





Free Facts: An Alternative to Inefficient Axioms in Dafny

Tabea Bordis¹ and K. Rustan M. Leino²

¹ Karlsruhe Institute of Technology, Karlsruhe, Germany
tabea.bordis@kit.edu

² Amazon Web Services, Seattle, WA, USA
leino@amazon.com

Abstract. Formal software verification relies on properties of functions and built-in operators. Unless these properties are handled directly by decision procedures, an automated verifier includes them in verification conditions by supplying them as universally quantified axioms or theorems. The use of quantifiers sometimes leads to bad performance, especially if automation causes the quantifiers to be instantiated many times.

This paper proposes *free facts* as an alternative to some axioms. A free fact is a pre-instantiated axiom that is generated alongside the formulas in a verification condition that can benefit from the facts. Replacing an axiom with free facts thus reduces the number of quantifiers in verification conditions. Free facts are statically triggered by syntactic occurrences of certain patterns in the proof terms. This is less powerful than the dynamically triggered patterns used during proof construction. However, the paper shows that free facts perform well in practice.

Keywords: SMT-based reasoning · proof brittleness · Dafny · formal verification

1 Introduction

Complex software is used in almost every domain, including safety-critical or security-critical domains that require strong guarantees of correctness. Formal methods have successfully been applied to guarantee correctness of large-scale, complex software (e.g., [12, 13, 21, 22, 26, 31, 36, 37]). Especially successful are Satisfiability Modulo Theories (SMT) solvers [5, 9, 33] and SMT-based, automated program verifiers, such as Dafny [29], Frama-C [25], AutoProof [19], VeriFast [24], and F* [39]. In Dafny, for example, the developer writes specifications and code, which are then translated into proof obligations that are automatically checked by an SMT solver. The verification result is displayed in the IDE, including descriptive error messages in case of a negative result.

The underlying idea of automated verifiers is to transfer most of the verification effort from the developer to the SMT solver, i.e., to automate the verification task as much as possible. For instance, axioms for properties that are known to be

true are automatically generated such that the developer does not have to specify and prove said properties themselves. As a result, automation has increased user-friendliness, leading to more users and the application to larger and more complex systems, including their application in industry. However, with increasing complexity, a certain *proof brittleness*¹ has been observed among SMT-based verifiers [22, 30]. Proof brittleness describes a problem where seemingly irrelevant changes, such as renaming a variable or using a newer version of the tool, can lead to a variation in the verification time and even result. The impact of this is severe, as it drastically increases debugging time and may even require refactoring of the code and specification. Additionally, confidence in the tool and the user experience decreases, as the focus shifts from developing correct software to refactoring the code to make the proof complexity manageable for the solver.

In Dafny, many axioms, for example, describing properties for collection types, are automatically defined in the background whenever a proof is triggered. The solver can then use this information provided by the axiom to prove the correctness of the program. There are some problems with axioms that affect proof brittleness, and which we found during our experiments to remove certain axioms from Dafny. (1) Axioms increase the complexity of the proof obligations. The more information there is for the solver, the more options there are to discharge a proof. (2) Universally quantified axioms provide properties and the solver decides where to instantiate the property. Therefore, the developer cannot control this. (3) Not every property is easily expressible in an axiom, because the solver can quickly run into matching loops, where the SMT solver keeps instantiating quantified axioms for new terms.

In this paper, we propose *free facts* as an alternative automatic mechanism for giving properties similar to those defined in axioms, but on concrete instances of the program code. For example, when we detect a set union of sets A and B, we generate a property describing how to retain the original sets from the union; the property specifically mentions the instances A and B, rather than using an axiom that quantifies over any two sets. The advantage is that the property is already instantiated without giving the solver the option to use it elsewhere. Furthermore, the developer does not have to change their behavior because the behavior is fully automated.

In this paper, we make the following contributions:

- We introduce *free facts* as an alternative to universally quantified axioms.
- We define *free facts* for properties of different collection types in Dafny.
- We provide an implementation of *free facts* in Dafny as a proof of concept.
- We evaluate the impact of *free facts* with regard to proof brittleness in Dafny and compare them to universally quantified axioms.

2 Dafny and Its Verifier

Our work is in the context of the Dafny programming language. Dafny supports formal reasoning about programs and for that purpose features an automated

¹ Also referred to as *proof instability*.

program verifier. The verifier operates in two phases: it first prescribes proof obligations (using the intermediate verification language Boogie) and then attempts to discharge these proof obligations using automatic decision procedures (which are provided in an SMT solver). In this section, we describe the general structure of how the proof obligations are prescribed. The full details of how (an early version of) Dafny is translated into the intermediate verification language Boogie are recorded in Marktoberdorf Summer School lecture notes [28].

2.1 Proof Obligations

To prescribe proof obligations, Dafny uses the Boogie intermediate verification language [6]. Boogie features first-order declarations of types, functions, and axioms, as well as imperative procedures. Procedure bodies consist of statements from a simple while language. The Boogie tool generates a verification condition for each procedure. More precisely, given axioms A , and with a procedure whose pre- and postconditions are Pre and $Post$, respectively, and whose body is a statement S , Boogie generates the logical verification condition

$$A \wedge Pre \implies wp[S, Post]$$

where $wp[S, Post]$ is first expanded to be the weakest precondition of S with respect to $Post$ [18].

