Formal Methods in Computer-Aided Design 2025

Static Coverage in Deductive Software Verification

Aaron Tomb
Amazon, Portland, OR, USA
aarotomb@amazon.com

Anjali Joshi
Amazon, Boston, MA, USA
anjalijs@amazon.com

***Abstract*—The results of software verification are only as trustworthy as the provided specification. Errors or incompleteness in the specification can result in unwarranted confidence in the implementation. Previous work, particularly in the realm of model checking, has investigated a notion of *coverage* in verification. Portions of the system that could be replaced arbitrarily without causing verification failure are considered uncovered. We show that the same notion of coverage used in model checking can be applied to deductive software verification, with a reasonable performance penalty when using an implementation based on unsatisfiable cores. This approach provides opportunities for identifying specification gaps by detecting vacuous proofs, unconstrained code, and unnecessary or redundant specifications. We describe an implementation of this approach for the Boogie intermediate verification language, the use of this implementation from the verification-aware programming language Dafny, and experimental results on a large corpus of industry-scale Dafny code.**

I. Introduction

Verification provides a high degree of assurance that software behaves as *specified*. Assurance that the software behaves as *intended* depends on the correctness and completeness of the specification — how well it agrees with the designer's goals. Although techniques to detect specification problems do exist in software verification, most focus entirely on detecting vacuous proofs, and most adopt a somewhat ad-hoc technique (most commonly, attempting to prove a contradiction at various points throughout the program). Prior work in the model checking community formalized a notion of the coverage of a verified program [22], saying that a transition system element is covered if modifying it could possibly cause verification to fail. Further work showed that coverage is dual to vacuity [21], and therefore that coverage analysis can be used to detect vacuity.

In this paper, we extend this prior work and apply it in the context of deductive software verification that supports separate specification and implementation constructs. We describe a technique for static coverage that in addition to detecting vacuous proofs, also identifies redundant specifications and unconstrained portions of the implementation. Each program construct that is not covered is relevant for identifying a specific type of specification gap:

- uncovered goals are proved vacuously,
- uncovered assumptions are unnecessary, and
- uncovered implementation statements are either unconstrained by the specification or unreachable.

We describe the technique for computing static coverage in deductive verification and an implementation of this technique within the Dafny verification-aware programming language [25] via the Boogie intermediate verification language [27]. Our experimental results evaluating this implementation demonstrate modest computational overhead relative to verification, making it practical to employ as integral part of verification. We also show how this technique can be used to generate actionable warnings without false positives.

A. Examples

Before exploring the technique in detail, we first consider concrete examples of these specification problems, such as the Dafny program in Fig. 1.

```
 1  function {:axiom}
 2  Find(xs:seq<int>, x:int,
 3       beg:int, end:int): (i:int)
 4    requires 0 <= beg < |xs| &&
 5             0 <= end < |xs|
 6    ensures beg < i < end
 7    ensures xs[i] == x
 8
 9  method CallFind() {
10    var xs := [42, 43];
11    var i := Find(xs, 47, 0, 1);
12    assert xs[i] == 45;
13  }
```

Fig. 1. Dafny code with a subtly contradictory specification. The postcondition on line 6 becomes unsatisfiable when instantiated with the concrete arguments passed in on line 11.

In this code, the `Find` function is specified but not implemented, as may be the case partway through the process of implementing a verified program, and the `{:axiom}` annotation emphasizes that its contract is assumed to hold. This function is intended to search for a given element, `x`, between indices `beg` and `end` of the sequence `xs`. For this to work, it requires both indices to be within range, and ensures that the returned index is between the provided indices. This seems reasonable at first, but there is a mistake in the postcondition. If `end` is `beg + 1`, no return value `i` is possible. In the `CallFind` method, the passed indices are adjacent, so the postcondition of `Find` is false. Because of this, Dafny can prove the clearly false assertion at the end of the method. However, by default, Dafny does nothing to indicate the fact that this proof succeeds due to an inconsistency. As we will

https://doi.org/10.34727/2025/isbn.978-3-85448-084-6_32

show, computing coverage will allow detection of the fact that this assertion was proved vacuously, with no additional effort from the programmer.

Similarly, consider the Dafny program in Fig. 2 that has a redundant specification. One could imagine believing (incorrectly, but perhaps reasonably) that the index bound safety of the standard implementation of binary search requires the array to be sorted, and therefore providing the precondition shown. Dafny would accept this without complaint and continue to prove that the implementation does not use out-of-bounds array indices (an implicit specification that Dafny attempts to prove of every program). Dafny would not, by default, do anything to indicate that this precondition is unnecessary. Computing coverage will allow detection of this redundancy.

```
 1  method BinSearch(a:seq<int>, k:int)
 2    returns (r:int)
 3    requires forall i,j ::
 4      0 <= i < j < |a| ==>
 5        a[i] <= a[j]
 6  {
 7    var lo, hi := 0, |a|;
 8    while lo < hi
 9      invariant 0 <= lo <= hi <= |a|
10    {
11      var mid := (lo + hi) / 2;
12      if k < a[mid]      {
13        hi := mid;
14      } else if a[mid] < k {
15        lo := mid + 1;
16      } else {
17        return mid;
18      }
19    }
20    return -1;
21  }
```

Fig. 2. Dafny code with an unnecessary precondition (lines 3, 4, 5) and code that is not constrained by the specification (lines 17, 20).

Finally, consider the specification of the same binary search implementation. Some of the code is necessary to ensure array bounds safety, such as the calculation of `mid`. However, other portions of the code, such as the `return` statements, could be changed arbitrarily without affecting the results of verification.

Computing coverage allows more useful verifier feedback for each of these examples than previously available, and more granular information than available from previous verification coverage analysis. This approach is not specific to Dafny, and could be adopted by any verifier based on weakest preconditions.

