A Framework for Debugging Automated Program Verification Proofs via Proof Actions

CAV Artifact Evaluation ★ Available

Chanhee Cho, Yi Zhou, Jay Bosamiya, and Bryan Parno

Carnegie Mellon University, Pittsburgh, PA, USA
`{chanheec,yeet,jaybosamiya,parno}@cmu.edu`

CAV Artifact Evaluation ★ ★ ★ Reusable

Abstract. Many program verification tools provide automation via SMT solvers, allowing them to automatically discharge many proofs. However, when a proof fails, it can be hard to understand why it failed or how to fix it. The main feedback the developer receives is simply the verification result (i.e., success or failure), with no visibility into the solver's internal state. To assist developers using such tools, we introduce ProofPlumber, a novel and extensible *proof-action* framework for understanding and debugging proof failures. Proof actions act on the developer's *source-level proofs* (e.g., assertions and lemmas) to determine why they failed and potentially suggest remedies. We evaluate ProofPlumber by writing a collection of proof actions that capture common proof debugging practices. We produce 17 proof actions, each only 29–177 lines of code.

1 Introduction

Software verification tools typically fall into two camps. The first camp relies on interactive proof assistants, such as Coq [1] or Lean [2]. These proof assistants show the developer the current *proof state*; i.e., the proof goal and all hypotheses in scope. Developers then complete a proof by invoking tactics (user-developed programs, e.g., in Ltac [3], that manipulate the proof state), viewing the effects of the tactics in sophisticated IDEs. By default, these tools typically provide less automation and generally require the developer to manipulate the proof at a low-level. Some tactics (e.g., `sledgehammer` [4,5,6] or `crush` [7]) provide considerably more automation, but as a result, their strengths and weaknesses tend to match those of tools in the second camp.

In the second camp are languages and tools designed specifically for automated program verification, e.g., Spec# [8], ESC/Java[9], VCC [10], Dafny [11], F* [12], Viper-based languages [13,14,15,16], and Verus [17]. With these tools, developers add specifications to their programs, and the tool attempts to automatically prove (potentially with help from developer-provided *source-level* assertions/lemmas) that the spec holds. Typically this is done by reducing the problem to a logical formula (e.g., via a weakest precondition calculus [18]) that can be checked by an SMT solver [19,20,21,22,23]. Because SMT solvers can discharge many proofs fully automatically, they enable developers to take large, logical steps.

However, when a proof fails, it can be hard to understand why it failed or how to fix it. The main feedback the developer receives is simply the verification result (i.e., success or failure), with no visibility into the solver's internal state. Exposing such internal state in a useful manner is challenging, since an SMT solver might generate millions of terms during its proof search. Hence, unlike in interactive proof assistants, it can be difficult to answer questions like "What propositions has the solver proven at this program point?" or "What additional facts are needed for the solver to complete the proof?". Indeed, because verification is generally undecidable, the developer does not even know initially if the proof failed because it is invalid, or because the tool's automation is incomplete. As a result, when developing verified code in automated program verification languages, significant time goes into trying to figure out why the code is failing to verify and what needs to be done to address the failure.

To assist developers using automated program verification techniques, we introduce ProofPlumber, a novel *proof-action* framework for understanding and debugging proof failures. Unlike tactics, which are primarily used to manipulate (low-level) *proof state*, proof actions act on the developer's *source-level proofs* (e.g., assertions and lemmas) to determine why they have failed and potentially suggest remedies. ProofPlumber comes prepackaged with a set of proof actions that automate a wide variety of standard proof debugging techniques, and it is also fully extensible, so that as developers devise new or project-specific techniques, those can be automated as well. Ultimately, we hope that ProofPlumber-style proof actions will reduce tedium for experienced developers and help bootstrap developers just starting to write verified programs.

We implement ProofPlumber[1] in the context of Verus [17], a verification-oriented language for Rust [24,25]. ProofPlumber offers three categories of APIs (with a total of 36API calls) for manipulating source-level Verus proofs. In §4, we demonstrate ProofPlumber's expressivity and extensibility by implementing a collection of 17 proof actions, each of which requires only 29–177 lines of code. We also show that ProofPlumber reduces Verus' trusted computing base (TCB).

2 Proof Debugging Considered Painful

2.1 Background on Automated Program Verification in Verus

ProofPlumber is implemented for Verus [17], an SMT-based verification language for formally verifying Rust programs. The basic syntax of Verus, shown in Figure 1, is similar to that used in most languages for automated program verification. The `requires` clause on line 8 describes the function's precondition, and the `ensures` clause on line 9 describes its postcondition. The `assert` statement on line 20 is a static check performed by an SMT solver; when a proof fails, proof engineers add assertions to a program to extract information from the solver.

