WIP: Tunable Automation in Automated Program Verification

Alexander Y. Bai[1,2][0009−0009−7458−7864], Chris Hawblitzel[2][0000−0002−5676−0362], and Andrea Lattuada[1][0000−0002−9303−452X]

[1] MPI-SWS abai@mpi-sws.org, andrea@lattuada.me
[2] New York University ayb5065@nyu.edu
[3] Microsoft Research chris.hawblitzel@microsoft.com

Abstract. Automated verification tools based on SMT solvers have made significant progress in verifying complex software systems. However, these tools face a fundamental tension between automation and performance when dealing with quantifier instantiation—the primary source of incompleteness and verification slowdown in SMT-based verifiers. Tools choose between aggressive quantifier instantiation that provides more automation but longer verification times, or conservative instantiation that responds quickly but may require more manual proof hints.
We present a mechanism that enables fine-grained control over the availability of quantified facts in verification contexts, allowing developers to selectively tune the level of automation. Our approach lets library authors provide different pre-defined automation levels while giving end-users the ability to further customize quantifier availability at the module, function, or proof context level.
We implement our techniques in Verus, a Rust-based verification tool, and evaluate them on multiple openly available codebases. Our empirical analysis demonstrates the automation-performance tradeoff and that selective quantifier management is needed for developers to select the appropriate level of automation in different contexts.

Keywords: SMT-based Reasoning · Verus · Formal Verification

1 Introduction

Formal verification of programs can guarantee their correctness with respect to a specification (and under certain assumptions of their operating environment). Verification tools based on automated theorem provers (such as SMT-solvers) have made significant progress in efficiency and scalability to larger projects [9,11, 13,22,32]. Fully automated proofs are only possible when writing specification and code so that it falls within a decidable fragment of first-order logic [3,20,21,24,26], but this imposes a significant limitation in expressivity and often prevents the developer from describing the specification and program in the most natural way [11]. General-purpose verifiers thus allow expressing programs and proofs in an undecidable logic (often first-order logic with uninterpreted functions, integer arithmetic, and unrestricted quantification).

The verification community's experience is that quantifier instantiation is the primary source of incompleteness and verification performance penalty in SMT-based verifiers [9, 12, 14, 17, 19, 23]. These tools express everything from core axioms to developer-stated properties using quantifiers, and then rely on the SMT-solver syntactic matching (e-matching) heuristic [18] to automatically instantiate quantified expressions with symbolic values from the proof context. This approach has two consequences: (i) if the syntactic "trigger" that results in instantiation is not present in the proof context, the user has to provide a hint to instantiate the quantifier as needed and (ii) the verification time can be proportional (often exponentially) to the number of quantifiers in the automated prover's scope and to how easily they are instantiated (based on their syntactic "triggers"). If quantifiers are not sufficiently instantiated, the verification tool's automation is more incomplete (it will fail verification, which can be fixed by a manual user hint). On the other hand, quantifiers with easily matched syntactic triggers may enable the prover to automatically find proofs in more cases, but result in a large potential search space, causing long verification runtimes before the tool reports success or failure. Semi-automated verification tools generally fall somewhere on this "automation spectrum" (Figure 1). For example, Dafny [15] employs a trigger selection algorithm that minimizes the need for user annotation and hints: this helps new Dafny users write proofs without immediately having to learn about the underlying mechanism [14]. On the contrary, Verus [11, 12] has conservative defaults optimized for short response times, at the potential cost of less out-of-the-box automation due to more incomplete instantiation.

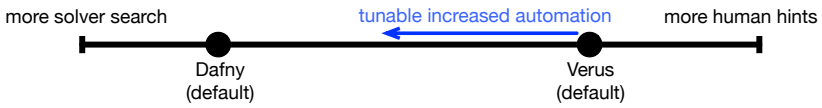

Fig. 1: The spectrum of quantifier-based automation.

In this work, we explore several ways to allow the user to tune the amount of automation (and its resulting cost) provided by quantifier instantiation in automated verification tools. We implement and evaluate our techniques in the Verus Rust-based verification tool and language [11, 12].

Our technical contribution is a `broadcast` mechanism to selectively include axioms and lemmas in the proof environment as quantified facts, allowing both coarse-grained and fine-grained control over which quantifiers are available to the SMT solver in different proof contexts. Quantified facts can be imported in bulk, or selectively, at the module, function, or isolated proof context level. The `broadcast` mechanism enables library authors to provide various default "automation levels" and end-users to select among these, or further tune the proof context. The user can also query the tool to determine which quantified facts were likely used in a proof, allowing fine-tuning once a proof is discovered.

We provide an empirical analysis of the `broadcast` mechanism's impact on verification time and proof burden across multiple openly available Verus codebases. Our evaluation clarifies the tuning tradeoff that allows shorter proofs at the cost of some verification performance. Additionally, we investigate trigger selection strategies and their effects on the automation-performance tradeoff.

