On the Impact of Formal Verification on Software Development

ERIC MUGNIER, University of California at San Diego, USA
YUANYUAN ZHOU, University of California at San Diego, USA
RANJIT JHALA, University of California at San Diego, USA
MICHAEL COBLENZ, University of California at San Diego, USA

Auto-active verifiers like Dafny aim to make formal methods accessible to non-expert users through SMT automation. However, despite the automation and other programmer-friendly features, they remain sparsely used in real-world software development, due to the significant effort required to apply them in practice. We interviewed 14 experienced Dafny users about their experiences using it in large-scale projects. We apply grounded theory to analyze the interviews to systematically identify how auto-active verification impacts software development, and to identify opportunities to simplify the use, and hence, expand the adoption of verification in software development.

CCS Concepts: • **Software and its engineering → Formal software verification**.

Additional Key Words and Phrases: User study, Dafny, Grounded theory

ACM Reference Format:
Eric Mugnier, Yuanyuan Zhou, Ranjit Jhala, and Michael Coblenz. 2025. On the Impact of Formal Verification on Software Development. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 403 (October 2025), 27 pages. https://doi.org/10.1145/3763181

1 Introduction

Verifiers such as Dafny [Leino 2010], F^* [Swamy et al. 2016] and Verus [Lattuada et al. 2023] are becoming increasingly prevalent in both academia and industry. Just in the past few years, projects verifying cloud controllers [Sun et al. 2024], and security modules [Zhou et al. 2024] have been published in communities outside of programming languages and formal methods, including operating systems and software engineering conferences, and more importantly, have been deployed in real-world systems [Chakarov et al. 2025; Swamy et al. 2022].

Despite the above successes, actual users of such verifiers remain few and far between. We speculate that the adoption of *and* hesitation in the use of such verifiers stems from their *auto-active* [Leino and Moskal 2010] nature. Unlike proof assistants, which require users to write proofs manually, auto-active verifiers use SMT solvers to attempt to automatically generate proofs. On the bright side, adoption is driven by the ability to automate low-level proof details to SMT solvers, enabling users to focus on higher-level aspects of their proofs. However, dark clouds swiftly appear when the automation inevitably fails, and the user is left to decipher what knowledge the verifier has (and lacks) to provide hints that can guide the verifier to a valid proof. Consequently, the automation provided by these verifiers places them in an *uncanny valley* in terms of usability,

Authors' Contact Information: Eric Mugnier, University of California at San Diego, La Jolla, USA, emugnier@ucsd.edu; Yuanyuan Zhou, University of California at San Diego, La Jolla, USA, yyzhou@ucsd.edu; Ranjit Jhala, University of California at San Diego, La Jolla, USA, rjhala@ucsd.edu; Michael Coblenz, University of California at San Diego, La Jolla, USA, mcoblenz@ucsd.edu.

ACM 2475-1421/2025/10-ART403
https://doi.org/10.1145/3763181

defeating the purpose of the tools, which was to make the fire of formal verification available to mere mortals without doctorates in formal methods.

How are verifiers used in practice? In this work, we chart this valley by investigating how auto-active verifiers are *actually used* in real-world software development. To this end, we recruited 14 experienced software engineers averaging five years of work experience, who have used auto-active verifiers like Dafny. We then interviewed them and analyzed these interview logs using *grounded theory* [Charmaz 2014] to identify patterns and relationships in the data to answer three questions: (RQ1) First, what *expectations* should developers have regarding how verification might impact the design and implementation of software? (RQ2) Second, what *practices* do experience developers use to apply verification effectively? (RQ3) Third, what *opportunities* exist to simplify the use and, hence, expand the adoption of verification in software development?

Formal-first vs. Engineering-first Developers We observed two distinct groups of participants. *Formal-first* developers have deep formal methods expertise that was then applied to real-world software engineering, and *engineering-first* developers were experienced software engineers who went on to use verification in a project. We then find that even though verified software development has many of the same phases as traditional software development—requirements, implementation, testing, review, packaging, maintenance, *etc*—developers from the different groups have very different expectations and practices for the phases, summarized via three concrete contributions.

1. Designing & Building Verified Software (§ 4) Our first contribution addresses RQ1 with an analysis of how the use of auto-active verification impacts the design and implementation of software. Of course, the design phase is impacted by the need to codify requirements as formal *specifications*. Crucially, the implementation phase is also affected, as now developers must write auxiliary assertions to prove lemmas, which are then combined to yield *proofs* of top-level specifications. There are two more key phases: *debugging* failed proof attempts and *hardening* the code to ensure that proofs continue to hold in the face of changes to SMT solver heuristics. When debugging, we find that formal-first developers often want to spend time to root cause the failures, while engineering-first developers might prefer to fix suggestions that let them move on. Similarly, we find that formal-first developers tend to appreciate the benefits of automation and how it avoids the need to spell out low-level details, while engineering-first developers consider the attendant proof-brittleness problems something that can be mitigated via discipline encoded in style guidelines.

2. Testing & Deploying Verified Software (§ 5,§ 6,§ 7) Our second contribution focuses on RQ2 via an analysis of how verification impacts testing, review, deployment, and downstream maintenance of software. Here, we find that a tight integration with classical software development practices is key, as *testing* and *code reviews* provide rare opportunities to catch errors in formal specifications, which would not be flagged by the verifier. However, verification increases the burden of reviewing, as each review requires looking at changes in specifications, implementations, and also at the code generated via transpilation into the executable target language. Interestingly, we find that during review formal-first developers tend to focus on the specifications while engineering-first developers focus on the code that will actually be executed. We find that engineering-first developers particularly value the fact that verifiers enhance coding *agility* by precisely automating change impact analysis, hence enabling aggressive code optimizations without fear of breaking functionality.

3. Opportunities for Improving Verified Software Engineering (§ 8) Our third contribution answers RQ3 by identifying several concrete opportunities from the interviews, for streamlining verified software engineering, by simplifying the processes of specification and proof, and tightening the integration with the other phases of the software development lifecycle. These include

```
method applyDiscount (x : real, percent : real) returns (res : real)
  requires 0.0 ≤ x
  requires 0.0 ≤ percent
  requires percent ≤ 100.0
  ensures 0.0 ≤ res
  // Example of MISSING POSTCONDITIONS
  // ensures (0.0 < x ⟹ res ≤ x) ∨ (percent = 0.0)
{
  var factor := percent / 100.0;
  assert 0.0 ≤ factor ≤ 1.0;
  // MISSING HINT
  // assert x ≤ 0.0 ∧ factor ≤ 0.0 ⟹ 0.0 ≤ x * factor;
  res := x * factor;
}
```

```
Error: a postcondition could not be proved on this return path
Could not prove: 0.0 ≤ res
```

Fig. 1. Example of a failing Dafny method with missing assertion and preconditions, reported by P9

conducting empirical studies that identify what constitutes clean proofs that are robust to solver changes and codifying them in style guides and linters; improving the interactivity of the verifier by providing an experience similar to that of the GNU Project Debugger (GDB), finding ways to automatically generate distinct review artifacts for specifications, proofs, and executed code; and finding ways to more seamlessly integrate tests and formal specifications, or even use unit testing as a proxy for proof.

Together, our study is the first of which we are aware that uses interviews with experts to understand the impact of automated verifiers on real-world software engineering projects. Some of our findings resonate with anecdotal evidence from blog posts [Tomb 2023], talks [Rungta 2024], and case-studies [Faria and Abreu 2023]. However, by systematizing this expert ("folk") knowledge, we hope to enable readers to exploit this knowledge to improve their own software engineering with verification, and to devise the new algorithms, techniques, and tools needed to make formal verification more commonplace in software development.

2 Background

We assume in this paper, a basic familiarity with auto-active verifiers like Dafny [Leino 2010].

Contracts and Verification Conditions As a brief refresher, consider the code in fig. 1, which shows a method that takes two inputs `x` and `percent` and returns an output `res`. The method's contract has a *pre-condition* specified by the `requires` clause, which says that the input `x` must be non-negative, and that the input `percent` must be between 0 and 100. The method's contract also has a *post-condition* specified by the `ensures` clause, which states that the output `res` is non-negative. Dafny uses the specification and the implementation to generate a verification condition (VC): a logical formula whose validity implies that the implementation satisfies its contract.

Auto-Active-Proof: Assertions, Hints and Lemmas While the SMT solver can prove many VCs (valid), more complex programs or specifications yield VCs that are outside the *decidable* theories implemented by the solver. For example, even the VC from fig. 1 uses a non-linear arithmetic, arising from the multiplication in the method body. When this happens, sometimes the SMT solver's heuristics suffice, but often, the developer has to provide an explicit *hint* in the form of an `assert`

#	Gender	Age	Occupation	Degree	Experience	Software verified	# LOC
P1	F	25-34	Student	≥ MS	< 1 year	Distributed systems	≥500
P2	F	25-34	Student	≥ MS	8-10 years	Compilers, distributed systems	≥10,000
P3	M	25-34	Student	BS	4 years	Mathematical proof	≥1000
P4	M	34-45	Computer Scientist	≥ MS	≥ 10 years	Security critical	≥1000
P5	M	25-34	Student	≥ MS	5-7 years	Cloud	≥1000
P6	M	34-45	Computer Scientist	≥ MS	≥ 10 years	Security critical	≥20,000
P7	M	45-54	Software Engineer	≥ MS	≥ 10 years	Authorization	≥5000
P8	M	25-34	Professor	≥ MS	5 years	Distributed systems	≥1000
P9	M	34-45	Computer Scientist	≥ MS	≥ 10 years	Business critical	≥1000
P10	M	25-34	Professor	≥ MS	2 years	File system	≥10,000
P11	M	45-54	Computer Scientist	≥ MS	≥ 10 years	Storage system	≥10,000
P12	M	54-64	Software Engineer	BS	≥ 10 years	Cryptography	≥10,000
P13	M	45-54	Software Engineer	BS	≥ 10 years	Cryptography	≥10,000
P14	M	45-54	Software Engineer	≥ MS	≥ 10 years	Blockchain	≥10,000

Table 1. Participants' backgrounds

statement (shown as a comment) that encodes the relevant mathematical fact as part of the VC. Sometimes, these mathematical facts are over recursively defined functions, must themselves be proved via separately defined recursive (inductive) functions which effectively act as *lemmas*, which may then be used to prove a given VC by appropriately invoking the lemma in an `assert`. Hence, Dafny, and related tools are often referred to as *auto-active* verifiers, to distinguish them from fully *interactive* verifiers like Rocq or Lean which do not use SMT automation.

