IronSpec: Increasing the Reliability of Formal Specifications

Eli Goldweber Weixin Yu Seyed Armin Vakil Ghahani Manos Kapritsos
University of Michigan
{edgoldwe, weixinyu, arminvak, manosk}@umich.edu

Abstract

The guarantees of formally verified systems are only as strong as their trusted specifications (specs). As observed by previous studies [22, 52], bugs in formal specs invalidate the assurances that proofs provide. Unfortunately, specs—by their very nature—cannot be *proven* correct. Currently, the only way to identify spec bugs is by careful, manual inspection.

In this paper we introduce IronSpec, a framework of automatic and manual techniques to increase the reliability of formal specifications. IronSpec draws inspiration from classical software testing practices, which we adapt to the realm of formal specs. IronSpec facilitates spec testing with automated sanity checking, a methodology for writing *Spec-Testing Proofs (STPs)*, and automated spec mutation testing.

We evaluate IronSpec on 14 specs, including six specs of real-world verified codebases. Our results show that IronSpec is effective at flagging discrepancies between the spec and the developer's intent, and has led to the discovery of *ten* specification bugs across all six real-world verified systems.

1 Introduction

Formal verification has emerged as a promising technique for increasing the robustness of complex systems by helping developers prove that their implementation meets a formal specification. As promising as this approach is, it has a fundamental Achilles' heel: its guarantees of eliminating *all* bugs in the implementation rely on the *specification being correct*.

The crucial observation that the guarantees of a mechanized proof are only as strong as their specifications is not new and was first identified in 1985 [33]. Specifications (a.k.a. specs) are inherently trusted, rather than proven correct. Relying on trust alone is not enough to ensure that specs remain bug free. If a spec contains a bug, proving that the system meets this spec may be meaningless; the *proven* system could also contain a bug that is hidden by the buggy spec.

The correctness of specifications is the rock upon which the entire edifice of formal verification is built.

Despite the importance of writing correct specs, current best practices rely solely on manual inspection. Developers argue [25, 26] that because specs are typically small compared to the size of the corresponding proof and implementation, it is feasible to manually inspect specs thoroughly enough to ensure that they **capture the intended behavior** of the system. While expert developers are more likely to write correct specs, they are not infallible. As formal verification becomes widely adopted, more and more non-experts will write specs, only exacerbating the risk of introducing bugs. Thus, it is imperative that the process of writing specs be as robust as possible.

In fact, several studies [22, 32, 52], through extensive manual effort, have shown that formally verified systems—many of which were developed by experts in formal verification—contain critical bugs, which originate with problems and inconsistencies in their specs. For example, in January 2022, Notional Finance found a double-spending vulnerability in a deployed verified smart contract missed by manual inspection [34]. In this case, part of the spec was vacuous, causing it to be too weak, and thus the proof would still pass with a *buggy* implementation.

Since a spec is a formal expression of a developer's intent, *proving* the spec correct is ultimately impossible. Ensuring a spec matches a developer's intent will always be best-effort. Whilst no approach can guarantee a bug-free spec, that does not mean attempts to do so must exclusively rely on extensive manual effort and system expertise to resolve. Indeed, there are no structured or automated approaches for a developer to debug this complicated state space. To fill this gap, we propose a means to better handle this challenge.

Inspired by classic testing techniques [17, 24, 48], we introduce IronSpec, a spec testing framework. To enable testing specs, IronSpec adapts the automation of mutation testing [19, 31] and sanity checking along with a customized manual testing approach inspired by unit testing. Together, this framework introduces a systematic way to boost assurance that a spec captures the intended behavior of the system.

If there is a bug in a spec, it originates in the same manner

as any other type of bug; there is a disconnect between the intent of the developer and what is written. A spec is *incorrect* if it is too weak, allowing for the existence of even a single implementation that exhibits undesired behavior, or if the spec is too strong, precluding some desired behavior. To identify spec bugs, we leverage the insight that spec bugs manifest themselves as a consequence of *a disconnect of intent* to search for and highlight such disconnects through structure and automation.

IronSpec aids in pinpointing where the developer's intent diverged from the current spec by providing various tools that encapsulate this notion. Common cases where the intent of the developer deviated from the spec can be flagged with IronSpec's Automatic Sanity Checker. If a system has a passing proof, IronSpec leverages this to provide additional automation with spec mutation testing. Mutation testing can automatically identify cases where the behavior of the implementation differs from the spec, by using the proof to identify relevant mutations. The hints of potential intent disconnect provided by automation are bolstered by a manual methodology for writing *Spec-Testing Proofs (STPs)*. STPs are inspired by traditional unit testing and allow developers to test if their understanding of what behavior the spec should allow matches the current spec. STPs can be used to investigate the hints provided by automation to either confirm the existence of a bug or to absolve the disconnect as intended behavior.

We evaluate IronSpec by testing six specs produced in-house, two specs containing artificial bugs that were studied in Abreu et al. [6], and six specs of open-source verified systems. We demonstrate the effectiveness of the automation and manual testing methodology of IronSpec by describing *ten* spec bugs found across a verified Distributed Validator Protocol [4], a verified SAT solver [7], a verified QBFT system [2], a formal spec of the Eth2.0 spec [1], daisy-nfsd [15], and a verified AWS Encryption SDK library [3].

Overall, this paper makes the following contributions:

- Introduces IronSpec, a spec testing framework that allows developers to pinpoint places where the current spec may have diverged from their original intent.
- Proposes an Automatic Sanity Checker, a testing methodology for writing *Spec-Testing Proofs (STPs)*, which are applicable to test specs even in the absence of a completed proof or implementation, and describes how to adapt mutation testing to specs to automatically identify divergences between the spec and the implementation.
- Demonstrates the effectiveness of IronSpec, by illustrating how we applied IronSpec to *six* real-world, verified systems leading to the discovery of *ten* spec bugs.

2 Manually Scrutinizing Specifications

Relying on manual inspection alone to ensure an intended specification is not practical. Fonseca et al. [22] performed a study aimed to challenge the assumption that just because a system is verified, it is bug-free. In this study, the authors thoroughly examined three formally verified distributed systems, IronFleet [26], Verdi [53], and Chapar [41] and identified sixteen bugs across their specifications, verification tools, and their unverified shim layers. Two of these bugs were found to be in specifications. This study was chiefly manual and required close examination of the respective specifications to identify. The authors do introduce some basic automation, yet their techniques still rely predominantly on manual effort and expertise in the system. This work demonstrated the need for and acknowledges the lack of a more rigorous and automated approach to testing formal specifications. Similarly, Yang et al. [54] conducted a bug study of compilers and discovered two bugs within the verified compiler CompCert due to underspecification, and similarly observed that specifications are complex and lack scrutiny.

The concerning discovery of these previous works identifies the gap that this work aims to fill; to provide a means for developers to help automatically and methodically identify specification bugs across the spectrum of specifications.

Complicating this problem, specifications can take on different forms, making uniform debugging approaches difficult. In their simplest form, specifications can be in-line predicate assertions [29]; boolean functions that check the state of the system against some property. A more specific class of predicate assertions based on the Floyd-Hoare style logic [21, 28] are preconditions and postconditions, which establish invariants about the state of the program before and after the execution of a piece of code. For more complex systems, rather than directly proving properties about the system, it can be easier [26, 53] to prove state machine refinement [37]. For refinement, the specification is an abstract state machine that encapsulates the desired behavior of the system.

To highlight the subtlety of trying to manually ensure a specification is correct, consider an *incorrect* specification for a simple `Sort` method found on line 3 of Specification 1. This `Sort` method takes a sequence of integers as input and promises to return a sorted sequence of integers in ascending order. The specification for this method is a single postcondition which ensures that the value at every index in the output sequence is less than or equal to the value at subsequent indices. At first glance, this may seem to be a correct specification for `Sort`—a mistake that many newcomers to verification make.

However, this specification is incorrect, as it neglects to mention any relationship between the input and output sequences. This is considered a buggy specification because a proof could still pass even with an *incorrect* implementation that exhibits undesired behavior, erroneously giving the illusion of correctness. For example, if the input sequence were $[1,6,7,2]$, an incorrect implementation could arbitrarily return $[1,42,100]$ or even the empty sequence $[\,]$. The *incorrect* implementation for `Sort` in Specification 1 is triv-

Specification 1 Incorrect Sort Spec

```
1  method Sort(input:seq<int>)
2    returns (out:seq<int>)
3    ensures forall i | 0 <= i < |output| - 1 ::
4              out[i] <= out[i+1]
5  { return []; }
```

Specification 2 Correct Sort Spec

```
1  method Sort(input:seq<int>)
2    returns (out:seq<int>)
3    ensures forall i | 0 <= i < |output| - 1 ::
4              out[i] <= out[i+1]
5    ensures multiset(input) == multiset(output)
6  { /* body omitted */ }
```

ial and always returns an empty sequence. Yet, the proof for this method would still pass, as this trivial implementation satisfies the incorrect, too-weak specification.