2 Background

2.1 Verus

Verus [11, 12] is an SMT-based semi-automated tool to verify the correctness of code written in Rust. Verus is heavily inspired by Dafny [15] and optimizes for short developer iteration time thanks to short response times [11] for both successful and unsuccessful proofs. The following Verus example briefly demonstrates Verus syntax and how it integrates a verification language in Rust:

```
spec fn divides(n: int, k: nat) -> bool { n % (k as int) == 0 }

spec fn is_prime(n: nat) -> bool {
 forall|k: nat| 2 <= k < n ==> !divides(n as int, k)
}

spec fn is_even(i: int) -> bool { divides(i, 2) }

proof fn even_gt_2_isnt_prime(i: nat)
 requires i > 2 && is_even(i as int)
 ensures !is_prime(i) { }

fn is_prime_impl(n: u64) -> (result: bool)
 requires n >= 2,
 ensures result == is_prime(n as nat)
{ /* ... implementation and proof ... */ }
```

The `spec` functions `divides`, `is_prime`, and `is_even` contain pure logical ghost expressions (including quantifiers), used to express function specifications or auxiliary conditions. `even_gt_2_isnt_prime` is a ghost `proof` function, which expresses a callable lemma, and - in this case - can be proven automatically by the solver. `is_prime_impl` is an executable function that is proven to compute whether the input is prime according to the `is_prime` definition (we omit the implementation and proof for brevity). Ghost code includes `spec` and `proof` functions, executable functions' pre- and post-conditions, and inline assertions, such as the following, which appears as part of the proof of `is_prime_impl`:

```
assert(divides(n as int, i as nat));
```

2.2 Quantifier Triggers

As discussed in Section 1, semi-automated verification tools heavily rely on quantifier instantiation to automatically discharge proof obligations. Syntactic matching (e-matching) via "triggers" [7] is a commonly used heuristic for automatic quantifier instantiation, and – while the underlying solver is often capable of automatically selecting quantifier triggers (sometimes known as "patterns") –

languages like Dafny and Verus perform trigger selection at the surface language level and allow manual selection and manual overrides through user-provided annotations. Performing trigger selection in the verification tool can help avoid expensive or vacuous triggers [14].

In Verus, like in similar tools, only function calls, field accesses, or arithmetic operators where one of the arguments is one of the quantified variables are allowed as trigger subexpressions. A trigger is composed of potentially multiple subexpressions and needs to mention all quantified variables at least once.

The following proof function has a precondition that all elements of a sequence are even numbers, and we want to prove that a specific entry is even.

```
1 spec fn is_even(i: int) -> bool { i % 2 == 0 }
2
3 proof fn seq_trigger_example(s: Seq<int>)
4   requires
5     5 <= s.len(),
6     forall|i: int| 0 <= i < s.len() ==> #[trigger] is_even(s.index(i)),
7 {
8   assert(s.index(3) % 2 == 0);
9 }
```

The valid triggers for the `forall` on line 6 are `is_even(s.index(i))` and `s.index(i)`, and the user has manually chosen `is_even(s.index(i))` with the `#[trigger]` annotation. Because `is_even(s.index(3))` does not appear anywhere in the context, the solver will not instantiate the quantifier with `i == 3` and thus the `assert` on line 8 will fail. Changing the trigger to `s.index(i)` will enable matching with `s.index(3)` and result in a successful proof.

2.3 Implicit Context

SMT-based semi-automated tools like Dafny and Verus include a default set of quantified facts provided as part of the "prelude" of the SMT problem that encodes the program verification condition. This default set defines, among others, the semantics of built-in concepts like sequences (`Seq` in Dafny). For example, proving the following

```
1 proof fn seq_axiom_usage(s1: Seq<nat>, s2: Seq<nat>)
2  requires s1.len() > 10 && s2.len() > 20
3  ensures s1.add(s2).len() > 30 { }
```

relies on this built-in Verus axiom:

```
1 • (#[trigger] s1.add(s2).len()) == s1.len() + s2.len()
```

In Dafny, this context is fixed and defined as part of verification condition generation (through Boogie [2]). In Verus, a default context is defined in a pure-SMT prelude and a collection of lemmas in the standard library explicitly marked to become contextual quantified facts. More quantified facts in the proof context improve the level of automation available to the users, but increase the proof search space available to the solver, potentially resulting in longer response times. In this work, we address the one-size-fits-all approach to contextual quantified facts by allowing users to construct and selectively import them.

3 Quantified Facts à la Carte

We introduce a mechanism in Verus to selectively publish quantified facts from `proof` functions (lemmas) with a user-provided marker (`broadcast`) and import them as additional context for a module or function (via the new keyword `broadcast use`).

We showcase the usage of broadcast in the following code examples.

This lemma fails to verify automatically in Verus:

```
pub proof fn push_contains(a: Seq<int>) {
 let b = a.push(3);
 assert(b.contains(3));
}
```

due to Verus' default context being insufficient to prove the fact that a sequence contains every element it contains before after pushing:

```
pub broadcast proof fn lemma_seq_contains_after_push<A>(s: Seq<A>, v: A, x: A)
  ensures (#[trigger] s.push(v).contains(x)) <==> v == x || s.contains(x),
{ /* ... manual proof ... */ }
```

However, once proven and marked `broadcast`, this quantified fact can be imported as the following universally quantified statement:

```
• forall|s: Seq<_>, v: A, x: A|
    (#[trigger] s.push(v).contains(x)) <==> v == x || s.contains(x)
```

As broadcast proofs share the same ability as moving the input parameters to a universally quantified variable in the ensures clause, they must adhere to trigger rules as if they are universally quantified.

