `-1`, indicating the point where the function's behavior deviates from the expected property. Hypothesis provides specific examples of inputs that cause the test to fail, which is a valuable aid in debugging. The clarity and specificity of the error reports make it easier to understand the nature of the problem and to fix it.

**Stateful Testing** Stateful testing in Hypothesis is a powerful extension of its standard property-based testing approach. It is particularly suited for testing systems where the state changes over time, such as databases, user interfaces, or any other system where operations have different effects depending on the history of interactions with the system. The key concepts of stateful testing in Hypothesis are:

- **Modeling System State:** In stateful testing, we can create a model of the system's state. This model is a simplified representation of the system's actual state, allowing Hypothesis to understand and track the changes that occur during testing.
- **Operations as Methods:** The test defines a set of operations that can be performed on the system. These operations are represented as methods in a class in Python. Each method modifies the state or asserts properties about the state.
- **Automatic Sequence Generation:** Hypothesis automatically generates sequences of operations to test the system. It tries different combinations and orders of operations to explore how the system behaves under various sequences of events.
- **State Verification:** After executing a sequence of operations, Hypothesis checks if the final state of the system is consistent with what would be expected based on the model. This helps in identifying bugs related to state changes or interactions between different parts of the system.

- Finding Minimal Failing Sequence: When a bug is found, Hypothesis attempts to minimize the sequence of operations that leads to the failure, making it easier to understand and fix the issue.

By automatically generating and minimizing operation sequences that lead to failures, Stateful testing helps uncover subtle bugs that might not be evident in traditional testing approaches.

**Stateful Testing Example** Let's consider a simple stack implementation in Python and a stateful test designed to find bugs in its behavior. Suppose we have a basic stack implementation:

```
class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if self.items:
            return self.items.pop()
        # Not raise IndexError("pop from empty stack")

    def size(self):
        return len(self.items)
```

We can use Hypothesis to write a stateful test for this stack. The test will simulate a series of push and pop operations and check if the stack behaves as expected.

```
from hypothesis import strategies as st
from hypothesis.stateful import RuleBasedStateMachine, rule

class StackStateMachine(RuleBasedStateMachine):
    def __init__(self):
        super().__init__()
        self.stack = Stack()
        self.modelStack = []

    @rule(item=st.integers())
    def push(self, item):
        self.stack.push(item)
        self.modelStack.append(item)

    @rule()
```

```

def pop(self):
    if self.modelStack:
        expected_item = self.modelStack.pop()
        assert self.stack.pop() == expected_item
    else:
        with self.assertRaises(IndexError):
            self.stack.pop()

```

```
TestStack = StackStateMachine.TestCase
```

Let's say there's a bug in the `pop` method: it does not raise an `IndexError` when popping from an empty stack. When the Hypothesis test runs, it might generate a sequence of operations that includes popping from an empty stack, which would lead to a failed assertion. Then it outputs:

```

Falsifying example:
test_run_state_machine(factory=TestStack, data=data(...))
state = TestStack()
state.pop() # Expected IndexError, but no exception was raised

```

The test simulates real-world usage of the stack by pushing and popping elements, mirroring how the stack would be used in an application. The `modelStack` acts as a simplified model of the stack's expected behavior. Each operation on the real stack is mirrored on this model, ensuring that the stack's state aligns with expectations. Hypothesis explores various sequences of push and pop operations, some of which might not be immediately obvious or intuitive to a developer writing manual tests. The test uncovers the bug where the `pop` method does not correctly handle popping from an empty stack, a critical edge case in stack implementations. Hypothesis provides a minimal sequence of operations that leads to the bug, aiding in quick identification and resolution of the issue.

**Recap** Hypothesis is a powerful and flexible Python library that has significantly contributed to the domain of property-based testing. It simplifies the creation of unit tests and enhances their capability to uncover hidden issues in the code. We provided a comprehensive case where conducting property-based testing and stateful testing with Hypothesis on a file-based database management system in Chapter 6. Source codes can be found in our UWaterloo Gitlab group repository.

The benefits of using Hypothesis for property-based testing are significant. It generates many more test cases than a developer might typically write, covering edge cases that might not have been previously considered. The library is particularly effective in handling complex data types, such as floating-point numbers and Unicode text, where traditional testing methods might fall short. Additionally, the act of defining properties encourages a deeper analysis and understanding of the problem, potentially leading to more robust and efficient solutions. By focusing on testing general properties rather than specific cases,

Hypothesis encourages a more thorough and insightful approach to testing, leading to more robust and reliable software.