3 Method

Participants We recruited 14 participants through referrals by Dafny maintainers, personal contacts, and snowball sampling. The main inclusion criterion was that the participants had worked on a large Dafny project that exceeded 500 lines of code. We stopped recruiting once the last few participants began repeating key ideas, which occurred after 14 interviews. Although we may not have reached idealized theoretical saturation at all levels, we are confident that the concepts that emerged capture the key concerns of all the participants. Additionally, this number is consistent with prior work, which found that saturation is typically achieved within about 12 interviews [Guest et al. 2006]. These 14 participants have a variety of backgrounds, as summarized in table 1. We labeled them from P1 to P14 in the order of their interviews. Two of the participants were students, four were computer scientists, four were software engineers, and two were professors. Their experience ranged from under one to more than ten years, although most of them were relatively senior, with 10 participants having at least four years of experience. The projects they worked on varied in size, from less than 1000 lines of code to more than 10,000 lines of code, covering a variety of domains.

Interview protocol We conducted interviews from September 2024 to March 2025 in a semi-structured format, using a predefined set of questions while allowing the conversation to flow naturally. We grouped the questions into four categories following the software development lifecycle —*planning*, *design*, *implementation*, and *review/testing/maintenance*. We also asked the participants about their background and their projects. The interviews lasted about an hour and were conducted over Zoom, except for the first two, which were held in person. Conversations were recorded and transcribed automatically using Zoom. The protocol was approved by our university's Institutional Review Board (IRB), and we obtained informed consent from our participants. To

protect participants' identities, we report only anonymous identifiers, such as P1, not their names. Participants could also skip questions whose answers could reveal sensitive information.

Analysis To analyze our interview transcripts, we used a constructivist grounded theory approach [Charmaz 2014]. Our method consisted of three main steps:

(1) Paraphrasing each relevant sentence of the transcript using gerunds. This resulted in 1081 segments.
(2) Extracting low-level codes from the paraphrased content, yielding 325 codes.
(3) Exploring relationships between codes primarily through diagrams. From these relationships, we identified six key categories – *learning curve*, *specification and proof technique*, *proof brittleness*, *assurance techniques*, *integration with software development*, *code changes*.

Although this resembles axial coding, our process relied more on visually mapping connections, often before writing memos, to document our ideas and observations about participants, as advised by Charmaz [2014]. Diagrams proved to be the most practical technique for extracting meaning from the data.

Threats to Validity Our study presents a snapshot of the field at the time of our interviews, but it may not capture all perspectives. None of our participants were complete Dafny novices, such as undergrad students, and we were constrained by the size of the Dafny community, which lacks diversity—for example, we were only able to recruit two women. Additionally, while we believe that most of our findings are generalizable to other automated verifiers such as Verus or Fstar, our participants mainly discussed their Dafny experience, which may not be entirely transferable.

As verification researchers, our knowledge of the field influenced how we conducted the interviews and analyzed the data. In fact, this study was motivated by observations of Dafny's use in industry. With backgrounds in programming languages, systems, and verifiers, we saw an opportunity to study its use in practice. We believe that the challenges observed on large-scale projects are not yet well understood. People with different backgrounds might have had a different perspective. For example, a sociological perspective might have revealed more findings on the collaborative approaches taken by the participants. All the coding was performed by the first author, but techniques such as memoing helped mitigate bias [McDonald et al. 2019]. Additionally, using a systematic grounded theory approach helped reveal our biases and limit their impact.

Finally, the interview process could also be a limitation. Interviews do not necessarily reflect reality, but rather the participants' personal experience. Moreover, confidentiality constraints may have restricted what participants could disclose, potentially omitting relevant insights.

4 Results: Impact of Verification on Development

First, we present our results on how formal verification affects the phase of *building* software. First, the *specification* phase is significantly impacted, as the informal requirements of classical software engineering are now augmented with formal artifacts that describe what the code should do (§ 4.1). Second, the use of verifiers significantly changes the *implementation* phase because now, in addition writing the code that runs, engineers must also write auxiliary assertions and lemmas that are needed to help the verifier formally establish that the code meets its specification (§ 4.2). Third, the notion of *debugging* is also transformed as a significant effort goes into understanding why the verifier fails to prove that the code meets its specification and to then determine the appropriate hints to provide to the verifier to help it succeed (§ 4.3). Finally, we find that verifiers that make extensive use of brittle SMT solver heuristics to automate verification, add a new phase to the process of building software: that of *hardening* the code so that the proofs continue to hold even as the specific solver heuristics might change across future versions (§ 4.4).

4.1 Specification

> *"Whenever you pick a language to write the pseudocode in, it starts to shape your idea of what the implementation is." — P13*

Developers report that verifiers influence *when* they write specifications, *what* sources they use to derive specifications, and *how* they actually write the specifications to minimize risks of unsoundness and simplify the downstream verification.

When to Specify

Participants report writing the specification at different stages of the development, depending on the risks they want to mitigate. Participants P12 and P13 develop the specification and implementation *concurrently*. This approach minimizes the cost as they can iterate between the two to incorporate missing details from the code into the specification or inversely. Some participants, like P5, P7, and P11, *start with the specification* to ensure the design makes sense. Specifying first allows to "get a better understanding of the code" (P7) and minimize the risk of writing an incorrect implementation by stating its invariants first. Finally, P6 and P9 *begin by writing a working implementation* in Dafny before attempting to prove it. Their goal is to have a working proof of concept that can be tested on real data before focusing on verification. Using this method, they can showcase the feasibility of the project quickly and secure their peers' support.

What to Specify

To write specifications, participants used different sources of data: — *documentation*, *design documents*, *tests*, and *code* — to determine the important properties of their system. Each of these sources provides partial information, and so developers choose to combine them to get a complete picture of the overall specification.

Specifications from Documentation P5 reports extracting specifications from documentation: "You can look at the documentation, and you can guess things are semi formalized, like sentences that are written in natural language, but that describes mathematical proprieties like boolean properties." However, documentation is often incomplete and vague, which introduces the risk of writing a specification that is too weak or potentially incorrect. To prevent the latter, P12 and P13 write an informal specification first, using RFC 2119 [Bradner 1997] semantics, with keywords such as `MAY` and `MUST`. This informal specification, inspired from their design document deliberately avoids pseudocode or reference to any elements of the code, as, participant P13 noted: "Whenever you pick a language to write the pseudocode in, it starts to shape your idea of what the implementation is, and you start to make assumptions."

To connect their specification with the code, P12 and P13 use Duvet [AWS Labs 2025], which inserts requirements text as comments in corresponding code sections. Developers then translate these informal specifications into Dafny contracts, as illustrated in the simplified example given by P13 shown in fig. 2. This example includes (1) a *comment* that is extracted by Duvet from the informal specification *aws-kms-keyring.md* (lines 2-4) and (2) a Dafny *post-condition* with the `ensures` keyword (lines 5-7). The latter is the developer's formalization of the natural language comment: if the input contains a plaintext data key, then the encryption context must be successful. P13 explained that this structured process has been highly effective: "the person who's writing the design [now has] a mechanism to make sure the person who's writing [the code] does what they want — give them some evidence that it's correct." This process avoids imprecision from natural language documentation, as every property is linked to an RFC-style statement. However, the process is costly to design and maintain since it introduces another layer of translation.

Specifications from Tests Some participants derive specifications from existing tests. For example, P6 described their method as "generalizing the tests which exist into theorems." The benefit of this method is that the tests provide a good sense of what should be true or not about each part of the system. The challenge with this approach is that these tests "are obviously like specific points within that space" meaning they must find a specification that covers "all the inputs."

Specifications from Code P5 studies the code: "it gives you a lot of implementation details," which are not always tested or documented, such as "what kind of error is sent when you have this thing that is invalid?" However, reading the code to infer the specification is often a tiresome task, that P4 described as: "extremely time-consuming to [...] stare at the Java and try to figure out the abstract properties." In other words, it is not always clear what the code *is* doing, and hence, even less what properties *should* hold about it. To address this, P4 resorted to ignoring details that are not pertinent to the specification, stating that "many implementation details in the middle were irrelevant."

How to Specify

Developers described using two main strategies to simplify formal verification: writing *purely functional* specifications that minimize reasoning about imperative updates, and then further *abstracting* the specifications to focus the aspects germane to downstream proof.

Pure (Functional) Specifications Participants often reported using functional `function` specifications that describe the behavior of imperative `method` implementations, as this eases verification by providing a pure (side-effect free) representation of the implementation. Verifiers have an easier time with functional code as it maps more directly to the logical representations needed for formal verification. It also enables equational reasoning while eliminating the need for complex *frame conditions* that specify what may change in the program state, the need to track *aliasing* relationships, and complex *control-flow*. The effectiveness of this strategy of writing pure functional specifications is well-established in the research literature [Hawblitzel et al. 2015; Klein et al. 2009; Leroy 2006].

P5, P7, P8, P10, and P11 reported using functional specifications to describe properties of imperative methods, allowing the use of *local*, *compositional* and *equational* or *algebraic* reasoning to prove properties of the implementation. Figure 3 shows an example extracted from a participant's project. The method `UpperBoundedAdditionImpl` implements the addition of two bounded integers, `x` and `y`, with an upper bound `u`. In the `ensures` clause, the method specifies that the result `sum` must be equal to the result of the `function` specification `UpperBoundedAddition`. `UpperBoundedAddition` is a pure function that takes two *unbounded* integers `x` and `y` and an upper bound `u` and returns the sum of `x` and `y` if it is less than or equal to `u`; otherwise, it returns the

```
0 //= aws-encryption-sdk-specification/framework/aws-kms/aws-kms-
      keyring.md#onencrypt
1 //= type=implication
2 //# If the input [encryption materials](../structures.md#encryption-
3 //# materials) do contain a plaintext data key, OnEncrypt MUST
4 //# attempt to encrypt the plaintext data key
5 ensures
6  ∧ input.materials.plaintextDataKey.Some?
7  ⟹
8   ∧ KMS.IsValid_PlaintextType(input.materials.plaintextDataKey.value)
9   ∧ 0 < |client.History.Encrypt|
```

Fig. 2. Simplified example of an informal Duvet specification manually translated into Dafny.

```
method UpperBoundedAdditionImpl(x: uint64, y: uint64, u: uint64) returns
    (sum: uint64)
  ensures sum as int = UpperBoundedAddition(x as int, y as int,
      UpperBoundFinite(u as int));
{
  if y ≥ u {
    sum := u;
  } else if x ≥ u - y {
    sum := u;
  } else {
    sum := x + y;
  }
}
```

```
function UpperBoundedAddition(x: int, y: int, u: UpperBound): int
{
  // LeqUpperBound checks if a given integer x
  // is less than or equal to the specified upper bound u.
  if LeqUpperBound(x + y, u) then x + y else u.n
}
```

Fig. 3. Example functional specification of an imperative method [Iro 2015]

upper bound `u`. The main differences between the implementation and the specification are (1) the types: the implementation uses `uint64` while the specification uses `int` and (2) the control flow: the implementation has multiple assignments to `sum` while the specification is a single expression.

