Failure-Proof Specification Writing: Smoke Tests in VerCors

Alexandra Jakovleva, Alexander Stekelenburg[1][0009-0008-7305-038X], and Marieke Huisman[1][0000-0003-4467-072X]

University of Twente,
Enschede, The Netherlands
{a.v.stekelenburg,m.huisman}@utwente.nl

Abstract. One important caveat in software verification is that specifications can contain inconsistencies, which can result in implementations passing verification, without actually being checked. To avoid this, software verification systems build in *smoke tests*: by injecting deliberately false properties that can only be proven if the proof context contains a contradiction. This paper discusses how a number of known smoke tests, developed earlier in the context of Frama-C, have been integrated into the VerCors verifier. We extensively test our implementation, and evaluate its performance overhead on the verification time.

1 Introduction

Over the past years, deductive verification has shown its power to provide formal guarantees about the behaviour of software [5]. However, the power of deductive verification depends on the quality of specifications. In particular, if specifications contain inconsistencies, proofs vacuously go through, and the result of the verifier gives a wrong sense of correctness. For example, if a function's precondition is logically equivalent to false, then every postcondition that is specified for the function will pass verification. Therefore, in addition to further improving the capacities of the verification engine and the supported program logic, we also need to have techniques that help us to detect such inconsistencies in specifications.

Recently, Blanchard et al. [2] addressed this issue for the deductive verifier Frama-C (WP-plugin) [6]. They presented various *smoke tests* for Frama-C, named after the practice of pumping smoke through pipes to find leaks. A deliberately false proof obligation is injected at a strategic program point, such that if the verifier proves it, the proof context must already be contradictory. Concretely, they present smoke tests for five categories of specification inconsistencies, which are implemented by inserting appropriate checks at dedicated points in the control-flow graph.

This paper investigates how to introduce smoke tests for the VerCors verifier [1], a deductive verifier for concurrent programs supporting multiple input languages such as Java and C. Up to now, VerCors only implemented one simple smoke test, flagging unsatisfiable preconditions. As the VerCors implementation

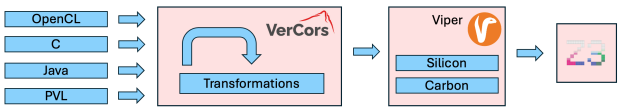


Fig. 1: The VerCors verification pipeline.

does not use a control-flow graph, but rather instruments the AST directly in multiple rewrite passes, the main challenge for the implementation was to identify how to properly instrument this AST with the appropriate smoke tests.

Concretely, after introducing the necessary background in Section 2, this paper first investigates which smoke test categories are relevant for VerCors in Section 3. Then Section 4 discusses the actual implementation using AST rewrites. Finally, in Section 5 the implementation is evaluated on a large number of system tests to ensure it properly implements the desired smoke tests. Moreover, we also evaluate the overhead on the verification performance of the various smoke tests. We end with related work and conclusions in Sections 6 and 7, respectively.

2 Background

VerCors [1] is a deductive verifier for concurrent and parallel programs, written in languages such as Java, C, OpenCL, and PVL (its internal prototypal verification language). Specifications are written as pre-post style contracts, using permission-based separation logic [3] as property specification language. The details of the VerCors specification language are irrelevant for the work described in this paper, for more information we refer the interested reader to the VerCors tutorial [11].

VerCors verifies an annotated program in a number of steps, as shown in Fig. 1: (1) the frontend parser produces an Abstract Syntax Tree (AST) from the input source; (2) a large number of *rewrite passes* progressively transform this AST into the Viper intermediate verification language [8]; (3) one of Viper's backends (Silicon or Carbon) generates verification conditions; and (4) the Z3 SMT solver [7] tries to prove the resulting verification conditions. The rewrite passes of VerCors use pattern matching on the relevant AST nodes, and instrument or rewrite them, leaving everything else unchanged for the next pass.