Manually identifying a spec bug, like that in Specification 1, can be challenging. In fact, a correct specification for `Sort` should also capture the relationship between the input and output by adding an additional post-condition to ensure that the multiset of the input is equal to the multiset of the output, see line 5 in Specification 2.

The opposite case, where a specification is too strong, can be equally as important and challenging to manually identify. For example, if we replace line 5 in Specification 2 with `ensures input == output`, the specification becomes unnecessarily strong. Multisets do not take order into account, whereas sequences do, so the updated postcondition is overly strong. The only input and output pair that could satisfy this specification is if the sequences are identical and already in ascending order. Even if one has a correct implementation of `Sort`, proving that the implementation upholds this specification is impossible. To debug the inevitably failing proof, the developer must examine their implementation for bugs, check their proof for missing invariants *and* manually inspect their spec to make sure it captures the intended behavior. Having high confidence in the spec would make this scenario much more unlikely and would give the developer more time to focus on the proof itself, knowing they are proving the right property.

3 How To Test A Specification

It is challenging to diagnose spec bugs because specs are trusted, and a buggy spec can often be at odds with a developer's original understanding of the system. Complicating the problem, specs are often intended to be abstract, allowing different, correct implementations to meet the spec. Hence, we introduce IronSpec, a framework for testing specs to help gain confidence that a spec is bug-free. This work represents the first systematic effort to bridge the gap between the mature and extensive work in software testing and the lack of rigor in ensuring spec correctness.

IronSpec is inspired by the insight that the existence of a spec bug is inherently due to a disconnect between what the developer intended and what properties were actually captured in the spec. IronSpec provides tools to allow a tester to identify and test possible occurrences where the original intent of the developer may have diverged from the current spec. Some aspects of IronSpec only rely on the spec and have no dependence on the existence of an implementation or a passing proof. However, if there is an implementation and a corresponding passing proof, IronSpec can leverage this to use the implementation as an additional reference point to help focus the testing process.

This section introduces and provides a high-level overview behind the ideas of why each testing component of IronSpec is useful in exposing disconnects between the intent of the developer and their spec. Section 4 discusses each in more detail.

3.1 Testing Specifications In The Absence Of A Passing Proof

Akin to test-driven development [10], it is desirable to test a spec without requiring a proof or corresponding implementation. If there is a bug in the spec when it comes time to write a proof, a developer may struggle and expend unnecessary manual effort in debugging in the wrong place. The Automatic Sanity Checker and *Spec-Testing Proofs (STPs)* provide two frames of reference for a tester to check their specs against, even in the absence of an implementation and proof.

Regardless of the context of the system, it is clearly never intended for a verified method to be permitted to return arbitrary values. If the spec is too weak, an incorrect implementation might be free to return *any* value, unconstrained by the spec. The Automatic Sanity Checker raises high-confidence flags when the spec of a verified method fails to properly constrain its output based on the given input. The sanity checker also alerts the developer to partially constrained input and output, which provide weaker hints to the existence of spec bugs but are also worthwhile to investigate further.

Because the Automatic Sanity checker only looks for under-constrained input and output, this technique can be used even in the absence of an implementation or proof. Section 4.1 describes the Automatic Sanity Checker in more detail.

The Automatic Sanity Checker excels at automatically finding common spec bugs by leveraging generic code patterns, but cannot leverage any user-provided hints and insights. We address this gap by introducing a methodology for manually writing *Spec-Testing Proofs (STPs)*. STPs are inspired by traditional unit tests and are proofs about the spec for context-specific input and output. STPs help developers expose differences between the expected behaviors they intend to include in the spec and what is currently permitted. This

testing methodology is useful in the presence of a passing or failing proof, but can also be applied in the absence of a proof. Section 4.2 explains how to write STPs and interpret their results.

An STP is, by definition, a proof; this is the key difference between STPs and standard unit tests. Since STPs are proofs, STPs help to answer different questions than what unit tests allow for. Instead of attempting to prove a general property, an STP demonstrates the validity of the spec for a specific, concretized value or a range of values. This testing methodology exploits the insight that crafting proofs for specific cases is often less challenging than producing a comprehensive proof and can frequently be proved by the verifier with minimal manual intervention. Each STP is a small proof about *distinct* properties of the spec. The steps of writing STPs are generic, and so can be useful tools in investigating the correctness of many variations of specs.

Differing from a failed unit test, if an STP fails to verify, it could be for various reasons. The STP may fail due to a divergence between the expectation of the spec and the STP, indicating a bug. If the tester suspects that a disconnect caused the failed proof, the appropriate next step is to write a concrete *Counterexample* STP. The counterexample proves that unintended behavior is permitted by the spec. Alternatively, an STP may fail because the STP body lacks sufficient proof annotations for the verifier to prove the final postcondition. Distinguishing between a spec bug and the need to add proof to the body of the STP is impossible to immediately diagnose for every case because in this work we are targeting undecidable programs.

3.2 Testing Specifications With The Assistance Of A Passing Proof

Even when a system is verified with a passing proof, it is still possible for the system to contain bugs if the spec itself is buggy; thus testing a spec at this point is still very valuable. A too-weak spec could allow for a proof to pass with an incorrect implementation, falsely giving the illusion of correctness. Alternatively, even if the current implementation contains no bugs, a too-weak spec could allow for a buggy update to the current implementation, such that a proof would still pass with the same too-weak spec. Relying on a developer to write a bug-free implementation given a buggy spec, goes against the very reason to verify systems in the first place; so it is just as vital to identify spec bugs when the proof passes.

Using the Automatic Sanity Checker and writing STPs are applicable when testing a spec with a passing proof, but the proof and implementation together contain untapped information that can further assist testing. Like the spec, the implementation also captures the intent of the developer. Identifying the difference of intent between the behavior allowed by the spec and what is actually in the implementation, calls the developer's attention to potential disconnects. IronSpec can take advantage of the proof and implementation to automatically test a spec with *mutation testing*. Mutation testing identifies cases when the spec is weaker than the current implementation. IronSpec uses the passing proof as a reference point to automatically distinguish cases where the existing implementation is weaker than the behavior allowed by the spec. Further details concerning how IronSpec adapts mutation testing to specs are described in Section 4.3.

Departing from traditional mutation testing, IronSpec starts with a spec, implementation, and passing proof and then only mutates the spec. IronSpec relies on an existing passing proof to indicate whether a mutation should be killed, whereas traditional mutation testing relies on a test suite. A mutation is kept and considered *alive* if the original proof still passes with the mutated spec, indicating that the implementation also meets this different spec. The behavior allowed by the original spec but not the mutated spec serves as an example of a subset of behavior that may not be intended.

The existence of even a single alive spec mutation serves as a flag to the developer. An alive mutation is clear evidence that a different spec still allows the proof to pass with the unmodified implementation, and represents specific behavior unaccounted for by the original spec. The difference between the original spec and the passing mutated spec is a strong hint for the tester to determine if that specific behavior is intended. Identifying alive mutations is accomplished automatically, but understanding the implication of any such alive mutation cannot be automated and ultimately still relies on the developer's intuition to understand.

An alive mutation is simply a hint highlighting a divergence between the spec and the implementation. However, not all alive mutations immediately lead to the discovery of a spec bug. If a spec is correct but also weaker than the current implementation, there is a chance for an alive mutation to be considered a false positive and marked as intended behavior. In a contrasting, albeit rare case, if both the spec and implementation are buggy, but the implementation is not weaker than the buggy spec, then no alive mutations may be found.

To reduce the chance of false positives only a subset of the generated mutations are eventually considered. Logically equivalent or weaker mutations than the original spec and mutations that trivially make the proof pass can be safely ignored. The details for how specs are mutated and what constitutes valid mutations are expounded upon in Section 4.3.

Note that we deliberately mutate only specs and not implementations for two reasons. Firstly, specs are smaller than implementations, therefore reducing the number of mutations necessary to consider. Secondly, mutating only the spec rather than the implementation is advantageous for automation. Specs, being boolean functions, enable automatic filtering of irrelevant mutations. Assuming the proof passes given the original spec, any logically weaker spec mutation will still allow the proof to pass and does not provide any new relevant information. By automatically checking the relative logical

strength of a mutated spec in relation to the original, weaker mutations can be identified and ignored. This automation is impossible when mutating the implementation, as determining relative logical strength is not possible in all cases. Logical relationships can be determined automatically for boolean functions, like specs, whereas not all imperative code shares this attribute. Implementation-based mutations would increase the manual burden on the tester, as many more false positives would be an unavoidable outcome that would require manual effort to sift through. The process of automatically filtering spec mutations is further explained in Section 4.3.2.