Now, this broadcasted proof can be imported in context with the directive:

```
broadcast use {lemma_seq_contains_after_push};
```

For example, the initial lemma verifies automatically by importing:

```
pub proof fn push_contains(a: Seq<int>) {
 broadcast use {lemma_seq_contains_after_push};
 let b = a.push(3);
 assert(b.contains(3));
}
```

Broadcastable quantified facts, i.e. `proof` functions marked with `broadcast`, can be defined in the Verus standard library, in third-party libraries, or in end-user code, and can be imported directly, or through `broadcast groups`, which name and collect related broadcastable facts together so they can be imported in context in bulk. As an example, `lemma_seq_contains_after_push` is now in fact a broadcastable quantified fact in the Verus standard library, and it is part of the `group_seq_properties` broadcast group defined, again, by the standard library:

```
pub broadcast group group_seq_properties {
  lemma_seq_contains,
  // ...
  lemma_seq_contains_after_push,
  // ...
}
```

This enables the user to import a larger set of quantified facts in context in bulk, as follows:

```
pub proof fn push_contains(a: Seq<int>) {
 broadcast use {vstd::seq_lib::group_seq_properties}; // imports all the quantified facts
     from the group
 let b = a.push(3);
 assert(b.contains(3));
}
```

Thanks to this mechanism Verus can provide a default set of quantified facts in context while allowing the end-user to tune the level of quantifier-instantiation-induced automation in each proof context. Library authors can also leverage the mechanism to provide (sets of) contextual quantified facts for their users' benefit.

3.1 Restricting the Set of Quantified Facts in the Context

Additional quantified facts in context improve automation at the cost of some verification performance, as we will see in our case studies in Section 5. We extended Verus to interpret the solver's "unsat-core" output, which contains the facts that were used in a successful proof, to display to the user the set of imported quantified facts that are likely necessary for the proof. When running Verus with the appropriate flag on the last example of the previous section (which relies on the bulk import of the `group_seq_properties` quantified facts), we get the following output:

```
checking this function used these broadcasted lemmas and broadcast groups:
    - (group) vstd::seq_lib::group_seq_properties,
    - vstd::seq_lib::lemma_seq_contains_after_push
```

This indicates that `lemma_seq_contains_after_push` was the only quantified fact relevant for the proof, and enables the end-user to trim the context by replacing the `broadcast use` for the entire `group_seq_properties` with just `lemma_seq_contains_after_push`, which results in a successful proof but with reduced context:

```
pub proof fn push_contains(a: Seq<int>) {
 broadcast use {vstd::seq_lib::lemma_seq_contains_after_push};
 let b = a.push(3);
 assert(b.contains(3));
}
```

The combination of bulk import of quantified facts via `broadcast use` of `broadcast group`s and the tooling to report those that are likely relevant to the proof enables a workflow where the developer first imports a broad set of facts in context, and when the proof is successful, trims the set of quantified facts to those relevant to the proof, potentially recovering proof performance by reducing the solver's search space.

The ability to import quantified facts in broader or more limited proof scopes also enables fine-grained control of where to increase automation. With `broadcast use`, a Verus user can import quantified facts in any proof contexts, or for an entire module. New proof contexts are introduced by

- proof functions (`proof fn`);
- `assert (expression) by { /* ... proof ... */ }`;
- calculational proofs ("calc" in Dafny and Verus).

In the previous example, the user can `broadcast use group_seq_properties` at the module level, and then fine-tune the level of automation by tailoring the set of imported quantified facts in more granular proof contexts to address verification performance slowdowns [8].

4 Case studies

4.1 Modularizing Proof Libraries

The `broadcast` mechanisms enable constructing self-contained abstractions that hide the internal details of type-specific proofs but expose relevant facts relating values in the abstraction. As an example, we look at the abstraction over keys used in IronKV, the Verus port [11] of the verified sharded key-value store from Ironfleet [10].

For most of the system, the concrete type representing a *key* is irrelevant for the proofs, which only rely on the fact that keys have certain properties. IronKV defines a Rust "trait" (similar to a Haskell-style typeclass) that (i) expresses the proof obligations for a type to be a valid *key*, and (ii) exposes the properties relevant to the clients of the *key* abstraction through the `broadcast` mechanism.

```
pub trait Key {
  ...
  proof fn key_obligations()
    ensures // ... conditions necessary for the type to be a valid key

  broadcast proof fn trans_lt_lt(a:Self, b:Self, c:Self)
    ensures a < b && b < c ==> a < c
  { /* justified thanks to the ensures of ‘key_obligations‘ */ }

  // ... additional properties ...
}

pub broadcast group group_key_cmp_properties {
  Key::trans_lt_lt,
  // ... additional ‘broadcast‘ proofs from the trait
}
```

Fig. 2: **Excerpt from the definition of IronKV's *key* trait: the example has been simplified for ease of presentation but without hiding any relevant details**.

Figure 2 shows the definition of the `Key` trait. A type implementing the trait must provide a proof for `key_obligations`, which ensures that the type has the necessary properties to be used as a `key`. In return, the trait exposes a number of broadcastable quantified facts (such as `trans_lt_lt`) which are grouped together in the `broadcast group group_key_cmp_properties`. Code that operates on keys need not know the details of the type used as keys, and can just rely on the properties exposed through the group by importing the quantified facts in context via the directive:

```
broadcast use {group_key_cmp_properties};
```

4.2 Verus Standard Library

As discussed in Section 2.3, built-in abstract datatypes' semantics are defined as axioms or proofs in the default implicit context of quantified facts. The `broadcast` mechanism offers greater flexibility in organizing the quantified facts for these datatypes, such as sequences, maps, and sets.

Dafny fully defines these datatypes' semantics as axioms in the Boogie "prelude" used during verification condition generation: some can be derived from other axioms with proofs, but they still appear as axioms in the prelude because Dafny lacks a general mechanism to include in the context additional quantified facts derived from proofs. In comparison, in Verus there are a much smaller number of axioms (resulting in a smaller Trusted Computing Base) and additional facts relating operations on the built-in datatypes are `broadcast proof` functions that are grouped together with the axioms in a default `broadcast group` which is automatically included in the proof context with the Verus' standard library. As a conservative choice to prioritize verification performance, this is a smaller set than what's included in Dafny's prelude.