For the purposes of this paper, it suffices to understand three kinds of statements in Boogie:

- Assignment statement $x := E$ evaluates expression E and then assigns its value to variable x .
- Assertion statement **assert** P adds condition P as a verification debit.
- Assumption statement **assume** Q adds condition Q as a verification credit.

Expressions in Boogie are *total*; that is, it is legal to apply operators and functions to any arguments. If the source language (Dafny, in our case) wants to prescribe a proof obligation for one of its expressions, then it must introduce an assertion in Boogie. For example, the translation of the Dafny statement $x := y/z$ into Boogie is

```
assert z ≠ 0; x := Div(y, z)
```

This instructs the Boogie tool to check the condition $z \neq 0$, reporting an error if the condition cannot be proved, and then assign the variable x .

An assumption is used to state a condition that the verifier is allowed to use. Sound verification requires that every assumption be justified in some way, but such justification lies outside the use of Boogie; Boogie allows verification-tool authors to introduce such assumptions. Sometimes, an assumption is justified by some property of the programming language or by a limitation of the program verifier. For example, a program verifier for a C-like language may choose to verify only those runs of a program that do not exceed the available memory. For an allocation statement $x := \mathbf{malloc}(1024)$, such a verifier would introduce an assumption along the lines of

```
x := ...; assume x ≠ 0 ∧ size(x) = 1024
```

Procedure pre- and postconditions (introduced, respectively, with **requires** and **ensures** clauses) provide a convenient way to introduce verification debits and credits at procedure boundaries. In particular, a precondition turns into an implicit **assert** statement at a call site and a matching implicit **assume** statement at the beginning of the procedure body. Conversely, a postcondition turns into an implicit **assert** statement at the end of the procedure body and a matching implicit **assume** statement on return from each call.

Here is a small Boogie example that illustrates these features:

```
axiom A
procedure M(x: X) returns (y: Y)
  requires Pre(x) ensures Post(x, y)
{
  assert P(x);
  assume Q(x);
  y := E(x)
}
```

Procedure M declares one in-parameter (x) and one out-parameter (y). For this example, Boogie generates the verification condition

$$A \wedge Pre(x) \implies P(x) \wedge (Q(x) \implies Post(x, E(x)))$$

and passes it to the decision procedures in Boogie's underlying SMT solver.

2.2 Axioms Versus Assumptions

In the example above, the proof goal $Post(x, E(x))$ has three antecedents: the axioms A , the precondition $Pre(x)$, and the assumption $Q(x)$. Further antecedents in other examples include guard conditions from control flow like **if** and the postconditions of any calls. Logically, there is no difference between these kinds of antecedents. Instead, they are all assumptions, but stated in different contexts and different scopes.

Stylistically, **assume** statements are used to introduce assumptions about local variables (like the result of a procedure call), whereas axioms are used to describe properties of global functions or operators of the language (like Div from a previous example above).

2.3 Expression Translation

As further background for our paper, let us describe the general translation of expressions from the source language (Dafny) into the intermediate verification language (Boogie). The expression translation is part of the translation of any statement, so we will use a Dafny assignment statement $x := E$ as a running example; other statements are similar (see [28]). There are three parts to the translation of expressions from Dafny to Boogie.

Translation Mapping. The translation of $x := E$ will map the Dafny variables into corresponding variables in Boogie. For our purposes, we will assume the Boogie variables have the same names. Thus, the left-hand side x in Dafny simply maps into a Boogie variable x .

The right-hand side of the assignment is more interesting. Its translation also needs to map Dafny operators and functions into Boogie counterparts. For example, we can imagine that Dafny’s integer-division operator $/$ is translated into a Boogie function *Div*, as we saw in an example above. For this purpose, we introduce a translation function *Tr*:

$$x := \text{Tr}[E]$$

where, for example, $\text{Tr}[E0/E1] = \text{Div}(\text{Tr}[E0], \text{Tr}[E1])$. (In many of our examples that follow, we will use the same operator symbol in Boogie as in Dafny.)

Checking Well-Formedness. As we mentioned above, expressions in Boogie are total. In contrast, operators and functions in Dafny can be *partial*. The translation from Dafny to Boogie therefore prescribes *well-formedness checks*, as we will indicate with the translation function *Wf*. With these in mind, the translation of an assignment statement $x := E$ becomes

$$\mathbf{assert} \text{ Wf}[E]; \quad x := \text{Tr}[E]$$

For example, we have

$$\text{Wf}[E0/E1] = \text{Wf}[E0] \wedge \text{Wf}[E1] \wedge \text{Tr}[E0] \neq 0$$

Introducing Assumptions. Translation functions like *Tr* and *Wf* have been described before (e.g., [27,28]). What Dafny also uses, but which has not been described, is a template for introducing assumptions. Previously, this part of the translation has been limited in focus, mostly to try to speed up verifier performance of function calls. We will not describe the details of these previous assumptions, since they are not the subject of this paper. Relevant to this paper is just that the translation process includes not only the translation mapping *Tr* and the well-formedness checks *Wf*, but also an *assumption generator* *Ag*. The assumption generator is used as follows, as we can now show the complete translation of expressions from Dafny to Boogie:

$$\mathbf{assert} \text{ Wf}[E]; \quad \mathbf{assume} \text{ Ag}[E]; \quad x := \text{Tr}[E]$$

As we mentioned, assumption generators have had limited use in Dafny. Indeed, for most expressions E , we have $\text{Ag}[E] = \text{true}$. It is into these assumption generators that we will incorporate our *free facts*, as we will describe next.

