

Kent Academic Repository

Full text document (pdf)

Citation for published version

Christakis, Maria and Leino, K. Rustan M. and Müller, Peter and Wüstholtz, Valentin (2016) Integrated Environment for Diagnosing Verification Errors. In: Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science (LNCS), 9636. Springer pp. 424-441. ISBN 9783662496732.

DOI

http://doi.org/10.1007/978-3-662-49674-9_25

Link to record in KAR

<http://kar.kent.ac.uk/58940/>

Document Version

Author's Accepted Manuscript

Copyright & reuse

Content in the Kent Academic Repository is made available for research purposes. Unless otherwise stated all content is protected by copyright and in the absence of an open licence (eg Creative Commons), permissions for further reuse of content should be sought from the publisher, author or other copyright holder.

Versions of research

The version in the Kent Academic Repository may differ from the final published version.

Users are advised to check <http://kar.kent.ac.uk> for the status of the paper. **Users should always cite the published version of record.**

Enquiries

For any further enquiries regarding the licence status of this document, please contact:

researchsupport@kent.ac.uk

If you believe this document infringes copyright then please contact the KAR admin team with the take-down information provided at <http://kar.kent.ac.uk/contact.html>

Integrated Environment for Diagnosing Verification Errors

Maria Christakis¹, K. Rustan M. Leino¹,
Peter Müller², and Valentin Wüstholtz³

¹ Microsoft Research, Redmond, USA
{mchri,leino}@microsoft.com

² Department of Computer Science, ETH Zurich, Switzerland
peter.mueller@inf.ethz.ch

³ The University of Texas at Austin, USA
valentin@cs.utexas.edu

Abstract. A failed attempt to verify a program’s correctness can result in reports of genuine errors, spurious warnings, and timeouts. The main challenge in debugging a verification failure is to determine whether the complaint is genuine or spurious, and to obtain enough information about the failed verification attempt to debug the error. To help a user with this task, this paper presents an extension of the Dafny IDE that seamlessly integrates the Dafny verifier, a dynamic symbolic execution engine, a verification debugger, and a technique for diagnosing timeouts. The paper also reports on experiments that measure the utility of the combined use of these complementary tools.

1 Introduction

Software developers today get more assistance than ever before from analyses running in their integrated development environment (IDE). These analyses scrutinize the code in shallow or deep ways and then display information, issue warnings, make suggestions, or rewrite the code. Examples include code formatting, intelligent code completion, semantic variable renaming, cyclomatic code complexity analysis, unit test generation, bounds checking, race detection, worst-case execution time analysis, termination checking, and functional-correctness verification. As the level of sophistication of an analysis goes up, so does the level of understanding required for a programmer to diagnose the output of the analysis and determine how to take corrective action.

In this paper, we consider the problem of diagnosing the output of a program verifier of the kind where the underlying reasoning engine, typically an SMT solver, runs without user interaction. Examples of such verifiers are Spec# [3], Frama-C [15], SPARK 2014 (for Ada) [20], AutoProof (for Eiffel) [40], and Dafny [29]. In particular, we consider three kinds of output:

- 1) **Timeouts:** While SMT solvers are generally both useful and fast in practice, they occasionally time out. When they do, the information available

may not be the same as in cases where they output counterexamples. Moreover, a timeout can mask other error messages because it abruptly ends the counterexample search.

- 2) **Spurious warnings:** The logical conditions that a program verifier needs to resolve are in general undecidable, so it would be too much to expect that every error message produced by a verifier indicates a real error. However, in practice, most warnings that are not indicative of errors in the executable code are not caused by undecidability but by the lack of strong enough auxiliary specifications (such as loop invariants) in the program.
- 3) **Genuine errors:** Sometimes when the program verifier reports a real error, the programmer’s response can be one of disbelief. Erroneously—perhaps by habit—assuming the error is caused by an infelicity in the verifier, the programmer spends time trying to coax the verifier into giving a different output, only to miss the blatant error that the verifier detected. Such an error can occur in either the executable code or in the program’s specifications.

The main challenge in debugging verification errors is to determine which of these cases applies and to obtain enough information about the failed verification attempt to debug the error. A single tool may not support the best kind of diagnosing for each output.

In this paper, we contribute comprehensive tool support in a single verification environment. The combination of our tools covers all steps of the typical diagnosis procedures for verification.

We use as our setting the Dafny programming language, verifier, and IDE. In addition to standard (sequential) imperative and functional constructs, the language includes constructs for specifications (aka *contracts*), auxiliary specifications, and proof authoring. The verifier uses these specifications to perform *modular* verification. For example, it reasons about a method call solely in terms of the callee method’s specification and about a loop solely in terms of the loop invariant.

Dafny has always had a program verifier. In this paper, we extend the Dafny IDE with a novel dynamic test generator (Delfy), the Boogie Verification Debugger (BVD) [28], and a new mode for diagnosing timeouts⁴. Using step-by-step recipes, we show how our seamless integration of these tools helps diagnose verification problems. Our paper also gives an experimental evaluation of our tool integration and its effect on diagnosing verification errors. Both Dafny and the IDE extension are available at <http://dafny.codeplex.com> (Delfy is currently not included).

In Sect. 2, we illustrate the use of the combination of our tools on small representative examples. We then describe in more detail the facilities that our integrated diagnosis environment offers: hover text in Sect. 3, Delfy in Sect. 4, BVD integration in Sect. 5, and timeout diagnosis in Sect. 6. We give our ex-

⁴ A preliminary integration of the verifier and BVD into the Dafny IDE has previously been described in an informal workshop paper [31]. The full integration of the tools is new here, as are the test generator and the timeout-diagnosis tool.

perimental evaluation in Sect. 7. The final sections of the paper discuss related work and conclude.