Ultimately, all properties that need to be verified get translated into the basic statements that Viper uses to track and query its internal prover state. The statements **inhale** and **exhale** operate on the full heap state, so they can transfer permissions from and to the heap, as well as assert pure facts, while **assume** and **assert** operate only on pure, non-heap facts and cannot transfer permissions. The operations **inhale** and **assume** add a fact without requiring a proof, while **exhale** and **assert** verify that the given expression holds. Notice that **assert** does so without modifying the state, since it carries no permissions to transfer, while **exhale** removes the asserted permissions from the heap once it succeeds, so the state changes. For example, a **requires** clause such as `x >= 0` is *inhaled* on entry, and an **ensures** clause such as `\result >= x` is *exhaled* at the return, checking that it already follows from the inhaled state. Finally,

refute `expr` is the dual of **assert** with the verdict flipped: it succeeds when `expr` is *not* provable, and fails when `expr` can be proven. This flipped behaviour makes **refute** a convenient statement to implement smoke tests.

3 Smoke Tests Overview

As mentioned above, Blanchard et al. [2] identified 5 smoke test categories. The first category, inconsistent preconditions, was already covered by VerCors. For the other categories, we investigated whether they were relevant for VerCors. Moreover, we also identified additional places where the smoke tests had to be inserted, because VerCors, compared to Frama-C, supports several additional programming language constructs such as locks, parallel blocks and atomic instructions. Below we describe the different smoke test categories that we added to VerCors[1].

3.1 Dead Code

The first category of smoke tests that we added to VerCors targets dead code, including code that becomes unreachable because of its specification. Blanchard et al. [2] illustrate this with a function whose defensive null-check is excluded by its own precondition, `\valid(x)`, which states that `x` is a valid, non-null pointer (see Listing 1).

Listing 1: Defensive code masked by a precondition

```
1 /*@ requires \valid(x); // inhales x is non-null
2     ensures \result == *x; */
3 int read(int *x) {
4   if (!x) return 42; // unreachable branch
5   ...
6 }
```

VerCors has precisely the same behaviour, which can be caused by *e.g.*, branches guarded by conflicting conditions, and parallel code blocks with contradicting block contracts. The dead-code regions checks that we implement in VerCors extend beyond Frama-C/WP's scope, covering branch bodies, loop bodies, post-loop continuations, switch cases, dead code following a state-narrowing **assume**/**inhale**, parallel block bodies, and atomic block bodies.

3.2 Postcondition Contradictions

Another cause of vacuous verification happens when a contract of an abstract function (without body) contains an inconsistent postcondition. Every caller of

[1] The last category of smoke tests identified by Blanchard et al. is related to Frama-C's `exits` clause. As VerCors has no equivalent clause, we do not consider it here.

the abstract function will assume this postcondition, while it is never verified. An example of such a case is shown in Listing 2. A similar issue can occur when code is verified that calls another method, whose postcondition is never checked because the method never terminates.

Listing 2: An abstract method's contradictory postcondition is trusted.

```
1 ensures \result > 0 && \result < 0;
2 int g(int x); //abstract: postcondition not verified
3
4 ensures \result != 0;
5 int f(int x) {
6   return g(x);
7 }
```

3.3 Invariant Satisfiability

As mentioned by Blanchard et al. [2], loop specifications can also cause other forms of vacuous specifications, which require care to be detected. A loop invariant must hold before the loop is entered, at the end of every iteration, and after exit. Thus, if a loop invariant is inconsistent, this leads to a verification failure upon loop entry. However, this verification failure can also be caused by the loop context not establishing the loop invariant. Therefore, it is important to have a smoke test that detects inconsistent loop invariants, so the two situations can be distinguished.

VerCors has three invariant specification constructs: *loop invariants*, checked at every loop iteration; *lock invariants*, inhaled into a thread's heap on every lock acquisition; and *atomic block invariants*, which guard access inside an `atomic` block. In all three cases, inconsistencies in the specifications can lead to vacuous verification: Listing 3 shows code with a self-contradictory lock invariant, which therefore verifies vacuously[2].

