





Better Counterexamples for Dafny

Aleksandar Chakarov¹, Aleksandr Fedchin² (✉) , Zvonimir Rakamarić¹ , and Neha Rungta¹

¹ Amazon Web Services, Seattle, WA, USA
aleksach, zvorak, rungta@amazon.com

² Tufts University, Medford, MA, USA
aleksandr.fedchin@tufts.edu

Abstract. Dafny is a verification-aware programming language used at Amazon Web Services to develop critical components of their access management, storage, and cryptography infrastructures. The Dafny toolchain provides a verifier that can prove an implementation of a method satisfies its specification. When the underlying SMT solver cannot establish a proof, it generates a counterexample. These counterexamples are hard to understand and their interpretation is often a bottleneck in the proof debugging process. In this paper, we introduce an open-source tool that transforms counterexamples generated by the SMT solver to a more user-friendly format that maps to the Dafny syntax and is suitable for further processing. This new tool allows the Dafny developers to quickly identify the root cause of a problem with their proof, thereby speeding up the development of Dafny projects.

Keywords: Dafny · Counterexample · Verification · SMT

1 Introduction

Dafny [12,11,6] is a verification-aware programming language popular in the automated reasoning community. Amazon Web Services (AWS), in particular, uses Dafny to develop critical components of their access management, storage, and cryptography infrastructures [5]. For these components, developers at AWS are writing Dafny programs that include the specification and the corresponding implementation. The advantage of using Dafny is that one can leverage the built-in verifier during the development process to automatically prove that the implementation of a method satisfies its specification. Finally, Dafny provides compilers for generating executable code in different target languages, such as C#, Java, and Go. For example, AWS developers have implemented the core AWS authorization logic in Dafny, and generated production Java code using a custom Java compiler. However, despite its advantages, Dafny has so far lacked in debugging functionality that could guide the developer to the root cause of a potential assertion (i.e., proof) failure. This was slowing down the developers, and it prompted the work on counterexample extraction that we present in this paper.

To confirm that an assertion holds, Dafny verifier first translates Dafny source into the Boogie [1,3] intermediate verification language. Boogie generates a verification condition and submits it to an SMT solver (in our case Z3 [13,15]). When an assertion is violated, the solver provides a counterexample (i.e., a *counterexample model*). Understanding such counterexamples is key to debugging a failing proof. However, due to the two translation steps separating Dafny code from the SMT query, the counterexamples provided by the solver are difficult to understand and inhibit the debugging process. The scope of the problem becomes apparent from the fact that a counterexample extraction tool was once developed for Boogie [10], a language that is much closer to the solver in the verification pipeline than Dafny.

Prior attempts to present Dafny counterexamples in a human-readable format [9,8] have been successful with integers and Booleans but yielded unsatisfying results for other types. Our main contribution is a tool that improves the readability of Dafny counterexamples for other basic types, user-defined types, and collections. The tool converts a counterexample generated by the solver to a format that is intuitive to Dafny developers. In addition to improving the user experience, our tool lays the foundation for automatic test case generation, as we discuss in Section 4.

2 Motivation

Fig. 1 shows our running example of a Dafny program. The program defines a class `Simple` with an instance method `Match` that returns true if argument `s` (of type `string` that is alias for `seq<char>`) matches the pattern `p`. For simplicity, we only allow the `'?'` meta-character in the pattern, which matches any character. The program also includes specifications in the form of preconditions, postconditions, and loop invariants. The Dafny verifier uses these to prove the correctness of the method implementation.

To demonstrate the usefulness of counterexamples and the need to present them in a human-readable format, we introduce a bug into the `Match` method. We do this by deleting the part of the guard highlighted on line 16, thereby turning the method into a string equality check. The implementation of the method and its specification are no longer in agreement, and the Dafny verifier reports that the postcondition on line 7 might be violated on line 18. Even in this simple case, the information that the verifier gives, although it might help in localizing the problem, does not make the cause of the bug apparent. The counterexample provided by the solver spans hundreds of lines and is difficult to read. For example, Fig. 2 gives a slice of this counterexample showing just that variable `s` has type `seq<char>`.