B. Overview of Paper

Our contributions beyond prior work are as follows:

- We formalize how coverage translates to deductive verification, specifically the verification condition (VC) generation process, using unsatisfiable cores.
- We define the type of specification gap implied by each type of uncovered element. Our interpretation allows automatic detection of:
  - contradictory assumptions or branches, leading to vacuous proofs;
  - redundant specifications that can be removed; and
  - unconstrained implementation code that could be specified or removed.
- We provide experimental results showing reasonable computational overhead and useful warnings on a large corpus of real-world Dafny code.

We first review prior work in more detail. Then we define more precisely what coverage means in the abstract, and how it applies to compositional verification of programs with contracts (as opposed to monolithic transition systems), before detailing how to calculate coverage. Finally, we describe how we implemented the approach in Dafny and Boogie, and provide measurements of running the analysis on several Dafny code bases, including performance, warning count, and warning classification.

II. Prior Work

The term *coverage* has been used since at least the 1960s [10], usually to describe the set (or percentage) of syntactic program elements executed by a test or set of tests. (Although some cases, such as path coverage and MCDC extend beyond program elements into more semantic notions.) We describe a notion of coverage intended to be compatible with that used in testing, but that captures the elements that took place in a proof.

Le, *et al.* [23], investigated the idea of combining coverage from proofs and tests. Their definition of coverage is based on the degree to which mutations to the program can cause proof or test failure. Like our work, they consider coverage in terms of the syntactic structure of the program. They focus on how to combine proof and test coverage, rather than on how to calculate either, though they imply the use of mutation testing.

In the verification community, significant research has gone into the problem of determining how well a specification describes an implementation, the most relevant being connected to one of (a) deductive software verification using SMT or first-order logic, (b) model checking with respect to temporal logic specifications, or (c) analysis of high-level modeling languages.

In the realm of deductive software verification, one common approach to identifying either specification problems or problems in unspecified programs is to do some form of reachability analysis to identify infeasible portions of code or specification in the context of any available assumptions [11], [19], [20], [29].

One variation of this approach, sometimes referred to as "smoke testing", is available in several verification tools, including Boogie [5], [24], [27], Why3 [4], [31], and Frama-C [2], [13]. This approach involves adding `assert false` statements in relevant locations. Any that are provable indicate either an inconsistent specification or dead code.

These smoke testing features address part of the problem we aim to solve, but somewhat less efficiently, and do not attempt to identify how much of a program is covered by the given specification, whether any parts of the specification are redundant, or any other issues with specification quality.

The model checking community has extensively explored the problem through the lens of coverage, concluding that coverage and vacuity are dual notions [21]. According to this duality, the question is to determine, given a system that successfully verifies, what parts can change while still allowing verification to succeed. If a portion of the implementation can change arbitrarily without causing verification to fail, then either that portion of the implementation is not covered by the specification or some portion of the specification is vacuous, allowing any implementation. If part of the specification can change without causing verification to fail, then that part of the specification is unnecessary for proving the goal. Generally, such portions of the specification can be removed.

The original work describing this duality assumed the use of mutation testing as a mechanism for evaluating coverage or vacuity [22]. By changing a program element and attempting to re-prove the program, it's possible to determine whether that element is necessary for the proof.

As a more efficient alternative, one can do a proof that the implementation satisfies the specification and identify the facts used to complete the proof, which can be done in several ways. When using an SMT solver, the clauses that appear in an unsatisfiable core provide an overapproximation of one possible set of facts needed to complete a proof. The VaqTree tool uses this approach to do two types of vacuity detection during bounded model checking, and extracts even more precise information by examining the resolution proof required to show that the core is unsatisfiable [28]. The related VaqUoT tool extracts similar information from BDD-based proofs [17]. Later work described the generalization of unsatisfiable cores to unbounded model checking, as inductive validity cores [15], and described how to use these to measure the portion of an implementation covered by a specification [16].

The work on bounded and unbounded model checking was based on transition systems rather than programming languages, however. Unlike deductive software verification, these proofs typically aren't "modular" in the sense that each completely describes the correctness of an element of code. We describe how to take a similar approach in the context of deductive verification of software, along with the opportunities that arise in the new context.

Working with declarative models rather than a combination of a specification and an implementation, Torlak *et al.*, describe an approach for calculating coverage of Alloy specifications using unsatisfiable cores [30], with the goal of helping detect mistakes.

Finally, Beckert, *et al.* [3] addressed the problem of analyzing incomplete proofs to determine how much of the program they had successfully verified. This work calculated coverage as a subset of the state space of a program rather than a subset of its structural elements. It diverges from our work in two dimensions: first by considering failed proofs and second by considering the semantic state space rather than syntactic elements.

III. Verification Coverage

A. Definition

We define *verification coverage* as a property of programs that meet their specifications as shown by a verifier. As in previous model checking work, we define the covered elements of the program (and its specification) to be those that, if changed, *could* cause verification to fail. Not all changes would necessarily lead to failure, but for each covered element there is at least one change that could lead to the program no longer verifying. Conversely, uncovered elements are those that could be changed arbitrarily (within the bounds of a well-formed program) without causing verification to fail. Uncovered specification elements indicate over-specification. Uncovered implementation elements indicate under-specification. Note that, in our interpretation, a subterm that appears identically more than once in a program may be covered in some locations and not others.

B. Interpretation

We begin by describing, at a high level, what program elements we consider atomic. This is of particular importance because the presence or absence of each type of element in the set of covered elements can mean something qualitatively different. Other choices are possible, including different choices in granularity, as we discuss further in Section III-E. But the following choices reflect the implementation in Boogie and Dafny that we have experimented with, and reflect an approach that would naturally apply to most verifiers for imperative languages, especially those that build verification conditions using the weakest precondition calculus.

In our formalization and implementation, we consider Boogie, a simple imperative language containing procedures and contracts on procedures. We include implementation statements and specification constructs as our atomic program elements. Implementation statements include assignments and calls to procedures. Specification constructs include assert statements, assume statements, loop invariants, and pre- and post-conditions on procedures.

One key type of information from coverage analysis is the identification of *uncovered* elements, and useful interpretation of coverage depends on the type of program element in the original language. For the constructs that show up in Boogie:

- An uncovered assignment or call statement indicates code that is either lacking specification (influencing no proof goals) or unreachable.
- An uncovered assume statement or precondition of the procedure being verified is a purely unnecessary assumption that can be removed.
- An uncovered precondition of a callee indicates that the precondition was established vacuously.