2 Systematic Diagnosis of Verification Failures

In this section, we present systematic approaches to diagnosing the two forms of verification failures: (1) verification errors, which may be spurious warnings and genuine errors, as well as (2) timeouts. For each approach, we describe the tool support we provide and illustrate the approach on a small example program. Details are described in the subsequent sections.

2.1 Diagnosis of Verification Errors

The main challenge in debugging a verification error is to determine if the complaint is spurious or genuine, and to obtain enough information about the failed verification attempt to debug the error. For genuine errors, this includes determining whether to fix the program or the specification. For spurious errors, it includes determining if more auxiliary specifications are required or if the error is caused by an incompleteness of the verifier (which happens in particular when the SMT solver cannot discharge a verification condition even though it holds).

Using the example in Fig. 1, we illustrate how we support this debugging process. The condition stated by the assert-statement in this program does not hold along all executions of the program, because `Max` erroneously computes the *minimum* of its arguments. But even if `Max` had been implemented correctly, the verifier would report a (spurious) error because the postcondition of `Max` is too weak to (modularly) prove the assertion.

Diagnosing verification errors typically proceeds in the following three steps.

Step 1: Fixing simple errors. For certain simple verification errors (such as omitting a precondition of the method being verified), the error message of

```

method Main(a: int) {
  var aSq := a * a;
  var r := Max(a, aSq);
  assert r = aSq; // verification fails
}

method Max(a: int, b: int) returns (max: int)
  ensures max = a ∨ max = b
{
  if a ≤ b { max := a; }
  else    { max := b; }
}

```

Fig. 1: A Dafny example that asserts that an integer is never bigger than its square. The assertion does not hold because method `Max` returns the minimum of its arguments; it fails to verify because the postcondition of `Max` is too weak to prove it. Note that integers in Dafny are unbounded and that calls are verified modularly, based solely on the callee’s specification.

the verifier provides enough information to diagnose and debug the error. To provide easy, demand-driven access to error messages, the Dafny IDE presents them in tool tags when hovering over the error location, which is indicated by red squiggly lines. The hover text also shows inferred specifications (such as termination metrics) and parts of the counterexample provided by the SMT solver (as we shall see later in Fig. 5). In our example, the error message is simply “assertion violation”, which does not point us to the source of the problem.

Step 2: Determining whether errors are spurious. Debugging genuine verification errors is fundamentally different from debugging spurious errors. For the former, one needs to determine which aspects of the program or specification are incorrect and fix them. For the latter, one needs to determine how to convince the verifier that the program is actually correct.

A common approach to determine if an error is spurious is to create an executable test from the counterexample given by the SMT solver [4, 16]. However, this approach has two major limitations. First, the counterexample reflects the (modular) verification semantics of a method, where calls are encoded via the callee’s specification, loops are encoded via loop invariants, etc. By the soundness of verification, any error in the execution semantics is also an error in the verification semantics, but not necessarily vice versa. Therefore, it is possible that a test case derived from the SMT solver’s counterexample does not reveal an error even though the program fails for other inputs. A programmer might then conclude incorrectly that the verification error is spurious. Second, SMT solvers sometimes produce invalid counterexamples, that is, valuations that do not actually falsify the verification condition. This may be due to an incompleteness in the SMT solver (e.g., when reasoning about non-linear arithmetic) [33]. Executing such counterexamples does not lead to meaningful conclusions. In fact, it may not even be possible to generate a test case from such a counterexample.

To avoid these problems, we do not execute counterexamples and instead apply dynamic symbolic execution (DSE) [8, 24] (also called concolic testing [35]) to generate test cases for the method that contains the verification error. We have equipped the Dafny IDE with Delfy, a DSE tool that instruments the executable code with runtime checks for assertions and then uses dynamic symbolic execution to systematically explore all paths through a Dafny method up to a given bound. DSE mitigates the limitations of counterexample execution as follows. First, it is based on the (non-modular) execution semantics, not on the verification semantics and, thus, attempts to find inputs for which the *execution* of a method leads to an assertion violation. Second, when some constraints in a proof obligation cause the SMT solver to produce an invalid counterexample during verification, the same problem may occur during DSE. However, DSE has the option of replacing symbolic inputs by concrete values, thereby simplifying the formula, which increases the chance of obtaining a valid counterexample.

Running DSE can have three different outcomes: (1) It produces a test case that leads to an assertion violation. In this case, we can conclude that the error is *definitely not spurious*. One can now use a conventional debugger to explore the execution of the test case and determine how to fix the error. (2) It is

able to verify the method. This is possible when the method can be tested without exceeding the bounds of DSE (for instance, the method contains no input-dependent loops) and when the SMT solver is able to produce concrete inputs for each constraint [11]. In this case, the error is *definitely spurious*. It is now possible to communicate this verification result to the verifier. (3) If DSE neither verifies nor falsifies the method, our best guess is that the error is *spurious*, and we proceed to step 3 below.

Running Delfy on method `Main` from our example reproduces the error by generating a test case where $a \leq a*a$ (necessarily, since this is a mathematical fact, and thus the then-branch of the conditional in method `Max` is executed) and $a \neq a*a$ (such that the assertion is violated), for instance, $a = 2$. Stepping through this test case in the debugger immediately reveals that method `Max` is incorrect. After fixing the error, verification still fails. Running Delfy again verifies method `Main`. We could now communicate this result to the verifier or—as we describe next—we could try to determine what additional facts are needed by the verifier to prove the method.