2.2 Examples of Proof Debugging

Today, developers typically debug their failed proofs by manually adding source-level assertions to their program. Such an assertion has no effect on the executable

[1] ProofPlumber's code and proof actions are available as open source at https://github.com/verus-lang/verus-analyzer.

code; instead, it breaks the proof into smaller steps, and it indirectly queries the solver's internal state, extracting two pieces of information: **(1)** Is the code's precondition sufficient to prove the new assertion? (determined by whether the assertion verifies); and **(2)** Is the assertion sufficient to prove the code's postcondition? (determined by whether the postcondition now verifies).

```
spec fn fibo(n: nat) -> nat {
 if n == 0 { 0 } else if n == 1 { 1 }
 else { fibo(n - 2) + fibo(n - 1) }
}

// first attempt
proof fn fibo_is_monotonic(i: nat, j: nat)
 requires i <= j,
 ensures fibo(i) <= fibo(j), // fails
{
 if i < 2 && j < 2 {}
 else if i == j {}
 else if i == j - 1 {
  fibo_is_monotonic(i, j - 1);
 } else {
  fibo_is_monotonic(i, j - 1);
  fibo_is_monotonic(i, j - 2);
 }
 // Debug by copying the failing ensures
 assert(fibo(i) <= fibo(j)); // fails
}
```

```
// second attempt
proof fn fibo_is_monotonic(i: nat, j: nat)
 requires i <= j,
 ensures fibo(i) <= fibo(j),
{
 if i < 2 && j < 2 {
  assert(fibo(i) <= fibo(j)); // succeeds
 } else if i == j {
  assert(fibo(i) <= fibo(j)); // succeeds
 } else if i == j - 1 {
  fibo_is_monotonic(i, j - 1);
  assert(fibo(i) <= fibo(j)); // fails
 } else {
  fibo_is_monotonic(i, j - 1);
  fibo_is_monotonic(i, j - 2);
  assert(fibo(i) <= fibo(j)); // succeeds
 }
 assert(fibo(i) <= fibo(j)); // fails
}
```

Fig. 1. "Stepping up" an assertion to identify the failing case.

Effective assertion choice and placement is a key part of the proof engineering process. Choosing the wrong assertion or inserting it in the wrong place sheds little light on the cause of the proof failure. Further, a single assertion is seldom sufficient; instead, multiple iterations are required to further break down the proof goal, until either the proof succeeds, or the developer determines the key missing facts the prover needs (or finds a bug in the code).

Unfortunately, assertion-based debugging is an arcane art. Beginners find it hard to understand what assertions to add, where to add them, and how to use them to break down the proof goal. We frequently see beginners become stuck randomly adding assertions that do not improve their understanding of the proof failure. Even for experts, assertion-based debugging is tedious and error prone.

We illustrate the challenges with two simplified examples. In real verification projects, the properties involved are much larger and more complex, making the manual manipulation of source-level assertions a remarkably laborious, error-prone process. While our examples are based on Verus, manipulating source-level proofs is the standard debugging technique in automated program verification; see, for example, Dafny's manual assertion guide [26], F*'s guide [27], and various proof debugging examples on StackOverflow [28,29].

The left side of Figure 1 presents a failing proof for `fibo_is_monotonic`; the postcondition does not hold. An experienced proof engineer typically starts by copying the failing postcondition to the end of the function, so that they can manipulate it for further debugging. When that assertion fails (as expected), the proof engineer still does not know which of the four branches is causing the proof failure, so she then copies the same assertion into each branch, as shown on the right side of the figure. When she calls the verifier again, she observes that the assertion in the third branch fails and can start fixing it.

```
 1 // easy proof for unbounded integers
 2 proof fn mul_inequality(x:int, y:int, z:int)
 3   requires x <= y && 0 < z
 4   ensures x * z <= y * z
 5 {...}
 6
 7 // first attempt for bounded integers
 8 proof fn mul_inequality_bounded(
 9   x: int, xbound: int, y: int, ybound: int)
10   requires x < xbound && y < ybound
11         && 0 <= x && 0 <= y
12   ensures x * y <= (xbound - 1) * (ybound - 1)
13 {
14  // precondition fails for both calls
15  mul_inequality(x, xbound-1, y);
16  mul_inequality(y, ybound-1, xbound-1);
17 }
```

```
18 // second attempt for bounded integers
19 proof fn mul_inequality_bounded(
20   x: int, xbound: int, y: int, ybound: int)
21   requires x < xbound && y < ybound
22         && 0 <= x && 0 <= y
23   ensures x * y <= (xbound - 1) * (ybound - 1)
24 {
25  // Step #2: split the assertion into pieces
26  assert(x <= xbound - 1); // succeeds
27  assert(0 < y);           // fails
28  // Step #1: inline precondition for first call
29  assert(x <= xbound -1 && 0 < y); // fails
30  mul_inequality(x, xbound-1, y);
31  mul_inequality(y, ybound-1, xbound-1);
32 }
```

Fig. 2. Inlining a precondition.