3 Free Facts

In this section, we present and discuss our concept for free facts. First, as motivation, we show a common pattern that currently needs manual proof effort. Then, we define its automated mechanism and provide free facts for Dafny’s collection types. Finally, we discuss free facts in terms of use cases and limitations.

3.1 Motivating Example

Axioms for Operations of Built-in Types. The general strategy for axiomatizing the operators of built-in types in Dafny is to define them in terms of primitive operators. For example, set operations are defined in terms of set membership as follows:

$$\forall x, S, T \cdot x \in S \cup T \iff x \in S \vee x \in T$$

This strategy gives rise to a kind of rewriting that moves toward smaller terms, and hence (by itself) terminates. However, this strategy alone does not give equality between terms, a property that logic gives the name *extensionality*. Extensionality becomes important when terms are used as arguments to other functions. Dafny thus also uses an extensionality axiom. For example, the one for sets looks like

$$\forall S, T \cdot S =_{set} T \implies S = T$$

where $=_{set}$ is the set-equality operator in Dafny and $=$ is the verifier's equality.

How Axioms Get Used. Universal quantifiers in Dafny's verifier are used through instantiation. To control this process, each quantifier has a *matching pattern* [17]. The verifier instantiates an axiom if, during proof construction, some of the prover's ground terms look like the matching pattern.

For example, the matching pattern for each of the quantifiers in the examples above are the left-hand side of the main connective in the quantifier body. So, if the prover's ground terms happen to contain $y \in A \cup B$ for some expressions y, A, B , then the first quantifier above is instantiated with $x, S, T := y, A, B$. But note that the quantifier is not instantiated if the ground terms only contain $y \in A$ and $y \in B$. In the same way, the quantifier in the extensionality axiom is instantiated only if there already is a ground term that mentions $=_{set}$.

Derived Properties. Using the defining axioms and the axiom of extensionality, it is possible to prove additional properties as theorems, such as

$$\forall x, S \cdot x \notin S \implies (S \cup \{x\}) \setminus \{x\} = S$$

This property is often used in proofs of Dafny programs. A sketch of a prototypical example thereof is a loop that wants to maintain $P(S)$ as a loop invariant, where P is some predicate on the set S , and the loop body contains an assignment $S := S \cup \{x\}$.

Since the theorem above is useful, it is tempting, as developers of the Dafny verifier, to include the theorem among the verifier's axioms. Unfortunately, it far too often instead has a negative effect on prover performance, because the quantifiers end up being instantiated too often. So, the theorem above is not included in Dafny. Instead, programs that need the property tend to include a user-defined assertion of the property, which the Dafny verifier proves and then uses. For example, it is typical to see Dafny code snippets like

```
S' := S ∪ {x};
assert S' \ {x} =set S
```

Similar code snippets are frequently used for other types and operators as well. For instance, here is an example that uses sequences:

```
x := A[0];
A' := A[1..];
assert [x] · A' =seq A
```

3.2 Free Facts

Our aim is to obtain the desired automation in cases like our motivating example, but without risk of causing the verifier to instantiate the derived-property theorems too many times. We do this by instantiating such theorems *before* sending verification conditions to the verifier. We call the result of such an instantiation a *free fact*, and we include free facts among the generated assumptions (translation function Ag in Sect. 2.3). For example, as motivated by the example in the previous subsection, the free facts we generate

$$\begin{aligned} \text{Ag}[S \cup T] &= \\ &\text{Ag}[S] \wedge \text{Ag}[T] \wedge \\ &\text{Tr}[S] = (\text{Tr}[S] \setminus \text{Tr}[T]) \cup (\text{Tr}[S] \cap \text{Tr}[T]) \end{aligned}$$

Note that this generalized property works for any set T , not just a singleton set $\{x\}$ as we showed in our motivating example above.

To support free facts Dafny, we first decide on some candidate theorems (more about that in Sect. 3.3). The mechanism we then use to control instantiations is similar to what the verifier does with matching patterns, but with an important difference: While the verifier's set of ground terms grows as the verifier performs inferences, the terms available to free-fact generation are those that occur syntactically in the program. To understand this syntactic-terms limitation, suppose we tried to encode the associativity of set union as a free fact:

$$\begin{aligned} \text{Ag}[S \cup (T \cup U)] &= \\ &\text{Ag}[S] \wedge \text{Ag}[T] \wedge \text{Ag}[U] \wedge \\ &\text{Tr}[S] \cup (\text{Tr}[T] \cup \text{Tr}[U]) = (\text{Tr}[S] \cup \text{Tr}[T]) \cup \text{Tr}[U] \end{aligned}$$

This would generate the free fact only if the Dafny program contained an expression of the form $S \cup (T \cup U)$. However, it would not generate the free fact if the syntax was slightly different. For example, the free fact would not be generated for a code snippet like

```
a := T ∪ U;
b := S ∪ a
```

since no single expression contains two union operators. Because of this syntactic limitation, free facts are most effective when the matching pattern has just one operator.