Listing 3: A self-contradictory lock invariant.

```
1 lock_invariant flag > 0 && flag < 0;
2 class C {
3   int flag;
4   requires committed(this);
5   void method() {
6     lock this; // inconsistent lock invariant is assumed
7     // critical section code
8     unlock this;
9   } }
```

[2] Note that also in this case, the inconsistency would make the commit statement where the lock invariant is initialised fail to verify, but again the user would not know whether this is caused by an inconsistency, or by insufficient information in the context to establish the invariant.

4 Smoke Test Implementation

This section discusses how the various smoke tests are added to VerCors. In Frama-C, the places to insert the smoke tests were computed using the control-flow graph (CFG) that the tool internally generates. VerCors does not compute a CFG, so this implementation approach could not be reused directly. Instead, our smoke test implementation instruments the AST directly, with one rewrite pass per smoke test, using two forms of instrumentation.

- **Direct instrumentation**: insert a **refute** `false` node into the existing tree.
- **Synthetic method generation**: build a separate method that inhales the relevant clause and ends with **refute** `false` for an isolated proof context.

To make it clear to the user at what program point the specification becomes inconsistent and/or the code becomes dead, and to not overwhelm the user with redundant warnings we implement *cascade suppression*. Cascade suppression is implemented by establishing relationships between smoke tests such that if the parent test fails then the child's warning will be suppressed.

As mentioned above, VerCors already implemented the precondition satisfiability check. This is implemented by generating a synthetic method with the precondition as its **requires** clause and **refute** `false` as its body. If this **refute** statement fails, it means that the precondition contains a contradiction.

4.1 Dead Code Detection

Unlike the precondition satisfiability check, dead code detection must be done via direct instrumentation, because reachability depends on the exact prover state accumulated along the execution path to the check point. The taken approach inserts **refute** `false` at the entry of each potential dead-code region. If `false` is not provable, then the region is reachable, and the **refute** passes silently. If `false` is provable, then the region is unreachable, the **refute** fires, and VerCors emits a `deadBranch` warning pointing to the exact source location. Notice that the use of **refute** is essential here. If we use **assert** `false`, this would result in a verification error inside every live branch.

The dead code detection rewrite pass uses pattern matching to instrument method bodies. We use two different instrumentation actions for detecting dead code: prepending a smoke test at the entry point of a code region and appending a smoke test after a state-narrowing statement. To each inserted smoke test we attach a list of previously inserted smoke tests that should suppress this smoke test's warning if they already fired. Our implementation of dead-code detection covers branch bodies, loop bodies, post-loop continuations, switch cases, code after a state-narrowing assumes and inhales, lock statements, parallel block bodies, and atomic block bodies. The pass does not instrument pure functions, since their bodies are single expressions without state changes or control-flow transfers, and therefore they cannot contain dead code.

There are cases where the user intentionally makes part of the code dead using an **assume** `false` or **inhale** `false` statement, usually to narrow the branches

that must be checked while debugging. We warn the user when these statements appear in the code even in cases where smoke tests would usually be suppressed. Additionally, if a part of the code is expected to be dead, the user may insert an **assert** `false` statement to be sure of this. In that case the pass marks the rest of the region as intentionally dead code, so the pass stops inserting checks there entirely, and it silently passes.

Listing 4 shows how the **refute** `false` statements (in green) are added as the result of the rewrite pass applied to a method covering several of these dead-code cases at once. Verifying this method produces an `assertFailed:false` from the misplaced **assert** `false` (line 5), plus two `deadBranch` warnings: one for the else-branch (line 8) and one for **inhale** `false` (line 9). All other `deadBranch` warnings, i.e. those from lines 11 and 13, are suppressed.

Notice that we have one fundamental limitation: if we have dead code due to for example a `return` statement, we cannot catch this with our approach. As we do not compute a CFG, we are not able to catch those cases.