These differences align with observations made by P4 and P5 regarding functional code. As the former highlights, functional code "makes state updates explicit" by reifying such updates as *named* values in the code which simplifies the subsequent verification as "Dafny has a significantly easier time proving things about functional code." P5 further emphasizes that "your specification needs to be pure," arguing that purity inherently improves design by making side effects explicit. In this case, `UpperBoundedAddition` being a pure function simplifies verification by removing the need for Dafny to reason about the intermediate (implicit) state changes of the imperative implementation.

Of course, one might wonder why the developers did not write the implementation in a functional style in the first place. The reason is that the imperative version is more computationally efficient: it short-circuits as soon as it finds a satisfying condition, avoiding computing the sum of `x` and `y` when unnecessary. Thus, we found developers write functional specifications to get a clear, concise, and verification friendly representations of efficient-to-execute imperative implementations.

Abstracting Specifications While a functional specification is easier to verify, its explicit nature can still bog down downstream verification with irrelevant information. Developers address this problem by abstracting the specifications to only expose properties that are relevant for verification. To illustrate the complexity of specifying imperative code, P8 referenced a paper showcasing the powers of Dafny [Leino 2008]. They described a simple method for adding an element to a singly linked list, which involves just a few lines of code. Yet, verifying that this method is correct required substantial effort as it requires "proving that you know that no pointers are pointing anywhere that you don't want them to be pointing. And it just takes a lot of work to say exactly where you want them to be pointing to." P11 explained that this verbosity can have negative consequences on the soundness of the specification, especially when the functional specification is similar to the

implementation "because you're just as likely to have a bug in your specification as you are in your implementation."

To improve clarity, P11 described using different levels of abstraction. First, they write a functional specification equivalent to the implementation, similar to the one described in Figure 3, where *implementation* types like vectors and hash maps are replaced by *logical* sequences and maps, to simplify reasoning. Second, they introduce another layer that "hides the details of the algorithm and exposes an interface." This separation, they noted is "valuable for verifying different components of your system independently of each other". Finally, as P7 mentioned, they define main theorems under the form of Dafny `predicate`, which are pure mathematical functions that return a boolean value, to "capture" the global invariants of the code. This approach encapsulates the low-level details of the implementation and summarizes the "theorem [...] in half a page of text".

4.2 Proof Development

"In Dafny, very, very little goes through with literally zero proof" — P10

Eleven of our 14 participants agreed that a proof checked by the tool was the only acceptable outcome of a verification effort. They enforced this standard to "only commit verified code" (P5). The remaining participants reported to sometimes trusting their implementation for proofs that are too demanding, given something they know to be correct.

The effort of ensuring "correctness all the time" (P10) is a result of the cost and the expectation put into verification. As P11 explained: "If you're going to put in the effort of proving, you might as well get the benefits of the proof, which is that you don't have to worry about software bugs." P5 confirmed that justifying the effort of verification through guarantees is essential to convince leaders and stakeholders to invest in the process.

While Dafny and similar *auto-active* verifiers use SMT solvers to automatically discharge many proof obligations, the developer must still spend substantial effort writing proofs of properties that lie outside the solver's capabilities. Indeed, previous work [Hawblitzel et al. 2015] shows that the number of lines of *proof* code can be 10× the number of lines of *implementation* code. Thus, we sought to study how verification affected the actual coding practices of our participants, to identify what techniques and approaches they use when developing proofs.

Regarding *when* to write proofs, participants described two techniques: (1) only verified code should be *committed* to version control; (2) proofs can be developed *gradually*, using assumptions as temporary placeholders in lieu of full proofs; Regarding *how* to write proofs, we found two approaches: (1) structuring proofs in a *top-down* fashion, starting with the top-level theorems, and then forking off the obligations needed to prove those theorems as helper lemmas, that are then assumed and then subsequently proved, perhaps using more lemmas, recursively; (2) structuring proofs in a *bottom-up* fashion by building up a library of proved lemmas, using those to prove more complex lemmas, until finally, the top-level theorem is proved.

Gradual Proof Due to the upfront cost of verification and the need to produce quick results, all participants reported using an incremental verification approach. P3 explained: "Dafny is very nice for this incremental development, you can just make a lemma pass with some, assert false, and see 'Oh, with this assumption, can I make these things true?' " Using this method, Dafny allows users to build their proofs step by step in the same way interactive proof assistants allow skipping goals via `admit` or `sorry`. Participant P11 wished Dafny would go further with a hybrid approach that would allow fuzzing or model checking the specification against the implementation to avoid fruitless proof effort when the implementation is *not* a refinement [Wirth 1971] of the specification.

Some participants felt that sometimes formal proof was unnecessary, and testing was sufficient. For example, P12 said "For most code, writing a dozen tests is actually going to test every interesting

code path and so proving it for all possible inputs doesn't actually get your code any more correct because it was already correct." Consequently, P12 and P13 reported not writing contracts at all, and skipping verifying well-tested or relatively straight forward functions. Of course, the participants noted that it is not always possible to bypass verification effort as Dafny forces you to write certain annotations to make the code compile. For example, in some situations not covered by Dafny's built-in heuristics, the developer must specify a `decreases` clause to convince Dafny that a piece of looping or recursive code terminates.

While such *gradual* techniques is essential to verified development, they can blur the distinction between verified and unverified code, so developers have to take care to understand which bits of code are verified and which are not.

Top-down Proof Participants P1, P3, P5, and P6, reported using a *"top-down"* approach, starting by proving the main theorem *before* proving the necessary lemmas. They deferred, as unproved assumptions, proofs of any secondary propositions (sub-goals) needed to prove the top-level goals. Often the decomposition into sub-goals is relatively straightforward as it often reflects the structure of the code. As P6 explained: "For each of the properties, there's a main lemma and, generally, the structure of the code means that these get split down into auxiliary lemmas about different parts of the code which are easy to combine." This connection between the code and the proof structure was echoed by P5, P10 and P13, with P5 describing it as "almost a compositional proof starting from the specification." This technique, however, suffers from the exclusion of low-level details that might impact the proof but which are only discovered when trying to prove some very low-level lemma at the "bottom" of the proof dependency graph, and which could then invalidate the entire preceding proof effort.

Bottom-up Proof To avoid such unpleasant surprises, P10 uniquely reported using a "bottom-up" approach where they started with self-contained low-level lemmas that were then incrementally used to build *up* to the overall top-level theorem, thereby minimizing the risk of barking up the wrong (proof) tree. However, P10 acknowledged that this approach only works for them as they know the main proof goal, "[Having] an idea of the whole design going in [makes it] a lot less likely that [I] verify something useless." Indeed, P10 notes this approach does not work for all projects, for instance "when it's more experimental, this [...] tends to waste time", as this localized, bottom-up approach can lead to proving lemmas not needed by the top-level proof.

Several participants reported using a *hybrid* of the top-down and bottom-up approaches as a middle ground. For example, P4 and P5 stated that when they knew the main theorem, and have some idea of what the implementation should be, they start top-down and then switch to bottom-up if needed. This helps, as P4 explains to "sanity check my backwards work by working forward a little bit, and seeing how far apart things are from meeting in the middle."

4.3 Proof debugging

"Who's wrong, you or Dafny?" — P6

In addition to the debugging process that arises in regular software development, auto-active verification introduces a new debugging phase: the developer must understand and fix their proof when it is *rejected* by the verifier. Proof debugging is particularly challenging because it requires the developer to determine whether the is code rejected (1) because the desired property *does not hold*, or (2) because Dafny simply *cannot prove* it. In the first case, the main reason for a proof failure is a mismatch between the contract and the code, as reported by six participants, and described in § 4.1. In the second case, users referred to "Dafny being wrong" as the verifier failing to prove a true statement without hints. (Note that while Dafny can be wrong due to unsoundness issues, only one user reported encountering such a case in our study.) Next, we illustrate the nature of the *proof*

debugging problem with an example, and describe how developers tackle it via a combination of attempting to *passively interpret* the verifier's error message, *actively probe* the verifier's internal state by adding assertions to the code, and *adding assumptions* to narrow the cause of the failure.

The Proof Debugging Problem Figure 1 shows an example from P9 that illustrates the challenge of proof debugging. The method `applyDiscount` calculates the percentage `percent` of a given number `x`. In this case, the author wanted to prove that, given a non-negative number `x` and a valid positive percentage `percent`, the result `res` is also non-negative. However, when trying to verify this method, Dafny gives an error message indicating that it cannot prove the postcondition `0.0 ≤ res` without explaining why. Now the developer has to determine whether (1) the *code fails* the postcondition, or (2) the *verifier fails* to prove the postcondition owing to some limitation of Dafny or the SMT solver, and if so, what hint would enable successful proof.

Interpreting Error Messages The first challenge of the debugging process is that of understanding what brought up an error message. P9 described the feelings of another inexperienced user when reading the error message from fig. 1: "The thing just tells you I can't prove that 0.1 * 0.1 is greater than 0 without any further explanation. Everyone will think that thing is [stupid]…They will think like this called an LLM or something. But the thing is that it's supposed to be good at math. You have to give a better error message." Other users echoed this frustration. P2 described the output as "spits out a no"; P3 called it "black box thing" and P7 remarked that it provides "no help." In fact, while the error message gives you the location of the problem—the path, along which the postcondition could not be proved—it does not explain *why* the proof failed. Expanding on this, P12 confessed to being lost when it came to identifying what information was wrong or missing: "How do I know which of the 1,000 things involved in that assert are what [Dafny is] having a problem with?" They also wished that Dafny could surface more information to the developer: "Down in the depths of Boogie (an intermediate verifier used by Dafny), it probably knows and there should be some way to percolate that back up." This frustration aligns with reports from P1, a beginner in Dafny, who suggested that the unclear error messages, on top of creating misunderstanding, can also lead to *mistrust* in the tool.

Actively Probing via Assertions as Breakpoints One standard method to debug proof errors like that in fig. 1 is to add intermediate assertions that help to the developer build a mental model of what the verifier can and cannot prove, which then, ultimately, helps them identify *hints* that can provide the information needed to complete the proof. P9 explained that the inexperienced user added assertions on line 6 as they thought the problem was about inferring the value of `factor`. However, the real problem was that the verification goal here involved *non-linear arithmetic*, which is undecidable in general. In this instance, the SMT solver did not recognize that multiplying two non-negative numbers always results in a non-negative product. Once this fact was provided as a *hint* — an explicit assertion on line 8 — Dafny was able to prove the assertion and the postcondition, removing the error message. This strategy of adding intermediate assertions or loop invariants is used by all participants to debug proofs as P10 noted, "in Dafny, very, very little goes through with literally zero proof." P12 described their approach: "sprinkle in a bunch of assert statements to see what the actual problem is" — a method similar to adding breakpoints to inspect the machine state during regular debugging. P6 talked about a more methodical approach of "asserting the post conditions at every exit point."