3.3 Free Facts for Collection Types

Collections are nontrivial data types, yet are widely used in software systems. In Sect. 2, we explained how axioms are used to describe properties of built-in types in Dafny. In Sect. 3.1, we gave an example of a property that performs poorly as an axiom and therefore has to be provided as a user-defined assertion. Hence, we looked for assertions on collection types in different systems that are implemented in Dafny and defined our free fact properties based on our findings. In Table 1, we show a complete list of all free facts that we defined for this paper. Only the last two properties are defined as axioms in the current version of Dafny. The other properties have to be defined by the developer in an assertion if needed and are therefore new properties in the automatic encoding.

Table 1. Free Fact Properties for Collection Types in Dafny.

Collection Type	Operation in Code	Free Fact Property
Set	$S \cup T, S \setminus T$	$S = (S \setminus T) \cup (S \cap T)$ $T = (T \setminus S) \cup (S \cap T)$
Multiset (allows duplicates)	$S \cup T, S \setminus T$	$S = (S \setminus T) \cup (S \cap T)$ $T = (T \setminus S) \cup (S \cap T)$
Map	$M + N$	$M.keys \cap N.keys = \emptyset \implies M = M + N - N.keys$ $M.keys \cap N.keys = \emptyset \implies N = M + N - M.keys$
Sequence	$X \cdot Y$	$X = (X \cdot Y)[0.. X]$ $Y = (X \cdot Y)[X .. (X \cdot Y)]$
	$X[i.. X], X[0..i]$	$X = X[0..i] \cdot X[i.. X]$
	$X[i..j]$	$X[0..j] = X[0..i] \cdot X[i..j]$
		$X[i.. X] = X[i..j] \cdot X[j.. X]$

Operations: Map merge: +, Map difference: -, Seq. concatenation: ·, Seq. length: |X|, Subsequence: X[i..j]
 See Dafny reference manual for further explanations of the operations [3].

Sets and Multisets. In contrast to ordinary sets, multisets allow duplicate entries. Apart from that, we define the same free fact properties for sets and multisets. Whenever we detect a (multi-)set union or (multi-)set difference in the Dafny code, we generate the two free fact properties in the very right column that describe a relationship between the (multi-)set operations \setminus , \cup , and \cap .

Maps. We define similar free fact properties as the ones for sets for finite maps, as well. The merge of two maps is not commutative, because values are overridden if the key already exists in the left-hand side map; hence, the key sets of the two maps need to be disjoint.

Sequences. For sequences, we define free fact properties for the concatenation of two sequences and the subsequence operation $X[i..j]$ (from index i inclusive to index j exclusive). For the concatenation of two sequences $X \cdot Y$, we get the original set X by taking the subsequence of the concatenation from 0 to the length of X ($|X|$). Respectively, set Y is equal to the subsequence of the concatenation from $|X|$ to $|(X \cdot Y)|$. For the subsequence operation, we define a free fact for the special case where one of the indices is 0 or the length of the

sequence, i.e., dropping the start or the end of the sequence, and a general one for arbitrary indices. The latter is generated only if the special case is not true.

3.4 Discussion

Applicability Limitations. Free facts rely on syntactically detectable operations in the Dafny code. Collections are particularly well suited, because (1) they often require additional properties in the form of assertions and (2) their operations are easy to detect since they are not scattered over multiple statements.

During our experiments, we encountered proof brittleness when using the non-linear arithmetic setting of the SMT solver. As an alternative, we tried to define free facts for non-linear arithmetic properties. Unfortunately, the distribution properties of $+$ and $*$ are not possible under our syntactic-terms limitation.

Free Facts as Replacement for Axioms. Our aim is not to replace axioms altogether. Axioms are an effective way of globally providing certain properties that do not require a proof. SMT solvers use the given information dynamically and quickly and decide about their instantiation. However, the instantiation of axioms must always be regulated by matching patterns. Otherwise, the solver will quickly run into matching loops. For some properties, it is difficult to define the matching pattern in a way that is not too restrictive, such that the axiom is never really instantiated or that it is instantiated too often, resulting in performance issues. For example, if we would define the first free fact property from Table 1 as a quantified axiom with $S \cup T$ as trigger, this leads to an endless instantiation of this property as the trigger matches part of the term. For these properties, we propose free facts as an alternative because they are defined on concrete instances of the code and the solver cannot instantiate them arbitrarily, i.e., even if a free fact that we generate is not used, it does not keep generating additional facts, like universally quantified axioms can. The overall goal is to replace only inefficient axioms and to add properties for further automation where it fits the conditions of free facts.

Increasing the Level of Automation. The high degree of automation in verification tools has led to complex queries for the SMT solver in the backend which increases proof brittleness. Therefore, it may seem unintuitive to propose another automatic mechanism to be added on top of Dafny to counter proof brittleness. With free facts, we propose a change in the automatic encoding of the proof obligations to *avoid* universally quantified axioms. Other parts of the automatic encoding could possibly also be improved. In other places, however, it may be better to reduce automation, giving control back to the developer. Overall, a composition of multiple solutions, not only for automatic encoding from Dafny to Boogie, but also for the way programs and proofs are defined and how the solver is used in the backend, will bring progress regarding the overall problem.

4 Evaluation

Proof brittleness is a problem that can occur when the proof obligations and information for the program verifier are too complex. In this section, we eval-

uate *free facts* in terms of their usefulness with regard to proof brittleness and compare them to axioms in Dafny.

4.1 Research Questions

In particular, we define the following research questions:

RQ1: Is it possible to define free facts in Dafny?