Figure 2 illustrates how proof engineers start debugging when the verifier tells them that a function's precondition fails to hold. In this example, after proving `mul_inequality` for unbounded integers, the proof engineer tries to prove a similar property (`mul_inequality_bounded`) for bounded integers. After drafting the proof on the left side, she learns that the precondition on line 3 fails at both callsites (lines 15 and 16). She then copies over the failing precondition, and replaces the lemma's formal arguments with the concrete values at the call site, as in line 29. When she runs the verifier again, the added assertion fails as expected. To learn which part of the conjunction fails, she splits the assertion into two separate assertions (lines 26 and 27). After running the verifier yet again, the proof engineer identifies the cause of the proof failure: `0 < z` does not hold.

2.3 Automated Proof Debugging with Proof Actions

We observe that much of the effort of proof debugging is consumed by steps that can clearly be automated. We therefore propose *proof actions*,[2] which can automatically transform a program and its proof. With proof actions, we can capture in an automated manner the existing proof-debugging practices used by experts. This reduces their tedium and transcription errors, and it enables the "wisdom" that these practices represent to be easily handed to new developers.

For example, the §2.2 examples can be automated with these proof actions:

Weakest Precondition Step. This proof action moves an assertion above the statement that precedes it: in the case of a branch statement, it moves the assertion to the end of each of the branch statements. More generally, the proof action implements the rules of the weakest precondition calculus.

Insert Failing Preconditions. When preconditions cannot be established at a call site, this proof action inlines the precondition in the caller's context.

[2] Inspired by the code actions supported by the Language Server Protocol (LSP) [30].

2.4 Challenges with Automatic Code Transformation

Automating proof debugging through proof actions is achievable but challenging, since it is hard to write programs that automatically and correctly transform programs [31,32,33]. Such transformations require information about the program's control flow, the types of expressions, and the definitions of variables, functions, and types. To act on this knowledge, we need easy and intuitive ways to manipulate source-level proofs. Finally, to understand proof failures, we need to interact with the verifier in an automated way. Existing program verifiers lack support for one or more of these features.

Furthermore, providing a fixed set of proof actions is insufficient. Since program verification is still rapidly evolving, proof engineers will need to add new proof actions. Moreover, different verification projects come with different proof styles, and hence each project may benefit from project-specific proof actions.

3 ProofPlumber: An Extensible Proof Action Framework

We start with the design of ProofPlumber's API, which provides the functionality for developing proof actions. We then discuss the API's implementation.

At a high level, ProofPlumber provides APIs that allow a proof action to **(1)** lookup context information (e.g., types and definitions), **(2)** manipulate the source-level program and proof, and **(3)** interact with the verifier. Proof actions can be exposed to proof engineers in various ways; our current implementation does so via the engineer's editor (e.g., Visual Studio Code), where she can, say, click on a failing assertion to invoke an appropriate proof action.

Figure 3 illustrates the workflow of a proof action built using ProofPlumber. Once the program text from the editor is parsed, it is lifted to a simplified form, and type-checked. The lifted and type-checked version of the program text is available to proof actions. After a proof action manipulates the proof using ProofPlumber's APIs, it is then pretty printed into the proof engineer's editor.

3.1 ProofPlumber's API Design

Fundamentally, a proof action is a procedure that edits the user's source program based on results from type checking and verification. The corresponding data structures in ProofPlumber are: **(a)** Transformation-Oriented Syntax Tree (TOST) nodes representing the user's source code; **(b)** the Context, which contains additional source-level information such as types and definitions; and **(c)** the Verus verifier, which contains information about failing assertions.

The TOST is the core data structure that represents the source program. It is an abstract syntax tree, with each language construct represented as an `enum` (e.g., `assertExpr, blockExpr`). Since the TOST is not a concrete syntax tree (CST), it omits semantically irrelevant syntactic details. The TOST thus allows easy manipulation of the user's source code, ignoring trivialities like whitespace. The TOST offers the following APIs (corresponding to *3.1 represents the program*, *4. modifies program*, and *5. Pretty printing* in Figure 3).