## 4.2 QuickCheck

**Overview** QuickCheck in Rust [8] is an influential and widely-used library in the Rust programming language ecosystem, designed for automated property-based testing with a focus on randomly generated input. It is a port of Haskell’s original QuickCheck [3]. The Rust implementation of QuickCheck, maintained by BurntSushi on GitHub, carries forward the philosophy and functionalities of its Haskell counterpart, adapting them to the Rust programming environment. All QuickCheck needs is a property function—it will then randomly generate inputs to that function and call the property for each set of inputs. If the property fails (whether by a runtime error like index out-of-bounds or by not satisfying your property), the inputs are “shrunk” to find a smaller counter-example. The shrinking strategies for lists and numbers use a binary search to cover the input space quickly.

**Core Features** QuickCheck in Rust allows for the generation and shrinking of various data types like integers, floats, tuples, booleans, lists, strings, options, and results. The library is accessible via `crates.io` and can be included as either a direct dependency or as a development dependency for test code. Additionally, if the `#[quickcheck]` attribute is to be used, the `quickcheck_macros` should also be added.

A typical use case involves defining a property function that QuickCheck will test against a variety of inputs. For example, a double reversal function could be tested to ensure that reversing a vector twice returns the original vector. This function is then annotated with `#[quickcheck]`, allowing QuickCheck to automatically test it with numerous randomly generated vectors.

QuickCheck can be combined with other crates like ‘fake’ for generating specific types of data, such as valid email addresses. This enhances the flexibility and applicability of property-based testing in more complex scenarios.

**Advanced Features** QuickCheck in Rust also provides a suite of advanced features that significantly enhance the effectiveness and depth of property-based testing:

- **Arbitrary Trait:** The `Arbitrary` trait is pivotal in QuickCheck’s functionality. It defines how random data is generated and shrunk for different types. Implementing this trait for custom types allows for more controlled and meaningful property testing.
- **Shrinking:** Shrinking is an essential feature of QuickCheck. When a test fails, QuickCheck attempts to simplify the failing input to the smallest example that still causes the test to fail, aiding in debugging and understanding the nature of the failure.

- Discarding Tests: QuickCheck also allows for discarding certain test results, particularly useful when testing properties that only hold for a subset of inputs. This is achieved through the `Testable` trait and returning a `TestResult` directly.

**A Simple Example** First, We create a simple ‘reverse’ function in Rust. This function takes a vector of integers and is intended to return the vector in reverse order:

```
fn reverse(vec: Vec<i32>) -> Vec<i32> {
    let mut rev = vec![];
    for &item in vec.iter() {
        rev.push(item); // Intentional bug: items are being added in the original order
    }
    rev
}
```

Notice the bug in this function: instead of inserting elements in reverse order, it’s just copying the original vector.

Then, we write a QuickCheck test to validate our `reverse` function. The test will check whether reversing a vector twice gives us back the original vector:

```
#[cfg(test)]
mod tests {
    use super::*;
    use quickcheck::quickcheck;
    quickcheck! {
        fn double_reversal_is_identity(xs: Vec<i32>) -> bool {
            xs == reverse(&reverse(&xs))
        }
    }
}
```

In this test, `quickcheck!` macro is used to automatically generate various inputs (`xs: Vec<i32>`) and check if the property `double_reversal_is_identity` holds true.

When the QuickCheck test runs, it will generate numerous random vectors and apply them to the `double_reversal_is_identity` function. QuickCheck will quickly find that our `reverse` function doesn’t actually reverse the vectors because the property we’re testing (reversing a vector twice returns the original vector) will fail.

For instance, if QuickCheck generates a vector like `[1, 2, 3]`, the test will fail because `reverse(reverse([1, 2, 3]))` will not equal `[1, 2, 3]` due to our flawed implementation of `reverse`. It will output:

```
FAILED: double_reversal_is_identity([1, 2, 3])
```



Once a failing test case is found, QuickCheck tries to **shrink** the input to the simplest form that still causes the test to fail. This makes it easier to understand and fix the bug. In this specific example, this could potentially lead to an even simpler case being reported, such as `[1, 2]`.

**Recap** QuickCheck for Rust is a powerful tool for property-based testing, bringing the advantages of randomized testing to the Rust ecosystem. While it has some limitations and requires consideration of the type of data and properties being tested, its ease of use and the depth of testing it enables make it a valuable asset in a Rust developer’s toolkit.