Narrowing Failure Causes via Branches and Assumes Another strategy, in the spirit of the top-down proof decomposition approach, was described by P10 as "breaking down" the proof by adding conditionals, splitting it into cases to precisely understand in which scenarios the proof fails. Building on this, P12 also reported using `assume` statements to identify missing facts "you might assume […] something you know is true […] if that fixes your problem, then you […] need

a lemma" (to prove that the `assume` proposition is indeed true.) Summarizing this experience, P10 described it as "how to get better, more error messages out of the tool."

Challenge: Building a Mental Model of Implicit Verifier State While these strategies are effective, the example in fig. 1 illustrates that even if the user is broadly familiar with the techniques to debug proofs, they may not be able to explicate the missing facts; they added an assertion but were not able to identify the one that was needed. To confirm this, both professors in our study (P10 and P8) reported that ultimately, the difficulty lies in building a *mental model* of the internal verifier state — *i.e.* of what Dafny "knows" at any point in the proof — as Dafny's error message only tells you *what* fails but not *why* it fails.

P10 and P8 compared the implicit proof context — slowly made visible via the addition of assertions — to the *explicit* proof state of interactive provers like Isabelle, Rocq or Lean, which display to the user *all* the facts that are "known" to the verifier at any point in the proof. While they acknowledge that both auto-active and inter-active tools can do similar things, P8 pointed out the challenge of Dafny's hidden verifier state was that "users basically have to have a perfect model of the Dafny proof context in their head in order to understand what's going on." One way users can clarify the context is by adding annotations that make the state explicit. However, this can sometimes require an overwhelming number of annotations: P8 recalled once requiring "some intermediate point [...] with 50 assertions". To alleviate this, users need to use their intuition and experience to decide which assertions to add, as enumerating everything is impossible due to the numerous cases or facts to consider. Ultimately, as P3 noted, proof debugging remains a complex challenge: "There is still [a gap] between the things that are intuitive for people, and that are intuitive for a computer", suggesting that tools and techniques that help bridge this gap will have a significant impact on the usability of auto-active verifiers.

4.4 Proof hardening

> *"There's something soul crushing about having to go back to things that you thought were done, and do them again." — P7*

A key strength of auto-active verifiers like Dafny relative to interactive ones like Rocq or Lean is their ability to automatically discharge proof obligations by relying upon SMT solvers. However, this automation comes at a cost: the proof obligations — such as the one from fig. 1 — often involve reasoning outside of decidable theories, and hence, rely on brittle SMT solver heuristics that can sometimes fail. Thus, verified software development has to include a new phase: *proof hardening*, where the developer modifies or alters their code and proofs so that verification remains robust in the face of subsequent changes to the SMT solver's heuristics or the surrounding context. Next, we illustrate the nature and extent of the *proof brittleness problem*, identify various *sources of brittleness*, and describe the common *hardening strategies* that developers use to address brittleness.

The Proof Brittleness Problem A brittle proof is a previously verified proof where trivial, unrelated changes can cause the proof to fail. This unpredictability was illustrated by P7, who explained introducing "some little fiddly change to the code, just maybe pass one more element in the state [...] and all the proofs would just stop working." Not only do small code modifications trigger failure, but P7, P10, P11, and P13 also reported that updating Dafny, changing the underlying Z3 solver, or even verifying on a different machine could break a brittle proof. While these issues were revealed by minor changes, their fixes were a major enterprise and major setbacks. Failures like these, where as P8 says, the verifier "goes off the deep end," were described as something that "bites you" by P5, as a "major roadblock" by P10, and ultimately, as inducing "existential dread" and "soul crushing" by P7.

We observed three categories of experience with proof brittleness. P1, P3, and P9 had not encountered brittle proofs, likely because their proofs were simple. Others *reactively* addressed brittleness issues. P10 explained "we would increase our [resource] limit on random functions, just to get things through expediently." Finally, some were highly impacted by brittleness and proactively addressed brittleness. P5, P6 and P7 faced significant delays, and P6 explaining that they performed "major refactoring" that took a month. As a result, P11 prioritized fixing brittle proofs above all else: "If you ever get a timeout, stop what you're doing. Don't just try to paper it over [...] and hide it, but actually [...] figure out why you have it and get rid of it."

Identifying Sources of brittleness

Participant P7 explained that the root cause of brittleness is an over-reliance on Dafny's automation: "[Dafny] encourages you to get into [...] the worst possible shape in a production project where [...] under the hood the complexity of what Z3 [can do] is at the limits of its ability." Major sources of brittleness included contexts, quantifiers, and frame conditions.

Contexts, Quantifiers Brittleness, according to P8, P7, and P5, is exacerbated by large proof contexts, which can result in gigabytes of queries sent to the solver. This overwhelming proof context size, P5 explained, arises from Dafny including "useless assertions" and "recursive function unrolling" both of which can unnecessarily increase the context size. Such large queries hinder debuggability and trust: "if the proof [...] requires five times the knowledge of humanity [...] I'm sorry, but you need to find a better argument [than the proof]" (P5). Despite this, P5, P7, and P8 reported that the solver sometimes lacked critical information that could simplify the proofs.

Quantifiers and Frame Conditions The amount of information was not the only challenge. P10 also cited *quantifiers* as a source for proof complexity [Zhou et al. 2023]. P6, P7, P8 and P10 identified *frame conditions* — which prove that an object remains valid after modification — as another source of which they "have no control" (P7). Beyond individual sources of complexity, P7 and P10 observed a broader trend: the sizes of the proof contexts increased as they moved higher in the proof tree, *i.e.*, as they proved higher-level properties using lower-level lemmas. This scaling issue led them to question the feasibility of larger proofs, with P10 remarking "if my project was twice as big, I'm not confident that things would work." This observation is counterintuitive as Dafny is a *modular* verifier where each function is checked in isolation, at, one might expect, roughly the same cost, regardless of where the function was in the call-graph. Indeed, one might expect higher-level proofs to be more abstract and therefore *less* complex. However, practical experience shows otherwise: the complexity of specifications frame conditions, and function definition unrolling snowballs as we go up the proof tree, making the proof-contexts ever larger and obligations harder to discharge.

Proof Hardening Strategies

Participants reported three main strategies to mitigate proof complexity: monitoring, adding hints, and changing the visibility of proof facts.

Resource Monitoring P5, P6, P7, P8, and P10 monitored the solver's resource count—an indicator incremented based on the number of solver operations used— to track the proof complexity. While, P5, P7 and P10 set a low resource count to make the proof fail if the limit was exceeded, no participant reported a reliable method for determining this limit. In fact, P7 and P10 reported having to increase the limit over time. This contradicts the intended goal of strict limits. P7 described their approach as "pushing left" by establishing a limit early to avoid major rework later, yet ultimately adjusting it as the proof grew.

Adding Hint Assertions P3, P5, P6, P7, P8, and P10 reported adding assertions as "hints" to help the solver. These assertions are not mandatory, but as P7 noted, they can reduce the proof time

from "a ton of time" to "immediate" as they can significantly reduce the space of terms where quantifiers are instantiated. Meanwhile, P5 and P10 mentioned removing assertions to minimize the context. To understand where to remove or add these assertions, P10 did so iteratively, while P5 used the `unsat core` functionality provided by Dafny. P13 reported using `assert false` to identify where changes are needed: if adding `assert false` (which skips part of proof) reduces the proof time significantly, then the skipped part is responsible for the complexity.

Controlling Visibility Eight participants optimized the visibility of proof facts, ensuring that only relevant information was available to the solver. To do so, P1, P3, P6, and P10 further refined the proof by breaking it up into smaller lemmas to simplify goals and reduce the context size. Additionally, P4, P5, P6, P7, P8, and P10 used a combination of `opaque` and `reveal` to hide or show elements in the proof context, especially for transparent functions. Participant P13 wished `opaque` was the default, as it would require fewer annotations (only when additional context is needed).

In contrast with the top-down approach to proof development (§ 4.2), P7 described doing most of these optimizations in a bottom-up fashion. P4 explained that a top-down approach makes proof optimization difficult: "it's also a dangerous path, because the prover can take five or ten seconds to prove something" (which is slow for an interactive experience).

Style Guides: Towards A Discipline of Proof Hardening To systematically mitigate against brittleness, P6, P7, P11, and P13 adopted structured approaches, including style guides, which aim to reduce automation—and hence the instability due to automation heuristics—as much as possible. For example, P5 advocated for minimal automation by saying "I want Dafny to be as stupid as possible and not help me at all [...]. Make me be really verbose, but [...] make sure the proofs are easy for [Dafny]." Another reported that they "don't play that game [...] if I'm needing to bump the resource count up, I break it down," meaning they divide the proof into smaller lemmas instead of increasing the resource count to automatically verify a large proof. To enforce this principle, they reported using the `isolate-assertion` flag to prove every assertion individually (rather than the default behavior of Dafny, which sends batches of assertions to the solver).

Happily, while brittleness was unanimously acknowledged as a major issue, participants noted that Dafny has improved over time, offering better automation, more visibility features, and clearer guidelines on mitigating the problem [Ver 2023]. As prescribed in these guidelines, several participants noted that brittleness is solvable when they "engineer their proof" appropriately. One participant remarked that "it's just kind of having discipline in the same way, like writing C++ code, you say you can't use these seven features, and you might have to write more code. They can't use templates or something, but they still kind of know what to do."

5 Results: Impact of Verification on Assurance Methods

In this section, we describe how formal verification affects the subsequent *deployment* of software. First, we show how verification influences *testing* by on the one hand, removing the need to exhaustively test implementations, but instead shifting some of that effort to testing formal specifications themselves (§ 5.1). Second, we demonstrate how verification impacts *code review*, again by moving effort to reviewing specifications (not proofs), and the transpiled code generated in various target languages from the (verified) Dafny sources (§ 5.2). Third, we consider how the verified code is *packaged* as libraries that can then be used by clients in production, and the constraints that such clients place on the verified development process itself (§ 6). Finally, we consider how verification impacts the long term *maintenance* of software, by enabling formal analysis to delimit the impact of changes, and thus, fearless optimization (§ 7).

5.1 Testing

"You really want to check that your spec is right, and one way to do that is testing. " — P7

Verification and testing are often seen as "orthogonal" (P7), and in the extreme, the former might be viewed as a replacement for the latter, as formal verification is a way to "test comprehensively [...] on all inputs" (P5). However, in practice, *all* participants that have engineered substantial verified software reported that they have used some form of testing along with verification. Six participants reported using testing as a way to *compare the specification* to some implementation or expected behavior. Thus, to understand how developers test their specifications, we first describe the two main *goals* of verification, and hence, testing, and then, the different *techniques* used to test during verified software development.