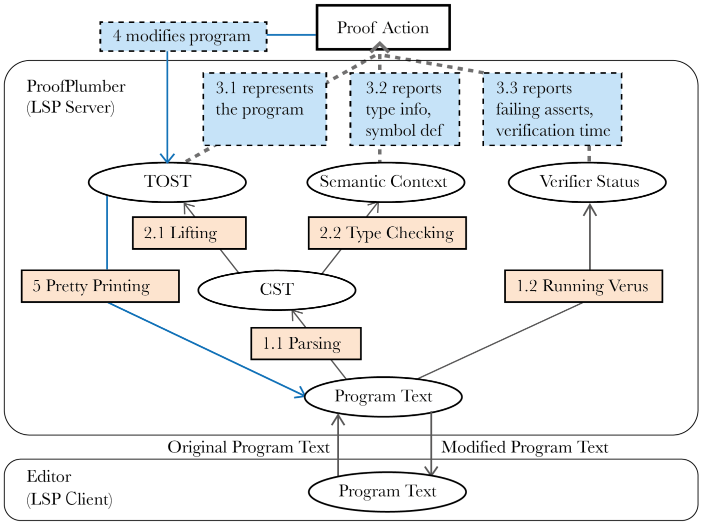

Fig. 3. Overview of ProofPlumber.

- *Traverse/Edit.* A TOST node allows direct access to its children. For example, `assertExpr` has a field `expr` that contains the asserted expression, which can be accessed and modified directly. Additionally, ProofPlumber offers a visitor pattern for recursively filtering or transforming TOST nodes.
- *Create.* When developing proof actions, it is often necessary to create new TOST nodes. While this can be done through each node's constructor, it can be tedious for large expressions; e.g., consider the expression "`x + y * 4`", which needs five constructor calls. To simplify this process, ProofPlumber provides an option to parse user-provided text into a TOST node.
- *Concretize.* When the proof action is done modifying the TOST, it needs to apply the changes to the program. Since the TOST is abstract, ProofPlumber provides an API to convert it back to a CST, hiding the details of the conversion.

Context is another core data structure for writing a proof action. It contains the following information not easily accessible from the TOST (corresponding to *3.2 reports type info and symbol def* in Figure 3):

- *Node in Scope.* A proof action generally acts within a specific scope indicated by the user; e.g., the user's cursor location may identify an expression that is inside a function, a file, a module, and a crate.
- *Type.* Needless to say, type information is crucial for understanding and manipulating the source program. Rust does not require type annotations for every variable or expression, so the type information is often unavailable at the source (or TOST) level. However, the type of every expression has been computed by the type checker, and this API provides that information.
- *Definition.* It is often necessary to look up the full definition for an identifier. For example, when case matching on an enum variable `c:Color`, it is necessary to look up the variants of `Color`. At the TOST level, the definition of `Color`

may be in a different module or even a different crate than the occurrence of `c`. This API provides the definition of an identifier, which can be a name for a struct, an enum, a function, etc.

The Verifier (Verus driver) is the last core data structure. It allows the proof action to interact with the verifier. A proof action does not have to finish all of its rewriting in one pass; instead, it can make a change, invoke the verifier, and then continue rewriting based on the verifier's response. Corresponding to *3.3 reports failing asserts and verification time* in Figure 3, this structure provides:

- *Errors.* The list of failing assertions, preconditions, and postconditions.
- *Time.* It often helps to know how long verification takes, since proofs with shorter verification time are often more robust [34]. If a proof takes too long, the proof action may choose a more efficient one.

3.2 ProofPlumber's Implementation

As Verus [17] is based on Rust, ProofPlumber extends rust-analyzer [35], the official language server for Rust, to understand Verus. As with rust-analyzer, ProofPlumber adheres to the Language Server Protocol (LSP) [30]. In turn, proof actions developed with ProofPlumber are compatible with editors that implement the client-side of the LSP. We construct our *Context* APIs by extending rust-analyzer's type checking implementation (*2.2 Type Checking* in Figure 3).

In our implementation, we have extended rust-analyzer's grammar and parser to obtain the Verus CST (*1.1 Parsing* in Figure 3). We then lift the CST to a TOST, eliminating details like whitespace (*2.1 Lifting* in Figure 3). After a proof action manipulates a TOST node using ProofPlumber's APIs, our pretty printer restores the TOST to a concrete program (*5 Pretty Printing* in Figure 3).

4 Evaluation

We evaluate ProofPlumber with the following three research questions.

- **RQ1**: Are proof actions expressive enough for real proof debugging tasks?
- **RQ2**: Does ProofPlumber make it easy to write proof actions?
- **RQ3**: Can proof actions reduce the TCB of automated program verifiers?