- An uncovered postcondition of a callee is an unnecessary assumption from the point of view of its caller. It may be necessary for other callers, but could be removed if it is not necessary for any caller.
- An uncovered postcondition, while verifying the procedure it is attached to, indicates that the postcondition is proved vacuously.
- An assert statement can be viewed as having two parts: an assertion that the expression is true at the point of the statement, and an assumption that it is true after the statement. If the former is uncovered, the assertion holds vacuously. If the latter is uncovered, the assertion is useful only for its top-level conclusion, not as an intermediate step toward proving later goals, and can be removed if it was intended for that purpose.
- A loop invariant that is uncovered while being proved indicates a vacuous proof.
- A loop invariant that is uncovered while assuming it in the body of the loop is an unncessary invariant.

Note that provenance is critical for the approach to be useful, especially given that most verification tools work by successively lowering code into simpler and simpler languages. We'll show below that we can consider a *passive* [1], [12] language with only assert and assume statements, but we can provide much more useful feedback if we know what construct in the original language gave rise to each of these statements.

To make this precise, we now formally specify a language and how to compute verification conditions for it that enable coverage tracking.

C. VC Generation with Coverage Tracking

We first define verification condition generation with coverage tracking on the standard language of unstructured *passive* programs [1], [12], [18] shown on the top of Fig. 3. In a passive program, each assignment to a variable is represented by an assumption about a separate incarnation of that variable.

Note that the full Boogie language includes a larger set of constructs than this, and Section III-D describes how to track the necessary information for coverage analysis when translating those constructs into passive programs.

A command can be an assertion or an assumption of a predicate P annotated with a label L, and a block consists of a sequence of commands followed by a transfer to one or more successor blocks.[1] We extend this language slightly, however, by adding a label to each assertion and assumption. Coverage analysis is then the problem of identifying which statements, identified by label, are required to complete the proof.[2] This is fundamentally the information that unsatisfiable cores provide, when doing proofs with an SMT solver.

To construct verification conditions from labeled commands, we introduce a fresh Boolean variable, v_L for each label L, as shown on the bottom of Fig. 3. These variables interact with the postcondition in the same way that the predicate embedded within the command does.[3]

$$\begin{array}{rcl} c & ::= & \mathsf{assert}_L\ P \\ & | & \mathsf{assume}_L\ P \\ b & ::= & (c;)^*\mathsf{goto}\ b^* \end{array}$$

$$\begin{array}{rcl} wp_C(\mathsf{assert}_L\ P, Q) & = & (v_L \wedge P) \wedge Q \\ wp_C(\mathsf{assume}_L\ P, Q) & = & (v_L \implies P) \implies Q \\ wp_C(\mathsf{goto}\ b^*, Q) & = & \bigwedge_b wp_C(b, Q) \\ wp_C(c_1; c_2, Q) & = & wp_{c_1, wp(c_2, Q)} \end{array}$$

Fig. 3. Top: syntax of passive programs. Bottom: weakest preconditions of passive programs with coverage-tracking labels added. All instances of v_L are fresh.

Once we have generated a verification condition C, we assert the entire negated VC, without a label, and assert each Boolean label along with a textual name. This leads to an SMT query like the following.

```
(assert v_L_1 :named "L_1")
...
(assert (not C))
(check-sat)
(get-unsat-core)
```

The SMT solver will respond to this query with a list of the labels that occur in the unsatisfiable core of the problem, corresponding to the covered program elements. Note that, in general, SMT solvers do not typically provide a *minimal* unsatisfiable core, and that multiple minimal cores may exist. This means that the possibility exists that a program element is included in the unsatisfiable core when, in fact, removing it may not lead to verification failure. Anecdotally, we have not seen lack of minimality cause issues in practice. However, an interesting avenue for future would would be to systematically measure how close cores are to minimal in practice, in this use case.

D. Desugaring with Labels

The Boogie language includes a wider range of commands than listed in the previous section, and already includes a transformation to desugar these into an acyclic graph of assertions and assumptions. This process is standard, and described elsewhere [1], [12], [18]. However, we provide an overview here of how it interacts with labels.

First, we define the syntax of labels used in Fig. 3 to include the alternatives shown in Fig. 4. In the surface Boogie syntax, an atomic label l can appear on an assignment statement, assert statement, assume statement, call statement, precondition, postcondition, or loop invariant. During desugaring, these labels are propagated as follows:

[1] Note that this syntax doesn't include labels on blocks and syntactically ensures that the program is acyclic, a standard guarantee of passive programs.

[2] Previous work in ESC/Java labeled assertions for the opposite reason, to help identify what portions of a program participated in a counterexample [26]. The Simplify theorem prover [9] had built-in support for these labels.

[3] The rule for assume statements already existed in Boogie, as a mechanism for tracking necessary assumptions as part of an effort to help better understand verification results. The portion of this focusing on errors has been published [6], but that publication doesn't discuss using the encoding to better understand successful proofs.

$$\begin{array}{lllll} L & ::= & l & & \\ & \mid & \mathsf{PreCheck}(l_1, l_2) & \mid & \mathsf{PreAssume}(l_1, l_2) \\ & \mid & \mathsf{Post}(l_1, l_2) & \mid & \mathsf{InvEstablish}(l) \\ & \mid & \mathsf{InvMaintain}(l) & \mid & \mathsf{InvAssume}(l) \end{array}$$

Fig. 4. Syntax of labels.

- Labels on assertions and assumptions are preserved.
- The label on an assignment is transferred directly to the corresponding assumption in the passive program.
- Each precondition (with label l_P) of a call (labeled l_C) is desugared into an assertion labeled with $\mathsf{Pre}(l_C, l_P)$ and an assumption labeled with $\mathsf{PreAssume}(l_C, l_P)$.
- Each postcondition (with label l_Q) of a call is translated to an assertion labeled with $\mathsf{Post}(l_C, l_Q)$.
- Each invariant (with label l) is broken into three pieces: an assertion labeled with $\mathsf{InvEstablish}(l)$, an assertion labeled with $\mathsf{InvMaintain}(l)$, and an assumption labeled with $\mathsf{InvAssume}(l)$.