Step 3: Finding the cause of spurious errors. When Delfy cannot reproduce a verification error, it is necessary to explore the verification semantics, which is reflected in the counterexample provided by the SMT solver. To do so in the Dafny IDE, a user can select a verification error by clicking on the red button next to the assertion (see Fig. 5). The IDE now highlights the program points along the trace leading to the error using blue buttons. By clicking on one of them, a user can bring up BVD and inspect the state at this program point as provided by the counterexample.

In our example, once method `Max` is fixed, the verification debugger shows for the program point after the call to `Max` that a is 2, aSq is 4, and r is 2. Since running Delfy did not reveal any error, we hypothesize that `Max` correctly computes the maximum of its arguments, and conclude that the counterexample values indicate that the verifier has insufficient information about the result of `Max`. We can fix this by strengthening its postcondition, and verification succeeds.

2.2 Diagnosis of Timeouts

The use of undecidable theories, especially quantifiers, in verification conditions can lead to a very large or even infinite search space for the SMT solver, for instance, when the verification conditions contain matching loops [19]. Therefore, Dafny and other automatic verifiers bound the time spent by the SMT solver, and report a verification failure when a timeout occurs [22]. However, if this happens, it is often unclear which fragments of a large verification condition cause the SMT solver to wander off. Moreover, because of the heuristics used in the SMT solver to instantiate quantifiers, timeouts are often caused by the interaction of different, often seemingly unrelated, terms in the program or its specification.

Verification of the example in Fig. 2 fails with a timeout. While trying to prove the last assertion in method `Test`, the SMT solver instantiates the universal quantifier in the postcondition of `FacUpTo` (and in the axiomatization of the sequence data type) indefinitely. For the verification to succeed, one needs to

```

method FacUpTo(n: int) returns (f: seq<int>)
  requires 1 ≤ n
  ensures |f| = n ∧ f[0] = 1
  ensures ∀ i • 1 ≤ i < |f| ⇒ f[i] = f[i - 1] * i
  Ⓜ{..}

method Test(n: int)
  requires 1 ≤ n
  {
    var f4 := FacUpTo(4);  assert f4[3] = 6;
    var f15 := FacUpTo(15); assert f15[14] ≠ 0;

    var fn := FacUpTo(n);
    assert fn[n - 1] ≠ 0; // verification times out
  }

```

Fig. 2: A Dafny example that computes the factorial of the first n natural numbers and asserts that they are positive. The proof requires generalization and induction, which Dafny does not perform automatically. Instead, the SMT solver keeps instantiating the universal quantifier in the post-condition of the call `FacUpTo(n)`, and verification times out even though, in principle, many other assertions could be proved.

instruct Dafny to prove by induction that all elements of sequence `fn` are non-zero, for instance, by adding the following assertion after the final call in Fig. 2:

```
assert ∀ i {induction} • 0 ≤ i < |fn| ⇒ fn[i] ≠ 0;
```

Diagnosing such timeouts typically proceeds in the following two steps.

Step 1: Determining whether the program satisfies its specification. Like for verification errors, it is useful to run the test case generator Delfy on the method whose verification times out. Note that the common approach of generating test cases from counterexamples is not applicable here since SMT solvers usually generate an incomplete counterexample or none at all in case of a timeout. In contrast, since Delfy relies only on the program and its specification, it can be used to diagnose timeouts. If Delfy generates a failing test, the program or its specification should be fixed before diagnosing the timeout. If Delfy manages to verify the method, Dafny can be notified such that it is no longer essential to debug the timeout. Delfy might succeed on examples that time out in the verifier because it uses a different axiomatization of data types such as sets and sequences. Moreover, Delfy’s SMT queries are constraints that describe a single path through a method, whereas Dafny’s verification conditions reflect all paths. Therefore, Delfy’s queries might provide fewer terms that are used by the SMT solver to instantiate quantifiers.

In the example from Fig. 2, Delfy neither generates a failing test nor manages to verify method `Test`; this is due to the input-dependent loop in the body (not shown) of method `FacUpTo`, which is called. Thus, we proceed to the second step.

Step 2: Narrowing down the cause of the timeout. We have developed a dedicated diagnostic mode for Dafny, which splits up the verification condition into smaller fragments and invokes the SMT solver multiple times to narrow down which assertions may cause the timeout. For each invocation, this algorithm tries to prove some of the fragments, and ignores the rest. If the SMT solver fails, an error is reported. If it succeeds, the algorithm recurs and attempts to verify the fragments previously ignored. If no such fragments exist, verification succeeds. Finally, if the SMT solver still times out, the algorithm recurs on fewer fragments or, if there is just a single fragment, “blames” that fragment for the timeout.

In our example, the timeout diagnosis determines that out of the nine assertions in method `Test` (three for precondition checks, three for bounds checks, and three for assert-statements), eight verify and only the last one times out. This clearly indicates that the user should provide more hints to help the verifier in proving this assertion.

The above recipes allow a programmer to systematically diagnose and debug all three kinds of verification failures. Our recipes are supported by a novel integration of the following components into the Dafny IDE: (1) an advanced hover text mechanism, (2) the Delfy test case generator, (3) the Boogie Verification Debugger, and (4) a technique for diagnosing timeouts. We describe these components in detail in the following sections.