Goals

We identified three kinds of testing goals pursued by participants. Some focused on *verified specifications*; their projects aimed to build a reference model of the deployed code, to formally establish desired properties of that model, and to then *compare* the behaviors of the deployed code (which is not verified *per se*), to the reference model. Others pursued *verified implementations*, aiming to formally verify various properties of the code that will actually be used in production to replace a legacy implementation. Here, the main testing requirement was to make sure that this verified implementation did not differ from the legacy (unverified) one. If any discrepancies arose, they needed to change the specification *and* implementation, ensuring that the verified code had the same behavior as the legacy implementation. Finally, six participants were focused on verifying new implementations of new specifications, which did not have any legacy compatibility requirements.

Testing Goals Compatibility is a common requirement. Most participants used testing to prevent the unpleasant surprises that occurred in the past where, per P7, "the [verified code] was wrong compared to the running code." However, the testing goals depended on the goals of the project. When the goal was to build a verified reference model, P2, P4, and P9 reported doing *conformance testing*, which ensured that the production implementation was equivalent to the verified reference model. Instead, when the goal was to build a verified implementation, P5, P6 and P7 reported doing regression testing, comparing behavior in the legacy (unverified) system to that in the new (verified) implementation. P7 motivated this style of testing as a way to "make sure to not break [clients]".

Thus, regardless of whether they were building verified specifications or verified implementations, participants reported testing their code in an *end-to-end* fashion, to make sure either that the reference model (or verified implementation) was equivalent to the deployed code (or legacy code). P4, P5, P7, P9, and P10 pointed out that such end-to-end testing was especially important to gain trust. P7 stated that "shadow mode"—where the verified implementation is run in parallel with the production code—was the "gold standard" and was "what made people believe we could ship." One of the reasons, as P7 explained, is that even after fuzzing the verified code extensively, they found seven or eight differences between the production answers and the verified code after months of running in shadow mode, emphasizing the better coverage of real data. P10 went a step further: "People don't care about our examples [...] they want to see real data in real data out."

Testing Assumptions and Performance As part of testing, P7, P10, and P11 emphasized the importance of using testing to validate any *assumptions* that arise from calls to external library code. Such libraries are typically modeled in Dafny using `:extern` functions with trusted contracts. P6 explained that the importance of testing these contracts (in addition to reading the library code) as "sane contracts don't tell you if it actually models reality." Finally, in the verified implementation paradigm, P7, P10, P11, and P12 reported using testing to measure the *performance* of the target

code generated by Dafny. As the Dafny code is never run as such (§ 6), the performance aspect is something that can only be checked post-transpiling as it depends on the target language.

Testing Correctness of Transpilation One participant reported that "verifying the Dafny code doesn't help if there's a bug in the code generation." The participant "had this little subroutine, and I verified it perfectly true … Fortunately I didn't just trust the verification. I had some tests, too, and the test failed because the code generation was wrong" which could defeat the entire purpose of verification, which doesn't "actually prove that the executing code is correct."

Testing techniques

The end-to-end testing was carried out using a variety of *existing* techniques.

Fuzzing and Unit Testing For example, P2, P5, P6, P7 described using *fuzzing* or *property-based* testing [Fink and Bishop 1997] and P2, P4 reported using unit tests. P5 pointed out that the advantage of using these techniques is that they produce well-established metrics recognized in an industrial setting especially "because there are a lot of company guidelines, [for example] your commit needs to be covered by all tests." Additionally, P6, P12, and P13 explained also having "unit tests which we write in Dafny and transpile alongside the code," not as a substitute for verification, but to ensure that the transpiling did not break any properties of the Dafny code.

Wrappers However, P7 and P10 noted that end-to-end testing is not the best fit to find problems with incorrect libraries axioms as these kinds of tests are expensive and provide poor error localization. To help with that, P10 remarked that a recent feature can "add dynamic checks […] at the interface boundaries," allowing code to fail fast when the assumptions are violated. P6 knew about this feature but had not used it yet. They explained: when there is a new feature they are "never the first person to try it out", illustrating the difficulty of staying up to date with the tool.

5.2 Code Review

> *"The glorious things about proofs is that they are self proving; there is no need to look over somebody else's proof. " — P11*

Ten of our participants reported using code reviews to check for errors. The three participants that did not mention code reviews were working on research projects where they were the sole contributor. Next, we describe how verification changes the nature of code reviews by increasing the *size* of each review and shifting the focus on what needs to be closely vetted by human eyes: *specifications* (especially those used as trusted assumptions) and *transpiled* execution targets.

Review Sizes Three participants (P5, P6 and P7) described code reviews as "quite large" compared to recommended review best practices [Ram et al. 2018]. P6 explained that this increase in size relative to regular software development is due to "multiple components all being updated at the same time": the specification, the implementation (proof), and the transpiled code can all be changed in the same commit, making some participants remark that reviews were "annoying to check because you had three times more things to check than a usual change." P4 and P12 also complained that "the proofs are intertwined with the code," affecting the readability of the review artifact as now methods can be longer than they would be otherwise since they also contain proof annotations.

Reviewing Trusted Specifications. Large review sizes push developers to prioritize review of components that they find critical. In particular, participants reported *not* reviewing proof code, as they were machine checked, and so "there is no need to look over somebody else's proof," thereby justifying not reviewing verified code. P6 summarized this prioritization as "the specification bit gets the most attention, and then the implementation, and then the transpiled [code]." P9 explained the focus on specifications: "reviews [are] the only time to check if the pre or post conditions are

strong enough." P11 goes even one step further, as they explain that they only review the code that is "trusted" (*i.e.* not machine verified) meaning the specifications for `extern` library methods. Indeed, the dependencies such specifications and the underlying implementation can be a source of errors as P6 attested of a case where "the actual code was changed but the contracts were not," thereby introducing an unsoundness in the specification arising from a mismatch between the semantics of the updated code and that of the assumed contract used for client verification.

However, P11 remarked having had to sometimes read the proof "if maintainability is a problem" (§ 4.4). In this case, their main concern is clarity as they "would ask them to put in either a comment or an assertion" in cases where the proof is too hard to understand. This choice between comments and assertions is interesting as they are not exactly equivalent as assertions are "active comments" which have an effect on the readability of the code and on the time taken by Dafny to carry out the verification. Thus, developers try to use assertions only when they help harden the proof (§ 4.4).

Reviewing Unnecessary Specifications One exception to the general trend of not reviewing verified code is that sometimes developers found it valuable to review (verified) specifications to detect any unnecessary clauses. Participants P4, P8, P9, and P10 reported one common and easy smell is a *missing correspondence* between the pre- and post-conditions for a procedure, *i.e.* where a contract contains pre-conditions that are *not necessary* to prove the contract's post-conditions. For example, P9 observed that engineers without formal methods background often "implemented data structures with preconditions but there was no protocol or model on the other side against which these preconditions would be verified."

P9 mentioned an example (shown in fig. 1): the method `applyDiscount`, which applies a percentage to a given number `x`. The problem is that the precondition line 3—*intended* to check that the discount is less than a 100—is not, in fact, required to establish the postcondition on line 4. If this precondition is indeed required, then the postcondition should be strengthened with **`ensures`** clauses such as the one on line 6. This issue stems from a disconnect in the user's mental model between what is true at *run-time* and what is relevant to establish the desired properties at *compile-time*. To review such contracts, P9 advised "You should always start from your postcondition and make that as tight as possible and then try to have the most liberal precondition on top. If you think your precondition is too liberal then you know your postcondition is too weak."

Reviewing Transpiled Targets As discussed in § 6, a major reason for selecting Dafny is that the verified implementation can then be *transpiled* into a variety of target languages. Of course, the transpiled code, in Java or Go, is not formally verified, and hence, developers spend time to carefully review it. P5, P6, P7, P11 and P13 acknowledged looking at the transpiled code but at different frequencies. At one extreme, P11 and P13 reported reviewing the target code only when they needed to investigate why something was not working correctly or to find opportunities for optimization. Developers can only directly investigate the transpiled target code as Dafny does not have any debugging and profiling tools that could be used on the verified source. On the other extreme, P5, P6 and P7 explained that they regularly inspected the transpiled code, because they lack trust in the transpiler compared to the case of classical compilation, *e.g.* of Java source, where they "don't check the JVM bytecode." These participants hoped that this trust deficiency will be remedied in the future. P7 observed that looking at the transpiled target was at odds with the idea of viewing that code as a "build artifact" and that "in principle we didn't need to get it code reviewed and just shove the changes into a version set."

Participants noted additional benefits beyond correctness from considering the transpiled code in the review. Participants described it as "more democratic" meaning that it *facilitates collaboration* with other teams that are more familiar with the target language than Dafny. P5 confirmed that "it will be the [transpiled] Java that will be read by engineering production or security teams" and

consequently, they care deeply about the quality of the transpiled code and prioritize reviewing it as they want it to "be perfect" before other teams review it.

6 Results: Impact of Verification on Packaging

"Any code you're taking from the outside cannot have any preconditions on it." — P7

Eight participants reported *packaging* their verified Dafny code as a library that can be called in the target language after transpiling the verified code into the target language. Next, we describe the motivation behind such *transpiler-based packaging* and the constraints it places on specification.

Transpiling into Target Packages

Many participants reported choosing Dafny in part because of its support for transpilation into multiple target languages. P7 compared Dafny to F*, a similar automated verification language but with which "connecting to *Go* would have been a challenge." Most of our participants report transpiling to a *single* target language: Rust (P2), Java (P4, P5, P6, P7, P9), C (P8) Go (P10), C# (P11).

Verify Once, Run Everywhere Only P12 and P13 reported using *multiple* targets: Java, C#, Go, Rust. Interestingly they report this was the main reason they chose Dafny, in fact, prioritizing the ability to target multiple languages over the verification capabilities. Dafny allowed them to write the implementation in one language which could then be transpiled to different targets. They preferred this to having to write multiple implementations in different languages, or using FFIs, the former being too complicated to maintain, and the latter over-burdening the library's users.

The Importance of Correct Transpilation The entire enterprise of verification only works if the transpiler for Dafny to the target language(s) works. Eight participants mentioned this risk, which forces them to review the generated code as discussed in § 5.2. P9 reported that "they caused a few compiler crashes by just using it for months." P12 also pointed out that bugs in the compiled artifact are sometimes a product of these target languages changing with the example of Go that "changes a lot between versions." We note, however, that Dafny has an extensive test suite for each of its runtimes and has also reported using XD Smith [Irfan et al. 2022], a fuzzing framework for Dafny, to perform differential testing between the different transpilers.