### 4.3 Proptest

**Overview** Proptest [20] is another property-based testing framework for the Rust programming language, inspired by the Haskell library QuickCheck [3] and its Rust counterpart. Hosted on GitHub and available through `crates.io`, Proptest is designed to create randomly generated input and test properties of functions and algorithms. Unlike QuickCheck, generation and shrinking is defined on a per-value basis instead of per-type, which makes it more flexible and simplifies composition.

**Features** Proptest’s core functionality lies in its ability to generate a wide variety of data types for testing, including integers, floats, strings, collections, and user-defined types. It allows for more intricate configuration of input generation than QuickCheck, which is particularly useful in testing scenarios where certain classes of inputs are more relevant.

The library is easily integrable into Rust projects and supports the creation of complex property tests with minimal boilerplate code. Functions under test are annotated with Proptest attributes, enabling the framework to automatically apply a range of inputs.

Proptest distinguishes itself with several advanced features:

- **Configurable Test Strategies:** Users can define detailed strategies for data generation, giving them precise control over the input space of their tests.
- **Custom Type Support:** Like QuickCheck’s ‘Arbitrary’ trait, Proptest allows users to define custom strategies for their types, enabling the framework to generate meaningful test data for user-defined structures.
- **Complex Test Scenarios:** Proptest excels in creating complex test scenarios, such as testing functions that operate on interrelated data structures or require inputs with specific properties.
- **Error Tracing and Shrinking:** When a test fails, Proptest attempts to minimize the failing case to a simpler form, aiding in quicker debugging. This process is more configurable compared to QuickCheck.

**A Simple Example** We define a function `concat_strings` in Rust that takes two `String` references and returns a new `String` which is the concatenation of the two:

```
fn concat_strings(a: &String, b: &String) -> String {
    if a.is_empty() {
        b.clone()
    } else {
        a.clone() // Bug: It should return a + b, but returns a instead
    }
}
```

We then write a Proptest test to check if `concat_strings` correctly concatenates two strings:

```
#[cfg(test)]
mod tests {
    use super::*;
    use proptest::prelude::*;
    proptest! {
        #[test]
        fn test_concat_strings(a: String, b: String) {
            let result = concat_strings(&a, &b);
            prop_assert_eq!(result, format!("{}", a, b));
        }
    }
}
```

When we run this test, Proptest will generate various pairs of strings and test them against our `concat_strings` function. Eventually, it will find a case where our function fails. The output might look something like this:

```
FAILED: test_concat_strings("abc", "def")
thread 'tests::test_concat_strings' panicked at 'assertion failed:
'(left == right)'
  left: "abc",
  right: "abcdef"', src/lib.rs:12:13
```

The test fails for the input strings "abc" and "def". Proptest successfully identifies that `concat_strings("abc", "def")` does not return "abcdef" as expected. The bug is in the logic of `concat_strings`. Instead of concatenating `a` and `b`, it returns `a` when `a` is not empty. This is why the test fails. Proptest's strength is showcased here. It generates a wide range of inputs, including edge cases that might not be immediately obvious during manual testing. If the initial failing case was more complex, Proptest would attempt to shrink the input to the simplest form that still causes the test to fail, aiding in pinpointing the exact nature of the bug.

**Recap** Proptest for Rust offers a powerful and flexible approach to property-based testing, particularly suited for scenarios where detailed control over test case generation is needed. Its ability to handle complex test scenarios and provide meaningful feedback through error shrinking makes it a valuable asset for Rust developers seeking comprehensive testing solutions.

#### 4.4 Comparison of Frameworks

In the area of property-based testing, various frameworks offer distinct features and capabilities tailored to different programming languages and testing needs. As we can see Table 1, Hypothesis, designed for Python, is good for its strategy-based test case generation and ease of integration with Python testing tools like pytest, making it ideal for robust edge case testing. QuickCheck in Rust, originating from Haskell, emphasizes random data generation in line with Rust’s functional programming aspects, though it offers less customization than its counterparts. Proptest, another Rust-based framework, extends the concepts of QuickCheck with more control over test case generation and shrinking, catering to Rust projects that demand detailed control over testing. On the JavaScript front, Fast-check [5] provides a balance between automatic test generation and user control, integrating seamlessly with popular JavaScript testing frameworks and being particularly suited for projects needing quick setup for property-based testing. Each of these frameworks excels in its environment, with Hypothesis and Fast-check standing out for ease of use, while QuickCheck and Proptest offer deeper integration within the Rust ecosystem, each bringing unique strengths to property-based testing.