3 Hover Text

Verifiers typically accumulate a lot of information, including error messages, inferred specifications (such as termination metrics), or verification counterexamples. However, most often, the user is interested only in a small fraction of this information, and specifically, in whatever helps to diagnose verification errors.

The hover text mechanism that we have integrated in the Dafny IDE addresses this need without overwhelming the user with too much information. Our mechanism uses the parser, type checker, and verifier to collect warnings, inferred specifications, and other information, which it attaches to the relevant parts of the Dafny abstract syntax tree. As a result, the IDE displays only the most critical information at all times (that is, squiggly lines for verification errors), and the user may access all other information on demand, by hovering over the relevant parts of the program text. For instance, a warning emitted by the verifier is shown when hovering over the corresponding squiggly line, and the values of the variables in a verification counterexample are shown when hovering over the variable usages (see Fig. 5).

4 Delfy, the Test Case Generator

In this section, we present Delfy, a dynamic test generation tool for Dafny. In addition to handling advanced constructs of the language, Delfy is designed to exchange information with Dafny about the verification status of all assertions via annotations in the code [12]. Consequently, Dafny does not need to check assertions that have already been proven correct by Delfy and vice versa.

4.1 Dynamic Symbolic Execution for Dafny

Delfy implements dynamic symbolic execution, in which the concrete and symbolic executions of a method under test happen simultaneously. Given a Dafny method under test, Delfy compiles the code into .NET bytecode and runs the compiled method. The compiled code includes call-backs that trigger the symbolic execution. All constraints are solved with Z3 [18].

Delfy introduces runtime checks for Dafny specifications, including loop invariants, termination metrics, pre- and postconditions, assumptions, assertions, and frame specifications, which serve as test oracles.

Delfy has support for features of Dafny that are typically not found in mainstream programming languages, for instance, non-deterministic assignments, non-deterministic if-statements, and non-deterministic while-statements. For each non-deterministic value, the symbolic execution in Delfy introduces a fresh symbolic variable, as if they were inputs to the method under test. Consequently, the symbolic execution collects constraints on such variables and generates inputs for them, which guide execution toward all those unexplored paths.

Dafny also supports uninterpreted functions and assign-such-that-statements, which assign a value to a variable such that a condition holds. Delfy handles these by introducing a fresh symbolic variable for the return value of an uninterpreted function or the assigned variable of an assign-such-that-statement. This symbolic variable is constrained by a condition of the form `ASSUME(c)`, saying that the variable must satisfy the function specifications or the such-that-condition in each test case.

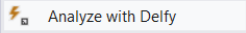

When the programmer provides a loop invariant for an input-dependent loop, Delfy can either impose a bound on the number of explored loop iterations or treat the invariant as a summary for the loop [10]. In the latter case, the symbolic execution of the loop body is turned off, and instead, the provided loop invariant serves as a symbolic description of the loop body. (Note that we abuse the term “summary” to express that reasoning about many loop iterations happens in one shot, although we do not refer to a logic formula of loop pre- and postconditions, as is typically the case in compositional symbolic execution [23, 1].) Summarization of an input-dependent loop might lead to spurious warnings when the loop invariant is too weak, in which case Delfy resembles the verifier. However, when the loop invariant is precise, this technique can be very useful in diagnosing verification errors and timeouts as it helps the exploration in covering the code after the loop.

A consequence of this approach for summarizing input-dependent loops is that the body of such a loop might not be thoroughly exercised since it is only executed concretely, and not symbolically; therefore, paths and bugs might be missed. To address this, Delfy supports a mode for thoroughly checking if an invariant is maintained by all iterations of an input-dependent loop [10].

4.2 Delfy in the Dafny IDE

We now present how we have integrated Delfy in the Dafny IDE. Fig. 3 shows the error emitted by the verifier (denoted by the red button) for the assertion

```

method Main(a: int) {
  var a := a * a;
  var r 
  assert r  == aSq; // verification fails
}

```

Fig. 3: A smart tag allowing the user to invoke Delfy on a method under test, and a verification error emitted by the verifier (denoted by the red button in the assertion).

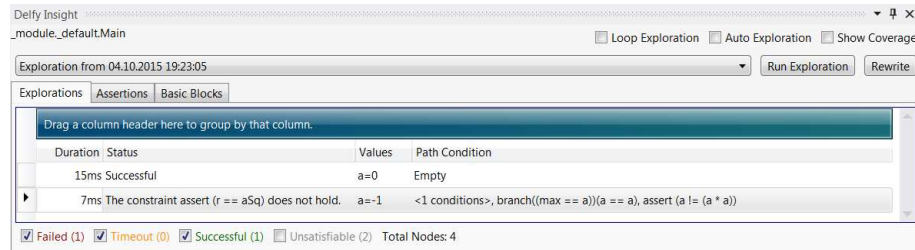
in method `Main` from Fig. 1. Delfy is run through a smart tag, shown in Fig. 3. Fig. 4 shows how the test cases generated by Delfy are displayed for method `Main` from Fig. 1.

The main characteristics of this IDE integration are as follows.

Color coding of assertions. To give users a sense of where they should focus their manual diagnosis, the IDE uses colors for assertions. A green color shows that the assertion has been proven, either by Dafny or Delfy. A red color denotes that an assertion definitely does not hold, that is, Dafny has emitted a verification error, and Delfy has generated a test case that fails due to this assertion. An orange color indicates that the assertion requires the attention of the user because Dafny has emitted a verification error, and Delfy has neither verified nor falsified it. One could further refine this color scheme by reflecting how thoroughly Delfy covered an orange assertion [10].