RQ2: To what extent can free facts reduce the proof brittleness in Dafny?

RQ3: Are free facts superior to axioms in terms of their verification time and resource count?

With **RQ1**, we want to assess the feasibility of free facts. By answering **RQ2**, we gain insights into how well suited *free facts* are to reduce the proof brittleness problem. With **RQ3**, we may estimate how *free facts* compare to axioms by comparing both approaches using the collection type properties (see Sect. 3.3).

4.2 Methodology

To answer our research questions, we created three different branches of Dafny that we used to compare the verification results. All branches can be used like the regular Dafny version.

Master: The master branch² is the original version of Dafny, and we use this branch as the baseline for our evaluation.

Free Facts: The free facts branch³ implements all properties of Sect. 3.3 as free facts and none as axioms.

Axioms: The axioms branch⁴ implements most properties from Sect. 3.3 as axioms and none as free facts. The differences between the axioms and the free facts in Table 1 are: The free facts for (multi-)set union and (multi-)set difference are implemented with only set difference as trigger. The free fact for sequence concatenation has a more restrictive trigger. Without adaptations, the SMT solver ran into matching loops.

To answer **RQ1**, we implement free facts for collection type properties as described in Sect. 3 in the free facts branch. Afterwards, we run the Dafny test suite to check whether the free fact branch of Dafny is working as intended. To reason about **RQ2** and **RQ3**, we perform a mutation-based analysis using an internal tool on three subject systems that are implemented in Dafny and compare the performance of our different Dafny branches. The idea of the mutation-based analysis is to syntactically mutate the subject systems to mimic the developer that observes proof brittleness when they make slight changes to their program. Furthermore, we collect metrics that indicate proof brittleness. For **RQ2**,

² <https://github.com/dafny-lang/dafny>, commit 2e7de95.

³ <https://github.com/dafny-lang/dafny/tree/tb-experiment-freefacts>, 62b2a90.

⁴ <https://github.com/dafny-lang/dafny/tree/tb-experiment-freefactaxioms>, e9a1bd2.

Table 2. Subject Systems and their Characteristics.

Subject Systems	# Procedures	LOC	# Specifica- tions	# CTs
Cedar ^a	1,454	19,695	5,974	589
Dafny Libraries ^b	6,398	15,729	4,668	638
Internal System	14,504	17,858	4,003	371

Procedures include methods, functions, and lemmas.

LOC — Non-whitespace lines of Code. # CTs — Explicit mentions of collection types.

^a <https://github.com/cedar-policy/cedar>

^b <https://github.com/dafny-lang/libraries>

we compare the free facts branch with the master branch, and for **RQ3**, we compare the free facts branch with the axiom branch. We describe the tool, the metrics, and the subject systems in the following.

Mutation-Based Analysis of Two Dafny Versions. For the evaluation, we use an internal tool that syntactically mutates our subject systems and collects different metrics for two different branches of Dafny. It mutates every procedure of the subject systems five times, randomly changing the names of all identifiers and the order of declarations, and runs each mutant with a random seed that the SMT solver uses when making decisions that can be arbitrary. The tool collects the following metrics for each procedure on the two different branches: The number of failed runs, the verification time, and the resource count (a Z3-specific metric for the proof complexity).

The number of failed runs can be used as an indicator for proof brittleness if some runs fail and some do not. We sum up the number of failed runs of the single procedures in one subject system to a total number of failed runs for each branch. If the number for branch A is higher than the number for branch B, this gives the indication that branch A is more brittle than branch B. To obtain a holistic evaluation, we combine the number of failed runs with two further metrics that indicate proof brittleness, namely, the average verification time and the average resource count. It has been observed that both metrics correlate with the proof brittleness problem [41].

Subject Systems. We evaluate *free facts* on three large-scale subject systems that are implemented in Dafny. One of the subjects is an internal policy-checking system. The other two systems, Cedar⁵ and the Dafny Libraries⁶, are openly accessible on GitHub. We selected the internal subject system because it has a high usage of sequence collections and it lets us discuss the results with domain experts to verify our results and prevent potential errors. In Table 2, we give an overview on the subject systems and provide metrics that show their size and complexity. All subject systems contain assertions in the code that are similar to our free fact properties (cf. Sect. 3.1).

⁵ <https://github.com/cedar-policy/cedar>.

⁶ <https://github.com/dafny-lang/libraries>.

4.3 Results and Discussion

RQ1: Is it possible to define free facts in Dafny?

We implemented free facts as described in Sect. 3. In the Dafny integration test suite, we found seven test cases with assertions that can be removed because of free facts. In some cases, such removal led to a higher resource count, but at the other end of the spectrum, one case had a resource count that was 12 times lower than the failing verification without free facts. We can therefore answer RQ1 positively for collection type properties. In Sect. 3.4, we have already discussed the applicability of our concept in terms of its limitations due to the syntactical detection.

RQ2: To what extent can free facts reduce the proof brittleness in Dafny?

In Table 3, we show the results of the mutation-based analysis of the master and the free facts branch for our three subject systems. We give the average improvement for the runtime and resource count in percent, and the difference between the total number of failed runs between the master and the free facts branch. For runtime and resource count, a positive percentage ($+x\%$) means that the free facts branch performed on average x percent faster/with less resources than the master branch. For the difference in failed runs, a negative number ($-y$) means that the free facts branch failed y fewer times than the master.