The Dafny language includes even more constructs, and Dafny programs are verified by generating Boogie programs and verifying those using the standard Boogie verifier. The rules for tracking labels originating in Dafny language constructs are more numerous, but follow the same intuitive structure as the rules in the list above.

E. Finer Granularity

When calculating coverage using unsatisfiable cores, we can also consider Boolean sub-expressions of any larger expression. For example, if a precondition of the form $P \implies Q$ exists in the program, a coverage result that includes only P indicates that P is unsatisfiable and therefore the content of Q is irrelevant. The Boogie implementation includes decomposition of the Boolean operators for conjunction, disjunction, and implication.

Other implementations could go farther. As one extreme case, for bit vector programs, each bit could be considered separately. We have not yet experimented with this, but it would follow from the same approach. The finite nature of the programs would allow measurement of state space coverage, as well.

F. Impact of Trigger-Based Quantifier Instantiation

SMT solvers can use triggers [8], [9], [14] to aid quantifier instantiation. Each quantified formula may be annotated with a term schema that indicates the shape of a particular expression. If an expression of this shape exists in the current clause database, the solver may consider instantiating the quantified formula. In some cases, it is necessary to introduce a term into a Dafny program, potentially as an additional assertion, to allow the appropriate quantified axioms to be instantiated, even if that assertion is otherwise unnecessary for completing the proof. Such an element will be considered uncovered, even though removing it would cause verification to fail. This is logically sound — the property to be proved is still derivable without that program element, even if doing it with SMT would require a perfect quantifier instantiation oracle in this case — but it can lead to a frustrating user experience in some cases.

IV. Implementation

To evaluate the precision, overhead, and usability of our approach on real-world verified programs, we implemented the analysis in two open-source tools: the foundation in Boogie, and extensions to Dafny to use this foundation.

The Boogie implementation integrates with the verification pipeline, building on the existing code for desugaring procedure calls and constructing an acyclic, passive program. The key changes needed were to allow statements to be annotated with labels and to extend labels during desugaring (as described in Section III-D). The implementation built on prior work intended to allow identification of necessary `assume` statements, making it also possible to identify covered assertions. This implementation consists of roughly 700 lines of C# code, almost all of which involves tracking the provenance of labels. The change to the VC generator exactly mirrors the description in Fig. 3.

Dafny verifies programs by first translating them into Boogie and then invoking the Boogie verifier. The Dafny implementation, consisting of around 2.5k lines of C# code, automatically adds a label to each Boogie statement that has a direct correspondence to a Dafny statement, and does additional provenance tracking to aid in producing useful messages. Based on its computation of coverage, Dafny implements warnings about contradictory or redundant specifications and source highlighting to indicate overall coverage and help identify unconstrained code.

A. Implementation Caveats

Building this analysis in Boogie and leveraging the existing Dafny to Boogie translation was mostly straightforward, but several subtle complications arose.

- Dafny programs sometimes include intentional proofs by contradiction. The goals of these proofs are typically uncovered, and therefore lead to warnings. To support these, we allow labeling an assert statement as taking part in an intentional proof by contradiction, silencing these warnings.
- Some of the Boogie code generated by Dafny includes assertions in locations that are unreachable by construction, presumably because it simplifies the translation and didn't previously lead to issues. We identify several classes of these and automatically exclude them from warnings.
- Each Dafny function definition is represented in Boogie as one or more quantified axioms, each of which includes most or all of the function body. This limits the degree to which we can track which subexpressions of a function body were covered by a particular proof. We can, however, track whether the function body was used at all. Inlining functions, rather than defining them with axioms, would

be one way to address this limitation, though it would not be possible for recursive functions.

V. Experiments

Previous work on verification coverage has reported substantial overhead [16]. In addition, most tools for automatically detecting problems in software or specifications are prone to false positives, or a low level of "interesting" true positives. And, finally, SMT-based verification can be prone to the problem of *brittleness*. We say that a pair of an SMT solver and a query is brittle if the solver can solve the inital query but fails to solve a minor, semantics-preserving variant of the query.[4] An analysis the increases brittleness too substantially may be infeasible to adopt in practice. Therefore, we aim to answer several research questions here.

RQ1 What is the typical overhead of calculating coverage, relative to doing verification without coverage analysis? We measure this in terms of both execution time and Z3 resource count (a deterministic measure of SMT solver work) per SMT query.

RQ2 Do the changes made to SMT queries to allow coverage analysis impact the brittleness of verification?

RQ3 How does the overhead compare to the "smoke testing" approach, where a tool attempts to prove `false` at various points throughout the program? In particular, how does it compare to the smoke testing implementation that already exists in Boogie?

RQ4 How useful are the warnings that Dafny generates? The possibility exists for entirely spurious warnings (due to encoding artifacts that lead to unreachable code or bugs in the implementation), but also for warnings about intentional proofs by contradiction or dead code, which may be considered uninteresting. The possibility also exists for false negatives, where program elements are determined to be covered due to the overapproximation of unsatisfiable cores.

To answer these questions, we ran Dafny on the code from several public repositories of Dafny code, and one non-public repository. These are:

- ESDK: AWS Encryption SDK for Dafny
- MPL: AWS Cryptographic Material Providers Library
- INT: A non-public project
- STD: Built-in Dafny `Std` library

We ran all measurements with a pre-release commit of Dafny (`67daee75a302077b4b49048b4b90e560bfbdf32b`), with Boogie v3.0.12 and Z3 v4.12.1, on an AWS c5.4xlarge host (16 3GHz Intel Xeon CPUs, 32GB RAM).