In certain cases, mutation testing is also useful in identifying too-strong specs. A spec in the Hoare-Logic style can also be considered incorrect by virtue of having a too strong precondition. IronSpec's mutation testing is still applicable in this case. If the spec mutation target is a precondition, rather than attempting to identify where the spec is disconnected from the implementation due to weakness, IronSpec reverses the criteria used to determine relevant mutations by considering mutations that are *weaker* than the original spec.

Mutation testing does not provide complete coverage of spec testing but rather focuses the attention of the tester on a disconnect between the spec and the implementation. STPs can be used to help fill this gap. Focusing on writing STPs about the discrepancy hinted at by an alive mutation leads to a more efficient way of identifying bugs. STPs guided by the hint of alive mutations can allow a tester either to arrive at a counterexample, showing a bug in the spec, or to absolve the alive mutation as intended behavior.

4 The IronSpec framework

IronSpec consists of three spec testing tools; an Automatic Sanity Checker, a methodology for writing *Spec-Testing Proofs (STPs)*, and an automatic mutation testing framework. Each assists in identifying and flagging divergences between the developer's intent and the existing spec.

The IronSpec prototype is built in C# as an extension to Dafny [40], a verification-aware programming language that enables verification with the Z3 SMT solver, and also supports practical imperative implementations by compiling to C#, Java, JavaScript, and Go. IronSpec was applied to test specs written in Dafny, but the concepts of how to test specifications are not Dafny-specific and could be re-implemented in other environments.

4.1 Automatic Sanity Checker

The Automatic Sanity Checker (ASC) examines the input, output, and spec of verified methods to identify cases where the spec may be weaker than intended. The ASC implementation consists of approximately 300 lines of C# code and achieves this check by traversing the AST of the method under test while maintaining some local state. Table 1 outlines the properties that are checked and their assigned severities. All of these properties can be determined by examining the AST, and as such, they can be checked efficiently without invoking a verifier. Either of the HIGH severity flags signifies a high likelihood of spec bugs, whereas the other severity levels indicate a cause for additional manual inspection. Both HIGH severity flags reveal a weakness in the flagged spec. If the postconditions do not depend on the input, then the weakness is in regard to the lack of necessity for that input. Whereas, if the postconditions do not constrain the output, an implementation with a passing proof could return arbitrary output values. Regardless of the particular functionality of the system, either case represents a clear disconnect between the intent of a correct spec and the current spec.

Table 1: Automatic Sanity Checking Flags

Flag Severity	Condition
LOW	Post conditions only depend on a portion of the input
MED	Only part of the output is constrained by the postconditions
HIGH	None of the postconditions depend on any of the input
HIGH	None of the output is constrained by any of the postconditons

The power of the Automatic Sanity Checker arises from exploiting the relationship between a spec and the input/output of its corresponding method. Both HIGH severity flags signal the condition when the spec constraints on the input/output of the method are non-existent. If no postcondition depends on any of the input values, then an obvious aspect of the spec is missing. The buggy sort spec in Specification 1 exemplifies this scenario. The spec is not constrained at all by the input, making the spec weak enough to allow for a proof of an incorrect implementation to pass. Similarly, if a method has an output not constrained by its postconditions, an incorrect implementation can return *any* output. The lower severity flags hint to partial violations of the general properties and do not immediately indicate bugs; rather, they signal a missing part of the spec that could be the source of a bug.

4.2 STP Methodology

The testing methodology outlines four classes of STPs. The first three help guide developers in understanding the **Usefulness**, **Correctness**, and **Provability** of their specs. Lastly, if there is a bug in the spec, developers can prove its existence with a **Counterexample** STP. The methodology focuses on specs written following the Hoare-Logic style [51] but can be applied to any type of predicate-based spec. All types of STPs enable the developer to prove a specific property about their spec. A developer proves that context-specific input and

Lemma 3 General Precondition STP

```
1  lemma PreconditionSTP(in:InType)
2    requires TestInputProperty(in)
3    ensures Precondition(in)
4            // or !Precondition(in)
```

Lemma 4 General Postcondition STP

```
1  lemma PostconditionSTP(in:InType,out:OutType)
2    requires TestInputProperty(in,out)
3    ensures Precondition(in)
4    ensures Postcondition(in,out)
5            // or !Postcondition(in,out)
```

output values are valid or invalid, and gauge if those results match their *intent* based on their understanding of what the spec should or should not permit. Lemma 5 is an example of an STP which tests if a sort spec is strong enough to reject specific *invalid* values. A passing STP shows that the intent of the developer matches the spec and is a proof for that particular property of the spec.

4.2.1 Writing STPs

The construction of different types of STPs share many similarities, but the results are interpreted differently. STPs also enable decoupling of pre- and postconditions so that they can be tested individually. The general form for these STPs are found in Lemmas 3 and 4.

Usefulness STPs help to answer the question of whether the preconditions are weak enough to remain useful; the preconditions should accept all intended valid inputs. Usefulness STPs follow the general form of Lemma 3. The specific input values are defined as part of the precondition for this lemma as the `TestInputProperty`, and should be a value that the test writer *expects* to be a valid input allowed by the spec. The postcondition for a Usefulness STP should be the preconditions from the spec, i.e. `ensures Precondition(in)`.

Correctness STPs examine whether the postconditions are strong enough to reject all intended invalid outputs. Writing Correctness STPs is based on the general form of Lemma 4. To test if the postcondition is strong enough to reject buggy behavior, the test writer supplies an output value that is *expected* to be invalid and should not be allowed by the spec i.e. `ensures !Postcondition(in)`. To isolate testing the postcondition from the precondition, the test writer should also prove that the undesired output does not satisfy the spec as a result of an invalid input value (Line 3 in Lemma 4), ideally with a separate Usefulness STP validating the input.

Conversely to Usefulness and Correctness STPs, Provability STPs test whether the preconditions are strong enough and whether the postconditions are weak enough for the existence of a provable implementation. Provability STPs are most useful before having a passing proof, as a passing proof is evidence that the spec has this property. That said, they

Lemma 5 Correctness STP Example - Incorrect Sort Spec

```
1  lemma CorrectnessSTPSort(
2      input:seq<int>, sorted:seq<int>)
3    requires input == [42, 1, 500]
4    requires sorted == [42, 500]
5    ensures !SortSpec(input, sorted)
6  { }
```

can still provide value in the presence of a passing proof, as they can test the strength of transitions in a state machine (see Section 5.2.2).

STPs for Provability are concerned with both preconditions and postconditions, thus follow the structure from both Lemmas 3, and 4. Precondition STPs prompt the test writer to prove that *expected* invalid input should not pass the precondition, i.e. `ensures !Precondition(in)`. Whereas postcondition STPs check that input and output *expected* to be permitted by the spec is allowed by the postconditions, i.e. `ensures Postcondition(in,out)`.

If suspecting a spec bug, a test writer can also directly write a Counterexample STP. A passing Counterexample STP is concrete evidence of a bug in the spec. Counterexample STPs can take on two different forms but are still derivative of Lemma 4 if concerned with postconditions and Lemma 3 for preconditions. A Counterexample STP can either show that an expected **valid input-output pair is rejected** by the spec or that an expected **invalid input-output pair is accepted**.

4.2.2 Adding Proof Help To STPs

When an STP fails to verify, it could be due to a divergence between the expectations of the test writer and the current spec, indicating a spec bug, which can be confirmed with a Counterexample STP, or it could be the result of the fundamental undecidability of this type of problem. If it is the latter case, it is possible to circumvent this roadblock in some instances by adding additional proof to the STP body.

The process of proving an STP is no different than writing a proof for any lemma, but the specificity of the STP narrows the scope necessary to reason about. However, before spending the manual effort to add proof annotations to the body of an STP, the first step is to negate the conclusion, i.e. the ensures of the STP. Negating the conclusion transforms an STP into a Counterexample STP. Thus, if the proof now passes there is a clear indication of a bug.