5 Exploring the Automation Tradeoff

We set out to study the effect of tuning the level of automation associated with reasoning about built-in Verus collection datatypes: `Seq`, `Map`, `Set`, and `Multiset`. We design experiments to answer the following questions:

RQ1: Does increasing the number of quantified facts in context result in more automation, i.e. fewer manual user-provided hints (in the form of `assert`s)?

RQ2: Does increasing the number of quantified facts in context hinder verification performance or the verification experience?

We grouped the `broadcast proof` functions corresponding to the quantified facts included in Dafny's prelude but excluded from Verus' default context in four `broadcast group`s, one per collection type: `group_<type>_properties`. The following table lists the number of translated Dafny prelude lemmas:

Collection datatype	**Seq**	**Map**	**Set**	**MultiSet**	**Total**
# of `group_<type>_properties` lemmas	25	2	10	12	49

We then used `broadcast use` to include these lemmas as quantified facts in the context of a number of openly available Verus projects and measured the impact on the number of manual hints necessary and on verification performance. In the following, we refer to imported quantified facts for collection types as *ambient facts* for brevity. For these experiments, we focus on the effects of including these facts in all proof contexts, and do not try to fine-tune the scope in which facts are made available.[4]

[4] All experiments are available at: https://github.com/ahuoguo/tunable-automation

5.1 Projects under Study

We considered the following projects as verification benchmarks:

- IronKV is a distributed key-value store which was described in Sec 4.1
- Splinter [1] is an ongoing work on a key-value store designed around a B^ϵ tree.
- Anvil [22] is a framework for building and formally verifying Kubernetes controllers. It provides a TLA-style temporal logic to reason with eventually stable reconciliation of controllers. We currently only focus on the anvil framework rather than the verification of specific controllers.
- CapybaraKV [13] is a storage system targeting persistent memory devices

5.2 Minimization

For the purpose of this experiment, we need a measure of "automation", for which we use a proxy metric: the number of hints provided to the solver via `assert` statements that are needed for the proof to succeed. Our projects under study contain more `assert`s than strictly necessary: they are used during proof development as a proof debugging strategy, and developers do not systematically remove `assert`s that become unnecessary once the proof is complete. `assert`s are also often intentionally retained as documentation of the proof structure. For the number of `assert`s to be a viable proxy metric for the level of automation, we need to determine the least number of such hints that still result in successful verification. We can automate such minimization by removing `assert`s and re-running verification, electing to keep only those that – when removed – result in a failure. However, this has combinatorial complexity: for a project with n `assert`s, there are, as a first approximation, 2^n candidates for a "minimal" proof. [5] We developed a Verus minimization tool that linearly scans through the proof code and removes the proof code (in the form of `assert` and `assert (...) by {...}`) if its removal does not cause a failure (i.e. if it is redundant as a hint to the solver). While obviously imperfect, we consider this an acceptable approximation to make it feasible to use this metric. Despite this simplification, it is Verus' performance that enables the use of this metric, for which we still need to re-run verification after each `assert` is tentatively removed[6]. To our knowledge, this is the first study of what fraction of `assert`s in a project are actually necessary hints to the solver. In a sense, we introduce a form of program slicing in the context of verification code [27].

To evaluate the impact of ambient facts on the level of automation, we run our Verus minimization tool twice on each project: first on the unmodified project,

[5] Consider a verification project with 100 asserts you want to minimize, and each run is 1 second. Then you have 2^{100} possible candidates to consider, which takes about 4×10^{22} years.

[6] Additional tooling may enable only re-running verification for the verification condition affected by the `assert`, but there is "cross-talk" between VCs (also known as "instability") which complicates this potential approach

and then again after importing the ambient facts. The number of `assert`s that become redundant when the ambient facts have been imported is a proxy metric for the resulting increase in automation (and thus, indirectly, proof effort).

It is important to note that counting the number of surviving `assert`s after "minimization" is a very rough measure of automation and might not reflect true developer experience. Developers use `assert`s to guide the solver, but also to debug proofs (and to document and clarify their own understanding of the proof strategy). Another limitation is our "minimizer" only targets assertions, namely, `assert(...)`, `assert <forall> ... by {...}` while it ignores calls to lemmas (`proof` functions) which can also be made redundant by ambient facts[7].

5.3 Effect on the Level of Automation

Table 1 contains the effect of ambient facts. One interesting observation is the number of extra assertions the target verification projects have; more than half of the assertions in these projects do not contribute to the SMT solver reaching an UNSAT.

System	**Original # of asserts**	**Minimized**	**Minimized with Ambient Facts**
IronKV	646	268(-58.5%)	245(-23, -8.6%)
Splinter	2678	1158(-56.7%)	1130(-28, -2.4%)
Anvil	701	343(-51.1%)	336(-7, -2.0%)
CapybaraKV	941	449 (-52.3%)	415 (-34, -7.6%)

Table 1: Assertion counts for verification systems after minimization and importing ambient facts

RQ1: Do more ambient facts about collection types bring more automation?

We see a reduction of the number of assertions for ambient facts, ranging from 2%-9%. Some of the assertions are non-trivial proof blocks ranging from 5-10 lines and were automatically discharged solely due to the ambient facts of collection types.

5.4 Impact on Verification Time

To answer RQ2, we first measure the ambient facts' impact on verification time. All experiments are done with 9 threads on a Dell PowerEdge M620 blade system (Intel Xeon E5 v2) with 16 cores using Verus version 0.2025.07.03.3105aa2. The results can be seen in Figure 3. The figure compares each function's original verification time to the verification time after having ambient facts and minimization. We can see that ambient facts do slow down verification by some amount,

[7] Deleting lemma calls requires semantic information, which is not available in our syntax-level minimization tool

with $\sim 2\%$ of the functions taking 2x of their verification time, and the worst slowdown in all projects ranges from 3x to 19x.