4.1 RQ1: Are proof actions expressive enough?

We demonstrate ProofPlumber's expressivity by writing 17 proof actions, divided into two groups: **(1)** proof actions inspired by Dafny's reference for verification debugging [26], which suggests a set of manual rewrites to perform while debugging proofs, and **(2)** proof actions distilled from the experiences of Verus developers. While these groups were developed independently, they overlap, as Dafny and Verus share proof styles common to automated program verification tools.

Figure 4 presents the list of proof actions and the ProofPlumber features used by each. As analyzed in detail below, we observe that ProofPlumber's APIs are expressive enough to implement all of the suggested rewrites from Dafny, excluding the ones that are not applicable to Verus or that are inherently manual (see §5 for details). Furthermore, ProofPlumber provides enough expressivity to implement eight distinct proof actions that automate the routine efforts of Verus developers. Figure 4 also demonstrates that all of ProofPlumber's features are utilized to implement these proof actions.

	Proof Actions	**Group**	**#lines**	**Verif. Errors**	**Verif. Time**	**TOST R/W**	**TOST Conc.**	**TOST Create**	**Lookup Node**	**Lookup Type**	**Lookup Defn.**
1	Split Implication in Ensures	Dafny	48			✓	✓		✓		
2	Split Smaller or Equal to	Dafny	70			✓	✓		✓		
3	Convert Implication into If	Dafny	40			✓	✓		✓		
4	Introduce Assert Forall	Dafny	42			✓	✓		✓		
5	Introduce Assert Forall Implies	Dafny	57			✓	✓		✓		
6	Introduce Assume False	Dafny	30			✓	✓	✓	✓		
7	Reveal Opaque Function above	Dafny	45			✓	✓		✓		✓
8	Reveal Opaque Function in block	Dafny	45			✓	✓		✓		✓
9	Add Seq "in-bounds" predicate	Dafny	54			✓	✓		✓	✓	✓
10	Insert Assert-By Block	Dafny	29			✓	✓		✓		
11	Weakest Precondition Step	Verus/Dafny(5)	177			✓	✓		✓	✓	✓
12	Decompose Failing Assertion	Verus/Dafny(1)	88	✓		✓	✓		✓	✓	✓
13	Insert Failing Postconditions	Verus	64	✓		✓	✓		✓		
14	Insert Failing Preconditions	Verus	45	✓		✓	✓		✓	✓	✓
15	Introduce Matching Assertions	Verus	84	✓		✓	✓	✓	✓	✓	✓
16	Remove Redundant Assertions	Verus	74	✓	✓	✓	✓		✓		
17	Apply Induction	Verus	133	✓	✓	✓	✓	✓	✓	✓	✓

Fig. 4. For each proof action, we report the source lines of code to implement it, and the ProofPlumber features it uses. For proof actions that overlap in both groups, the number of corresponding Dafny rewrite suggestions is in parentheses; e.g., five of Dafny's rewrite suggestions are special cases of the *Weakest Precondition Step*. *Weakest Precondition Step* and *Decompose Failing Assertion* generalize their Dafny counterparts.

Proof Actions Inspired by Dafny. Dafny provides 29 suggested rewrites for manual proof debugging [26], of which we have implemented 16 as proof actions. We exclude the other 13 rewrites, as 9 of them are not applicable to Verus, and 4 are inherently manual (§5). Of the 16 suggested Dafny rewrites we implemented, 12 are purely syntactic. Among the remaining 4, the first (9 in Figure 4) automatically adds an "in-bounds" predicate for a sequence's index; it uses the `Lookup Type` API to decide if an assertion contains a sequence. The second rewrite (one of the five in *Weakest Precondition Step* – rewrite 11 in Figure 4) adds the precondition and postcondition of a function at the callsite. The last two (7 and 8 in Figure 4) are related to `reveal`, which is used to control the visibility of a function to the verifier. These automatic rewrites lookup the function's definition and the function's "proof visibility".

Proof Actions for Verus. We implemented 7 additional proof actions to automate routine proof engineering tasks in Verus. We discuss *Weakest Precondition Step* and *Insert Failing Preconditions* in §2.3, and the rest below.

Decompose Failing Assertion. When a complex assertion fails to verify, it is not always obvious which portion of the failing expression is responsible. This proof action automates the process of decomposing and isolating the failing sub-formulas. We discuss it in more detail in §4.2.

Insert Failing Postconditions. A common proof failure is that a procedure's postconditions cannot be established. Since there can be multiple postconditions and multiple exit points (e.g., due to an early `return`), developers often employ a tedious manual process to pinpoint the failing conditions at each exit point. This proof action automatically adds the failing postcondition(s) at each exit.