Selective test generation. Delfy allows the user to select an assertion that has not been verified by Dafny, and explore only those paths that reach this assertion. If a programmer selects a red button in a method under test and runs Delfy, then only those test cases that exercise the corresponding unverified assertion are generated, regardless of whether there are other unverified assertions in the method under test. We determine which test cases to generate using a technique based on static symbolic execution [10].

Debugging failing tests. Delfy also makes it possible to debug the generated test cases. A smart tag allows users to run a failing test case in the .NET debugger, such that they can step through the execution and observe the values of variables.



Duration	Status	Values	Path Condition
15ms	Successful	a=0	Empty
7ms	The constraint assert (r == aSq) does not hold.	a=-1	<1 conditions>, branch((max == a))(a == a), assert (a != (a * a))

Failed (1)
 Timeout (0)
 Successful (1)
 Unsatisfiable (2)
 Total Nodes: 4

Fig. 4: Delfy displays the generated tests. The user can choose to inspect all generated tests, or categorize them based on their outcome.

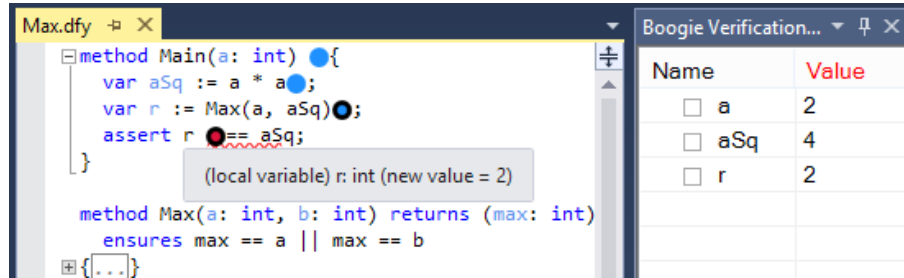


Fig. 5: Inspecting values from the counterexample for the error in method `Main` of Fig. 1. The hover text shows the value of variable `r` and the BVD window on the right shows the values of all variables.

5 Integration of the Verification Debugger

Counterexamples, which are provided by the verifier and the underlying solver, often include valuable information for diagnosing verification errors. Since these counterexamples reflect the verification semantics (for instance, by reasoning about method calls modularly), this holds in particular for intricate verification errors that cannot be reproduced by Delfy. (Recall that Delfy is based on the non-modular execution semantics.) BVD makes the verification counterexamples accessible through the Dafny IDE, which allows users to inspect the values of variables (including heap locations), much like in a conventional debugger. However, unlike in most runtime debuggers, a user can inspect the counterexample at any relevant point during the execution.

BVD is invoked by clicking on the red button that is associated with each verification error. Now, several blue buttons appear along the trace that leads to the error (see Fig. 5). Clicking on any of them shows the counterexample state at that program point. For instance, a user may diagnose a verification error by starting at the failing assertion and gradually moving backward in the program to understand how the failing state was reached.

6 Timeout Diagnosis

As discussed in Sect. 2.2, users occasionally encounter timeouts when verifying non-trivial programs. Timeouts often indicate that the verifier is unable to derive a certain fact on its own, and requires hints from the user. To detect timeouts quickly and to ensure a responsive user interaction, the Dafny IDE defaults to a time limit of ten seconds per method or function.

If this time is not enough, the user can increase the limit or use our technique for diagnosing timeouts. In the latter case, we instruct the verifier to produce slightly different verification conditions, which can be decomposed more easily and on demand. This makes it possible to split up the verification conditions and, thereby, identify those assertions that are responsible for the timeout.

Conceptually, our alternative verification conditions insert an assumption $F_k \implies A_k$ before every assertion A_k , where a F_k is an undefined boolean function. Initially nothing is known about these functions. That is, the solver needs

```

procedure diagnose(VC, U, D, T) {
  if (|U| = 0) {
    if (0 < |T|) {
      report the timed-out assertions in T;
      return Timeout;
    }
    return Verified;
  }
  choose S, such that  $S \subseteq U \wedge |S| = \max(|U| / D, 1)$ ;
  var R := check_some(VC, S, TL);
  if (R = Error) {
    return R;
  } else if (R = Verified) {
    return diagnose(VC, U \ S, 1, T);
  } else {
    if ( $2 \leq (|U| / D)$ ) {
      return diagnose(VC, U, 2 * D, T);
    } else {
      return diagnose(VC, U \ S, 1, T  $\cup$  S);
    }
  }
}

```

Fig. 6: Algorithm for diagnosing timeouts.

to consider the case that all F_k functions yield false and, thus, this instrumentation does not affect verifiability of the verification condition. However, once a timeout occurs, we can define some of the F_k functions to yield true, thus, *temporarily* disabling assertions and simplifying the verification task.

Fig. 6 shows our algorithm for decomposing the verification task once there has been a timeout. Procedure `diagnose` takes four arguments: (1) the current verification condition `VC`, (2) the set of unverified assertions `U` (initially contains all assertions in the verification condition), (3) the integer `D` (for denominator) to determine what fraction of these assertions to check next (initially set to 2), and (4) the set of timed-out assertions `T` (initially empty).