The experiment data showed that the ambient facts improve automation by some amount, but may have noticeable runtime slowdown in one or two specific functions. We argue that our new technique for broadcastable quantified facts is a way to combat verification slowdowns on particular functions. Quantified facts can be tuned down by importing them at a finer-grained level, rather than providing the ambient facts everywhere, as we discussed in Section 3.1.

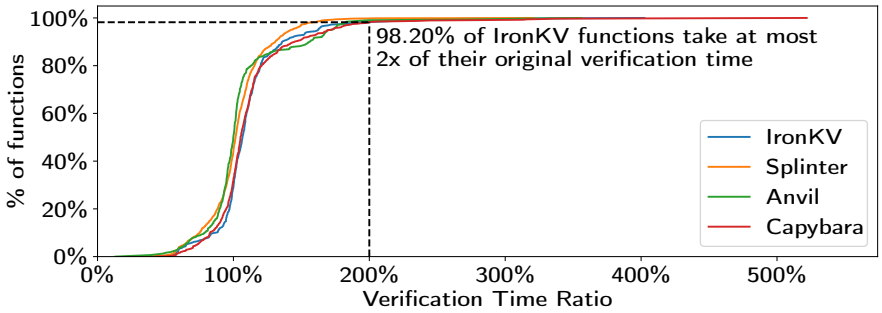

Fig. 3: Cumulative distribution of verification time ratio for each function, ratio is between ambient facts (minimized) and original verification time. We have removed two extreme cases of 10x verification slowdown, one in Splinter, where the runtime bumped from 34ms to 669ms (19.6x), and one in Anvil, where the runtime bumped from 3508ms to 43563ms (12.4x)

5.5 Impact on Verification Failure Time

Although Verus's run time for successful verification tasks is important when we are importing ambient facts, the time for reporting a verification failure is also important to the verification engineer's experience. As we may have dramatically increased the search space by importing more ambient facts, this might cause a similarly dramatic increase in the time Verus takes to report a verification failure. Once the solver finds a proof, it stops, which does not happen if the proof is incomplete or incorrect. Thus, it is also important to see if more ambient facts impact user experience by slowing down the reporting of failures. After minimization, removing an assertion will almost certainly result in verification failure. Therefore, we randomly sampled 20 assertions to remove and recorded the time it took to report the failure, and computed the ratio to the successful verification time for the function left intact. The results can be seen in Figure 4. We conclude that ambient facts did result in slowdowns in a small fraction of functions that failed to verify, with 78% of the failed functions still verifying in

2x of their original verification time. Thus, ambient facts did not result in an explosion in verification failure time.

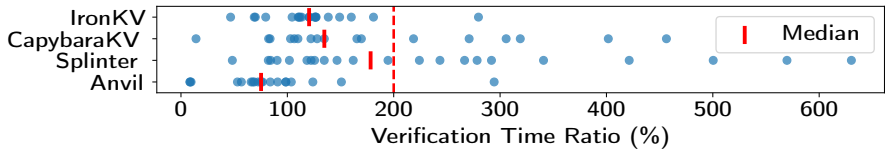

Fig. 4: Runtime "Slowdown" Ratio for 20 runs (each with one of the randomly sampled asserts removed) against successful verification. For each of the 20 runs, we report the ratio between a function's verification time when a failure was introduced and the verification time for the same function left intact. The plot also includes the median of these ratios, and a line indicating 2x slowdown.

6 Triggers

Quantified facts are instantiated via syntactic matching of "triggers" (Section 2.2). When the verification tool selects the "trigger" expressions for a quantifier, it can default to selecting every possible valid, non-redundant trigger, or it can conservatively only select *safe* triggers, i.e. those that are less likely to significantly increase the search space that the solver can explore, leading to longer verification times.

6.1 Trigger Selection Strategy

Verus' default automatic trigger selection is quite conservative: it selects a few "safe" triggers. A quantifier can be annotated with `#![all_triggers]`, which switches the trigger selection strategy to pick all valid, non-redundant triggers for the quantified expression[8]. The `#![all_triggers]` strategy is generally still more conservative than Dafny's default trigger selection strategy (which employs some more advanced techniques where quantifiers are automatically split and triggers shared, see Sec 2.1 of [14]).

6.2 More Automation with More Triggers

Axioms An immediate question follows: Is it possible to obtain a more extensive automation (that is, considering our metric, a smaller number of required hints

[8] Due to technical reasons related to polymorphism, there is an exception: quantifiers where the quantified variables are used as arguments to both arithmetic operations and function calls. For these quantifiers, the `#![all_triggers]` strategy remains more conservative

to the solver as `asserts`) by changing how triggers are selected for the default set of quantified facts imported when using the Verus standard library?

The answer is no. By manual inspection of most axioms of `Seq, Set, Map, Multiset`, we have observed that their triggers are quite straightforward, and most trigger candidates that are not selected would generally lead to "matching loops" which result in a very large search space (these result in the solver timing out). For example, for the following axiom of lengths of sequences (an excerpt of the default set of the Verus' standard library ambient facts):

```
pub broadcast axiom fn axiom_seq_add_len<A>(s1: Seq<A>, s2: Seq<A>)
  ensures
    #[trigger] s1.add(s2).len() == s1.len() + s2.len(),
;
```

The two possible trigger sets are

```
• s1.add(s2).len()
• s1.len(), s2.len()
```

But the second one is almost always unhelpful, as this axiom is usually only relevant in contexts where there is a call to `add`. Moreover, if the second trigger is included, this axiom will be instantiated excessively: once on each pair of `.len()` calls in the program and proof context.