Introduce Match Case Assertions. A special case of assertion decomposition is when the assertion is about an enum. Today, the developer tediously writes a match statement for the enum and then adds assertions to each case to identify where the problem lies. This proof action emits a boilerplate match statement for the enum, but only presents the failing variants.

Remove Redundant Assertions. During proof debugging, to understand the solver's state, proof engineers typically introduce multiple assertions, most of which are redundant (i.e., they help the human, not the verification). Hence, after debugging a proof, to maintain source code readability, developers manually remove these redundant assertions. This proof action mechanizes the process.

Apply Induction. If the selected variable is a natural number or an abstract datatype, this proof action generates the boilerplate code for an inductive proof, including the base and inductive cases. When the selected variable is an enum, it introduces a match statement with an empty proof block for each variant, generating the recursive call to the lemma when the variant is defined recursively.

4.2 RQ2: Does ProofPlumber make it easy to write proof actions?

Figure 4 shows that each proof action from §4.1 needs only 29–177 lines of code. To qualitatively illustrate how ProofPlumber's API enables the easy creation of a proof action, Figure 5 presents a snippet from `Decompose Failing Assertion` from §4.1. This proof action uses all three of ProofPlumber's APIs to analyze a failing assertion with a conjunction of clauses and present the specific clauses that fail. Specifically, the proof action retrieves the surrounding function using the `Context` API (line 6). Inside the function, the original assertion is replaced (using the `TOST` API) with an assertion of one conjunct (line 11). The proof action then uses the `Verifier` API to invoke the verifier on this modified function (line 12) and check if the new assertion fails. If so, it is added to the source code.

While ProofPlumber primarily focuses on automating proof debugging, we observe that ProofPlumber's extensibility also supports the broader proof engineering process. In §4.1, we described *Remove Redundant Assertions* which helps with proof refactoring, and *Apply Induction* which helps with proof development.

```
fn decompose_failing_assertion(
  api: &AssistContext<’_>, // handle for API calls
  assertion: AssertExpr,   // TOST Node to modify
) -> Option<BlockExpr> {   // ‘‘None’’ indicates the proof action is not applicable
  let split_exprs = split_expr(&assertion.expr)?; // split into logical conjuncts
  let this_fn = api.tost_node_in_scope::<Fn>()?; // Find assertion’s enclosing ‘‘Fn’’
  let mut stmts: StmtList = StmtList::new();
  for e in split_exprs {
    // make each logical conjunct into an assertion of its own
    let split_assert = AssertExpr::new(e);
    let modified_fn = api.replace_statement(&this_fn, assertion, split_assert)?;
    if api.run_verus(&modified_fn)?.is_failing(&split_assert) {
      stmts.statements.push(split_assert.into());
    }
  }
  stmts.statements.push(assertion.into()); // restore the original assertion
  Some(BlockExpr::new(stmts))
}
```

Fig. 5. Main routine for the `Decompose Failing Assertion` proof action.

4.3 RQ3: Can proof actions reduce the verifier's TCB?

ProofPlumber can reduce a verification tool's TCB by replacing baked-in debugging support. For example, Verus provides a command line option called `--expand-errors`, which tries to localize the cause of a proof failure. The code implementing this functionality is intertwined with the process of verification condition generation (VCG). However, this functionality is essentially a combination of *Decompose Failing Assertion*, *Insert Failing Postconditions*, and *Insert Failing Preconditions*. Similarly, Dafny's VCG includes custom code similar to our *Apply Induction*. In both cases, we can replace functionality inside the trusted VCG with external, untrusted proof actions. Given the importance of the VCG for sound verification results, this enhances the trustworthiness of those results.

5 Limitations

While ProofPlumber provides automation for most of the existing proof debugging practices for automated program verification, some debugging practices are still not automatable. To illustrate this, we elaborate on Dafny's four rewrite suggestions that we considered inherently manual in §4.1.

The first two are related to quantifiers (`exists` and `forall`). If a failing assertion contains a `forall`, the Dafny manual suggests that the proof engineer should replace the `forall` binding with a "guessed" concrete value that is likely to fail. Similarly, if a failing assertion contains an `exists`, the manual suggests that the proof engineer should replace the `exists` binding with a "guessed" concrete value for which the updated assertion is likely to hold. The "guessing" part is inherently manual as SMT solvers often do not produce a useful concrete model (which could be used to build the source-level counterexample) when they return an "unknown" result, which is the common case when a Verus or Dafny proof fails.