The largest effect of free facts was seen in the Dafny Libraries, where we measure an average improvement in the runtime of 41%. In contrast, the resource count worsened by 4% on average. The difference in the number of failed runs is only minor with 25 additional failed runs. For the internal system, we observe an improvement in runtime by 9% and resource count by 8%. However, the difference in the number of failed runs increased by 367 failed runs. Compared to the total number of runs (72,520) this is still just a change of 0.5%. Besides the average of the whole system, we also looked at the maximum average improvement of the procedures in Cedar and the Dafny libraries, which was +748% runtime and +160% resource count for Cedar and +19597% runtime and +277% for the Dafny libraries. The procedure in the Dafny libraries that had the extreme improvement in runtime actually went from being brittle with only 2/5 succeeding runs on the master to being stable with 5/5 succeeding runs with free facts.

Overall, we can answer RQ2 neither positively nor negatively. The average resource count suggests a slight deterioration compared to the master branch for Cedar and the Dafny Libraries. With free facts, we generate certain properties automatically that previously had to be defined manually. As long as those assertions are not removed from the code, the assertions still provide a verification debit for the solver and therefore obfuscate the benefit of free facts. We discuss this in detail in Sect. 4.4. In contrast, the runtime improved for all subject systems. We conclude from this that, even though the proofs seem to be slightly more complex (potentially because the subject systems still contain

Table 3. Results: Master vs. Free Facts

Subject Systems	Runtime	Resource Count	# Failed Runs		Diff.
			total		
	Avg. Improvement	Avg. Improvement	master	free facts	
Cedar ^a	+1%	-6%	38	66	+28
Dafny Libraries ^b	+41%	-4%	392	417	+25
Internal System	+9%	+8%	51	418	+367

The total number of all runs ($5 * \#procedures$): Cedar - 7,270. Libraries - 31,990. Internal - 72,520.

^a <https://github.com/cedar-policy/cedar>

^b <https://github.com/dafny-lang/libraries>

Table 4. Results: Axioms vs. Free Facts

Subject Systems	Runtime	Resource Count	# Failed Runs		Diff.
			total		
	Avg. Improvement	Avg. Improvement	axioms	free facts	
Cedar ^a	+7%	-1%	40	69	+29
Dafny Libraries ^b	+23%	+9%	557	511	-46
Internal System	+9%	+6%	386	391	+5

The total number of all runs ($5 * \#procedures$): Cedar - 7,270. Libraries - 31,990. Internal - 72,520.

^a <https://github.com/cedar-policy/cedar>

^b <https://github.com/dafny-lang/libraries>

assertions that could be removed), but easier and therefore faster to close given the additional information. Additionally, we observed strong improvements for individual procedures, such as in the case of the maximum values, which shows the great potential of free facts as a concept. We expect an alignment of the resource count to the positive trend of the runtime once free facts are deployed, and developers adapt their behavior to the free fact generation of Dafny.

RQ3: Are free facts superior to axioms in terms of their verification time and resource count?

In Table 4, we summarize the results for the mutation-based analysis between the free facts and the axiom branch. While the difference in the number of failed runs between the two branches is rather small (+29 failed runs for Cedar, -46 failed runs for the Dafny libraries, and +5 failed runs for the internal system), there is an improvement in favor of the free facts branch in both the average runtime and resource count. The average improvement of free facts for the Dafny libraries and the internal system is higher (+23% runtime and +9% resources for the libraries and +9% runtime and +6% resources) than the one for Cedar (+7% runtime and -1% resources), which we explain again by the fact that Cedar uses fewer collections.

Overall, we can answer RQ3 positively, since the free facts branch took less time and resources on average than the axiom branch. Note that we were not able to implement all free fact properties as axioms, as some led to matching loops, which would quickly lead to a timeout for a majority of the procedures. In fact, this supports our argument that free facts are an alternative automatic mechanism that is superior in certain cases where axioms are inefficient.

4.4 Threats to Validity

Removal of Assertions in the subject systems. As explained in Sect. 3.1, we are automating the generation of certain properties in Dafny that previously had to be defined as assertions by the developer. Therefore, the full potential of free facts can only be observed if these assertions are removed from the code as they are translated into a proof debit for the SMT solver. Leaving the assertions in the code does not affect the correctness, but it does affect the runtime and resource count. In our evaluation, we did not remove these assertions from the subject systems, as the manual effort would be too high since we used large-scale systems. As a result, this benefit is not measurable in our experiments, and the values for newly built systems using free facts might be better as we expect developers to adapt their proofs accordingly. It is conceivable that such a change in behavior would make free facts better overall.

Reproducibility. Parts of our experiments are not reproducible for externals because we used (1) an internal tool to perform the variability analysis and (2) an internal subject system. However, the implementation of all branches is publicly available, as well as the other two subject systems. With that, a similar report can be generated, also including mutations. Detailed instructions can be found in the Dafny documentation in Section *Measuring proof brittleness* [3].

Since both the mutation-based analysis and the decision process of the SMT solver involve randomness, and the subject systems are productively used systems that are regularly modified, the exact results will still vary from run to run. However, the overall trend is reproducible.

Transferability. We have only performed our evaluation on Dafny with Z3 as SMT solver. We cannot claim that our results are fully representative for other automated verifiers or SMT solvers. Nevertheless, the concept is transferable to other verifiers as well. We believe that our results provide valuable insight into the proof brittleness problem and may influence future work.