User Code We saw that there are little to no opportunities to expand the triggers of quantified facts in the standard library, but user-level code may contain more complex quantified expressions with more trigger candidates. In this section, we study the effects of selecting a broader set of triggers for all quantifiers.

In the following experiment, we picked IronKV and removed all manual triggers if possible. We then used `#![all_triggers]` wherever it is supported, and ran the `assert` minimization tool.

The change only breaks one proof. In the end, we added 206 `#![all_triggers]` annotations in the 261 quantifiers in IronKV.

We measure the impact on automation using our `assert`-based metric, and report results in Table 2. For this project, a more liberal trigger selection strategy improves automation (according to our metric) more than importing ambient facts for collection types. We measure the performance impact (on success and failure), with a methodology similar to the one described in Section 5.4 and Section 5.5, and report the results in Figure 5.

System	Original	Min	All Triggers	Ambient Facts
IronKV	646	268(-58.5%)	235(-33, -12.3%)	245(-23, -8.6%)

Table 2: Assertion counts for IronKV after minimization with all_triggers

Unfortunately, this experiment cannot scale well to other verification projects considered in Sec:5.1, as they have at least twice as many triggers throughout the project, and verifying the project using `all_triggers` pervasively will result

in a large amount of verification failures. We suspect these failures are due to increased search space and misguided heuristics. However, our data indicated that more trigger candidates on the user level quantified expressions have a non-trivial positive impact on proof automation. We leave further exploration of the automation level of different trigger levels as future work.

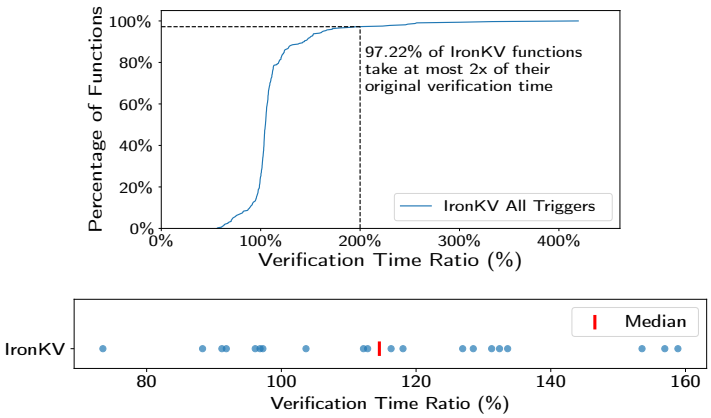

Fig. 5: Cumulative distribution of verification time, and verification "slowdown" ratio for 20 sampled asserts for IronKV with `all_triggers` enabled. Detailed descriptions of the two figures can be seen in Figure 3 and Figure 4.

7 Related Work

7.1 Dafny

The authors of Dafny argue that trigger-based installation should be available at the source code instead of relying on SMT heuristics, and provide a trigger selection algorithm [14].

Increasing automation is crucial for past large-scale verification projects in Dafny. One way of publishing additional quantified facts pervasively on a module level is to declare a `ghost predicate` in a module with an `ensures` clause. The following is an example of publishing the fact that the uninterpreted function `BinOp` is commutative.

```
function BinOp(x: int, y: int) : int

ghost predicate BinOpAuto(x: int, y: int)
  ensures BinOpAuto(x, y) ⟹ BinOp(x, y) == BinOp(y, x)
  // BinOpAuto is equivalent to
  //  ∀ x,y. BinOpAuto(x, y) ⟹ BinOp(x, y) == BinOp(y, x)
```

By importing the module containing this `ghost predicate`, the quantified fact derived from the `ensures` is effectively available on the module level.

The authors of the VeribetrKV crash-safe key-value store [9, 16] report that this methodology has been crucial in their development. However, this approach to introducing additional quantified facts in context does not allow for granular control of where they are imported (in contrast to `broadcast proof`).

7.2 Instability

Instability (or brittleness) is a common phenomenon observed in SMT-based verification tools. Namely, some proofs break under semantically equivalent changes [5, 25, 30, 31]. Shake [29] showed that irrelevant context is one of the culprits for instability.

To fight proof instability, a technique known as “free facts" was proposed in Dafny, which is a set of generated facts based on syntactic matching during the verification condition generation [4]. In a sense, free facts perform a weaker version of pattern-based instantiation before SMT solving. The free facts authors indicated a strong interest in studying their verification projects with a reduced number of assertions, as the extra assertions take up verification time and resource usage. Our empirical evaluation on Verus suggested that reducing the number of assertions does not necessarily reduce verification time, and we leave exploration in this direction as future work.

Axolocl is a tool to automatically localize proof context in local proof blocks [8]. Their proof localization algorithm moves relevant axioms for verifying an assertion into a local proof context, based on information from UNSAT-CORE and quantifier instantiation, in Boogie [2], an intermediate verification language for Dafny and Viper. We believe that, with Verus proof blocks and broadcastable quantified facts, we can achieve source-level translation to minimize proof contexts.

7.3 Other Verus Work

RagVerus [28] is the first work that considered repository-level proof automation. Though the methodologies are very different from this project, the proof automation aspect is very similar to our evaluation of quantified facts in Section 5, and the Verus Property Extractor uses a very similar technique to our Verus assert minimizer.

ProofPlumber [6] introduced several automated strategies to insert assertions in the right places to guide the proof engineer to see where the proof has gone wrong. It’s interesting future work to incorporate the minimizer and axiom-usage-info as automatic strategies.