**Table 1.** Property-based Testing Frameworks Comparison

Features	Hypothesis	QuickCheck	Proptest
<b>Language Support</b>	Python	Rust	Rust
<b>Testing Approach</b>	Strategy-based	Random generation	Controlled gen.
<b>Ease of Use</b>	High	Medium	Medium-High
<b>Customization</b>	High	Medium	High
<b>Community Support</b>	Strong	Good	Growing
<b>Tool Integration</b>	Excellent	Good	Excellent
<b>Ideal Use Case</b>	Edge case testing	Functional tests	Detailed control

## 5 Shared storage

### 5.1 Crash Consistency in SharedStore

SharedStore uses a protocol based on soft updates to handle crash consistency, a common issue in storage systems [2]. It defines two properties: persistence and

forward progress. Persistence ensures operations saved before a crash are readable after unless replaced by a later saved operation. Forward progress ensures that after a shutdown without a crash, every operation’s dependency shows it is persistent.

To verify these properties, authors in [2] used property-based testing to generate and check crash states. In the previous part, we conducted a detailed study on property-based testing. They add a DirtyReboot operation to the operation set [2]. This operation’s parameter determines which data in volatile memory is saved to disk during a crash. After a reboot and recovery, the test checks each operation’s dependencies against the two properties. If the check fails, there’s a crash consistency bug.

This approach generates crash states at a broad level. To increase the chance of finding bugs, the operation set includes flush operations for each component that can be mixed with other operations. They have also implemented a version of DirtyReboot that checks crash states at the block level, similar to BOB and CrashMonkey [2]. However, this exhaustive approach hasn’t found additional bugs and is much slower to test, so they don’t use it by default [2].

ID	Component	Description
<i>Functional Correctness</i>		
#1	Chunk store	Off-by-one error in reclamation for chunks of size close to PAGE_SIZE
#2	Buffer cache	Cache was not correctly drained after resetting an extent
#3	Index	Metadata was not flushed correctly during shutdown if an extent was reset
#4	API	Shards could be lost if a disk was removed from service and then later returned
#5	Chunk store	Reclamation could forget chunks after a transient read IO error
<i>Crash Consistency</i>		
#6	Superblock	Superblock Dependency for extent ownership was incorrect after a reboot
#7	Superblock	Mismatch between soft and hard write pointers in a crash after an extent reset
#8	Buffer cache	Writes did not include a dependency on the soft write pointer update
#9	Chunk store	Reference model was not updated correctly after a crash during reclamation
#10	Chunk store	Reclamation could forget chunks after a crash and UUID collision
<i>Concurrency</i>		
#11	Chunk store	Chunk locators could become invalid after a race between write and flush
#12	Superblock	Buffer pool exhaustion could cause threads waiting for a superblock update to deadlock
#13	API	Race between control plane operations for listing and removal of shards
#14	Index	Race between reclamation and LSM compaction could lose recent index entries
#15	Chunk store	Reference model could re-use chunk locators, which other code assumed were unique
#16	API	Race between control plane bulk operations for creating and removing shards

**Fig. 2.** SharedStore issues prevented from reaching production by validation.

## 5.2 Crash Consistency example: Issue 10

Next, we examine an example to check crash consistency. In the above figure, Issue #10 was a crash consistency bug related to chunk reclamation. SharedStore automated testing allowed the developer to discover this before code review [2].

The problem stemmed from how chunk data is serialized. Each chunk has a two-byte magic header and a random UUID for validation. The issue occurred when a chunk’s UUID spilled onto a second page.

A crash caused data loss on page 1, corrupting the chunk. However, there was no consistency issue as the chunk wasn’t considered persistent. After recovery, a second chunk was written to the same extent and flushed to disk, making it persistent. During reclamation, the first chunk was decoded successfully due to a UUID and magic byte collision, causing the second chunk to be skipped and become inaccessible after reclamation. This resulted in a consistency violation as the second chunk was considered persistent but was lost after reclamation.

This subtle issue involved a specific UUID choice, a chunk size that spilled onto a second page, and a crash that only lost the second page. Despite this, our checks automatically discovered and minimized this test case.

### 5.3 Check concurrent execution

The validation approach used in SharedStore primarily deals with sequential correctness, focusing on deterministic single-threaded executions [2]. However, in a real-world scenario, any distributed storage system like SharedStore operates with high concurrency, handling multiple requests and tasks simultaneously. While Rust’s type system ensures freedom from data races, it doesn’t provide guarantees against higher-level race conditions. This is challenging to test and debug due to their non-deterministic nature.