If set `U` is empty, we are done. We return `Verified` if set `T` of timed-out assertions is empty, and `Timeout` otherwise. If set `U` is non-empty, we choose a subset `S` of the unverified assertions and check only these assertions for a fixed time limit `TL` (set by default to 10% of the time limit for the entire method or function). If we find a failing assertion, we terminate immediately. If the check successfully verifies the assertions in `S`, we recursively diagnose the timeout among the remaining assertions. Otherwise, we try to check a smaller set of assertions by invoking procedure `diagnose` with $2 * D$. If doubling `D` is not possible without exceeding the cardinality of `U`, we have found assertions to blame for the timeout, collect them in `T`, and proceed to also check the remaining assertions. If the algorithm reports any blamed assertions, it is reported that each of them timed out individually, given time limit `TL`. This shows exactly which assertions the user should focus on in order to prevent the timeout.

The procedure `check_some` checks the verification condition after temporarily disabling some assertions. To do so efficiently, it makes use of scopes in the solver that push and later pop constraints about the F_k functions for assertions that are not in set `S`.

Challenge	Error ID	Spurious?	Extension			
			Hover text (w/o CEX)	Hover text (only CEX)	Delfy	BVD
SUMMAX	1	no	✓	✓	–	✓
	2	no	✓	✓	✓	✓
	3	yes	✓	✓	✓	✓
	4	yes	✗	✓	✓	✓
MAXARRAY	5	no	✓	✓	–	✓
	6	no	✓	✓	✓	✓
	7	yes	✓	✓	✓	✓
	8	yes	✓	✓	✓	✓
	9	yes	✓	✗	✓	✓
	10	yes	✓	✓	✓	✓
	11	yes	✗	✗	✓	✓
BINARYSEARCH	12	no	✓	✓	–	✓
	13	no	✓	✓	✓	✓
	14	yes	✓	✓	✓	✓
	15	no	✓	✓	✓	✓
	16	no	✓	✓	–	✓
	17	no	✓	✓	✓	✓
	18	no	✓	✗	✓	✓
	19	no	✓	✓	–	✓
	20	yes	✗	✗	✓	✓

Tab. 1: Errors diagnosed while solving three verification challenges.

7 Experimental Evaluation

In this section, we evaluate our extensions of the Dafny IDE on diagnosing both verification errors and timeouts.

7.1 Verification Errors

To demonstrate that even simple programming tasks exhibit different forms of verification errors, we have evaluated our extensions on Dafny solutions we developed to three challenges posed in verification competitions and benchmarks. We used the Dafny IDE to diagnose each verification error we encountered during the three verification sessions, and report the results in Tab. 1.

Challenge SUMMAX is taken from verification competition VSComp-2010 [27]. It consists in computing the *sum* and *max* of the elements in an array and proving that $sum \leq N * max$, where N is the length of the array. Challenge MAXARRAY is taken from verification competition COST-2011 [6]. Given a non-empty integer array, MAXARRAY requires that we verify that the index returned by a given method points to an element maximal in the array. Challenge BINARYSEARCH is taken from a set of verification benchmarks [41], and consists in verifying an implementation of binary search over an array. All versions of our solutions to these challenges are numbered by a verification-error identifier, which is shown in the second column of the table, and can be provided upon request. The third column indicates that roughly half of the verification errors are spurious, which is not uncommon.

To diagnose the errors, we used hover text information about error messages and inferred specifications (fourth column), hover text information about verification counterexamples (fifth column), Delfy (sixth), and BVD (seventh). As

described earlier, each of these extensions may provide complementary insights to the user about the cause of verification errors. In the table, we indicate helpful insights (✓) as well as information that did not help in the diagnosis of a verification error (✗). However, note that such insights are not necessarily sufficient for diagnosing the error—multiple steps may be needed and the use of more than one of our extensions; also, different users may find some feedback more insightful than others. For instance, the counterexample information (through the hover text or the verification debugger) is perhaps more suitable for experienced users. Consequently, in particular for spurious errors, there is usually no definite answer about which extension pinpointed the source of an error.

Note that we have created a separate column for the counterexample information that is available in the hover text to highlight the difference with BVD. As shown in the table, the hover text is sufficient to diagnose most verification errors. BVD only becomes essential when inspecting values within data structures, such as arrays, which are not shown in the hover text. Fixing spurious errors without counterexample information would require significant *mental effort* and *time* from users since they would often need to resort to trial-and-error to identify which information the verifier is missing. In principle, Delfy could provide help with such cases. However, since all of our programs contained input-dependent loops, Delfy was not able to show that an error is *definitely* spurious.

In a few cases (indicated by a ‘–’ in the table), Delfy was not applicable. This was the case when the cause of a verification error was a specification that Dafny guessed heuristically, such as a termination metric. Even though, at the moment, Delfy does not support runtime checks for such guessed specifications, it *automatically and reliably* detected all other genuine errors. Without Delfy, this would have required *manual* effort from the user, for instance, to inspect counterexamples. In other words, no extension of the Dafny IDE is absolutely indispensable, but each extension can significantly reduce the user effort for diagnosing errors.

We also found situations where the hover text about error messages and inferred specifications (fourth column of the table) provided limited support. In particular, there is no indication of how much progress a user makes in fixing a verification error. For instance, they might add one of two loop invariants that are necessary for proving a failing assertion, but the error message remains unchanged. They are, therefore, not confident that the change is a step in the right direction by only reading the hover text. In contrast, our other extensions provide better support in such cases; for instance, in this example, the counterexample state after the loop would now be different due to the additional invariant.

7.2 Timeouts

We have evaluated our technique for diagnosing timeouts by running it on 39 programs taken from real verification sessions, which were recorded with the Dafny IDE [32] and can be provided upon request. We compare two configurations that only differ by parameter TL from Fig. 6: (1) Low (10% of the time limit per method/function), and (2) High (20% of the time limit per method/function).