As an example of the process of writing an STP, consider the **Correctness** STP in Lemma 5 for the incorrect sort spec from Specification 1. This STP tests that the sort spec should reject the case when the output sequence is sorted, but only contains a subset of the original input. The Sort method does not have a precondition, so any input sequence would satisfy a Usefulness STP, so this step can be skipped. Running a verifier on this STP would initially result in failing to prove the postcondition automatically. Before spending manual effort to

Table 2: Mutation Operators

Operator	Description
AOR	Arithmetic Operator Replacement
LOR	Logical Operator Replacement
ROR	Relational Operator Replacement
COI	Constant Operator Insertion
UOR	Unary Operator Replacement
ENO	Expression Negation Operator
VNOR	Variable Name Operator Replacement
SOR	Set Operator Replacement
HOR	Heap Operator Replacement

prove this STP, the first step is always to negate the postcondition (i.e. changing Line 5 to `ensures SortSpec(input, sorted)`), transforming the Correctness STP into a Counterexample STP. The attempt to prove this counterexample would now pass and serves as a concrete example of where the spec has diverged from the test writer's understanding.

4.3 Mutation Testing

If the system has a passing proof, IronSpec can leverage the proof and implementation as a reference point for further automation. IronSpec systematically generates a set of specification *mutations*, slight syntactical modifications of the original spec, but only considers those that are not weaker than the original spec. If the original proof still passes with one of these stronger specs, this alerts the developer to the original spec being weaker than the implementation; a disconnect that may hint at an unintentional spec weakness.

All mutations are subjected to three verification-assisted checks, outlined in the following subsections. Each of these checks filter the set of mutations by discarding irrelevant mutations; any discarded mutation is deemed *killed*. If a mutation is still *alive* after all three checks, it serves as a hint of a potential spec bug.

4.3.1 Mutation Generation

We generate mutations inspired by the method-level mutation operators from MuJava [44,45] and from a study that used the Z3 SMT solver to optimize a set of mutation operators based on subsumption relationships [23]. We further introduce an additional predicate-based mutation operator, Set Operator Replacement (SOR). SOR introduces mutations about set inclusion, for example, an expression, $e \in s$, would be mutated to become, $e \notin s$ or vice versa.

The IronSpec prototype is implemented in Dafny, so all mutations are applied to expressions in the Dafny AST. For Dafny expressions that reason about the heap, we introduce the Heap Operator Replacement (HOR) mutation operator, which mutates expressions containing the Dafny keyword `old`. The full list of the mutation operators used in IronSpec is shown in Table 2.

Lemma 6 *IsAtLeastAsWeak* Lemma

```
1  lemma IsAtLeastAsWeak(p:Params)
2      requires OriginalPredicate(p:Params)
3      ensures MutatedPredicate(p:Params)
```

Predicate 7 *Mutation* Target Example

```
1  predicate SafetyProperty(p:Params)
2  { SubPredA(p) ==> SubPredB(p) }
```

Each generated mutated spec is the result of IronSpec applying a single mutation operator at a time. The set of all mutated specs consists of all possible single-operator mutations for a given spec applied to each subexpression in the mutation target.

An example of one of the many possible spec mutants starting with the single postcondition from Specification 1 would be: `forall i | 0 <= i < |output| - 1 :: out[i] < out[i+1]`. This mutation is generated using the Relational Operator Replacement (ROR) mutation operator which generates mutations by replacing relational-based operators from the set of `{==, <, <=, >, >=, !=}`. One application of this mutation operator results in replacing the `<=` to a `<` in the RHS of the forall expression.

4.3.2 First Pass: Logical Redundancy

Not all mutations produced from the original spec are relevant. A spec defines a set of behaviors, and a passing proof shows that the behavior of a specific implementation is a subset of the behavior allowed by the spec. A spec that is weaker than the original would allow a larger set of implementations to satisfy this subset property. Any mutated spec that is logically equivalent or *weaker* than the original spec would not provide any new information to the tester about the current implementation and can be safely disregarded.

A mutation can cause a spec to become weaker if it weakens a postcondition or if it strengthens a precondition. Either case allows for a larger set of implementations to satisfy the spec. Therefore, for each mutation to a postcondition, IronSpec tests if the mutation *is at least as weak* as the original spec.

Definition 4.1. Given predicates S and S' with parameters p, S *is at least as weak* as predicate S' iff $\forall p. S'(p) \implies S(p)$

IronSpec captures this definition by automatically formulating Lemma 6 for the original and mutated specs. If this lemma passes, then the mutated spec must be equivalent to or weaker than the original spec, indicating that it can be killed.

Conversely, if the mutation modified a precondition, IronSpec checks the opposite, to see if the mutation *is at least as strong* as the original spec. The lemma to check if a spec is at least as strong as the original is similar to Lemma 6, but with the requires and ensures reversed.

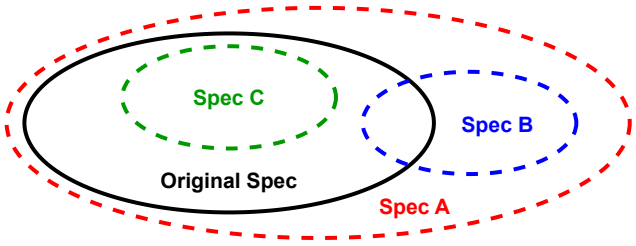

Figure 1: A mutated spec can either be strictly weaker (Spec A), strictly stronger (Spec C), logically equivalent, or partially stronger and weaker (Spec B) than the original spec.

As an example of Definition 4.1, consider Figure 1, where each circle represents the set of behaviors allowed by each respective spec. Any behavior in the circle encapsulated by the Original Spec is still inside the set of allowable behaviors of Spec A, making Spec A strictly weaker than the Original Spec and thus would be automatically discarded based on the result of Lemma 6. Both Specs B and C are not at least as weak as the Original Spec and would survive the *Logical Redundancy* pass.

The *IsAtLeastAsWeak* lemma is generated automatically to test the spec's overall safety property, rather than directly testing the mutated expression. This is important to avoid false positives. For example, consider if `SubPredB(p)` in Predicate 7 contains the mutated expression. Testing the *IsAtLeastAsWeak* Lemma with just `SubPredB(p)` may fail, indicating that this mutation is stronger than the unmodified version of `SubPredB(p)`, but this mutation may cause its caller, Predicate 7, to become weaker.

4.3.3 Second Pass: Vacuity

IronSpec's second pass aims to identify the mutations that cause vacuity [36]. For example, if a mutation to `SubPredA(p)` from Predicate 7 resulted in it always evaluating to *false*, then Predicate 7 would always be true. A vacuous spec would allow for the system's proof to pass trivially because any behavior of the system would be allowed by the vacuous spec. Checking vacuity is more complicated than purely checking if the mutated predicate is itself vacuous, as the conditions of a predicate that calls the mutated predicate, in conjunction with the mutated predicate could result in the caller becoming vacuous. This is especially important with specs that are state machines where a mutation could cause a state transition to become *false*, removing that behavior from the spec. IronSpec automatically generates a lemma to check for vacuity by considering the full call path.

4.3.4 Third Pass: Full Proof

The final check is to see if the full proof will pass with the mutated spec. In this final pass, the system is re-verified with the addition of the mutated spec to ensure that no intermediary lemmas now fail. If the full proof passes, the mutation is considered *alive* and serves as a flag to the developer to re-examine the spec.

4.3.5 Hierarchical Classification of Alive Mutations

Rather than providing the tester with a list of all alive mutations, IronSpec performs an additional pass to characterize the alive mutations, minimizing the output to the most relevant. To maximize the hint provided by an alive mutation, IronSpec evaluates the set of alive mutations to calculate a Direct Acyclic Graph (DAG) indicating which mutations are weaker or stronger in relation to one another. The DAG is structured so that each node is stronger than all of its children. The tester need only further concern themselves with the root of each connected component of this *mutation DAG*, as all children are weaker than the root in each component. This hierarchical classification is inspired by previous research to classify and remove equivalent mutations [8, 23, 46, 50].

4.4 Using Alive Mutations As Hints For STPs

When testing specs, human intuition is always the final oracle, thus, STPs are still needed to finish the investigation started by mutation testing. On their own, alive mutations only indicate a relative divergence between the spec and the implementation, but these hints can be used to write focused STPs. The relative strength of an alive mutation can be used to shrink the state space necessary to test, focusing on the divergence between the original spec and the mutation.

Armed with an alive mutation, a test writer can effectively exploit its hint by deviating from the standard guidelines of writing STPs and work **backwards** from the *spec difference*. The spec difference is the set of behaviors allowed by the original spec S and not by the alive mutation S'; essentially $S - S'$. The spec difference embodies the fundamental insight IronSpec is based on; it captures a specific disconnect between the original spec and the implementation. The behavior allowed by this reduced expression is permitted by the original spec, but not by the more restrictive mutation. The spec difference uniquely presents the tester with this subset of behavior to determine if that particular disparity is intended.