To address this, the SharedStore team wrote harnesses for key properties and validated them using stateless model checking, a method that explores concurrent program interleavings [2]. This approach is used to check both concurrent executions of the storage system and validate SharedStore-specific concurrency primitives. Stateless model checkers can validate concurrency properties and test for deadlocks.

Ideally, the team aims to check that concurrent SharedStore executions are linearizable with respect to the sequential reference models. They use the Loom stateless model checker for Rust to check fine-grained concurrency properties [2]. However, due to scalability issues with Loom, they developed the stateless model checker Shuttle for larger tests. Loom is used to check soundness on all interleavings of small, critical code, and Shuttle is used to randomly check interleavings of larger tests.

An example of a Loom test harness for SharedStore’s LSM-tree-based index is provided. The test initializes the index’s state, spawns three concurrent threads, and checks read-after-write consistency for a fixed read/write history.

A race condition in the LSM tree implementation, identified as Issue 14 [2], was caught by the Loom test harness before code review. The LSM tree implementation uses an in-memory metadata object to track the set of chunks storing LSM tree data on disk. This metadata is mutated by two concurrent background tasks.

From our understanding, the SharedStore team is making significant efforts to ensure the crash consistency and concurrency of their storage system. They

are utilizing advanced techniques and tools like Loom and Shuttle to validate and test their system. Despite the challenges posed by concurrency and non-deterministic executions, they have been successful in identifying and resolving complex issues, demonstrating the effectiveness of their approach.

#### 5.4 2 additional classes of issues were identified in the development of SharedStore

Now we are going to dive into two additional classes of issues identified in the development of SharedStore.

Firstly, let's discuss undefined behavior. SharedStore was developed in Rust [2], a low language known for its memory safety guarantees and gaining popularity over C/C++. However, SharedStore, being a low-level system, necessitates the use of some unsafe Rust code, primarily for interacting with block devices. This 'unsafe' code is not subject to all of Rust's safety guarantees, thereby introducing the potential for undefined behavior. To mitigate this risk, the developers employed the Miri interpreter [21], a dynamic analysis tool integrated into the Rust compiler. This tool can detect certain types of undefined behavior, enhancing the robustness of the system.

From our own understanding, the reliance on external tools like Miri indicates a potential vulnerability in the SharedStore system's design. The need to extend Miri to support basic concurrency primitives also suggests that Rust's native capabilities may not fully meet the needs of complex, low-level systems like SharedStore.

Secondly, we are talking about serialization. SharedStore employs numerous serializers and deserializers to convert data between in-memory and on-disk formats. Given that both bit rot and transient failures can corrupt on-disk data, the developers wisely treat data read from disk as untrusted. They aim to ensure that the deserialization code is robust enough to handle such corrupted data. To achieve this, they collaborated with the developers of Crux [17], a bounded symbolic evaluation engine, to prove the panic-freedom of their deserialization code. This is a commendable approach, as it ensures that their deserializes won't crash due to out-of-bounds accesses or other logic errors, regardless of the input.

However, from our perspective, the reliance on fuzzing for larger inputs due to the small size bound required for symbolic evaluation indicates a potential scalability issue.

#### 5.5 Recap of SharedStore

In conclusion, the SharedStore team has made significant strides in addressing these issues, the reliance on external tools and the potential scalability concerns suggest areas for further research and optimization. As a graduate student studying this, I would be interested in exploring alternative approaches to handle undefined behavior and serialization, potentially leveraging Rust's evolving ecosystem of libraries and tools. It would also be intriguing to investigate how

these challenges are addressed in other storage systems and whether those strategies could be adapted for SharedStore. This could lead to a more robust and scalable system, reducing the reliance on external tools and making the system more self-contained.

## 5.6 Further Exploration

In [2], the authors have provided a comprehensive overview of their validation process for SharedStore. From our own understanding, we believe the following areas could benefit from further exploration, and here is our suggestion and findings.

**Bug Detection and Validation Effort** The authors have done a good job of preventing bugs from reaching production. However, the example provided about the cache-miss path bug raises questions about the comprehensiveness of their testing strategies. It would be beneficial to understand how they plan to address these limitations in the future. The paper states that the validation tests comprise 31% of the code base, which seems quite substantial. However, it's unclear how this compares to industry standards or other similar projects. Providing such a comparison could help contextualize this figure better.