	TIME LIMIT	
	Low	High
TimeOut (IN %)	57.89	50.00
Error (IN %)	17.11	20.69
Verified (IN %)	25.00	29.31
AVERAGE NUMBER OF SOLVER QUERIES	65.67	51.00
AVERAGE TIME (RELATIVE TO TIME LIMIT PER METHOD/FUNCTION)	6.24	9.25
AVERAGE NUMBER OF ASSERTIONS TO BLAME	2.67 (0.15%)	1.84 (0.11%)

Tab. 2: Comparison between two configurations for diagnosing timeouts.

Tab. 2 demonstrates the different trade-offs. While configuration Low is significantly faster by using a larger number of short solver queries, it results in timeouts more often and is able to narrow down the set of timed-out assertions less effectively. For verification conditions that still result in a timeout, configuration Low reports on average 0.15% (at most 10 assertions) of all assertions in that method/function as responsible. For configuration High, these numbers are significantly lower (0.11% on average, at most 4 assertions).

Independently, both configurations are able to prevent a large number of timeouts by decomposing the verification tasks (as shown by the first three rows in Tab. 2). For instance, with configuration High, the algorithm from Fig. 6 returns the result `Verified` or `Error` for 50% of the timed-out verification conditions. Therefore, for these verification conditions, none of the assertions required more time than the limit. This suggests that the user might be able to prevent the timeout by increasing the time limit for the corresponding method or function.

8 Related Work

Verification IDEs. Several verification tools are integrated into development environments and show verification errors either continuously or at the touch of a button, e.g., [3, 15, 13, 20, 26, 14]. Our work goes beyond the integration of a single tool, instead providing in one package a collection of tools with complementary strengths.

The Isabelle environment for mathematical formulas integrates both interactive proof assistance and automatic counterexample search [42, 5].

The Eiffel Verification Environment analyzes programs in two independent ways [39]. Essentially, one way strives to fully verify the program, whereas the other cuts corners in order to provide quick turnaround with understandable error messages. This two-step verification resembles the combination of two of our tools, the Dafny verifier and Delfy.

Dynamic symbolic execution. Dynamic symbolic execution has been implemented in many popular tools over the last decade, e.g., SAGE [25], EXE [9], jCUTE [34], Pex [38], KLEE [7], BitBlaze [37], and Apollo [2]. In contrast to these tools, Delfy targets a verification language for proving functional correctness of programs and, therefore, supports specification constructs and operations that are not found in mainstream programming languages.

Delfy implements dynamic, rather than static, symbolic execution for two important reasons. First, DSE can alleviate the limitations of an underlying SMT solver by replacing complex symbolic conditions in SMT queries with their

concrete values [24]. Second, the dynamic aspect has applications beyond the scope of this paper, in particular for learning specifications [21, 17, 36].

Exploring counterexamples. BVD [28] lets one inspect counterexamples to verification conditions generated by Boogie, VCC [13], and Dafny. Besides integrating BVD into the Dafny IDE, we provide easy access to excerpts from the counterexample through hover text. OpenJML [14] also provides such hover text, but not the full BVD experience.

An alternative to a dedicated counterexample debugger is to generate an executable program that encodes the verification semantics and the counterexample, for instance, by extracting a value for a non-deterministic choice from the counterexample [33]. This approach allows one to use a conventional debugger to explore the counterexamples.

Several tools generate executable tests from counterexamples [4, 16]. In contrast, Delfy lets one explore the program independently of the verification semantics that is reflected in the counterexample.

Timeouts. Unlike Boogie’s existing verification-condition splitting [30], our technique for diagnosing timeouts is not concerned with parallelizing verification tasks. Instead of iteratively creating smaller and smaller program fragments that are fed to the verifier, our technique generates a single verification condition once and uses the SMT solver to decompose it in case of a timeout. Besides this, our technique is able to identify all assertions that time out individually after a given time limit.

9 Concluding Remarks

In this paper, we have enhanced the IDE of the verification-aware language Dafny with a comprehensive set of problem-diagnosing tools, including a new timeout-diagnosis tool and the novel Delfy dynamic test generator. The seamless integration of these tools, alongside the on-demand information that the IDE now provides via hover text, lets a user obtain useful feedback when trying to understand and remedy verification failures. While in this work we have made the sophisticated diagnostic information easily accessible to users, we hope in future work to also see automatic suggestions of remedies.

Acknowledgments. We are grateful to Patrick Emmisberger and Patrick Spetel for their contributions to Delfy.

References

1. S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *TACAS*, volume 4963 of *LNCS*, pages 367–381. Springer, 2008.
2. S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. M. Paradkar, and M. D. Ernst. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *TSE*, 36:474–494, 2010.
3. M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: The Spec# experience. *CACM*, 54:81–91, 2011.
4. D. Beyer, A. J. Chlipala, and R. Majumdar. Generating tests from counterexamples. In *ICSE*, pages 326–335. IEEE Computer Society, 2004.