5 Related Work

Resolving Proof Brittleness. Few approaches have addressed proof brittleness, mainly because: (1) it is a rather new challenge stemming from the recent verification of complex systems, and (2) the complexity of the SMT-solvers decision process using heuristics and randomness to a certain degree. We categorize the papers into the Dafny pipeline, writing program, specification, and proof

(proof engineering); automatic encoding of Dafny to SMT; the SMT solver itself. Five of the papers have been presented at this year’s Dafny workshop [1].

Proof Engineering. McLaughlin et al. [32] introduce Dafny64, a mode of using Dafny that significantly reduces verification resources by stripping back automation for proofs. Cutler et al. [15] improved the stability of type safety proofs in Dafny by making functions opaque (i.e., making the body of the function unavailable) and specifying them manually such that the solver does not have to reason about multiple large definitions simultaneously. Ho and Pit-Claudel [23] improved the debugging of brittle lemmas by using Dafny’s abstract modules to achieve an induction principle similar to that in the theorem prover Coq [7]. These papers highlight the necessity to reduce automation, such that the developer gains more control over the verification task. Although we agree with this suggestion, with *free facts* we aim to improve the automatic encoding in the next step of the pipeline. We reduce the options for the solver while maintaining the usability of Dafny as an automatic verifier. The proposed approaches and free facts can be applied in parallel to maximize results.

Automatic Encoding. Srinivasan et al. [38] identify the boundaries where information from other modules should be made opaque to leverage the automation of Dafny in the best possible way. In particular, they concentrate on quantifier instantiations. With *free facts*, we also propose a technique that improves the automatic encoding; however, we focus on code-based detection of the need and instantiate properties to avoid universally quantified axioms.

SMT Solver. Mugnier et al. [34] propose a portfolio of SMT solvers meaning that different SMT solvers and different versions of SMT solvers are used to get more performant proofs. This work is orthogonal to *free facts* and can be applied later to further increase the performance.

Detection of Proof Brittleness. The detection of proof brittleness is indirectly related to our work, because detecting proof brittleness does not directly reduce brittleness. However, detection can be used to evaluate approaches to resolve proof brittleness and may lead to a better understanding of the problem and more targeted solutions in the future.

As a first measure, for early detection, Dafny and F* provide a command-line flag to execute multiple randomized verification runs [2, 3]. Mariposa [41] is a tool that performs a mutation-based analysis to detect and quantify SMT-based proof brittleness. In their paper, they performed an evaluation on six different verification projects, provide a benchmark, and describe their findings. The Axiom Profiler [10] is a tool that analyzes instantiation problems, e.g., matching loops caused by axioms, by logging information of SMT runs.

Quantifier Instantiation. Since the Simplify prover introduced E-matching [17], it has been adapted and improved in a number of SMT solvers [4, 8, 16], as well as the pattern selection in many SMT-based automated verifiers (including Dafny) [14, 30, 35]. However, there are also a few approaches that focus on avoiding quantifier instantiation altogether. The tool Leon [11] (a predecessor of Stainless [20]) is an SMT-based verifier for programs written in Scala. It avoids quantifiers by unfolding recursive definitions as needed. Liquid Haskell adds refinement types to Haskell and implements a recursive technique similar to Leon called refinement reflection [40]. The idea of both approaches is comparable to free facts, but they are applied to functions and the detection of the need to unfold is more dynamic than our syntactic detection.

6 Conclusion

Proof brittleness is a persistent issue in automated verification, particularly with complex and large-scale software. Research is still in its early stages of understanding and improving the sources of proof brittleness. The complexity stems from the high degree of automation. We believe that the entire process, from proof engineering to the automation of the verifier and the SMT solver itself, requires revision. We aim to improve the automatic encoding in Dafny by generating pre-instantiated *free facts*. This approach reduces the number of quantifiers in the verification conditions compared to quantified axioms without increasing manual effort for the developer. With collections, we found a good use case for free facts, and we plan to experiment with further use cases in the future.

References

1. Dafny 2024 - POPL 2024. <https://popl24.sigplan.org/home/dafny-2024#event-overview>. Accessed 15 Mar 2024
2. Understanding how F* uses Z3 - Proof-Oriented Programming in F* documentation. https://fstar-lang.org/tutorial/book/under_the_hood/uth_smt.html. Accessed 01 July 2024
3. Dafny Documentation (2024). <https://dafny.org/dafny/DafnyRef/DafnyRef.html>. Accessed 18 Mar 2024
4. Bansal, K., Reynolds, A., King, T., Barrett, C., Wies, T.: Deciding local theory extensions via e-matching. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 87–105. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21668-3_6
5. Barbosa, H., et al.: cvc5: a versatile and industrial-strength SMT solver. In: TACAS 2022. LNCS, vol. 13243, pp. 415–442. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_24
6. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_17