A. Results on Examples

Before presenting statistics about the performance of coverage analysis on large code bases, we show the results it provides on the examples shown in Section I. Recall that, in the first example (see Fig. 1), the `CallFind` method invokes `Find` with arguments that make its postcondition unsatisfiable. Dafny reports this problem as follows:

```
Find.dfy(12,2): Warning: proved using
contradictory assumptions: assertion
always holds. (Use the `{:contradiction}`
attribute on the `assert` statement to
silence.)
   |
12 |   assert xs[i] == 45;
   |   ^^^^^^

Find.dfy(12,9): Warning: proved using
contradictory assumptions: index in range
   |
12 |   assert xs[i] == 45;
   |          ^^


Dafny program verifier finished with 2
verified, 0 errors
```

Next, recall that, in the second example (see Fig. 2), the `BinSearch` method has an unnecessary precondition. Dafny reports this problem as follows:

```
BinarySearch.dfy(3,11): Warning: unnecessary
requires clause
  |
3 |   requires forall i,j ::
  |            ^^^^^^^^^^^^^


Dafny program verifier finished with 2
verified, 0 errors
```

B. Overhead of Coverage Analysis

To determine how much overhead coverage analysis has relative to normal verification (RQ1), we measured the following (shown for each project in Fig. 5).

- the total number of goals for each project;
- time (in seconds) and resource count (RC, a deterministic proxy for difficulty provided by Z3) for each SMT query, for both normal verification (N) and verification with coverage tracking (C), along with statistics about the distribution of these values; and
- failure count, with and without coverage tracking.

In this data, we include the median, mean, and trimmed mean (a.k.a., truncated mean). This last value is calculated as the mean of the middle $n\%$ of values [7]. So the 98% trimmed mean (M98) discards the 1% smallest and 1% largest values. We calculate this because we observe that most Dafny verification tasks include a few very difficult queries and a huge number of relatively easy queries, and that the most difficult queries are frequently the most brittle. The trimmed mean discards the extreme outliers, giving a more characteristic picture of typical performance.

Based on these measurements, we compute an overhead, in terms of both time and resource count, for each project and for each statistic. These computations show that the overall time overhead ranges from around 20% to over 73%, depending

[4] Other authors have used the terms *instability* or the *butterfly effect* [32] for this phenomenon.

		ESDK	**MPL**	**INT**	**STD**
Goals		2122	4981	11042	4647
Time	N	$7.8{\times}10^{2}$	$7.2{\times}10^{2}$	$4.9{\times}10^{3}$	$2.8{\times}10^{2}$
	C	$1.1{\times}10^{3}$	$1.2{\times}10^{3}$	$8.4{\times}10^{3}$	$3.3{\times}10^{2}$
Mean	N	$3.7{\times}10^{-1}$	$1.4{\times}10^{-1}$	$4.4{\times}10^{-1}$	$6.0{\times}10^{-2}$
	C	$5.1{\times}10^{-1}$	$2.4{\times}10^{-1}$	$7.6{\times}10^{-1}$	$7.1{\times}10^{-2}$
	O	40.24%	65.66%	73.34%	19.97%
Med.	N	$1.5{\times}10^{-1}$	$5.2{\times}10^{-2}$	$1.2{\times}10^{-1}$	$2.4{\times}10^{-2}$
	C	$1.5{\times}10^{-1}$	$5.3{\times}10^{-2}$	$1.1{\times}10^{-1}$	$2.5{\times}10^{-2}$
	O	1.13%	2.51%	-4.27%	5.85%
M98	N	$2.1{\times}10^{-1}$	$1.2{\times}10^{-1}$	$2.8{\times}10^{-1}$	$5.3{\times}10^{-2}$
	C	$2.1{\times}10^{-1}$	$1.3{\times}10^{-1}$	$3.0{\times}10^{-1}$	$5.5{\times}10^{-2}$
	O	**3.49%**	**9.17%**	**7.48%**	**4.37%**
RC	N	$9.4{\times}10^{8}$	$2.2{\times}10^{9}$	$4.7{\times}10^{9}$	$8.0{\times}10^{8}$
	C	$1.6{\times}10^{9}$	$4.9{\times}10^{9}$	$6.0{\times}10^{9}$	$7.2{\times}10^{8}$
Mean	N	$4.4{\times}10^{5}$	$4.5{\times}10^{5}$	$4.3{\times}10^{5}$	$1.7{\times}10^{5}$
	C	$7.7{\times}10^{5}$	$9.9{\times}10^{5}$	$5.5{\times}10^{5}$	$1.5{\times}10^{5}$
	O	74.22%	121.37%	27.44%	-10.10%
Med.	N	$2.7{\times}10^{5}$	$8.7{\times}10^{4}$	$1.0{\times}10^{5}$	$5.4{\times}10^{4}$
	C	$2.8{\times}10^{5}$	$9.3{\times}10^{4}$	$1.2{\times}10^{5}$	$5.9{\times}10^{4}$
	O	3.33%	6.70%	14.18%	9.21%
M98	N	$3.8{\times}10^{5}$	$2.9{\times}10^{5}$	$2.3{\times}10^{5}$	$1.1{\times}10^{5}$
	C	$4.2{\times}10^{5}$	$3.4{\times}10^{5}$	$2.8{\times}10^{5}$	$1.2{\times}10^{5}$
	O	**10.56%**	**17.59%**	**21.11%**	**6.77%**
Fail.	N	2	1	3	0
	C	4	2	15	7

Fig. 5. Overhead of coverage-enabled verification with respect to normal verification. Each row is labeled with one of: "N" for normal verification, "C" for coverage-enabled verification, or "O" for calculation of the overhead of coverage analysis.

on the codebase. However, the median time overhead ranges between the much more tolerable -4% and 6%, and the trimmed mean between 3.5% and 9%. Similar patterns occur for resource counts. This suggests that, by constructing Dafny programs to avoid extreme outliers in verification time, we can keep the overhead of coverage analysis low enough to be almost imperceptable in practice. Our experience is that Dafny programs can be refactored to achieve this, and that doing so reduces verification brittleness in general.