8 Conclusion

We explored several directions for tunable automation in the Verus Rust verification tool. To this end, we introduced *broadcastable quantified facts* as a mechanism to tune automation on a finer-grained level, and demonstrated its usefulness through case studies. We also conducted multiple experiments on openly available

large Verus projects to evaluate the cost and benefits of increasing automation along two axes: increasing quantified facts in context, and increasing the level of automation from triggers.

Acknowledgements

We thank Jialin Li, Xudong Sun, and Zhizhen Cai for their help with setting up Verus projects. We also thank Alexandre Moine, and the audience of Verus Retreat 2025 and FP Teatime for their helpful feedback. The first author wants to especially thank Jay, Travis, and Bryan at CMU REUSE 2023 for getting him involved in the Verus world.

References

1. Verified betrfs. https://github.com/vmware-labs/verified-betrfs
2. Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures. LNCS, vol. 4111, pp. 364–387. Springer (2005). https://doi.org/10.1007/11804192_17, https://doi.org/10.1007/11804192_17
3. Blanc, R., Kuncak, V., Kneuss, E., Suter, P.: An overview of the leon verification system: verification by translation to recursive functions. In: Proceedings of the 4th Workshop on Scala. SCALA '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2489837.2489838, https://doi.org/10.1145/2489837.2489838
4. Bordis, T., Leino, K.R.M.: Free facts: An alternative to inefficient axioms in dafny. In: Formal Methods: 26th International Symposium, FM 2024, Milan, Italy, September 9–13, 2024, Proceedings, Part I. p. 151–169. Springer-Verlag, Berlin, Heidelberg (2024). https://doi.org/10.1007/978-3-031-71162-6_8, https://doi.org/10.1007/978-3-031-71162-6_8
5. Cebeci, C., Bjørner, N., Candea, G., Pit-Claudel, C.: A Conjecture Regarding SMT Instability. 23rd International Workshop on Satisfiability Modulo Theories (2025)
6. Cho, C., Zhou, Y., Bosamiya, J., Parno, B.: A framework for debugging automated program verification proofs via proof actions. In: Gurfinkel, A., Ganesh, V. (eds.) Computer Aided Verification. pp. 348–361. Springer Nature Switzerland, Cham (2024)
7. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52**(3), 365–473 (May 2005). https://doi.org/10.1145/1066100.1066102, https://doi.org/10.1145/1066100.1066102
8. Gopinathan, K., Spiliopoulos, D., Goyal, V., Müller, P., Püschel, M., Sergey, I.: Accelerating automated program verifiers by automatic proof localization. In: International Conference on Computer-Aided Verification (CAV). Springer, Zagreb, Croatia (2025), 37th International Conference on Computer-Aided Verification
9. Hance, T., Lattuada, A., Hawblitzel, C., Howell, J., Johnson, R., Parno, B.: Storage systems are distributed systems (so verify them that way!). In: Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI) (2020)

10. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S., Zill, B.: IronFleet: Proving practical distributed systems correct. In: Proceedings of the ACM Symposium on Operating Systems Principles (SOSP) (Oct 2015)
11. Lattuada, A., Hance, T., Bosamiya, J., Brun, M., Cho, C., LeBlanc, H., Srinivasan, P., Achermann, R., Chajed, T., Hawblitzel, C., Howell, J., Lorch, J.R., Padon, O., Parno, B.: Verus: A practical foundation for systems verification. In: Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles. p. 438–454. SOSP '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3694715.3695952, https://doi.org/10.1145/3694715.3695952
12. Lattuada, A., Hance, T., Cho, C., Brun, M., Subasinghe, I., Zhou, Y., Howell, J., Parno, B., Hawblitzel, C.: Verus: Verifying rust programs using linear ghost types. Proc. ACM Program. Lang. **7**(OOPSLA1) (Apr 2023). https://doi.org/10.1145/3586037, https://doi.org/10.1145/3586037
13. LeBlanc, H., Lorch, J.R., Hawblitzel, C., Huang, C., Tao, Y., Zeldovich, N., Chidambaram, V.: PoWER Never Corrupts: Tool-Agnostic Verification of Crash Consistency and Corruption Detection. In: Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI). USENIX Association (Jul 2025)
14. Leino, K.R.M., Pit-Claudel, C.: Trigger selection strategies to stabilize program verifiers. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification. pp. 361–381. Springer International Publishing, Cham (2016)
15. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Proceedings of the Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR) (2010)
16. Li, J., Lattuada, A., Zhou, Y., Cameron, J., Howell, J., Parno, B., Hawblitzel, C.: Linear types for large-scale systems verification. In: Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) (November 2022)
17. Moskal, M.: Programming with triggers. In: Proceedings of the 7th International Workshop on Satisfiability Modulo Theories. p. 20–29. SMT '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1670412.1670416, https://doi.org/10.1145/1670412.1670416
18. de Moura, L.M., Bjørner, N.S.: Efficient e-matching for SMT solvers. In: Pfenning, F. (ed.) Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4603, pp. 183–198. Springer (2007). https://doi.org/10.1007/978-3-540-73595-3_13, https://doi.org/10.1007/978-3-540-73595-3_13
19. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 41–62. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)
20. Murali, A., Rivera, C., Madhusudan, P.: Predictable verification using intrinsic definitions. Proc. ACM Program. Lang. **8**(PLDI) (Jun 2024). https://doi.org/10.1145/3656450, https://doi.org/10.1145/3656450
21. Nelson, L., Bornholt, J., Gu, R., Baumann, A., Torlak, E., Wang, X.: Scaling symbolic evaluation for automated verification of systems code with Serval. In: Proceedings of the ACM Symposium on Operating Systems Principles (SOSP) (2019)