Working backwards allows the test writer to find concrete values that satisfy only the spec difference, achieving more concentrated STPs. Typically, when writing STPs, a tester starts by manually specifying values they intend for the spec to allow or disallow. This process increases in difficulty with the additional constraint that these intended values also need to satisfy the spec difference. The shift of working backwards helps to alleviate this burden.

Driven by the insight that the actual semantic change between the mutation and the original spec is small—only a single mutation—the expression of the spec difference is min-

Table 3: Spec bugs identified using the Automatic Sanity Checker. All bugs were confirmed with Counterexample STPs based on the initial hint of either MED or HIGH flags.

Bug	Specification	Method Name	Flag
TS1	TrueSat [7]	Formula Ctor	HIGH
TS2	TrueSat [7]	Start	MED
ETH1	Eth2.0 [1]	on_block	MED
AES1	AWS ESDK [3]	Encrypt	MED
AES2	AWS ESDK [3]	Decrypt	MED

imal. Working backwards allows the test writer to generate *any* input and output, and then use the verifier to check that the input and output are accepted by the expression constituting the spec difference. After generating such values, the final decision relies on the tester to decide whether the input-output pair is intended. At this stage, the existence of an *unintended* value is a counterexample to the original spec.

5 Evaluation

We evaluate the effectiveness of the IronSpec prototype by applying the Automated Sanity Checker, the STP Methodology and the automated mutation testing framework to test 14 different specifications written in Dafny [40]. Six of these specifications are produced in-house and include artificially introduced bugs, with an additional two specs containing artificial bugs described by Abreu et al. [6]. Six of the specifications are of real-world, open-source verified systems, which include: QBFT [2], DVT [4], TrueSat [7], Eth2.0 [1], daisy-nfsd [15] and an AWS Encryption SDK library [3].

When testing a spec, the ultimate oracle is the test writer, thus the final step is always to write an STP. When testing a spec, a tester could start with any aspect of IronSpec. We discuss the various facets of IronSpec by highlighting their use in supplying the initial hints used to discover *ten* spec bugs, all confirmed by their corresponding authors.

We consider all spec bugs identified and discussed in this section useful and significant; all could have allowed or did allow an incorrect implementation that would violate safety **while still allowing the proof to pass**.

The IronSpec artifact is publicly available on GitHub [5].

5.1 Automatic Sanity Checking Evaluation

Applying the Automatic Sanity Checker to the six open-source verified systems led to the discovery of five spec bugs across three specs, listed in Table 3.

Of the bugs identified, only TS1 was identified immediately with a HIGH severity flag, whereas the other four bugs were each discovered in less than an hour by writing STPs based on the hint of MED severity flags. The corresponding implementation for all five spec bugs appeared correct, but the specs were buggy, being too weak. To confirm these spec bugs, we wrote buggy implementations as Counterexample STPs for each spec and demonstrated that the proof still passed.

Spec bugs TS2, ETH1, AES1, and AES2 were identified by investigating each respective MED severity flag. We found that in these cases, the bug was a result of the output consisting of a complex datatype with many sub-fields and having postconditions concerning only a subset of these fields. This combination allows for a different implementation to update the remaining unspecified fields arbitrarily.

A MED severity flag is not as strong of a hint of a spec bug as a HIGH severity flag because *the unspecified fields may or may not be critical for safety*. The HIGH severity flag raised for TS1 was; "None of the postconditions depend on any of the input." This spec bug allows a buggy implementation to completely ignore the input values when constructing the output. The authors have remedied bugs TS1 and TS2 with a pull request we submitted.

The two bugs, AES1 and AES2, from the AWS Dafny Encryption SDK library (ESDK) [3] are both cases of spec weakness. The Dafny ESDK is a verified SDK used as a reference to build ESDKs for other languages. These bugs exist for the high-level methods of `Encrypt` and `Decrypt`. They are caused by a combination of the postconditions under-constraining the output and because the postconditions of sub-methods are not exported. This underspecification allows for the proof of trivially incorrect implementations for `Encrypt` and `Decrypt` to pass, such as returning a ciphertext or plaintext consisting of a zero byte regardless of the input.

Specs with output containing complex datatypes with many sub-fields are a critical source of spec bugs. Judging from the results of applying the Automatic Sanity Checker, under-constraining complex output can easily be overlooked. To avoid these types of spec bugs, it is vital to specify the expected values for all sub-fields of the output.

5.2 STP Methodology Evaluation

In this section we describe our experience in writing Usefulness, Correctness, and Provability STPs for specs following the Hoare-Logic style; and in the cases of identifying a spec bug, Counterexample STPs. We discuss the effectiveness of these STPs to expose differences in what behaviors the spec allows in contrast to a test writer's expectations in the presence of artificially introduced bugs.

We also discuss a case study, where following the STP methodology we discovered three spec bugs in a verified QBFT protocol. We wrote STPs for all open source specs, and when an STP failed to verify and the result conflicted with our understanding of the spec, we wrote Counterexample STPs to prove the existence of a spec bug. For brevity, the case study in this subsection focuses on the QBFT spec, where STPs acted as the initial flag that hinted at the existence of a spec bug.

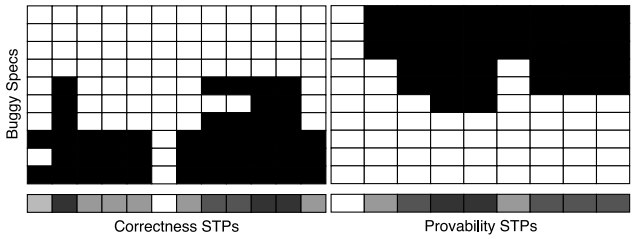

Figure 2: STP coverage for various buggy sort specs

5.2.1 STPs For Contrived Spec Bugs

We wrote STPs for five specs written in-house that follow the Hoare-Logic style. These specs include methods for finding the max of two integers (*Max*), sorting a sequence of integers (*Sort*), searching for an integer in a sequence (*Binary Search*), a cryptocurrency token creation contract (*Token-with-revert-external-wre*), and an auction contract (*SimpleAuction-with-revert-external-wre*). The *Token* and *SimpleAuction* specs were modified from Cassez et al. [14].

We wrote a total of 70 STPs for all variations of the contrived specs. The bugs introduced into the specs vary, but the resulting specs comprised an approximately equal split between too-strong and too-weak specs. Only 30% of these STPs (21) needed additional proof help and of those, each STP needed, on average, an additional 1.7 lines of proof help. The STP suite was successful in all cases in identifying introduced spec bugs—confirming the notion that a failing STP is a reliable flag suggesting a discrepancy between the intent encoded in the STPs and the spec.

Not all classes of STPs are useful in identifying all spec bugs because of the different natures of each type of STP. To demonstrate why writing a diverse suite of STPs is useful, consider Figure 2 that shows the coverage of 21 STPs across 10 different bugs for a `Sort` spec. In this experiment, the suite of STPs consisted of Correctness and Provability STPs because the different `Sort` specs were all postconditions, and Usefulness STPs are only concerned with preconditions. Each row corresponds to a different introduced bug and each column matches to a different STP. A darkened cell indicates that a specific STP successfully identified the bug after transforming the failing STP into a passing Counterexample STP. The bug was identified if a row contained a **single darkened cell**, whereas each column gives insight into the **coverage** of a single STP in identifying different bugs. The bottom row is a heat map corresponding to the ratio of bugs identified by each STP. The two STPs that uncovered zero spec bugs tested trivial enough cases such that they passed with all of the buggy specs. Depending on the spec bug, there were cases where the Correctness STPs were insufficient to identify the bug on their own, and there were cases where the same was true of the Provability STPs. So, when writing STPs it is important to have good coverage of different types of STPs to increase testing effectiveness.

Lemma 8 Simplified QBFT Provability Adversary STP

```
1 lemma AdversaryForwardMessageSTP(
2     a:Adversary,
3     a':Adversary,
4     inMsgs: set<Message>,
5     outMsgs: set<Message>)
6   requires validAdversaryConfig(a,a',inMsgs)
7   requires inMsgs == {ProposalMsg(CS1,block)}
8   requires outMsgs == {NewBlockMsg(CS1,block)}
9   ensures AdversaryNext(a, a', inMsgs, outMsgs)
```

5.2.2 STPs: QBFT Case Study

Writing STPs for the QBFT spec [2], a Byzantine fault-tolerant consensus protocol used in the Ethereum ecosystem [47], led to the discovery of three spec bugs confirmed by the authors. The spec in this system consists of a single safety property, *Blockchain Consistency*, and the environment, which includes the high-level distributed system, the network, and the adversary which are all modeled as a state machine. Upon manual inspection of these STPs, we found that the adversary spec was *incomplete*, based on our understanding of what the adversary spec should be. The overall proof still passed even in the presence of these three bugs because they essentially cancel each other out; two making the spec weaker than it should be, and the other making the spec stronger.