We also show scatter plots comparing the resource count (Fig. 6) and time (Fig. 7) of each query between normal and coverage-enabled verification. Both plots include a random subsample of 1000 points from the total data set, and exclude a small number of outliers above 10^7 resource units or 3 seconds, to make the pattern of the remaining data clearer. These plots show that, although coverage-enabled SMT queries are usually more difficult, they can also sometimes be easier, and the vast majority use less than twice the resources.

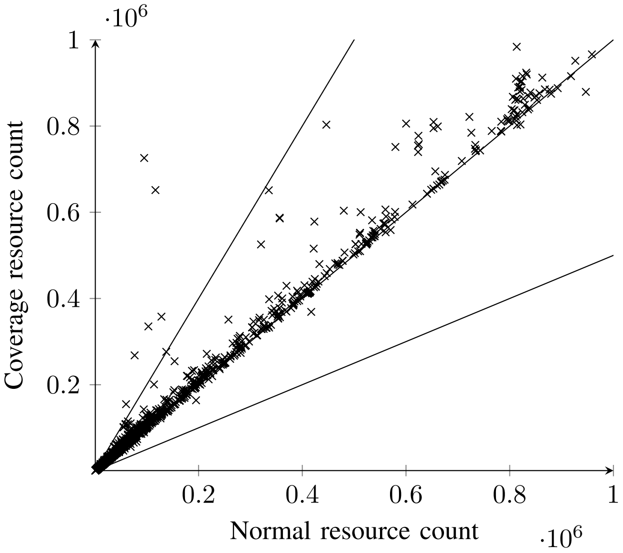

Fig. 6. Comparison of solver resource counts between normal (x-axis) and coverage-enabled (y-axis) verification, with lines for $y = 0.5x$, $y = x$, and $y = 2x$.

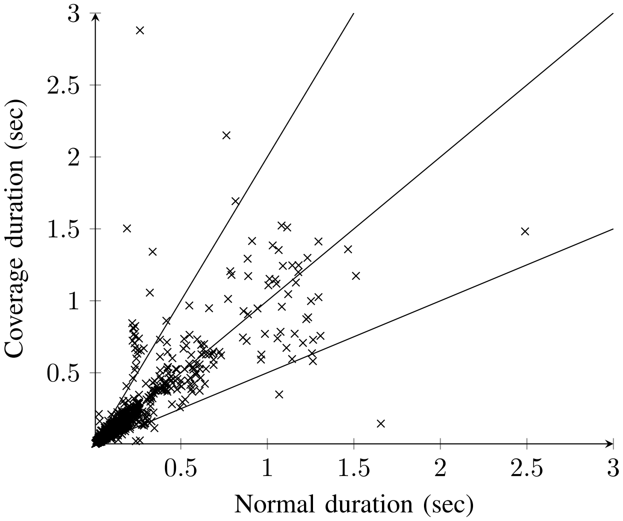

Fig. 7. Comparison of solver execution time between normal (x-axis) and coverage-enabled (y-axis) verification, with lines for $y = 0.5x$, $y = x$, and $y = 2x$.

C. Impact on Brittleness

To determine how much coverage analysis impacts the brittleness of verification (RQ2), we use a Dafny feature that allows each verification condition to be randomized along several dimensions: ordering of SMT declarations, names of SMT variables, and the random seed used by the SMT solver to make arbitrary decisions. The results of running 5 iterations of verification with this feature enabled, both with and without coverage calculation, appear in Fig. 8. [5] The increase in failure count and the standard deviation of resource count shows that enabling coverage analysis can increase brittleness — that is, it can increase the likelihood that verification will fail after small, semantics-preserving changes, and can increase the degree to which proof difficulty varies after such changes. However, the M98 trimmed mean shows only a slight increase in time and resource count, suggesting that the technique may be difficult to apply to already-brittle codebases, but is unlikely to introduce brittleness that is not already present.

	Mode	**INT**	**STD**
Time	N	1.4×10^{4}	1.4×10^{3}
	C	3.6×10^{4}	1.9×10^{3}
Mean	N	2.6×10^{-1}	6.1×10^{-2}
	C	6.5×10^{-1}	8.3×10^{-2}
Mean (98%)	N	1.3×10^{-1}	5.4×10^{-2}
	C	1.5×10^{-1}	5.7×10^{-2}
Std. dev.	N	5.2	2.6×10^{-1}
	C	1.2×10^{1}	1.3
RC	N	1.9×10^{10}	3.7×10^{9}
	C	3.5×10^{10}	5.0×10^{9}
Mean	N	3.5×10^{5}	1.6×10^{5}
	C	6.4×10^{5}	2.1×10^{5}
Mean (98%)	N	2.3×10^{5}	1.1×10^{5}
	C	2.8×10^{5}	1.2×10^{5}
Std. dev.	N	6.2×10^{6}	3.0×10^{6}
	C	1.0×10^{7}	6.9×10^{6}
Failures	N	18	13
	C	85	42

Fig. 8. Impact of coverage-enabled verification on brittleness.

D. Comparison with Smoke Testing

To compare coverage-enabled verification with smoke testing (RQ3), we focus on overall time and resource use, because smoke testing works entirely using additional queries, rather than modifying existing queries, and these therefore don't align with queries produced by normal or coverage-enabled verification. This also prevents a comparison in the form of a scatter plot. In addition, the implementation of smoke testing is less robust than that of coverage analysis, and fails on many of our benchmarks, so we could perform a meaningful comparison on only one element of our benchmark suite. The lack of robustness meant that it also needed to be run with slightly different Dafny flags, so the statistics in this table differ from those in Fig. 5. A summary of the comparison with smoke testing appears in Fig. 9.

	INT
Total coverage time	8.4×10^{3}
Combined smoke time	1.1×10^{4}
Total coverage resource count	6.0×10^{9}
Combined smoke resource count	2.1×10^{10}

Fig. 9. Overhead of smoke testing relative to normal verification, showing the total time and resource count taken by verification with coverage analysis enabled compared to the total time and resource count taken by verification with smoke testing enabled.

These results show that, in this instance, smoke testing is more expensive than coverage analysis. Other implementations of smoke testing could have different performance profiles. However, it is not clear that their performance could be directly compared to the implementation described in this paper, since they work on different languages.