22. Sun, X., Ma, W., Gu, J.T., Ma, Z., Chajed, T., Howell, J., Lattuada, A., Padon, O., Suresh, L., Szekeres, A., Xu, T.: Anvil: Verifying liveness of cluster management controllers. In: Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI) (Jul 2024)
23. Swamy, N., Hriţcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.Y., Kohlweiss, M., Zinzindohoue, J.K., Zanella-Béguelin, S.: Dependent types and multi-monadic effects in f*. SIGPLAN Not. **51**(1), 256–270 (Jan 2016). https://doi.org/10.1145/2914770.2837655, https://doi.org/10.1145/2914770.2837655
24. Taube, M., Losa, G., McMillan, K.L., Padon, O., Sagiv, M., Shoham, S., Wilcox, J.R., Woos, D.: Modularity for decidability of deductive verification with applications to distributed systems. In: Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI) (2018), https://doi.org/10.1145/3192366.3192414
25. Tomb, A., Tristan, J.B.: Avoiding verification brittleness in dafny. https://dafny.org/blog/2023/12/01/avoiding-verification-brittleness/ (Dec 2023)
26. Vazou, N., Seidel, E.L., Jhala, R., Vytiniotis, D., Peyton-Jones, S.: Refinement types for haskell. SIGPLAN Not. **49**(9), 269–282 (Aug 2014). https://doi.org/10.1145/2692915.2628161, https://doi.org/10.1145/2692915.2628161
27. Weiser, M.: Program slicing. In: Proceedings of the 5th International Conference on Software Engineering. p. 439–449. ICSE '81, IEEE Press (1981)
28. Zhong, S., Zhu, J., Tian, Y., Si, X.: Rag-verus: Repository-level program verification with llms using retrieval augmented generation (2025), https://arxiv.org/abs/2502.05344
29. Zhou, Y., Bosamiya, J., Li, J., Heule, M.J., Parno, B.: Context pruning for more robust smt-based program verification. In: 2024 Formal Methods in Computer-Aided Design (FMCAD). pp. 1–11 (2024). https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_12
30. Zhou, Y., Bosamiya, J., Takashima, Y., Li, J., Heule, M., Parno, B.: Mariposa: Measuring SMT instability in automated program verification. In: Nadel, A., Rozier, K.Y. (eds.) Formal Methods in Computer-Aided Design, FMCAD 2023, Ames, IA, USA, October 24-27, 2023. pp. 178–188. IEEE (2023). https://doi.org/10.34727/2023/ISBN.978-3-85448-060-0_26, https://doi.org/10.34727/2023/isbn.978-3-85448-060-0_26
31. Zhou, Y., Shah, A., Lin, Z., Heule, M., Parno, B.: Cazamariposas: Automated instability debugging in smt-based program verification. In: Proceedings of the International Conference on Automated Deduction (CADE) (Jul 2025), to appear
32. Zhou, Z., Chen, W., Hawblitzel, C., Cui, W.: VeriSMo: A verified security module for confidential VMs. In: Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI) (Jul 2024)

A Appendix

A.1 Mariposa

One interesting experiment is to use Mariposa to answer whether more ambient facts introduce more “instability", partly answering RQ2. This is not included in the main paper as Verus programs tend to be more “stable" than other auto-active verifiers. Nevertheless, Table 3 contains the results of running Mariposa on the four projects. IronKV Original, for some reason, has twice the number of queries, though not impacting the impact of the data. Except for Anvil having a reverse trend, most projects have a slight increase in instability when exposed to a larger set of ambient factors, which aligns with our expectation.

Benchmark	**Stable**		**Unstable**		**Unsolvable**		**Total**
	Number	*%*	*Number*	*%*	*Number*	*%*	
IronKV Original	726	100.00%	0	0.00%	0	0.00%	726
IronKV Minimized	363	100.00%	0	0.00%	0	0.00%	363
IronKV Broadcasted	362	99.72%	1	0.28%	0	0.00%	363
Splinter Original	1460	99.18%	3	0.20%	9	0.61%	1472
Splinter Minimized	1444	98.10%	18	1.22%	10	0.68%	1472
Splinter Broadcasted	1441	97.89%	22	1.49%	9	0.61%	1472
Anvil Original	314	99.37%	2	0.63%	0	0.00%	316
Anvil Minimized	313	99.05%	3	0.95%	0	0.00%	316
Anvil Broadcasted	315	99.68%	1	0.32%	0	0.00%	316
CapybaraKV Original	723	99.86%	1	0.14%	0	0.00%	724
CapybaraKV Minimized	721	99.59%	3	0.41%	0	0.00%	724
CapybaraKV Broadcasted	719	99.31%	5	0.69%	0	0.00%	724

Table 3: Mariposa Results on All Projects, “Broadcasted" means minimized with ambient facts, described in Table 1

A.2 Implementation

The underlying implementation uses fuels to indicate whether the broadcasted proof wants to be included in the context. If a proof function is marked as `broadcast`, then it will create a broadcast version that moves the parameters to be universally quantified. If `broadcast use` appears, Verus will activate the fuel such that the broadcasted proof function can be used, thus bringing the broadcasted proof function in scope.

Timeline: Verus@bc1fe provided an initial implementation; however, only proof functions without a proof body (mostly axioms) can be treated as ambient facts, as at the time we can’t check if there are no cycles in the lemmas. The following PR Verus#850 implements SCC order for SMT commands to make SMT prove the broadcasted proof functions before they get imported globally. Finally, Verus#1022 implements the generalized broadcastable quantified facts.