In contrast, our tool, released with Dafny v3.3.0, generates the following counterexample that triggers the postcondition violation:

```
s:seq<char> = (Length := 1, [0] := 'A');
this:Simple = (p: @1);
@1:seq<char> = (Length := 1, [0] := '?');
```

```

1 class Simple
2 {
3     var p:string
4
5     method Match(s: string) returns (b: bool)
6         requires |p| == |s|
7         ensures b <==> forall n :: 0 <= n < |s| ==>
8             s[n] == p[n] || p[n] == '?'
9     {
10        var i := 0;
11        while i < |s|
12            invariant i <= |s|
13            invariant forall n :: 0 <= n < i ==>
14                s[n] == p[n] || p[n] == '?'
15        {
16            if s[i] != p[i] && p[i] != '?'
17            {
18                return false;
19            }
20            i := i + 1;
21        }
22
23        return true;
24    }
25 }

```

Fig. 1: A Dafny program that matches a string against a pattern. The highlighted code is removed to introduce a bug as described in Section 2.

Here, the first line indicates that argument `s` is a sequence of characters (i.e., a string) of length 1, where the character at index 0 is `A`. Field `p` of the receiving object (`this`) points to object `@1`, where `@1` is a string of length 1 with the `?` meta-character at index 0. With these inputs, the buggy implementation of method `Match` returns false because the pattern and argument are not identical, even though they should match according to the specification.

Before we incorporated our tool into Dafny, it would report the following counterexample for this same program:

```
s = [Length 1](T@U!val!71); this = (T@U!val!75);
```

Clearly the counterexample generated by our tool is much more informative. Among the tools in this space that we know of, only Why3 [7] has counterexample generation functionality of similar complexity.

```

s#0 -> T@U!val!71 // Boogie variable s#0 has ID 71
BoxType -> T@T!val!15 // Boogie's Box type has ID 15
type -> { // The Boogie type of variable s#0 has ID 22
  T@U!val!71 -> T@T!val!22
}
SeqTypeInv0 -> { // Boogie type of s#0 is Seq Box:
  T@T!val!22 -> T@T!val!15
}
$Is -> { // Dafny type of variable s#0 has ID 76
  T@U!val!71 T@U!val!76 -> true
}
Tag -> { // Type with ID 76 is a subtype of a type with ID 13
  T@U!val!76 -> T@U!val!13
}
TagSeq -> T@U!val!13 // Dafny type with ID 13 is seq
TChar -> T@U!val!1 // Dafny type with ID 1 is char
Inv0_ISeq -> { // Dafny type with ID 76 is seq<char>
  T@U!val!76 -> T@U!val!1
}

```

Fig. 2: An extract of a counterexample model generated by Z3 for the code in Fig. 1 that shows that variable `s` has type `seq<char>`.

3 Design and Implementation

We implemented our tool on top of the existing Dafny counterexample extraction functionality by adding key new features such as the ability to extract types from the Z3 model and support complex types (e.g., sequences) beyond just integers and Booleans. Our type extraction supports type parameterization and type renaming, and makes extracted counterexamples useful beyond improved user experience, e.g., automatic test case generation (see Section 4).

We illustrate how the counterexample generation tool works using our running example from Fig. 1. Before the tool can look up the types and values of specific variables, it must first identify the variables and program states¹ relevant to the given counterexample. In our example, there are four relevant program states: the initial state, the state following the initialization of `i`, the state at the loop head, and the state preceding the return statement. There are three relevant variables: `this`, `s`, and `i`. Our tool inherits the extraction of this information from the Z3 model from the existing counterexample generator.

Once we identify the relevant variables and states, we determine the type of each variable. This is a two-step process. First, we extract the Boogie type of a variable in the Boogie translation from the Z3 model (e.g., `Seq Box` for `s` in Fig. 2). Then, we map it to its corresponding Dafny type (`seq<char>` for `s` in

¹ Dafny to Boogie translator marks Dafny program states with the `:capturedState` annotation in Boogie.