Keep Dafny Close to the Target The main challenge when targeting multiple languages is that "writing Dafny code which is easily compilable to one language doesn't mean you've got Dafny code which is easily compilable to a different language." A few participants, P5, P7, and P9, also reported caring about readability of the target code, as per P5, "it will be the [transpiled] Java code that will be read by engineering and production teams." P13 observed that types and data structures are different between languages. Indeed, seven participants noted that the transpiler produces unoptimized, slow code. P7 described the example of the Dafny built-in strings which, on transpilation, would induce "a bunch of painful, slow, useless copying of inputs."

To improve readability and performance, participants reported avoiding relying on the transpiler being smart, instead writing Dafny close to idiomatic target code. For example, instead of using Dafny's built-in strings, they directly used types closer to those in the intended target. This meant having to separate some part of the implementation depending on the target language, which has the downside of incurring a lot of additional work (instead of being able to rely on the transpiler.) To reconcile this tension, P5, P6 and P7 reported using a bespoke transpiler: "we are using this idiomatic compiler which is not destroying the code, because once you write things in Dafny, potentially, the original Dafny compiler may change data types and doesn't give you control over which data, operation or library you want to use."

Package Specifications

Explicating Assumptions Several participants reported that packaging into libraries encourages developers to write concise specifications (§ 4.1). P11 observed: "if the user of your program is another program then it's more likely that its API is understandable and compact." Further, as P6 observed, library packaging "brings consistency out of the box" through a unified specification that "makes it easier for service teams to know what the default behavior should be." This consistent behavior can be used as an argument for better reliability, and is also a way "to reduce the efforts the engineers have to put integrating." Next, we describe how packaging affects the *assumptions at the boundaries*.

Enforcing Assumptions via Dynamic Checks Indeed, avoiding assumptions on the input is the safest bet as invalid assumptions can impact the guarantees provided by verification. To avoid that, P7 reported that they checked that the inputs are valid, through "a wrapper [...] that would crash if the input didn't satisfy the preconditions that you need." Such run-time checks are not at odds with static verification guarantees, as these fail fast cases are included in the specification and are part of the contract of the library. Interestingly, one participant reported using the opposite approach where they were willing to *trust* that the unverified serializer and deserializer were giving them inputs that were correct, as they had control over the implementation of those components.

7 Results: Impact of Verification on Maintenance

Our findings contrast with the conventional wisdom that verification renders the code difficult to change and maintain, as each change requires updating the *specification* and *proof* in addition to the code. In particular, our study shows participants found that formal specifications enabled the precise analysis of the *impacts* of code changes, and hence, formal verification allowed for aggressive code *optimizations* without fear of breaking functionality.

Specification Enables Impact Analysis

Change impact analysis [Arnold and Bohner 1996] is about determining the implications of code changes. Several participants mentioned how specifications were a way to scope the effects of change, or as P5 put it, "another approximation of what will break [when code is changed]." Indeed, P9 explains that this is one of the reasons they chose to verify their (reference) model in the first place as it allows them to "see what happens if something has changed." The rationale is that if after a change in the code the specification fails to hold, participants could quickly infer what the behavior has been affected. This allowed them to narrow the scope and move from "changing a whole thing into changing small things," as P5 explained, potentially focusing testing efforts solely on the affected parts. Additionally, repairing or retrofitting the specification and the proof with "small fixes" gave more confidence to P10 that their change did not have any outstanding effects.

Finally, the specification itself can be used to *argue for* a change. Participant P7 reported an example where the specification exposed a "non-local control flow" that was not obvious in the code. This resulted in pushing for a simplification of the code, which translated to a simpler specification and proof. Importantly, as P5 pointed out, all of this is only possible "after the first [verified] deployment," as you can make these changes by using the existing specification.

Verification Enables Fearless Optimization

Several participants reported that verification allowed them to implement, as P11 put it, "optimizations that normal developer[s] would never [consider]". In fact, P12 explained that it is "where Dafny really pulled its weight" as "localize[d] changes are easy and straightforward," meaning that changes that do not deeply affect the specification can be quickly and confidently implemented.

Optimizing Computations P5 described these changes as "algorithmic optimization": they do not modify the end result but instead simplify the underlying computation. P7 reported using this as way to show results of verification early. They individually verified one of the key functions of their system, that they then optimized before verifying the full system. This allowed them to demonstrate the value of verification and get buy-in from various stakeholders.

Optimizing Redundant Checks P12 and P13 reported using verification to remove redundant checks on function arguments, thereby speeding up the code, especially if the function is called often. P12 explained that this is made possible by using `requires` clauses, which can check statically that the arguments are valid, and thus do not need to check them at runtime.

Modularity and Change We observed a tension between breaking Dafny code into smaller pieces (methods and functions) to make checking faster and avoid brittle proofs § 4.4 versus maintainability. For example, P10 noted "you want methods to be very short [...] So this creates a bunch of work. You have to write a spec for the intermediate function," which can complicate refactoring. It can be difficult to build compact interfaces when each function is very small, as compact interfaces might require bundling together multiple functions —something often avoided to keep proofs small.

8 Opportunities for Improving Verified Software Engineering

> *"We have just spent less time thinking about specifications and proof as first class citizens than we have for programs." — P4*

We now distill our findings into recommendations for improving verified software engineering. We first discuss the implications of our findings on how the process of specification and proof could be simplified. We then highlight ways to tighten the integration between verification and the rest of the development process, and to streamline the long-term maintenance of verified code.

8.1 Engineering Specification and Proof

The assessment from P4 at the start of the section summarizes the sentiment that if we are to fully exploit the capabilities of verifiers, we need to find ways to make *proving* as accessible as *programming*. One indication of the current gap between the two is the observation that the perception of the verifier depends on the background of the participants. On one hand, participants with a formal methods background appreciate Dafny's "simplicity" (P7), and reported that they learned it "on the fly" (P5). On the other hand, participants with a software engineering background complained that the learning curve is "steep" (P1) and sometime struggle to see the value of the extreme effort they put to not only learn but also to use the verifier. We can learn from both perspectives. The formal methods view suggests that verification in Dafny is feasible for many large-scale software systems but requires substantial effort and specialized skills. The software engineering view suggests that we are not yet at a point where the available resources and tooling make this effort feel pleasurable or worthwhile for most software engineers.

To this end, we recommend two areas of improvement to advance verified software engineering as a more mainstream and accessible practice. First, we should study what constitutes a clean proof. Second, we should enhance the interactivity of the verifier. We believe Dafny as language has made significant strides towards this goal, with more than fifty releases since its creation, introducing new features, bug fixes, performance improvements, better error messages, and an improved client. However, there are still complicated research problems that we need to solve as a community.

What is a Good Proof?

Further research should identify what constitutes a good proof and develop ways of *enforcing* these lessons via style guides or linters. As we have seen, techniques can help write the specification (§ 4.1)

and the proof (§ 4.2) to prevent problems such as unsoundness or incompleteness. However, how to write clear, performant, and robust (not brittle) verified code is still an open question. The answer will likely differ across users or projects. While some style guides exist in different projects [Daf 2023; Research 2015], they are not extensive, and they do not explain the rationale behind the style choices. Although the Dafny IDE provides visual cues to identify potential problems along with the warnings in the client itself, it is not configurable. Additionally, the available warnings are somewhat limited, as for example they do not cover aspects such as variable naming.

Recommendation: Style Guides and Linters Since different users will likely need different guidelines, we need to be able to enforce these guidelines through linters in a configurable manner. Thus, we recommend that the auto-active verification community work on extensible linters like Megdiche et al. [2022] and quantitative analyses like Huch and Stathopoulos [2023] to identify good proof styles and patterns. Further, we recommend adapting efforts like Martin [2008] and Ousterhout [2018], to the intersection of specifications, code, and proof. These efforts would go beyond existing style guides by providing general guidelines for naming conventions, code organization, and proof structure.

How to Improve Interactivity

Development with verifiers can feel like a slog in the dark, trying to decipher error messages and avoid unpredictable timeouts, while guided only by pricks of light from manually inserted assertions (§ 4.3) Dafny integrates *counterexamples* into the Dafny IDE and CLI [Chakarov et al. 2022] that are supposed to help with debugging. However, in our study, only P4 reported using counterexamples, and with mixed results: "It can't really give you a counterexample, or the counterexample it gives you is because it has insufficient visibility into the objects of the counterexample, and distinguishing those situations from genuine counterexamples [...] requires experience." Alternatively, tools like ProofPlumber [Cho et al. 2024] can suggest potential fixes, and Laurel [Mugnier et al. 2025a] can automatically generate assertions. In practice, however, they remain sparsely used, due to limited support and engineering resources. The situation may be improved by introducing better *feedback mechanisms* that make the verifier's context more explicit, and hence the process of proving more *interactive*, which might help users more easily build a mental model. "You could visualize the context of what the verifier receives, and you could define what that means [...] that would help bring it closer to the predictability of Rocq proofs" (P10). Of course, achieving this is not straightforward, as it requires *defining* what "context" means in the auto-active verifier's case, and finding a good way to *visualize* such contexts.

Defining Contexts One approach is to define the context as all the assumptions gathered from the code that apply to a given assertion, which would reduce the need to constantly navigate back and forth in the code to check lemma definitions and their postconditions. However, this would not entirely bridge the gap between the verifier and the user, as the SMT solver can infer new facts that are *consequences* of the above assumptions, but are not explicit in the code. Explicating such facts is hard for two reasons. First, these facts may, for example, capture relationships that are not present in the code. Indeed, some relationships may be syntactically inexpressible, as, *e.g.* they may be about the values of the same variable but at different points in time. Second, we want to avoid overwhelming the developer with excessive information that is irrelevant or is overly complex.

Visualizing Contexts Another challenge is determining the best way to visualize proof contexts. One approach is to adopt something similar to Rocq or Lean, where the user can see the context of the proof at any time. However, it is unclear whether this is the best approach, as unlike proof assistants, verifiers might present incomplete or redundant information. Another possibility is to allow users to interactively query the solver, similar to debuggers that display values at runtime.

Participant P10 described this idea: "you're running an application [...] it throws a seg fault [...] and then [in GDB] you're paused in the in the current stack effectively. I would love [...] being able to drop into some kind of interactive environment where I could probe [...] the state of the solver, or [...] ask to solve normal questions more interactively." A counterargument to this approach is that users are already effectively doing this when they insert assertions to debug their proofs.

Recommendation: Incremental & Interactive Contexts Thus, we believe the path forward is to find ways to appropriately determine the entire context that is relevant at a given program point or place where a verification error is reported and find ways to either visualize it completely—or interactively in the style of a debugger.