5. J. C. Blanchette and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In *ITP*, volume 6172 of *LNCS*, pages 131–146. Springer, 2010.
6. T. Bormer, M. Brockschmidt, D. Distefano, G. Ernst, J. Filiâtre, R. Grigore, M. Huisman, V. Klebanov, C. Marché, R. Monahan, W. Mostowski, N. Polikarpova, C. Scheben, G. Schellhorn, B. Tofan, J. Tschannen, and M. Ulbrich. The COST IC0701 verification competition 2011. In *FoVeOOS*, volume 7421 of *LNCS*, pages 3–21. Springer, 2011.
7. C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224. USENIX, 2008.
8. C. Cadar and D. R. Engler. Execution generated test cases: How to make systems code crash itself. In *SPIN*, volume 3639 of *LNCS*, pages 2–23. Springer, 2005.
9. C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *CCS*, pages 322–335. ACM, 2006.
10. M. Christakis. *Narrowing the Gap between Verification and Systematic Testing*. PhD thesis, ETH Zurich, 2015.
11. M. Christakis and P. Godefroid. Proving memory safety of the ANI Windows image parser using compositional exhaustive testing. In *VMCAI*, volume 8931 of *LNCS*, pages 373–392. Springer, 2015.
12. M. Christakis, P. Müller, and V. Wüstholz. Collaborative verification and testing with explicit assumptions. In *FM*, volume 7436 of *LNCS*, pages 132–146. Springer, 2012.
13. E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009.
14. D. R. Cok. OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse. In *Formal-IDE*, volume 149 of *Electronic Proceedings in Theoretical Computer Science*, pages 79–92. Open Publishing Association, 2014.
15. L. Correnson, P. Cuoq, F. Kirchner, V. Prevosto, A. Puccetti, J. Signoles, and B. Yakobowski. *Frama-C User Manual*, 2011. <http://frama-c.com//support.html>.
16. C. Csallner and Y. Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *ICSE*, pages 422–431. ACM, 2005.
17. C. Csallner, N. Tillmann, and Y. Smaragdakis. Dysy: Dynamic symbolic execution for invariant inference. In *ICSE*, pages 281–290. ACM, 2008.
18. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
19. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. *J. ACM*, 52:365–473, 2005.
20. C. Dross, P. Efstathopoulos, D. Lesens, D. Mentré, and Y. Moy. Rail, space, security: Three case studies for SPARK 2014. In *ERTS*, 2014.
21. M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69:35–45, 2007.
22. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, pages 234–245. ACM, 2002.
23. P. Godefroid. Compositional dynamic test generation. In *POPL*, pages 47–54. ACM, 2007.
24. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, pages 213–223. ACM, 2005.

25. P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *NDSS*, pages 151–166. The Internet Society, 2008.
26. B. Jacobs and F. Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, 2008.
27. V. Klebanov, P. Müller, N. Shankar, G. T. Leavens, V. Wüstholtz, E. Alkassar, R. Arthan, D. Bronish, R. Chapman, E. Cohen, M. Hillebrand, B. Jacobs, K. R. M. Leino, R. Monahan, F. Piessens, N. Polikarpova, T. Ridge, J. Smans, S. Tobies, T. Tuerk, M. Ulbrich, and B. Weiß. The 1st Verified Software Competition: Experience report. In *FM*, volume 6664 of *LNCS*, pages 154–168. Springer, 2011.
28. C. Le Goues, K. R. M. Leino, and M. Moskal. The Boogie verification debugger. In *SEFM*, volume 7041 of *LNCS*, pages 407–414. Springer, 2011.
29. K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.
30. K. R. M. Leino, M. Moskal, and W. Schulte. Verification condition splitting. Technical report, Microsoft Research, 2008.
31. K. R. M. Leino and V. Wüstholtz. The Dafny integrated development environment. In *Formal-IDE*, volume 149 of *Electronic Proceedings in Theoretical Computer Science*, pages 3–15. Open Publishing Association, 2014.
32. K. R. M. Leino and V. Wüstholtz. Fine-grained caching of verification results. In *CAV*, volume 9206 of *LNCS*, pages 380–397. Springer, 2015.
33. P. Müller and J. N. Ruskiewicz. Using debuggers to understand failed verification attempts. In *FM*, volume 6664 of *LNCS*, pages 73–87. Springer, 2011.
34. K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *CAV*, volume 4144 of *LNCS*, pages 419–423. Springer, 2006.
35. K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *ESEC*, pages 263–272. ACM, 2005.
36. R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori. A data driven approach for algebraic loop invariants. In *ESOP*, volume 7792 of *LNCS*, pages 574–592. Springer, 2013.
37. D. X. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *ICISS*, volume 5352 of *LNCS*, pages 1–25. Springer, 2008.
38. N. Tillmann and J. de Halleux. Pex—White box test generation for .NET. In *TAP*, volume 4966 of *LNCS*, pages 134–153. Springer, 2008.
39. J. Tschannen, C. A. Furia, M. Nordio, and B. Meyer. Program checking with less hassle. In *VSTTE*, volume 8164 of *LNCS*, pages 149–169. Springer, 2013.
40. J. Tschannen, C. A. Furia, M. Nordio, and N. Polikarpova. AutoProof: Auto-active functional verification of object-oriented programs. In *TACAS*, volume 9035 of *LNCS*, pages 566–580. Springer, 2015.
41. B. W. Weide, M. Sitaraman, H. K. Harton, B. M. Adcock, P. Bucci, D. Bronish, W. D. Heym, J. Kirschenbaum, and D. Frazier. Incremental benchmarks for software verification tools and techniques. In *VSTTE*, volume 5295 of *LNCS*, pages 84–98. Springer, 2008.
42. M. Wenzel. Isabelle/jEdit—a prover IDE within the PIDE framework. In *AISC/-Calculus/DML/MKM/CICM*, volume 7362 of *LNCS*, pages 468–471. Springer, 2012.