Variable	Constraint	Counterexample
<code>b:bv6</code>	<code>b == 1</code>	<code>b:bv6 := 0</code>
<code>r:real</code>	<code>r != 0.2</code>	<code>r:real := 1.0/5.0</code>
<code>c:char</code>	<code>c != 'c'</code>	<code>c:char := 'c'</code>
<code>c:char</code>	<code>c == 'c'</code>	<code>c:char := 'A'</code>
<code>d:M.DType</code>	<code>d.i > 4</code>	<code>d:M.DType = A(i := -34)</code>
<code>a:array2?<int></code>	<code>a.Length0 < 2 a.Length1 < 2 a[1,1] != 3</code>	<code>a:_System.array2?<int> := (Length0 := 2, Length1 := 40, [1,1] := 3)</code>
<code>s:set<int></code>	<code>1 in s</code>	<code>s:set<int> = {1 := false}</code>
<code>s:set<int></code>	<code>1 !in s</code>	<code>s:set<int> = {1 := true}</code>
<code>s:seq<int></code>	<code> s < 1 s[0] != 3</code>	<code>s:seq<int> = [3]</code>
<code>s:seq<int></code>	<code> s < 2 s[1] != 3</code>	<code>s:seq<int> = (Length := 2, [1] := 3)</code>
<code>m:map<int, char></code>	<code>1 !in m</code>	<code>m:map<int,char> = (1 := 'A', 2 := 'B', 3 := 'C', 4 := 'D')</code>

Table 1: Counterexamples generated for different constraints.

Fig. 2). The latter step may require choosing among the different types listed by the model (e.g., between `string` and `seq<char>`). We give preference to the original type names (`seq<char>`) to clearly separate user-defined from built-in types. We also take special care to extract type parameters and reconstruct the Dafny type name from its Boogie translation, for example, `Module.Module2.Class` from `Module_mModule2.Class`.

After determining the type of a variable, our tool extracts the string representation of the variable’s value. The way the value is specified in the counterexample model depends on the variable type. In method `Match` in Fig. 1, the receiver is an instance of a user-defined class `Simple`, so the tool looks up the value of its only field `this.p`. This field is itself a non-primitive variable, and so we recurse into its definition until we reach a value of a primitive type, which we then use to construct the non-primitive value. In case the model does not specify a value for some variable of primitive type, the tool automatically generates an adequate value that is different from any other value of that type in the model or source code.

Our implementation of the counterexample extraction tool supports all basic types, user-defined classes, datatypes, arrays, and the three most commonly used collections (sequence, sets, and maps). See Table 1 for concrete examples of the tool’s output. Previously, the counterexample generator could only show the values of integer and Boolean variables, constructor names used to create a datatype, or the length of a sequence. The differences between our new implementation and past versions are mostly due to the support we added for new types and collections (e.g., chars, bit vectors, maps). However, we also had to revamp and bring up-to-date some of the previously implemented features that have since ceased to function as intended. For instance, Krucker and Schaden [9]

show that they could once extract the values of object’s fields, but this functionality had not been maintained and it stopped working properly. We speculate that the lack of automated testing likely contributed to the failure to adapt the counterexample extraction to the rapidly evolving Dafny infrastructure. To ensure maintainability, we have developed an extensive test suite as part of this work. The test suite contains 54 tests covering all supported types and collections, and is executed as part of the continuous integration process of Dafny.

To benefit from the counterexample extraction feature while working in Visual Studio Code IDE, the user needs only to install the Dafny plugin.² In addition to visualizing counterexamples in the VS Code plugin, the counterexample extraction tool provides a public API and can be imported as a dependency by any C# project. Finally, we made our accompanying artifact publicly available to improve the reproducibility of our contributions [4].