Coverage analysis also provides additional capabilities, including detection of unnecessary assumptions, indication of what portions of an implementation have been verified, and indication of what facts were used in a proof.

E. Quality of Warnings

Finally, we analyze all of the warnings produced by coverage analysis and compare them with the warnings produced by smoke testing (RQ4). Coverage analysis warnings consist of redundant assumptions and contradictory assumptions. Redundant assumptions can be further categorized into:

- Redundant preconditions and assumption statements.
- Redundant assertions that were presumably added to aid in the proof of a later postcondition but turn out to be unnecessary.

The contradictory assumptions we encountered can be further categorized into:

- direct instances of `assume false`,
- dead code due to infeasible branches,
- inconsistencies in Dafny's background axioms,
- intentional proofs by contradiction,
- redundant tests of facts guaranteed by proof, and
- dead proof or specification code due to infeasible cases.

The number of each of these warnings appear in Fig. 10. Direct instances of `assume false` are a mechanism for avoiding proof of difficult properties, and occur in two of the projects analyzed. Dead code is rare, occurring in only two instances, in one project. The process of performing these experiments uncovered an inconsistency in the background axioms Dafny generates related to the freshness of heap objects that affects one project (and has since been fixed). Three of the four projects include intentional proofs by contradiction. Two of the projects included redundant runtime tests of properties that were guaranteed by proof. And three of the projects included either specification expressions in which some subexpression was constant or branches in proofs that covered impossible

[5] A bug in Dafny unrelated to coverage analysis prevented us from collecting brittleness data on the ESDK and MPL benchmarks. This bug has since been fixed, but not in time to be included in these results.

	ESDK	**MPL**	**INT**	**STD**
Coverage warn.	38	301	78	290
Redund. assum.	24	164	23	59
Redund. assert.	11	30	25	201
False assum.	0	0	8	9
Dead code	0	2	0	0
Prelude contrad.	0	69	0	0
Proof by contrad.	2	15	0	14
Redundant test	1	8	0	0
Dead proof/spec.	0	13	20	7
Smoke warn.	-	-	2897	-

Fig. 10. Warnings produced from coverage-enabled verification and smoke testing.

cases. These could be removed, simplifying the specification and proofs.

The inconsistencies due to Dafny's prelude axioms were the most significant specification mistake detected in this analysis. However, all of the others, except the intentional proofs by contradiction, indicate implementation or specification code that could be improved.

Smoke testing generates warnings in terms of Boogie code rather than Dafny code. That, plus the large number of warnings, made it infeasible to categorize each one. We instead simply note that the sheer number, even if they were in Dafny terms, would make them infeasible for the developers to triage, as well.

VI. Conclusion

The concept of verification coverage originated in the model checking world, where the monolithic nature of many model checking systems meant that coverage analysis was often expensive. This analysis has a direct analog in compositional deductive verification of software, however, where individual proof goals tend to be smaller and simpler. We have shown that coverage analysis can be smoothly integrated into a verification-aware programming language like Dafny with reasonable performance overhead and problem reports free of false positives (with caveats for partially unused elements and intentional proofs by contradiction), though false negatives (failures to detect true specification gaps) are possible.

This work only begins the investigation into coverage in deductive software verification. Several avenues for future work are immediately apparent:

- Coverage can be calculated with finer granularity (see Section III-E). Labeling additional program elements may make it possible to provide more precise feedback to the programmer about the quality and scope of their specifications.
- Unsatisfiable cores can be minimized (with additional computational expense). This has the potential to reduce or eliminate the possibility of false negatives, although multiple minimal cores may exist in some cases. Alternatively, analyzing the structure of the proof of unsatisfiability, rather than the clauses in the unsatisfiable core, also has the potential to produce more precise results [28].
- Information about the facts used in a proof can be used to optimize future searches for the same or similar proofs. A Dafny lemma, for example, could be updated to restrict the facts sent to the SMT solver to include only those that were necessary to complete the proof, reducing the solver's search space.

References

[1] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. *SIGSOFT Softw. Eng. Notes*, 31(1):82–87, September 2005.

[2] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. The dogged pursuit of bug-free C programs: the Frama-C software analysis platform. *Commun. ACM*, 64(8):56–68, July 2021.

[3] Bernhard Beckert, Mihai Herda, Stefan Kobischke, and Mattias Ulbrich. Towards a notion of coverage for incomplete program-correctness proofs. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Verification*, pages 53–63, Cham, 2018. Springer International Publishing.

[4] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011. https://hal.inria.fr/hal-00790310.

[5] Boogie. https://github.com/boogie-org/boogie.

[6] Maria Christakis, K. Rustan M. Leino, Peter Müller, and Valentin Wüstholz. Integrated environment for diagnosing verification errors. In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 424–441, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[7] Anirban DasGupta. *Asymptotic Theory of Statistics and Probability*, chapter 18. Springer Texts in Statistics. Springer-Verlag New York, 2008.

[8] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[9] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, May 2005.

[10] William R. Elmendorf. Controlling the functional testing of an operating system. *IEEE Transactions on Systems Science and Cybernetics*, 5(4):284–290, 1969.

[11] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. *SIGOPS Oper. Syst. Rev.*, 35(5):57–72, October 2001.

[12] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, page 193–205, New York, NY, USA, 2001. Association for Computing Machinery.

[13] Frama-C. https://www.frama-c.com/.

[14] Yeting Ge, Clark Barrett, and Cesare Tinelli. Solving quantified verification conditions using satisfiability modulo theories. In Frank Pfenning, editor, *Automated Deduction – CADE-21*, pages 167–182, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[15] Elaheh Ghassabani, Andrew Gacek, and Michael W. Whalen. Efficient generation of inductive validity cores for safety properties. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 314–325, New York, NY, USA, 2016. Association for Computing Machinery.

[16] Elaheh Ghassabani, Andrew Gacek, Michael W. Whalen, Mats P. E. Heimdahl, and Lucas Wagner. Proof-based coverage metrics for formal verification. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 194–199, 2017.