**Early Detection and Adoption by Engineering Team** The paper mentions that the results do not measure the effectiveness of detecting bugs early in development. This is a significant aspect of any testing strategy, as early detection can save considerable time and resources. From the software development life cycle aspect, the team would live to catch the bugs as early as possible. As we get into the late stages, the bugs would require much more effort to fix. Thus, it would be meaningful and beneficial to have some quantitative data or evidence about the types of bugs caught early and the impact of early detection on the overall development process. It's encouraging to see that the SharedStore engineering team, who do not have prior formal methods experience, has taken over most of the work on the reference models and test harnesses. However, the paper could provide more insight into the learning curve and challenges faced by the team during this transition. This kind of lesson could be helpful to any other teams, or companies who don't have formal method experience to adopt the formal methods.

**Limitations and Future Directions** The authors acknowledge that their validation checks can miss bugs, which is an important admission. However, the example provided about the cache-miss path bug raises questions about the comprehensiveness of their testing strategies. It would be beneficial to understand how they plan to address these limitations in the future. The authors mention their plans to extend their work to include reasoning about SharedStore's role in the S3 system and to strengthen the guarantees provided by their validation

work. These are important areas of focus, and it would be great to have more details about these future plans, such as the specific strategies they plan to use and the expected impact on the overall system.

**Use of Rust and Role of Formal Methods Experts** The decision to write the models in Rust seems to have been a key factor in the success of the project. It would be interesting to know more about why Rust was chosen over other languages and how it contributed to the overall effectiveness of the validation process. The authors mention that the first iteration of the reference models was written by formal methods experts, but they have been updated almost exclusively by the engineering team. It would be beneficial to understand more about this transition process and how the expertise of the formal methods experts was transferred to the engineering team.

## 6 Property-based Testing on pickleDB Using Hypothesis

### 6.1 Introducing pickleDB

PickleDB [6] is a lightweight, file-based database management system. At its core, PickleDB employs file handling mechanisms for database data management. This approach suggests that the database is persisted in a file format on disk storage, making it an apt choice for applications that necessitate durable storage without the complexities of a server-based database system.

Key Functionalities of PickleDB:

- Object-Oriented Design: The foundation of PickleDB is the `PickleDB` class, underscoring an object-oriented methodology. This design enhances the system’s modularity and facilitates efficient handling of database processes.
- File Handling Capabilities: Leveraging file operations, PickleDB manages its database, indicating that it stores data in a file-based format. This trait is particularly beneficial for applications requiring persistent storage with minimal overhead.

PickleDB is developed as a straightforward, file-based database solution. It is well-suited for Python applications where simplicity, ease of integration, and basic database functionalities are prioritized. The simplicity of its API enhances its appeal for swift deployment in applications requiring basic data persistence features.

### 6.2 Modifying pickleDB Source Code

The original `_loaddb()` method in `pickleDB` employed a straightforward approach to loading the database:



```
def _loaddb(self):
    '''Load or reload the json info from the file'''
    try:
        self.db = json.load(open(self.loco, 'rt'))
    except ValueError:
        if os.stat(self.loco).st_size == 0:
            self.db = {}
        else:
            raise
```

The file was opened without a context manager, potentially leading to resource leaks if an exception occurred before the file was closed. To address these issues, I made the following modifications:

```
def _loaddb(self):
    '''Load or reload the json info from the file'''
    try:
        # Using context manager to open the file
        with open(self.loco, 'rt') as file:
            self.db = json.load(file)
    except ValueError:
        if os.stat(self.loco).st_size == 0:
            self.db = {}
        else:
            raise
```

Using `with open(...) as file` ensures that the file is properly closed after its suite finishes, even if an exception is raised. This prevents resource leaks. This modifications have significant implications for property-based testing, particularly when using the Hypothesis framework. Tests can now run in a more stable environment since the modified code properly manages file resources, reducing the likelihood of tests failing due to resource-related issues.

### 6.3 Original Unit Tests in pickleDB

The existing unit tests [19] for PickleDB in the original Github repositories offer a foundational assessment of its functionalities. The current tests encompass key database operations including data loading, setting and retrieving entries, deletion mechanisms, and handling of complex data structures like lists and dictionaries. The test suite includes both positive (valid input) and negative (invalid input) testing, ensuring a basic level of robustness and error handling in typical scenarios.