7. Barras, B., et al.: The Coq Proof Assistant Reference Manual—*Coq: Version*. vol. 6, p. 1 (2006)
8. Barrett, C., et al.: Cvc4. In: Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, pp. 171–177. Springer, Heidelberg (2011)
9. Barrett, C., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Model Checking, pp. 305–343. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_11
10. Becker, N., Müller, P., Summers, A.J.: The axiom profiler: understanding and debugging SMT quantifier instantiations. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11427, pp. 99–116. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17462-0_6
11. 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 2013), pp. 1–10. Association for Computing Machinery (2013)
12. Bornholt, J., et al.: Using lightweight formal methods to validate a key-value storage node in Amazon S3. In: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP 2021), pp. 836–850. Association for Computing Machinery (2021)
13. Chudnov, A., et al.: Continuous formal verification of Amazon s2n. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 430–446. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_26
14. Cohen, E., et al.: VCC: a practical system for verifying concurrent C. In: Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, 17–20 August 2009, pp. 23–42. Springer, Heidelberg (2009)
15. Cutler, J.W., Hicks, M., Torlak, E.: Improving the Stability of Type Safety Proofs in Dafny (2024). <https://popl24.sigplan.org/details/dafny-2024-papers/3/Improving-the-Stability-of-Type-Safety-Proofs-in-Dafny>. in [1]
16. De Moura, L., Bjørner, N.: Efficient E-matching for SMT solvers. In: Automated Deduction—CADE-21: 21st International Conference on Automated Deduction Bremen, 17–20 July 2007, pp. 183–198. Springer, Heidelberg (2007)
17. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. *J. ACM* **52**(3), 365–473 (2005)
18. Dijkstra, E.W.: *A Discipline of Programming*. Prentice Hall (1976)
19. Furia, C.A., Nordio, Martín and Polikarpova, N., Tschannen, J.: AutoProof: auto-active functional verification of object-oriented programs. *Int. J. Softw. Tools Technol. Transf*
20. Hamza, J., Voirol, N., Kunčak, V.: System FR: formalized foundations for the stainless verifier. *Proc. ACM Program. Lang.* **3**, 1–30 (2019)
21. Hance, T., Lattuada, A., Hawblitzel, C., Howell, J., Johnson, R., Parno, B.: Storage Systems are Distributed Systems (So Verify Them That Way!), pp. 99–115 (2020)
22. Hawblitzel, C., et al.: IronFleet: proving practical distributed systems correct. In: Proceedings of the 25th Symposium on Operating Systems Principles (SOSP 2015), pp. 1–17. Association for Computing Machinery (2015)
23. Ho, S., Pit-Claudel, C.: Incremental Proof Development in Dafny with Module-Based Induction (2024). in [1]
24. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: a powerful, sound, predictable, fast verifier for C and Java. *NASA Formal Methods* **6617**, 41–55 (2011)
25. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: a software analysis perspective. *Formal Aspects Comput.* **27**(3), 573–609 (2015)

26. Klein, G., et al.: seL4: formal verification of an operating-system kernel. *Commun. ACM* **53**(6), 107–115 (2010)
27. Leino, K.R.M.: Ecstatic: an object-oriented programming language with an axiomatic semantics. In: *The Fourth International Workshop on Foundations of Object-Oriented Languages* (1997)
28. Leino, K.R.M.: Specification and verification of object-oriented software. In: Broy, M., Sitou, W., Hoare, T. (eds.) *Engineering Methods and Tools for Software Safety and Security*, NATO Science for Peace and Security Series D: Information and Communication Security, vol. 22, pp. 231–266. IOS Press (2009)
29. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) *LPAR 2010. LNCS (LNAI)*, vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
30. Leino, K.R.M., Pit-Claudel, C.: Trigger selection strategies to stabilize program verifiers. In: Chaudhuri, S., Farzan, A. (eds.) *CAV 2016. LNCS*, vol. 9779, pp. 361–381. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_20
31. Liu, J., et al.: P4v: practical verification for programmable data planes. In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM 2018)*, pp. 490–503. Association for Computing Machinery (2018)
32. McLaughlin, S., Jaloyan, G.A., Xiang, T., Rabe, F.: Enhancing Proof Stability (2024). <https://popl24.sigplan.org/details/dafny-2024-papers/14/Enhancing-Proof-Stability>, in [1]
33. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) *TACAS 2008. LNCS*, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
34. Mugnier, E., McLaughlin, S., Tomb, A.: Portfolio Solving for Dafny (2024). <https://popl24.sigplan.org/details/dafny-2024-papers/8/Portfolio-Solving-for-Dafny>, in [1]
35. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: a verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) *VMCAI 2016. LNCS*, vol. 9583, pp. 41–62. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_2
36. 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 27th ACM Symposium on Operating Systems Principles (SOSP 2019)*, pp. 225–242. Association for Computing Machinery (2019)
37. Protzenko, J., et al.: EverCrypt: a fast, verified, cross-platform cryptographic provider. In: *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 983–1002 (2020)
38. Srinivasan, P., Padon, O., Howell, J., Lattuada, A.: Domesticating Automation (2024). <https://popl24.sigplan.org/details/dafny-2024-papers/2/Domesticating-Automation>. in [1]
39. Swamy, N., et al.: Dependent types and multi-monadic effects in F*. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*, pp. 256–270. Association for Computing Machinery (2016)

40. Vazou, N., et al.: Refinement reflection: complete verification with SMT. Proc. ACM Program. Lang. **2**(POPL), 1–31 (2017)
41. Zhou, Y., Bosamiya, J., Takashima, Y., Li, J., Heule, M., Parno, B.: Mariposa: measuring SMT instability in automated program verification. In: Proceedings of the 23rd Conference on Formal Methods in Computer-Aided Design (FMCAD 2023), pp. 178–188 (2023)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