The other two rewrite suggestions are about controlling "proof visibility" when proof engineers encounter a very slow and/or unstable proof [34]. In Dafny, a function's body is available to the solver by default. If an engineer suspects a function definition is contributing to the proof's problematic performance, the Dafny manual first suggests making the function's body invisible to the solver. This change can cause the proof to fail in several locations that previously relied on information about the function's body. Therefore, it then suggests making the function's definition locally available to the solver "only when it is necessary". Deciding if the definition is necessary for the proof to succeed often involves the proof engineers' judgment regarding the proof context, potentially relevant lemmas (that might complete the proof without revealing the function's definition), and their tolerance for proof instability.

6 Related Work and Conclusion

Frameworks such as Meta-F* [36] for F* [12] and Tacny [37] for Dafny [11] focus on developing tactics to help write proofs, whereas ProofPlumber's primary focus is on proof debugging. Another line of work [38,39,40] attempts to reconstruct a source-level counterexample from an SMT solver's model [41]. However, when the solver returns 'unknown', the common case in program verification queries, the model is typically only partial and may be inconsistent with in-scope quantifiers, due to the incomplete heuristics used to for quantifier instantiation. Such inconsistency can be quite confusing for developers.

In everyday software engineering, a counterexample is critical in debugging, as a software engineer can inspect the program's concrete state, and hence localize the error. In contrast, a verification failure can happen for various reasons, including incorrect executable code, incorrect proofs, or even the SMT solver's incompleteness. Therefore, proof engineers still need debugging support to understand the failure.

Conclusion We have presented ProofPlumber, a framework for understanding and debugging proof failures in automated program verification proofs. ProofPlumber supports custom proof actions, which act on the developer's *source-level proofs* to determine why they have failed and potentially suggest remedies. Our evaluation shows that ProofPlumber can automate today's manual debugging practices and provides the extensibility needed for a rapidly evolving area.

Acknowledgments

The authors thank the CAV reviewers and the Verus team for feedback and support. This work was also supported by an Amazon Research Award (Fall 2022 CFP), a gift from VMware, the Future Enterprise Security initiative at Carnegie Mellon CyLab (FutureEnterprise@CyLab), NSF grant CCF 2318953, and funding from AFRL and DARPA under Agreement FA8750-24-9-1000. Chanhee Cho is additionally supported by the Kwanjeong Educational Foundation.

References

1. Coq Development Team, "The Coq Proof Assistant," https://coq.inria.fr/.
2. L. d. Moura and S. Ullrich, "The Lean 4 Theorem Prover and Programming Language," in *International Conference on Automated Deduction*, July 2021.
3. D. Delahaye, "A Tactic Language for the System Coq," in *Proceedings of the 7th International Conference on Logic for Programming and Automated Reasoning*, November 2000.
4. J. C. Blanchette, S. Böhme, and L. C. Paulson, "Extending Sledgehammer with SMT Solvers," in *International Conference on Automated Deduction*, July 2011.
5. J. Meng and L. C. Paulson, "Translating Higher-Order Clauses to First-Order Clauses," *Journal of Automated Reasoning*, vol. 40, no. 1, January 2008. [Online]. Available: https://doi.org/10.1007/s10817-007-9085-y
6. L. C. Paulson and K. W. Susanto, "Source-Level Proof Reconstruction for Interactive Theorem Proving," in *Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics*, September 2007.
7. A. Chlipala, *Certified Programming with Dependent Types: a Pragmatic Introduction to the Coq Proof Assistant.* MIT Press, 2022.
8. M. Barnett, K. R. M. Leino, and W. Schulte, "The Spec# Programming System: An Overview," in *Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, 2004. [Online]. Available: https://doi.org/10.1007/978-3-540-30569-9_3
9. P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll, "Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2," in *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*, 2005. [Online]. Available: https://doi.org/10.1007/11804192_16
10. M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte, "VCC: Contract-based Modular Verification of Concurrent C," in *31st International Conference on Software Engineering - Companion Volume*, 2009.
11. K. R. M. Leino, "Dafny: An Automatic Program Verifier for Functional Correctness," in *Logic for Programming, Artificial Intelligence, and Reasoning*, 2010.
12. N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Béguelin, "Dependent Types and Multi-Monadic Effects in F*," in *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 2016.
13. P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A Verification Infrastructure for Permission-Based Reasoning," in *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation*, 2016. [Online]. Available: https://doi.org/10.1007/978-3-662-49122-5_2
14. F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira, and P. Müller, "Gobra: Modular Specification and Verification of Go Programs," in *Computer Aided Verification (CAV)*, 2021.
15. M. Eilers and P. Müller, "Nagini: A Static Verifier for Python," in *Computer Aided Verification*, 2018.
16. V. Astrauskas, P. Müller, F. Poli, and A. J. Summers, "Leveraging Rust Types for Modular Specification and Verification," in *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, October 2019. [Online]. Available: http://doi.acm.org/10.1145/3360573
17. A. Lattuada, T. Hance, C. Cho, M. Brun, I. Subasinghe, Y. Zhou, J. Howell, B. Parno, and C. Hawblitzel, "Verus: Verifying Rust Programs using Linear Ghost

Types," in *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, December 2023.