The first bug identified in the adversary spec was an example of the spec being too strong; limiting the actions of what an adversarial node *should* be able to do. The initial hint indicating the possibility of this case was provided by failing Provability STPs. The initial reason for writing Provability STPs was to answer if the adversary spec was too strong; which is answered by the general form of Provability STPs. An overly restricted adversary model would weaken and perhaps invalidate the guarantees of the overall proof. Following the guidelines for writing STPs in Section 4.2.2, negating the conclusion of the failing Provability STPs led to the discovery of a passing Counterexample STP. Lemma 8 is a simplified example of such a failing Provability STP.

The failing simplified STP in Lemma 8 hints at the fact that the ability of the adversary to extract signed data structures is unnecessarily restricted. In the system model for QBFT, and other Byzantine fault-tolerant consensus protocols, a Byzantine node should be allowed to behave arbitrarily while not violating cryptographic assumptions. In this QBFT spec, adversaries are only able to extract and forward `CommitSeals(CS)` from a subset of received message types. The STP in Lemma 8 specifies the behavior of an adversary node receiving a `Proposal` message signed with a quorum of `CS1`, and constructing and sending a *NewBlock* message containing the `block` and `CS1` data structures copied from the `Proposal` message. The postcondition for this STP stipulates that this scenario constitutes a valid state transition from state `a` to state `a'`. After observing that this STP failed to immediately

Lemma 9 Simplified QBFT Adversary Correctness STP

```
1 lemma AdversaryForgeMessageSTP(
2     a:Adversary,
3     a':Adversary,
4     inMsgs: set<Message>,
5     outMsgs: set<Message>)
6   requires validAdversaryConfig(a,a',inMsgs)
7   requires outMsgs == {ProposalMsg(CS2)}
8                       // forged msg
9   ensures !AdversaryNext(a,a',inMsgs,outMsgs)
```

verify, negating the conclusion to, `!AdversaryNext(a, a', inMessages, outMessages)`, resulted in the proof passing for this transformed counterexample STP.

While investigating the implications of the behavior in the failing STP, we modified the adversary spec, weakening it to allow an adversary to forward `CS1` regardless of what message first contained it. After making this change, the full system's safety proof failed. To differentiate the proof failure from a now incomplete lemma, we constructed and proved a concrete counterexample resulting in a violation of the system's safety property, confirmed by the authors.

The second bug in the adversary spec is an example of the spec being too weak. This weakness is the reason why we can show a concrete counterexample to safety after addressing the first spec bug. The spec allows an adversarial node to send a `Proposal` message containing a `block` data structure with arbitrary values, including using the `CommitSeals` of honest nodes even if the adversary had not previously received such `CommitSeals` in previously received messages. `CommitSeals` are only used to make a final decision of committing a block, but this weakness in the spec allows an adversary to propose new blocks containing `CommitSeals` as if from honest nodes. This behavior allows an adversary to send a message that appears to be signed by an honest node, violating the security assumptions made by the QBFT system model. The STP in Lemma 9 is a simplified version of the Correctness STP used to discover this spec bug.

The third bug identified is related to the previous bug and is concerned with the underspecified spec of the function *getNewBlock()*. This function is empty-bodied and only contains the spec. Due to the underspecification of this function, a caller of this function, including an honest node can immediately send a message, such as a `Proposal` message, containing a full quorum of commit seals. If a buggy implementation is provided for this function, it too, could lead to a violation of the safety property.

5.2.3 STP Discussion

STPs enable fine-grain testing of specs and have been effective at helping to identify all ten spec bugs in the six open-source verified systems. By leveraging the insight that writing proofs for specific values is easier than a general proof, the manual effort required to write STPs remains minimal.

In the QBFT spec the presence of three spec bugs, two manifesting as a weakness in the spec and the other counteracting the first by overly restricting the adversary, makes manually or automatically identifying these bugs extremely difficult. Following the STP testing methodology, we efficiently, and without being experts in the system, identified these disconnects between what was written in the spec and our understanding of the intent of the spec.

5.3 Mutation Testing Evaluation

We applied IronSpec's automatic mutation testing to a set of six in-house specs, the two spec examples from Abreu et al. [6], and the spec of six open-source verified codebases. The evaluation attempts to answer how prevalent *alive* mutations are in specs, and how useful the provided hints are in assisting to identify spec bugs.

All mutation testing experiments were performed on a cluster of 21 servers where each node was equipped with two Intel E5-2660 v2 10-core CPUs at 2.20 GHz and with 256GB ECC Memory. In each experiment, one root node would create all mutations and send all subsequent verification requests in each stage of the mutation testing process to be processed in parallel at the other 20 nodes in the cluster using Dafny version 3.8.1. The results from running IronSpec can be found in Table 4 and are further explained in the following subsections.

5.3.1 In-House Specifications

In addition to the five in-house specs introduced in Section 5.2.1, we applied mutation testing on a simple key-value store state machine spec.

The top half of Table 4 contains the experimental results of running mutation testing on the in-house specs. Each buggy spec was tested with a correct implementation (C) and an incorrect implementation (I). Mutation testing all in-house specs with a correct spec and a correct implementation resulted in no alive mutations.

Mutation testing identified relevant alive mutations, regardless of whether the implementation is correct. For all six incorrect specs, mutation testing resulted in helpful alive mutations. The only exception is Sort (C), whose implementation contained additional loop invariants that caused the proof to fail when using weaker preconditions. In all other cases alive mutations were useful hints in manually identifying a weakness in the spec.

5.3.2 Alive Mutations in Open Source Systems

The Div and NthHarmonic specs are simple buggy specs introduced by Abreu et al. [6], where the authors proposed initial techniques to repair simple spec errors in Dafny. The alive mutations IronSpec found for these specs coincide with

Table 4: Results from running IronSpec's automatic mutation testing. In-House, buggy, specs marked with "(C)" correspond to experiments with a buggy spec but a correct implementation, whereas "(I)" indicates a buggy spec with an incorrect implementation. The Predicate Name is the specific mutation target within a spec. Spec LOC is the size of the mutation target, and Proof/Impl LOC is the size of the full end-to-end implementation and proof. Mutations are the total number of mutations generated, Alive Mutations indicate the number of alive mutations after all three passes and hierarchy classifications.

	Specification	Predicate Name	Spec LOC	Proof/Impl LOC	# Mutations	# Alive Mutations	Time
In-House Specs	**Max (C)**	maxSpec	2	5	80	1	11.3s
	Max (I)			7		4	7.5s
	Sort (C)	sortSpec	1	55	50	0	4.5s
	Sort (I)			4		1	7.3s
	Binary Search (C)	searchSpec	4	31	170	1	10.4s
	Binary Search (I)			18		2	24.3s
	KV SM (C)	Query Op	4	187	37	7	21s
	KV SM (I)					7	28.8s
	Token-wre (C)	GInv	1	87	13	1	7.8s
	Token-wre (I)			91		1	7.8s
	SimpleAuction-wre (C)	GInv	9	181	187	3	15.25s
	SimpleAuction-wre (I)					3	15.5s
Open-Source Specs	**Div**	Div	3	14	50	3	3.5s
	NthHarmonic	NthHarmonic	1	4	11	2	3s
	QBFT	NetworkInit	3	15071	44	3	80 min
	QBFT	AdversaryNext	48		197	7	162 min
	QBFT	AdversaryInit	3		35	4	80 min
	Distributed Validator	AdversaryNext	23	24747	110	7	191 min
	daisy-nfsd	GETATTR	4	18	35	1	4.3 min
	daisy-nfsd	WRITE	7	54	119	3	4.6 min

the conclusions made by Abreu et al. in demonstrating that these specs are buggy by being too weak.

QBFT Of the 44 generated mutants for the initial state of the network state machine spec, `NetworkInit`, three mutations remained *alive* as the roots of their respective components in the *mutation DAG*. Upon manual inspection of the surviving mutants, the spec differences all referenced an aspect of the Network's state that was never mentioned elsewhere. Thus, any value for part of the state would be considered "safe". These mutations do not imply the existence of a bug, but neither are they strictly false positives; rather they are examples of spec bloat. These alive mutations should still serve as flags to the developer, forcing them to answer the question of whether this state is needed, and if so why are these parts of the state not referenced?