8.2 Integrating Verification with Other Development Processes

Our study shows that other than the specification and proof phases (§ 4), many of the other phases of the verified software development process are similar to those of (non-verified) software engineering. For example, as noted by P7 (§ 5.1), testing and verification are orthogonal, and projects that pushed their products to production still relied on some form of testing.

However, our study also shows that these processes are currently disconnected. Many participants reported having to build custom tooling to integrate their verified code into the traditional development process. For example, P12 and P13 discussed Duvet, a tool developed in house, to connect informal *requirements* and formal specifications, and P4 built a client to *test* the reference model itself. Similarly, P7 and P6 created custom scripts to generate separate *code reviews* for the different artifacts, *i.e.*, the generated (transpiled) code and the specification. Additionally, several participants expressed a desire for tools that would bridge the gap between the verified code and the broader software development process. Consequently, we recommend lines of work to more tightly integrate the formal specifications and proof with *testing*, *review*, and *versioning*.

Specification & Testing

Several participants expressed a desire for better integration between testing and specification.

Recommendation: Testing External Library Contracts Tests are particularly useful in validating the axiomatically trusted contracts for external library code (§ 5.1). For example, P7 noted that "It would be nice to have a standard library for Java that was tested against all of the axioms we wrote about those about the library functions." One could imagine a tool that automatically generates tests to confirm or refute such contracts [Goldweber et al. 2024; Seidel et al. 2015]. While such a tool would not provide the same level of guarantees as verifying the library, it could identify issues prior to end-to-end testing, which is when participants reported discovering them. P13 proposed a more radical approach: a framework where specifications "rests its truth on execution." In this model, a lemma would be converted into tests, allowing developers to bypass proof obligations and instead confirm correctness by testing. Such a framework could serve as a practical way to *bootstrap* the specifications, particularly for well-known algorithms where proving correctness is tedious.

Recommendation: External Library Contracts from Tests Dually, we envision using existing tests to be automatically generate the library contracts. As discussed in § 4.1, the primary challenge when using tests as a way to infer the specification is to generalization, perhaps by using ideas from the literature on generalizing dynamic executions into invariants [Ernst et al. 2007].

Verification & Code Review

Verified code reviews tend to be long and error-prone (§ 5.2). In contrast, in the programming languages community, proof assistants are increasingly used to *shorten* papers by omitting explicit

proofs, letting reviewers focus on the theorems, since the proof is machine verified [Ringer 2020]. Unfortunately, in the case of verified code, we contend with the additional challenges of (1) the proof being intertwined with the implementation, and (2) the requirement that the code itself be readable and maintainable: reviewing the specification alone is often insufficient as the reviewer must also assess algorithmic complexity of the code and the maintainability of proofs (§ 5.2).

Recommendation: Multiview Reviews Consequently, we recommend rethinking how verified code is reviewed, by adopting a more tool-structured separation of concerns. Currently, a review has a single monolithic textual diff, making it difficult to focus on individual aspects. Instead P12 proposed an approach where a tool that enabled targeted reviews by "Hiding all the comments, all the asserts, the requires, the assumes, and everything that doesn't generate code. And let me just see the actual executable bits just to be able to see what's happening." A strictly one-component-at-a-time approach might be too simplistic, as it might blind reviewers to relationships between different parts of the verified code. A more flexible design could enable reviewers to toggle their focus while still maintaining a holistic view of the codebase.

9 Related work

Usability of automated verifiers To our knowledge, Dafny is the main automated verifier whose usability has been studied, primarily through case studies or experience reports. Faria and Abreu [2023] presents ten cases of verifying well-known algorithms in Dafny, highlighting challenges such as proof debugging and the need for specification testing. We found that these issues also apply to larger scale verification projects and explores the impact of the software development process, including testing, deployment, and maintenance. Other studies have examined the experience of verifying programs in Dafny [de Muijnck-Hughes and Noble 2024; Noble et al. 2024] or teaching Dafny to students [Noble 2024]. Notably, they also recognize that while verifying in Dafny is time-consuming, its similarity to imperative programming makes it accessible. de Muijnck-Hughes and Noble [2024] compares the Idris proof assistant to Dafny, finding that Dafny enables shorter proofs but lacks transparency about its correctness. This supports our findings in § 4.3: while proof assistants may be more complex, they seem more interactive than automated tools such as Dafny.

Pearce [2015] presents usability hypotheses based on their experience with Whiley [Pearce and Groves 2013]. Our findings confirm some of their hypotheses, particularly regarding the difficulty of writing assertions and the necessity of testing specifications to detect errors. Gamboa et al. [2025] studies the usability of LiquidHaskell [Vazou et al. 2014], which is integrated with Haskell [Marlow 2010]. While some of their findings are specific to LiquidHaskell, such as the unclear divide between Haskell and the Liquid types, they share some of our findings, such as the difficulty of understanding certain error messages.

Usability of verification tools at large Prior studies have applied human-centered interaction approaches to understand the challenges users face with proof assistants. While they differ in programming models and automation techniques, they ultimately achieve the same goal and can provide valuable insights. Shi et al. [2025] presents a contextual study of Rocq and Lean users, analyzing how users engage with these tools within their own projects. Although they focus on immediate interaction rather than the development process at large, they similarly highlight the importance of interactivity and proof design. Aitken et al. [1998] study seven HOL [Nipkow et al. 2002] users implementing list data structures. They found that experts completed the task more efficiently than novices. While one might expect that verifiers' automation would alleviate this problem, we observe similar issue due to the lack of interactivity.

Other works [Andronick et al. 2012; Staples et al. 2014] have examined one of the largest formal verification projects, the seL4 microkernel [Klein et al. 2009], documenting insights from real-world

verification. Similarly to our study, some of their findings only appeared at scale, such as the relation between verification and performant code, underscoring the need to study large projects.

Survey studies [Huang et al. 2024; Ringer et al. 2019] have analyzed formal verification projects and the lessons they offer. Ringer et al. [2019], presents the concept of proof engineering, —designing scalable proofs through careful structuring. While Dafny requires different design considerations, we also observe the importance of engineering proofs effectively.

Usability of programming languages and software development tools Several works have applied qualitative methods to understand the usability of programming languages [Coblenz et al. 2021; Fulton et al. 2021; Lubin and Chasins 2021; Rennels and Chasins 2023] and their tools [Barke et al. 2023]. Like our study, Fulton et al. [2021], also interviewed developers to understand the benefits and challenges of a language, Rust. While Rust offers different guarantees from Dafny, it has a steep learning curve and requires significant effort to write correct code, similar to what we found with Dafny. Finally, several studies have explored software development practices and their impact on the final deliverables, for example focusing on how software development is applied [Adolph et al. 2012] or the effect of agile methodologies [Waterman et al. 2015]. Our study uses a similar approach to understand a different domain—verification with automated verifiers.

10 Conclusion

We identified key ways that auto-active verification affects software development processes. In addition to the usual software development steps, developers must also debug their proofs and harden their code and proofs to keep them valid in light of future changes. They continue to use traditional testing approaches, in part because they need to ensure the specifications are correct and in part because some properties may be left unspecified. However, verification opens new opportunities for software engineers, who can improve performance and make other changes with lower risk of introducing regressions. Developers from formal backgrounds have different expectations of verification tools and techniques for using them than developers from software engineering backgrounds. Our findings offer new opportunities to improve the process of verified software engineering.

Data-Availability Statement

Our artifact is available on Zenodo [Mugnier et al. 2025b] and consists of two documents:

(1) our interview materials, including the recruitment email and the information sheet that was provided to the participants
(2) our study codebook, that was used to analyze the transcripts.

Acknowledgments

We thank our anonymous reviewers for their valuable feedback. This work was supported by the Qualcomm Chair Endowment.

References

2015. UpperBound.i.dfy. https://github.com/microsoft/Ironclad/blob/2fe4dcdc323b92e93f759cc3e373521366b7f691/ironfleet/src/Dafny/Distributed/Impl/Common/UpperBound.i.dfy#L10.

2023. Dafny VMC Guidelines. https://github.com/dafny-lang/Dafny-VMC/blob/main/Guidelines.md.

2023. Verification Optimization. https://dafny.org/latest/VerificationOptimization/VerificationOptimization

Steve Adolph, Philippe Kruchten, and Wendy Hall. 2012. Reconciling perspectives: A grounded theory of how people manage the process of software development. *Journal of Systems and Software* (2012). https://doi.org/10.1016/j.jss.2012.01.059

J.S. Aitken, P. Gray, T. Melham, and M. Thomas. 1998. Interactive Theorem Proving: An Empirical Study of User Activity. *Journal of Symbolic Computation* 25, 2 (1998), 263–284. https://doi.org/10.1006/jsco.1997.0175

June Andronick, Ross Jeffery, Gerwin Klein, Rafal Kolanski, Mark Staples, He Zhang, and Liming Zhu. 2012. Large-scale formal verification in practice: A process perspective. In *2012 34th International Conference on Software Engineering (ICSE)*. 1002–1011. https://doi.org/10.1109/ICSE.2012.6227120

RS Arnold and SA Bohner. 1996. An introduction to software change impact analysis. *Software Change Impact Analysis* (1996), 1–26.

AWS Labs. 2025. Duvet: A requirements traceability tool. https://github.com/awslabs/duvet.

Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. OOPSLA1 (2023). https://doi.org/10.1145/3586030

Scott O. Bradner. 1997. Key words for use in RFCs to Indicate Requirement Levels. RFC 2119. https://doi.org/10.17487/RFC2119

Aleksandar Chakarov, Aleksandr Fedchin, Zvonimir Rakamaric, and Neha Rungta. 2022. Better Counterexamples for Dafny. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13243)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 404–411. https://doi.org/10.1007/978-3-030-99524-9_23

Aleks Chakarov, Jaco Geldenhuys, Matthew Heck, Mike Hicks, Sam Huang, Georges Axel Jaloyan, Anjali Joshi, Rustan Leino, Mikael Mayer, Sean McLaughlin, Akhilesh Mritunjai, Clément Pit Claudel, Sorawee Porncharoenwase, Florian Rabe, Marianna Rapoport, Giles Reger, Cody Roux, Neha Rungta, Robin Salkeld, Matthias Schlaipfer, Daniel Schoepe, Johanna Schwartzentruber, Serdar Tasiran, Aaron Tomb, Emina Torlak, John Tristan, Lucas Wagner, Mike Whalen, Remy Willems, Jenny Xiang, Tae Joon Byun, Joshua Cohen, Ruijie Wang, Junyoung Jang, Jakob Rath, Hira Taqdees Syeda, Dominik Wagner, and Yongwei Yuan. 2025. Formally verified cloud-scale authorization. (2025). https://www.amazon.science/publications/formally-verified-cloud-scale-authorization

Kathy Charmaz. 2014. *Constructing grounded theory*. SAGE publications Ltd.