18. E. W. Dijkstra, *A discipline of programming.* Prentice-Hall, 1976.
19. L. De Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Tools & Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.
20. H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli *et al.*, "cvc5: A Versatile and Industrial-Strength SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2022.
21. B. Dutertre, "Yices 2.2," in *Computer Aided Verification (CAV)*, 2014.
22. A. Niemetz and M. Preiner, "Bitwuzla," in *Computer Aided Verification (CAV)*, 2023. [Online]. Available: https://doi.org/10.1007/978-3-031-37703-7_1
23. A. Niemetz, M. Preiner, C. Wolf, and A. Biere, "BTOR2, BtorMC and Boolector 3.0," in *Computer Aided Verification (CAV)*, 2018.
24. N. D. Matsakis and F. S. Klock, "The Rust Language," *ACM SIGAda Ada Letters*, vol. 34, no. 3, October 2014. [Online]. Available: https://doi.org/10.1145/2692956.2663188
25. S. Klabnik and C. Nichols, *The Rust Programming Language.* No Starch Press, 2018.
26. "Verification Debugging When Verification Fails." [Online]. Available: https://dafny.org/dafny/DafnyRef/DafnyRef#sec-verification-debugging
27. "Sliding Admit Verification Style." [Online]. Available: https://github.com/FStarLang/FStar/wiki/Sliding-admit-verification-style
28. "StackOverflow Question: With Dafny, Verify Function to Count Integer Set Elements less than a Threshold." [Online]. Available: https://stackoverflow.com/questions/76924944/with-dafny-verify-function-to-count-integer-set-elements-less-than-a-threshold/76925258#76925258
29. "StackOverflow Question: Hint on FStar Proof Dead End." [Online]. Available: https://stackoverflow.com/questions/61938833/hint-on-fstar-proof-dead-end
30. "Language Server Protocol." [Online]. Available: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#textDocument_codeAction
31. R. van Tonder and C. Le Goues, "Lightweight Multi-Language Syntax Transformation with Parser Parser Combinators," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019. [Online]. Available: https://doi.org/10.1145/3314221.3314589
32. J. I. Maletic and M. L. Collard, "Exploration, Analysis, and Manipulation of Source Code Using SrcML," in *Proceedings of the 37th International Conference on Software Engineering*, 2015.
33. P. Klint, T. van der Storm, and J. Vinju, "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation," in *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, 2009.
34. Y. Zhou, J. Bosamiya, Y. Takashima, J. Li, M. Heule, and B. Parno, "Mariposa: Measuring SMT instability in automated program verification," in *Proceedings of the Formal Methods in Computer-Aided Design (FMCAD) Conference*, October 2023.
35. "Rust Analyzer." [Online]. Available: https://github.com/rust-lang/rust-analyzer
36. G. Martínez, D. Ahman, V. Dumitrescu, N. Giannarakis, C. Hawblitzel, C. HriŢcu, M. Narasimhamurthy, Z. Paraskevopoulou, C. Pit-Claudel, J. Protzenko, T. Ra-

mananandro, A. Rastogi, and N. Swamy, “Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms,” in *European Symposium on Programming*, 2019.

37. G. Grov and V. Tumas, “Tactics for the Dafny Program Verifier,” in *Tools and Algorithms for the Construction and Analysis of Systems*, 2016.
38. M. Christakis, K. R. M. Leino, P. Müller, and V. Wüstholz, “Integrated Environment for Diagnosing Verification Errors,” in *Tools and Algorithms for the Construction and Analysis of Systems*, 2016.
39. A. Chakarov, A. Fedchin, Z. Rakamarić, and N. Rungta, “Better Counterexamples for Dafny,” in *Tools and Algorithms for the Construction and Analysis of Systems*, 2022.
40. C. Le Goues, K. R. M. Leino, and M. Moskal, “The Boogie Verification Debugger,” in *Software Engineering and Formal Methods*, 2011.
41. C. Barrett, P. Fontaine, and C. Tinelli, “The Satisfiability Modulo Theories Library (SMT-LIB),” 2016. [Online]. Available: www.SMT-LIB.org