[17] Mihaela Gheorghiu and Arie Gurfinkel. VaqUoT: A tool for vacuity detection. Technical report, University of Toronto, Department of Computer Science, 2006.

[18] Radu Grigore, Julien Charles, Fintan Fairmichael, and Joseph Kiniry. Strongest postcondition of unstructured programs. In *Proceedings of the 11th International Workshop on Formal Techniques for Java-like Programs*, FTfJP '09, New York, NY, USA, 2009. Association for Computing Machinery.

[19] Jochen Hoenicke, K. Rustan M. Leino, Andreas Podelski, Martin Schäf, and Thomas Wies. It's doomed; we can prove it. In Ana Cavalcanti and Dennis R. Dams, editors, *FM 2009: Formal Methods*, pages 338–353, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[20] Mikoláš Janota, Radu Grigore, and Michal Moskal. Reachability analysis for annotated code. In *Proceedings of the 2007 Conference on Specification and Verification of Component-Based Systems: 6th Joint Meeting of the European Conference on Software Engineering and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, SAVCBS '07, page 23–30, New York, NY, USA, 2007. Association for Computing Machinery.

[21] Orna Kupferman, Wenchao Li, and Sanjit A. Seshia. On the duality between vacuity and coverage. Technical Report UCB/EECS-2008-26, EECS Department, University of California, Berkeley, March 2008.

[22] Orna Kupferman, Wenchao Li, and Sanjit A. Seshia. A theory of mutations with applications to vacuity, coverage, and fault tolerance. In *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*, FMCAD '08. IEEE Press, 2008.

[23] Viet Hoang Le, Loïc Correnson, Julien Signoles, and Virginie Wiels. Verification coverage for combining test and proof. In Catherine Dubois and Burkhart Wolff, editors, *Tests and Proofs*, pages 120–138, Cham, 2018. Springer International Publishing.

[24] K. Rustan M. Leino. This is Boogie 2. June 2008.

[25] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[26] K. Rustan M. Leino, Todd Millstein, and James B. Saxe. Generating error traces from verification-condition counterexamples. *Sci. Comput. Program.*, 55(1–3):209–226, March 2005.

[27] K. Rustan M. Leino and Philipp Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 312–327, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[28] Jocelyn Simmonds, Jessica Davies, Arie Gurfinkel, and Marsha Chechik. Exploiting resolution proofs to speed up ltl vacuity detection for bmc. In *Formal Methods in Computer Aided Design (FMCAD'07)*, pages 3–12, 2007.

[29] Aaron Tomb and Cormac Flanagan. Detecting inconsistencies via universal reachability analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, page 287–297, New York, NY, USA, 2012. Association for Computing Machinery.

[30] Emina Torlak, Felix Sheng-Ho Chang, and Daniel Jackson. Finding minimal unsatisfiable cores of declarative specifications. In Jorge Cuellar, Tom Maibaum, and Kaisa Sere, editors, *FM 2008: Formal Methods*, pages 326–341, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[31] Why3 – where programs meet provers. https://www.why3.org/.

[32] Yi Zhou, Jay Bosamiya, Yoshiki Takashima, Jessica Li, Marijn Heule, and Bryan Parno. Mariposa: Measuring SMT instability in automated program verification. In *2023 Formal Methods in Computer-Aided Design (FMCAD)*, pages 178–188, 2023.

Appendix

A. Experimental Reproduction

This appendix describes how we gathered experimental data for each benchmark listed in the paper. Each of the following subsections describes one benchmark, giving the URL of the repository on GitHub, the SHA of the commit we ran on, the subdirectory of the repository we ran the verification in (if not the root), and the verification script we ran. We used different verification scripts to mirror each project's standard build setup as closely as possible.

1) Common Options:

1) Common Options: Each of these scripts passes its arguments to Dafny. This is so that we can use the same verification scripts for each mode of analysis we performed. For each experiment type, we passed in old-style Dafny CLI flags, because that is what each project used in its build system at the time of writing. These arguments are as follows:

For all experiments:

- `/verificationLogger:csv`

For coverage:

- `/warnRedundantAssumptions`
- `/warnContradictoryAssumptions`

For brittleness analysis:

- `/randomizeVcIterations:5`

For smoke testing:

- `/smoke`
- `/trace`
- `/vcsCores:1`
- `/prune:0`
- `/restartProver`

2) STD:

- Repository URL:
  - https://github.com/dafny-lang/dafny
- Commit hash:
  - `67daee75a302077b4b49048b4b90e560bfbdf32b`
- Subdirectory:
  - `Source/DafnyStandardLibraries/src/Std`
- Verification script:

```
dafny \
  /compile:0 \
  /trackPrintEffects:1 \
  /definiteAssignment:3 \
  /readsClausesOnMethods:1 \
  /rlimit:1000000  \
  `find . -name "*.dfy" | \
      grep -v "TargetSpecific"` $*
```

3) MPL:

- Repository URL:
  - https://github.com/aws/aws-cryptographic-material-providers-library
- Commit hash:
  - `29c6a2c20d4cdeaadeacd242dc4764ce0012193d`
- Verification script:

```
find . -name "*.dfy" | \
  grep -v "libraries/" | \
  xargs -P 4 -n 1 \
    dafny \
      /compile:0 \
      /definiteAssignment:3 \
      /quantifierSyntax:3 \
      /functionSyntax:3 \
      /unicodeChar:0 \
      /timeLimit:150 $*
```

4) ESDK:

- Repository URL:
  - https://github.com/aws/aws-encryption-sdk-dafny
- Commit hash:
  - `24ddf9a7b2840bea4a37c633c45d60cdfc177184`
- Verification script:

```
find . -name "*.dfy" | \
  grep -v "libraries/" | \
  grep -v "mpl/" | \
  grep -v "test/" | \
  xargs -P 4 -n 1 \
    dafny \
      /compile:0 \
      /definiteAssignment:3 \
      /quantifierSyntax:3 \
      /functionSyntax:3 \
      /unicodeChar:0 \
      /timeLimit:150 $*
```