However, the tests are limited to straightforward use cases, lacking in complex or edge-case scenarios such as handling of large data sets, concurrent operations, and data integrity under system failures. Current tests are example-based and do not automatically explore a wide range of input variations, potentially missing

out on identifying unexpected behaviors or hidden bugs. The tests fall short in extensively verifying data persistence over multiple sessions, a critical aspect for any database system to ensure long-term data integrity.

Utilizing Hypothesis for property-based testing can dramatically increase the breadth of test scenarios, including random and varied inputs to thoroughly assess database robustness. Hypothesis excels at automatically uncovering edge cases, crucial for a database system where handling of atypical inputs must be graceful and error-free. By defining and testing invariants, Hypothesis ensures consistent behavior of database operations, such as the reliability of set and get functions. Also, Hypothesis supports stateful testing, a significant advantage for databases. This approach can mimic real-world usage patterns, ensuring consistent behavior across various operations and state transitions.

So, while the existing unit tests for PickleDB lay a solid groundwork for basic functionality validation, incorporating property-based testing with Hypothesis is essential for a database system, as it not only broadens the testing scope to include a diverse range of scenarios but also ensures the system’s resilience and reliability in handling complex, real-world data operations.

#### 6.4 Switching to property-based testing

The `test_pickledb_properties.py` script in our UWaterloo Gitlab group repository represents a sophisticated effort to rigorously test the pickleDB library using property-based testing methodologies. By leveraging the Hypothesis library within the Python `unittest` framework, the tests effectively probe the database’s behavior under a wide array of generated scenarios, going beyond conventional example-based unit test paradigms.

The tests are structured as a class extending `unittest.TestCase`, providing a familiar and organized testing environment. The setup (`setUp`) and teardown (`tearDown`) methods are utilized to initialize the testing environment, including creating a temporary database instance for each test case, ensuring isolation and repeatability of the tests.

Hypothesis is used to generate a variety of data types and structures, probing the database’s ability to accurately store and retrieve diverse data formats. The use of Hypothesis’ `strategies` module allows for the creation of complex test scenarios, pushing the database to its operational limits and ensuring thorough coverage of potential use cases. By generating edge cases and unusual data inputs, the tests assess the robustness of pickleDB’s error handling mechanisms, ensuring stability in the face of unexpected or erroneous input.

The passing of all property-based tests suggests that our testing has uncovered areas where pickleDB performs exceptionally well, particularly in standard data handling and retrieval scenarios.

#### 6.5 Advancing to stateful property-based testing

In our significant advancement of the property-based testing methodology for pickleDB, the approach has evolved into a stateful property-based testing using

Hypothesis, as demonstrated in the `test_pickleddb_properties_stateful.py` script in our UWaterloo Gitlab group repository. This advanced strategy offers a more dynamic and realistic simulation of database interactions, capturing a broader spectrum of potential issues and behaviors [22].

The stateful testing paradigm implemented in our new code differs fundamentally from traditional property-based testing. While the original approach focused on generating a wide range of inputs and testing individual functions independently, stateful testing introduces the concept of maintaining an internal state that mimics the expected state of the database. This approach allows for the simulation of a sequence of operations, mirroring more closely real-world usage patterns.

A critical aspect of our stateful testing approach is the use of an internal model (`self.model`), a dictionary that maintains the expected state of the database. This model is used to verify the consistency of the database's state throughout the testing process. Through the `RuleBasedStateMachine` class, our test script defines various operations like `set_value`, `get_value`, `delete_key`, `update_value`, `list_keys`, and `dump_database`. These operations are not only tested in isolation but also in sequences that reflect more complex and realistic usage scenarios. The use of `@invariant` marks a method (`db_matches_model`) that continually checks the consistency between the internal model and the actual state of the database after each operation, ensuring that the database behaves as expected at all times.

The stateful approach brings several advantages. By testing sequences of operations rather than isolated actions, the tests can uncover issues that might only arise in more complex operational contexts. The internal model provides a powerful tool for continuously verifying the database's integrity, ensuring that each operation not only succeeds in isolation but also maintains overall database consistency. This approach is particularly effective in detecting and diagnosing errors related to state management, such as improper updates or deletions, which might not be apparent in traditional property-based testing.

The advancement to stateful property-based testing marks a significant leap in the robustness and comprehensiveness of the testing process for pickleDB. By more accurately simulating real-world usage and maintaining an internal model of the expected database state, this method provides a deeper and more nuanced understanding of the database's behavior and potential vulnerabilities. This approach not only strengthens the reliability of pickleDB but also serves as a model for advanced testing methodologies in the development of robust software systems.