Chanhee Cho, Yi Zhou, Jay Bosamiya, and Bryan Parno. 2024. A Framework for Debugging Automated Program Verification Proofs via Proof Actions. In *International Conference on Computer Aided Verification (CAV)*. https://www.microsoft.com/en-us/research/publication/a-framework-for-debugging-automated-program-verification-proofs-via-proof-actions/

Michael Coblenz, Gauri Kambhatla, Paulette Koronkevich, Jenna L. Wise, Celeste Barnaby, Joshua Sunshine, Jonathan Aldrich, and Brad A. Myers. 2021. PLIERS: A Process that Integrates User-Centered Methods into Programming Language Design. *ACM Trans. Comput.-Hum. Interact.* 28, 4, Article 28 (July 2021), 53 pages. https://doi.org/10.1145/3452379

J. de Muijnck-Hughes and J. Noble. 2024. Colouring Flags with Dafny & Idris.

Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.* 69, 1-3 (2007), 35–45. https://doi.org/10.1016/J.SCICO.2007.01.015

João Pascoal Faria and Rui Abreu. 2023. Case Studies of Development of Verified Programs with Dafny for Accessibility Assessment. In *Fundamentals of Software Engineering*.

George Fink and Matt Bishop. 1997. Property-based testing: a new approach to testing for assurance. *SIGSOFT Softw. Eng. Notes* 22, 4 (July 1997), 74–80. https://doi.org/10.1145/263244.263267

Kelsey R. Fulton, Anna Chan, Daniel Votipka, Michael Hicks, and Michelle L. Mazurek. 2021. Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study. In *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*. USENIX Association, 597–616. https://www.usenix.org/conference/soups2021/presentation/fulton

Catarina Gamboa, Abigail Reese, Alcides Fonseca, and Jonathan Aldrich. 2025. Usability Barriers for Liquid Types. *Proc. ACM Program. Lang.* PLDI (June 2025). https://doi.org/10.1145/3729327

Eli Goldweber, Weixin Yu, Seyed Armin Vakil Ghahani, and Manos Kapritsos. 2024. IronSpec: Increasing the Reliability of Formal Specifications. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 875–891. https://www.usenix.org/conference/osdi24/presentation/goldweber

Greg Guest, Arwen Bunce, and Laura Johnson. 2006. How Many Interviews Are Enough?: An Experiment with Data Saturation and Variability. *Field Methods* 18, 1 (2006), 59–82. https://doi.org/10.1177/1525822X05279903 arXiv:https://doi.org/10.1177/1525822X05279903

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*. https://doi.org/10.1145/2815400.2815428

Li Huang, Sophie Ebersold, Alexander Kogtenkov, Bertrand Meyer, and Yinling Liu. 2024. Lessons from Formally Verified Deployed Software Systems (Extended version). arXiv:2301.02206 [cs.SE] https://arxiv.org/abs/2301.02206

Fabian Huch and Yiannos Stathopoulos. 2023. Formalization Quality in Isabelle. In *Intelligent Computer Mathematics: 16th International Conference, CICM 2023, Cambridge, UK, , September 5–8, 2023 Proceedings* (Cambridge, United Kingdom). Springer-Verlag, Berlin, Heidelberg, 142–157. https://doi.org/10.1007/978-3-031-42753-4_10

Ahmed Irfan, Sorawee Porncharoenwase, Zvonimir Rakamarić, Neha Rungta, and Emina Torlak. 2022. Testing Dafny (experience paper). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*

(Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 556–567. https://doi.org/10.1145/3533767.3534382

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. https://doi.org/10.1145/1629575.1629596

Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types. *Proc. ACM Program. Lang.* OOPSLA1 (2023), 286–315. https://doi.org/10.1145/3586037

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Edmund M. Clarke and Andrei Voronkov (Eds.). https://doi.org/10.1007/978-3-642-17511-4_20

K Rustan M Leino and Michał Moskal. 2010. Usable auto-active verification. In *Usable Verification Workshop*.

Rustan Leino. 2008. Specification and Verification of Object-Oriented Software. In *Marktoberdorf International Summer School 2008*. https://www.microsoft.com/en-us/research/publication/specification-verification-object-oriented-software/

Xavier Leroy. 2006. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd ACM symposium on Principles of Programming Languages*. ACM Press, 42–54. http://xavierleroy.org/publi/compiler-certif.pdf

Justin Lubin and Sarah E. Chasins. 2021. How statically-typed functional programmers write code. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 155 (Oct. 2021), 30 pages. https://doi.org/10.1145/3485532

Simon Marlow. 2010. Haskell 2010 Language Report. (07 2010).

Robert C. Martin. 2008. *Clean Code: A Handbook of Agile Software Craftsmanship* (1 ed.). Prentice Hall PTR, USA.

Nora McDonald, Sarita Schoenebeck, and Andrea Forte. 2019. Reliability and Inter-rater Reliability in Qualitative Research: Norms and Guidelines for CSCW and HCI Practice. *Proc. ACM Hum.-Comput. Interact.* 3, CSCW, Article 72 (Nov. 2019), 23 pages. https://doi.org/10.1145/3359174

Yecine Megdiche, Fabian Huch, and Lukas Stevens. 2022. A Linter for Isabelle: Implementation and Evaluation. *CoRR* abs/2207.10424 (2022). https://doi.org/10.48550/ARXIV.2207.10424 arXiv:2207.10424

Eric Mugnier, Emmanuel Anaya Gonzalez, Nadia Polikarpova, Ranjit Jhala, and Zhou Yuanyuan. 2025a. Laurel: Unblocking Automated Verification with Large Language Models. 9, OOPSLA1 (2025). https://doi.org/10.1145/3720499

Eric Mugnier, Yuanyuan Zhou, Ranjit Jhala, and Michael Coblenz. 2025b. Artifact for "On the Impact of Formal Verification on Software Development". https://doi.org/10.5281/zenodo.15761040

Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. 2002. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, Berlin, Heidelberg.

James Noble. 2024. Learn 'em Dafny!

James Noble, Julian Mackay, Tobias Wrigstad, Andrew Fawcet, and Michael Homer. 2024. Dafny vs. Dala: Experience with Mechanising Language Design. In *Proceedings of the 26th ACM International Workshop on Formal Techniques for Java-like Programs*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3678721.3686228

John Ousterhout. 2018. *A Philosophy of Software Design* (1st ed.).

David J. Pearce. 2015. Some usability hypotheses for verification. In *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2015)*. https://doi.org/10.1145/2846680.2846691

David J. Pearce and Lindsay Groves. 2013. Whiley: A Platform for Research in Software Verification. In *Software Language Engineering*. Springer International Publishing, Cham, 238–248.

Achyudh Ram, Anand Ashok Sawant, Marco Castelluccio, and Alberto Bacchelli. 2018. What makes a code change easier to review: an empirical investigation on code change reviewability. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 201–212. https://doi.org/10.1145/3236024.3236080

Lisa Rennels and Sarah E. Chasins. 2023. How Domain Experts Use an Embedded DSL. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 275 (Oct. 2023), 32 pages. https://doi.org/10.1145/3622851

Microsoft Research. 2015. Ironfleet Style Guide. https://github.com/microsoft/Ironclad/blob/2fe4dcdc323b92e93f759cc3e373521366b7f691/ironfleet/STYLE.md.

Talia Ringer. 2020. Mechanized Proofs for PL: Past, Present, and Future. https://blog.sigplan.org/2020/01/29/mechanized-proofs-for-pl-past-present-and-future/.

Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. 2019. QED at Large: A Survey of Engineering of Formally Verified Software. *Found. Trends Program. Lang.* (2019). https://doi.org/10.1561/2500000045

Neha Rungta. 2024. Trillions of Formally Verified Authorizations a day! https://2024.splashcon.org/details/splash-2024-keynotes/3/Trillions-of-Formally-Verified-Authorizations-a-day-.

Eric L. Seidel, Niki Vazou, and Ranjit Jhala. 2015. Type Targeted Testing. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9032)*, Jan Vitek (Ed.). Springer, 812–836. https://doi.org/10.1007/978-3-662-46669-8_33

Jessica Shi, Cassia Torczon, Harrison Goldstein, Benjamin C. Pierce, and Andrew Head. 2025. QED in Context: An Observation Study of Proof Assistant Users. *OOPSLA*. https://doi.org/10.1145/3720426

Mark Staples, Ross Jeffery, June Andronick, Toby Murray, Gerwin Klein, and Rafal Kolanski. 2014. Productivity for proof engineering. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (Torino, Italy). Association for Computing Machinery, New York, NY, USA, Article 15, 4 pages. https://doi.org/10.1145/2652524.2652551

Xudong Sun, Wenjie Ma, Jiawei Tyler Gu, Zicheng Ma, Tej Chajed, Jon Howell, Andrea Lattuada, Oded Padon, Lalith Suresh, Adriana Szekeres, and Tianyin Xu. 2024. Anvil: Verifying Liveness of Cluster Management Controllers. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 649–666. https://www.usenix.org/conference/osdi24/presentation/sun-xudong

Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/2837614.2837655

Nikhil Swamy, Tahina Ramananandro, Aseem Rastogi, Irina Spiridonova, Haobin Ni, Dmitry Malloy, Juan Vazquez, Michael Tang, Omar Cardona, and Arti Gupta. 2022. Hardening attack surfaces with formally proven binary format parsers. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 31–45. https://doi.org/10.1145/3519939.3523708

Aaron Tomb. 2023. Clear Separation of Specification and Implementation in Dafny. https://dafny.org/blog/2023/08/14/clear-specification-and-implementation/.

Niki Vazou, Eric L Seidel, and Ranjit Jhala. 2014. LiquidHaskell. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell*. ACM.

Michael Waterman, James Noble, and George Allan. 2015. How Much Up-Front? A Grounded theory of Agile Architecture. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. https://doi.org/10.1109/ICSE.2015.54

Niklaus Wirth. 1971. Program development by stepwise refinement. *Commun. ACM* 14, 4 (April 1971), 221–227. https://doi.org/10.1145/362575.362577

Yi Zhou, Jay Bosamiya, Yoshiki Takashima, Jessica Li, Marijn Heule, and Bryan Parno. 2023. Mariposa: Measuring SMT Instability in Automated Program Verification. In *2023 Formal Methods in Computer-Aided Design (FMCAD)*. 178–188. https://doi.org/10.34727/2023/isbn.978-3-85448-060-0_26

Ziqiao Zhou, Anjali, Weiteng Chen, Sishuai Gong, Chris Hawblitzel, and Weidong Cui. 2024. VeriSMo: A Verified Security Module for Confidential VMs. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 599–614. https://www.usenix.org/conference/osdi24/presentation/zhou

Received 2025-03-26; accepted 2025-08-12