The alive mutations for the `AdversaryNext` and `AdversaryInit` predicates, both parts of the adversary state machine spec can be considered false positives. The alive mutations were all *stronger* mutations, but it is always safe to restrict the actions allowed by an adversarial node. Some alive mutations implied that the proof would still pass with no adversaries in the system, or only taking trivial actions. This observation led us to question and then to test with STPs if the adversary spec was initially more restricted than it should be, leading to the bugs discussed in Section 5.2.2.

DVT The Distributed Validator Technology Protocol (DVT) spec and proof [4] captures the behavior of an Ethereum Validator, where a group of nodes coordinates to perform the Ethereum validator duties. The DVT spec consists of the desired *non-slashable attestation* property and the environment, with the latter defined as the high-level distributed system, an adversary, and the network. All aspects of the environment are modeled as state machine specs. The *non-slashable attestation* property ensures that the system avoids committing a slash-able offense and produces valid attestations.

Applying mutation testing to the `AdversaryNext` predicate in the adversary spec resulted in seven *alive* mutations. One of the mutations was a false positive. Three of the mutations hinted towards a limitation of the messages allowed to be sent by an adversary, leading to a similar discovery as in the first QBFT bug. The remaining three alive mutations were concerned with the creation of attestations. This weakness lies in the spec's lack of specificity regarding the attestations an adversary can create. Armed with this observation, we show with a counterexample that this weakness could lead to a safety violation.

daisy-nfsd Applying the mutation framework to daisy-nfsd's [15] top-level NFS API spec resulted in alive mutations in two different methods' specs, `GETATTR` and `WRITE`. These mutations hint at the same spec weakness that both methods contain; one that would allow for a different trivial implementation to always return an error. This bug was confirmed by the authors as a known issue in their spec.

5.3.3 Combining STPs With Mutation Testing Hints

An alive mutation is a compelling hint that the spec may be weaker than intended, but it is just a hint; writing STPs (Section 4.4) is always the final step in testing. Consider the DVT spec bug from Section 5.3.2. The original mutation target predicate is non-trivial and consists of 22 lines including multiple quantified conjuncts. Working backwards from the alive mutations and focusing only on the expression derived by the difference between the original spec and an alive mutation, resulted in shrinking the 22-line predicate into only a single conjunct. Writing STPs concerning this single conjunct is much more tractable than writing STPs for the entire predicate. The tradeoff of the slightly increased manual effort to calculate the simplified expression and writing STPs concerning it outweighs the effort needed to consider the entire spec.

5.3.4 Mutation Testing Discussion

Mutation testing supplied the hints that led to the discovery of spec bugs in two verified codebases. These results exemplify the usefulness of adding automation to search for disconnects between the implementation and the spec. The insight of identifying tangible differences as potential areas of disconnected intent is a beneficial hint that can be leveraged to identify spec bugs. The results in Table 4 demonstrate that even with a small set of mutations, we were successful in identifying spec weaknesses.

The large increase in execution time of running mutation testing taken between different specs can be attested to the varying sizes of the full system proof and the time that it takes to verify the entire proof with the mutations that survive the first two passes. For instance, even running the full end-to-end proof once of the unmodified QBFT system can take approximately an hour to complete. The cost of running IronSpec on a large verified system is worth the execution time to debug a potential spec bug.

The results of testing specs with mutation testing demonstrate the effectiveness of this approach, but we did find that not all alive mutations led to the discovery of spec bugs. While the possibility of discovering false positive alive mutations exists, all cases were quickly diagnosed. Of the 61 alive mutations identified across all tested specs, we consider 13 to be false positives, because the spec weaknesses they hint at were deemed intended. All mutations for QBFT's `AdversaryNext` and `AdversaryInit` were considered false positives. A single alive mutation in the set of alive mutations for both DVT `AdversaryNext` and daisy-nfsd `WRITE` were also characterized as false positives. The one false positive mutation found in DVT `AdversaryNext` was classified as such because it would have only allowed the adversary to make attestations already created, which would not have led to any unintended, incorrect behavior.

Verified methods that modify ghost state are at a higher risk of mutation testing producing false positives. Ghost state is only maintained for the sake of the proof, and often, underconstrained postconditions related to ghost state would not result in a buggy implementation. The false positive mutation in daisy-nfsd's `WRITE` method hinted towards underconstrained ghost state that was modified in the method's implementation. Nevertheless, the daisy-nfsd authors confirmed that a different implementation, which modified this ghost state differently, would not lead to a safety violation or break the proof. However, they did acknowledge that this weakness was not immediately apparent.

Rather than finding false positives, it is also possible, especially with larger systems, for no alive mutations to be identified. For QBFT, DVT, and daisy-nfsd there were other spec mutation targets we applied IronSpec to, which resulted in no alive mutations. For example, in both QBFT and DVT, the alive mutations identified were part of the trusted specified environment, whereas no alive mutations were found for their respective safety properties, *Blockchain Consistency* and *non-slashable attestation*.

The IronSpec prototype takes the first steps to bring automation and structure to testing specs. The prototype targets Dafny specs, but the conceptual techniques are not tied to Dafny.

5.4 Amount Of Manual Effort Required

In the same way testing traditional software systems requires developer effort, testing specifications does too. IronSpec provides a framework and automation aid to help developers in this endeavor. If one is willing to spend the effort to verify their system, it is worth spending a few additional hours to gain confidence in proving the intended property. While manual effort is unavoidable, this effort can be greatly reduced as the automation of IronSpec helps to focus the developer's attention on a few potentially problematic aspects of the spec.

The majority of manual effort we expended in applying IronSpec was spent on understanding each system well enough to interpret the hints from the automation of IronSpec and to write appropriate STPs. Even so, the amount of manual effort expended remained relatively low despite not having specific expertise in each system. For example, the specification bug identified in daisy-nfsd took approximately 1-2 hours to determine from first examining the code base and running IronSpec to confidently identifying the spec bug. When test-

ing daisy-nfsd, it took minutes to run the mutation framework, and the rest of the time was spent comprehending the hint of the alive mutation, writing STPs, and understanding the underlying system well enough to determine if the flagged behavior was intended. In conversations with the daisy-nfsd authors, they admitted that this spec bug was subtle. Familiarity and expertise in a system and its spec will only help to further reduce the necessary manual effort.

Writing STPs for complex systems takes more effort than for simple examples. Yet, the limited scope of writing STPs with concrete values drastically limits the size of any potential additional proof annotations needed for those STPs in comparison to what would be necessary for proof of the unconstrained behavior. The effort of writing STPs varied per the complexity of the system. Writing a comprehensive suite of STPs ranged from a few hours to multiple days worth of effort for the larger QBFT and DVT specs. Writing-focused STPs based on alive mutations ranged from tens of minutes to hours per alive mutation.

6 Related Work

Kemmerer [33] first identified the potential benefits of testing specifications. Kemmerer proposed a technique based on symbolic execution to check if a spec satisfied the English-based functional requirements. Since then, several studies have proposed techniques to test informal user requirements [13, 18, 35, 42, 43].

The closest related previous work is the study by Fonseca et al. [22], which manually and painstakingly identified weaknesses in verified codebases, including two spec bugs. Other works have also begun to apply more structured approaches to increase reliability in formal methods. Kupfeman [36] discussed the possible advantages of vacuity and coverage checks for temporal-logic model-checking tools. Inspired by vacuity testing, and the concept of *unit proofs* from Chong et al. [16], Priya et al. [52] performed a case study of some AWS verified libraries, uncovering some hidden bugs. Bernardi et al. [11] also identified formal specifications as a weak point in the verification process, and proposed to reuse specifications once correct, for smart contracts. Le Traon et al. [38] even discussed the notion of applying a mutation analysis to Eiffel contracts.

With verification becoming more commonplace and the discovery of spec bugs in verified systems, a few, mostly manual efforts have attempted to identify spec bugs. The 2022 Notional Finance bug found in verified code inspired Certora to investigate ways to introduce testing into the verification process [49]. Recently, Abreu et al. [6] proposed initial efforts in using the dynamic invariant inference tool *Daikon* [20] to aid in automatically repairing specifications. When faced with a failing proof, their prototype assumes that the implementation is correct, and uses the implementation to generate test cases for the spec. Any failing tests present an opportunity to attempt to fix the spec by suggesting strengthening or weakening modifications. Of course, this approach only works if the implementation is correct, which partially defeats the purpose of performing verification in the first place.

Testing and formal methods share a close relationship and a common goal. Often, rather than questioning specs, developers have relied on specs or other formal methods to assist in testing traditional software [11, 16, 27, 39]. Works concerned with the Oracle problem [9, 12] have often utilized specs thus. There has even been work to test verification tools [30].