4 Conclusions and Future Work

This paper presents the new, improved version of Dafny’s counterexample extraction tool, which now extracts values of all variables of basic or user-defined types as well as variables representing algebraic datatypes, arrays, sequences, sets, and maps. We integrated the tool into the Dafny plugin for Visual Studio Code, and released it with Dafny v3.3.0. The tool has already been used by Dafny developers to assist them during the proof debugging process.

Note that a counterexample reported by the Dafny verifier might occasionally be a spurious one. This is a well-known problem that users of these verifiers struggle with. It is typically due to the incompleteness of the underlying SMT solver, for example, in the presence of quantifiers. A possible solution to identifying spurious counterexamples is to generate a concrete test case from the counterexample, execute the program concretely using the test case, and observe whether the concrete execution violates the same property [2,14]. The counterexample extraction tool presented in this paper, with its ability to extract the type and concrete value of any variable, can be used for test case generation as well. As future work, we plan to build on this functionality and implement extensions for identifying spurious counterexamples as well as for automatic unit test generation.

Acknowledgements We thank Christoph Amrein, Rustan Leino, Bryan Parno, Shaz Quadeer, Robin Salkeld, and Remy Wilems for reviewing our pull requests to Boogie and Dafny, as well as for their feedback that helped us to improve the early versions of the tool. We also thank Sean McLaughlin, Matthias Schlaipfer, and Jenny Xiang for their insights into the usability of the tool in practice, and to Jeff Foster for reviewing the initial drafts of the paper.

² Developed by the Correctness Lab at OST Eastern Switzerland University of Applied Sciences [9,8].

References

1. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: International Symposium on Formal Methods for Components and Objects. pp. 364–387 (2005). https://doi.org/10.1007/11804192_17
2. Becker, B.F.H., Lourenço, C.B., Marché, C.: Explaining counterexamples with giant-step assertion checking. In: Workshop on Formal Integrated Development Environment. EPTCS, vol. 338, pp. 82–88 (2021). <https://doi.org/10.4204/EPTCS.338.10>
3. Boogie, <https://github.com/boogie-org/boogie>
4. Chakarov, A., Fedchin, A., Rakamarić, Z., Rungta, N.: Better counterexamples for Dafny artifact (2021). <https://doi.org/10.5281/zenodo.5571033>
5. Cook, B.: Formal reasoning about the security of Amazon web services. In: International Conference on Computer Aided Verification. pp. 38–47 (2018). https://doi.org/10.1007/978-3-319-96145-3_3
6. Dafny, <https://github.com/dafny-lang/dafny>
7. Daillier, S., Hauxar, D., Marché, C., Moy, Y.: Instrumenting a weakest precondition calculus for counterexample generation. *Journal of Logical and Algebraic Methods in Programming* **99**, 97–113 (2018). <https://doi.org/10.1016/j.jlamp.2018.05.003>
8. Hess, M., Kistler, T.: Dafny Language Server Redesign. Term project, HSR Hochschule für Technik Rapperswil (2019)
9. Krucker, R., Schaden, M.: Visual Studio Code Integration for the Dafny Language and Program Verifier. Bachelor’s thesis, HSR Hochschule für Technik Rapperswil (2017)
10. Le Goues, C., Leino, K.R.M., Moskal, M.: The Boogie verification debugger (tool paper). In: International Conference on Software Engineering and Formal Methods. pp. 407–414 (2011). https://doi.org/10.1007/978-3-642-24690-6_28
11. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: International Conference on Logic for Programming Artificial Intelligence and Reasoning. pp. 348–370 (2010). https://doi.org/10.1007/978-3-642-17511-4_20
12. Leino, K.R.M.: Accessible software verification with Dafny. *IEEE Software* **34**(6), 94–97 (2017). <https://doi.org/10.1109/MS.2017.4121212>
13. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340 (2008). https://doi.org/10.1007/978-3-540-78800-3_24
14. Nilizadeh, A., Calvo, M., Leavens, G.T., Le, X.B.D.: More reliable test suites for dynamic APR by using counterexamples. In: IEEE International Symposium on Software Reliability Engineering (2021), to appear
15. Z3, <https://github.com/Z3Prover/z3>

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.