## 7 Conclusion

In conclusion, this project has given us a valuable opportunity to look into the practical aspects of software verification specifically in distributed storage and property based testing. We have explored its broader applications, understood the reasons behind its underutilization in some projects, and identified the areas

that can be improved, even we truly appreciated the authors great effort. Our focus on property-based testing has shed light on its potential to enhance the reliability of software systems.

Our critical review of the validation process of SharedStore, a distributed storage system, has allowed us to dive into their bug detection and validation efforts, early detection strategies, and the role of formal methods experts. We have also examined the limitations of their current approach and discussed potential future directions.

Overall, this study has enriched our understanding of software verification and its practical implications. It has underscored the importance of continuous learning and improvement in any development process. We believe that the insights gained from this study will be beneficial in the application of the concepts learned in the ECE653 course and beyond. We look forward to applying this knowledge to real software engineering projects and continuing our journey in the field of software testing, verification, and maintenance.

## References

1. Arts, T. et al.: Testing telecoms software with quviq QuickCheck. In: Proceedings of the 2006 ACM SIGPLAN workshop on Erlang. pp. 2–10 Association for Computing Machinery, New York, NY, USA (2006). <https://doi.org/10.1145/1159789.1159792>.
2. Bornholt, J. et al.: Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3. In: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. pp. 836–850 Association for Computing Machinery, New York, NY, USA (2021). <https://doi.org/10.1145/3477132.3483540>.
3. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming. pp. 268–279 Association for Computing Machinery, New York, NY, USA (2000). <https://doi.org/10.1145/351240.351266>.
4. Dahl, O.-J. et al.: Structured programming. Academic Press Ltd. (1972).
5. DUBIEN, N.: dubzzz/fast-check, <https://github.com/dubzzz/fast-check>, (2023).
6. Erd, H.: patx/pickledb, <https://github.com/patx/pickledb>, (2023).
7. Fink, G., Levitt, K.: Property-based testing of privileged programs. In: Tenth Annual Computer Security Applications Conference. pp. 154–163 (1994). <https://doi.org/10.1109/CSAC.1994.367311>.
8. Gallant, A.: BurntSushi/quickcheck, <https://github.com/BurntSushi/quickcheck>, (2023).
9. Hance, T. et al.: Storage Systems are Distributed Systems (So Verify Them That Way!). Presented at the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20) (2020).
10. Lampropoulos, L. et al.: Coverage guided, property based testing. Proc. ACM Program. Lang. 3, OOPSLA, 181:1-181:29 (2019). <https://doi.org/10.1145/3360607>.
11. Löscher, A., Sagonas, K.: Targeted property-based testing. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 46–56 Association for Computing Machinery, New York, NY, USA (2017). <https://doi.org/10.1145/3092703.3092711>.
12. Matela Braquehais, R.: Tools for Discovery, Refinement and Generalization of Functional Properties by Enumerative Testing. University of York (2017).

13. Musuvathi, M., Qadeer, S.: Fair stateless model checking. *SIGPLAN Not.* 43, 6, 362–371 (2008). <https://doi.org/10.1145/1379022.1375625>.
14. Paulson, T.N.L.C., Wenzel, M.: A Proof Assistant for Higher-Order Logic, <http://src.gnu-darwin.org/ports/math/isabelle/work/Isabelle/doc/tutorial.pdf>, (2013).
15. Runciman, C. et al.: Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In: *Proceedings of the first ACM SIGPLAN symposium on Haskell*. pp. 37–48 Association for Computing Machinery, New York, NY, USA (2008). <https://doi.org/10.1145/1411286.1411292>.
16. Sigurbjarnarson, H. et al.: Push-button verification of file systems via crash refinement. In: *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*. pp. 1–16 USENIX Association, USA (2016).
17. Tomb, A.: Crux: Introducing Our New Open-Source Tool for Software Verification, <https://galois.com/blog/2020/10/crux-introducing-our-new-open-source-tool-for-software-verification/>.
18. hypothesis/hypothesis-python at master · HypothesisWorks/hypothesis, <https://github.com/HypothesisWorks/hypothesis/tree/master/hypothesis-python>.
19. pickledb/tests.py · <https://github.com/patx/pickledb/blob/master/tests.py>.
20. proptest-rs/proptest, <https://github.com/proptest-rs/proptest>, (2023).
21. rust-lang/miri, <https://github.com/rust-lang/miri>, (2023).
22. Stateful testing Hypothesis 6.92.1 documentation, <https://hypothesis.readthedocs.io/en/latest/stateful.html>.