7 Conclusion

The correctness of specifications is the rock upon which the entire edifice of formal verification is built. As formal verification becomes increasingly popular, it is imperative that the foundation be as solid as possible.

This work proposes IronSpec, a systematic framework of manual and automated approaches to aid developers in finding bugs in their specs. We show how IronSpec was used to identify a number of subtle bugs in the specs of open-source codebases, without requiring copious amounts of expertise on the proven system. We believe IronSpec is a necessary step forward towards writing correct software.

Acknowledgements

We want to thank Shuangyu Lei for her work on an early version of the mutation testing framework. We thank the anonymous OSDI reviewers and our shepherd, Baptiste Lepers, for their insightful and useful feedback that we used to improve this paper. This work was supported by National Science Foundation grants CCF-2118512 and CCF-2018915.

References

[1] Eth2.0-dafny. https://github.com/Consensys/eth2.0-dafny/tree/master, 2021.

[2] Qbft formal specification and verification. https://github.com/Consensys/qbft-formal-spec-and-verification, 2021.

[3] Aws encryption sdk for dafny. https://github.com/aws/aws-encryption-sdk-dafny, 2023.

[4] Formal verification of the distributed validator technology protocol. https://github.com/Consensys/distributed-validator-formal-specs-and-verification, 2023.

[5] Ironspec. https://github.com/GLaDOS-Michigan/IronSpec, 2024.

[6] A. Abreu, N. Macedo, and A. Mendes. Exploring automatic specification repair in dafny programs. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, pages 105–112. IEEE, 2023.

[7] C.-C. Andrici and Ş. Ciobâcă. Verifying the dpll algorithm in dafny. *arXiv preprint arXiv:1909.01743*, 2019.

[8] D. Baldwin and F. Sayward. *Heuristics for determining equivalence of program mutations*. Yale University, Department of Computer Science, 1979.

[9] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5):507–525, 2014.

[10] K. Beck. *Test driven development: By example*. Addison-Wesley Professional, 2022.

[11] T. Bernardi, N. Dor, A. Fedotov, S. Grossman, N. Immerman, D. Jackson, A. Nutz, L. Oppenheim, O. Pistiner, N. Rinetzky, et al. Wip: Finding bugs automatically in smart contracts with parameterized invariants. *Retrieved July*, 14:2020, 2020.

[12] M. Böhme, C. Cadar, and A. Roychoudhury. Fuzzing: Challenges and reflections. *IEEE Softw.*, 38(3):79–86, 2021.

[13] M. Brockmeyer. Using modechart modules for testing formal specifications. In *Proceedings 4th IEEE International Symposium on High-Assurance Systems Engineering*, pages 20–26. IEEE, 1999.

[14] F. Cassez, J. Fuller, and H. M. A. Quiles. Deductive verification of smart contracts with dafny. In *International Conference on Formal Methods for Industrial Critical Systems*, pages 50–66. Springer, 2022.

[15] T. Chajed, J. Tassarotti, M. Theng, M. F. Kaashoek, and N. Zeldovich. Verifying the {DaisyNFS} concurrent and crash-safe file system with sequential reasoning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 447–463, 2022.

[16] N. Chong, B. Cook, K. Kallas, K. Khazem, F. R. Monteiro, D. Schwartz-Narbonne, S. Tasiran, M. Tautschnig, and M. R. Tuttle. Code-level model checking in the software development workflow. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, pages 11–20, 2020.

[17] E. Daka and G. Fraser. A survey on unit testing practices and problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 201–211. IEEE, 2014.

[18] G. De Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. Automated abstractions for contract validation. *IEEE Transactions on Software Engineering*, 38(1):141–162, 2010.

[19] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[20] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Science of computer programming*, 69(1-3):35–45, 2007.

[21] R. W. Floyd. Assigning meanings to programs. *Program Verification: Fundamental Issues in Computer Science*, pages 65–81, 1993.

[22] P. Fonseca, K. Zhang, X. Wang, and A. Krishnamurthy. An empirical study on the correctness of formally verified distributed systems. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 328–343, 2017.

[23] R. Gheyi, M. Ribeiro, B. Souza, M. Guimarães, L. Fernandes, M. d'Amorim, V. Alves, L. Teixeira, and B. Fonseca. Identifying method-level mutation subsumption relations using z3. *Information and Software Technology*, 132:106496, 2021.

[24] P. Hamill. *Unit test frameworks: tools for high-quality software development*. " O'Reilly Media, Inc.", 2004.

[25] T. Hance, A. Lattuada, C. Hawblitzel, J. Howell, R. Johnson, and B. Parno. Storage systems are distributed systems (so verify them that way!). In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 99–115, 2020.

[26] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. Ironfleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 1–17. ACM, 2015.

[27] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, et al. Using formal specifications to support testing. *ACM Computing Surveys (CSUR)*, 41(2):1–76, 2009.

[28] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[29] S. Igarashi, R. L. London, and D. C. Luckham. Automatic program verification i: A logical basis and its implementation. *Acta Informatica*, 4(2):145–182, 1975.

[30] A. Irfan, S. Porncharoenwase, Z. Rakamarić, N. Rungta, and E. Torlak. Testing dafny (experience paper). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 556–567, 2022.

[31] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2010.

[32] J. Kang, Y. Kim, C.-K. Hur, D. Dreyer, and V. Vafeiadis. Lightweight verification of separate compilation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 178–190, 2016.

[33] R. A. Kemmerer. Testing formal specifications to detect design errors. *IEEE transactions on software engineering*, (1):32–43, 1985.

[34] U. Kirstein. Post-mortem analysis of the notional finance vulnerability — a tautological invariant, January 2022. [Online; posted 17-January-2022].

[35] A. Knüppel, L. Schaer, and I. Schaefer. How much specification is enough? mutation analysis for software contracts. In *2021 IEEE/ACM 9th International Conference on Formal Methods in Software Engineering (FormaliSE)*, pages 42–53. IEEE, 2021.

[36] O. Kupferman. Sanity checks in formal verification. In *International Conference on Concurrency Theory*, pages 37–51. Springer, 2006.

[37] L. Lamport. Specifying systems: the tla+ language and tools for hardware and software engineers. 2002.

[38] Y. Le Traon, B. Baudry, and J.-M. Jézéquel. Design by contract to improve software vigilance. *IEEE Transactions on Software Engineering*, 32(8):571–586, 2006.

[39] O. Legunsen, W. U. Hassan, X. Xu, G. Roşu, and D. Marinov. How good are the specs? a study of the bug-finding effectiveness of existing java api specifications. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 602–613, 2016.

[40] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.

[41] M. Lesani, C. J. Bell, and A. Chlipala. Chapar: certified causally consistent distributed key-value stores. *ACM SIGPLAN Notices*, 51(1):357–370, 2016.

[42] M. Li and S. Liu. Reviewing formal specification for validation using animation and trace links. In *2014 21st Asia-Pacific Software Engineering Conference*, volume 1, pages 263–270. IEEE, 2014.

[43] S. Liu, J. A. McDermid, and Y. Chen. A rigorous method for inspection of model-based formal specifications. *IEEE Transactions on Reliability*, 59(4):667–684, 2010.

[44] Y.-S. Ma and J. Offutt. Description of mujava's method-level mutation operators. *Update*, 2016.

[45] Y.-S. Ma, J. Offutt, and Y. R. Kwon. Mujava: an automated class mutation system. *Software Testing, Verification and Reliability*, 15(2):97–133, 2005.

[46] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Jozala. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. *IEEE Transactions on Software Engineering*, 40(1):23–42, 2013.

[47] H. Moniz. The istanbul bft consensus algorithm. *arXiv preprint arXiv:2002.03613*, 2020.

[48] G. J. Myers, C. Sandler, and T. Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[49] S. Phipathananunth. Using mutations to analyze formal specifications. In *Companion Proceedings of the 2022 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, pages 81–83, 2022.

[50] A. V. Pizzoleto, F. C. Ferrari, J. Offutt, L. Fernandes, and M. Ribeiro. A systematic literature review of techniques and metrics to reduce the cost of mutation testing. *Journal of Systems and Software*, 157:110388, 2019.

[51] V. R. Pratt. Semantical considerations on floyd-hoare logic. In *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*, pages 109–121. IEEE, 1976.

[52] S. Priya, X. Zhou, Y. Su, Y. Vizel, Y. Bao, and A. Gurfinkel. Verifying verified code. *Innovations in Systems and Software Engineering*, 18(3):335–346, 2022.

[53] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 357–368, 2015.

[54] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 283–294, 2011.