Listing 4: Instrumented code after dead code detection.

```
1  requires x > 0;
2  void dead(int x) {
3    if (x > 0) {
4      refute false;
5      assert false;
6      x = x + 100;
7    } else {
8      refute false;
9      inhale false;
10     while (x < 10) {
11       refute false;
12       x = x + 1; }
13     refute false; }}
```

4.2 Postcondition Satisfiability Checks

The postcondition satisfiability check uses the same approach as for the preconditions: for every contract whose postcondition is not trivially `true` or `false`, we generate a synthetic method. This synthetic method takes the original's parameters plus a stand-in variable for **result**. Since the postcondition might contain `\old` or depend on non-heap conditions from the precondition, the generated method must also have the same precondition. The body of the synthetic method then inhales the postcondition, followed by a **refute** `false`.

The example in Listing 5 illustrates that it would be incorrect to ignore the preconditions, and just generate a method that requires the postcondition (as we did in the precondition satisfiability check). If we were to generate a method where the postcondition is assumed as a precondition then we cannot use `\old` to refer to the precondition context and non-heap-related assumptions from the precondition might be missing, which can make the expression not well-defined. See Listing 6 for the generated method to check the postcondition of `addPos`.

Listing 5: Callee needing precondition context

```
1  class Counter { int i; }
2
3  requires n > 0;
4  pure boolean isPos(int n) = n > 0;
5  requires n > 0;
6  requires Perm(c.i, write);
```

```
7  ensures isPos(n); //this call requires n > 0;
8  ensures Perm(c.i, write) ** c.i == \old(c.i) + \result;
9  int addPos(int n, Counter c) { c.i = c.i + n; return n; }
```

Listing 6: Generated postcondition check

```
1  requires n > 0 ** Perm(c.i, write);
2  void checkPostSat_addPos(int n, boolean result) {
3    // Exhale the precondition to empty the heap
4    exhale n > 0 ** Perm(c.i, write);
5    inhale isPos(n) ** Perm(c.i, write) ** c.i==\old(c.i)+result;
6    refute false; // Check for a contradiction
7  }
```

4.3 Invariant Satisfiability Checks

To detect unsatisfiable invariants, we need to test satisfiability of the invariant formula, independent of the calling context. This requires a fresh isolated proof context rather than an inline check, so we generate synthetic methods for this. All three invariant checks share the same pattern: generate an isolated method that inhales the invariant and then has a **refute** `false`.

Lock invariants differ from loop/block invariants in how and where this method is generated. For lock invariants the method is generated directly at class level, making the check independent of whether the lock is ever acquired. Loop and block invariants may refer to local variables. Since the check is performed in its own standalone method, it loses direct access to the method's own local variables. We solve this by threading those variables through as parameters of the generated method.

Loop and block invariants require two additional measures that lock invariants do not, because they are method-scoped and more expressive. Since they might contain `\old` or depend on non-heap conditions from the precondition, the generated method must also have the same precondition. Additionally, well-definedness errors are silenced to avoid duplicating an error. A well-definedness error occurs when an expression is evaluated without the permissions or preconditions it depends on already holding. VerCors already checks the well-definedness of invariants, so the duplicate error is suppressed.

VerCors has a labelled `\old[label](e)` expression which refers to the heap at a specific point in a method. However, the satisfiability check wraps the invariant in an isolated proof context. Since that proof context does not contain the labelled heap we do not know what permissions are available at that program point while performing the transformation pass. Therefore, we do not generate smoke tests for invariants that contain labelled `\old` expressions.

Firing an invariant satisfiability check can also suppress body and post-loop dead code errors, as shown in Listing 7.

Listing 7: Cascade suppression for an unsatisfiable loop invariant.

```
1 loop_invariant x > 0 && x < 0; //WARNING
2 while (x < 10) {
3   x = x + 1; //dead, suppressed
4 }
5 x = x - 1;  //post-loop check - suppressed
```

5 Testing and Performance Evaluation

We test the correctness of our smoke test implementation, and evaluated its impact on the performance of VerCors.

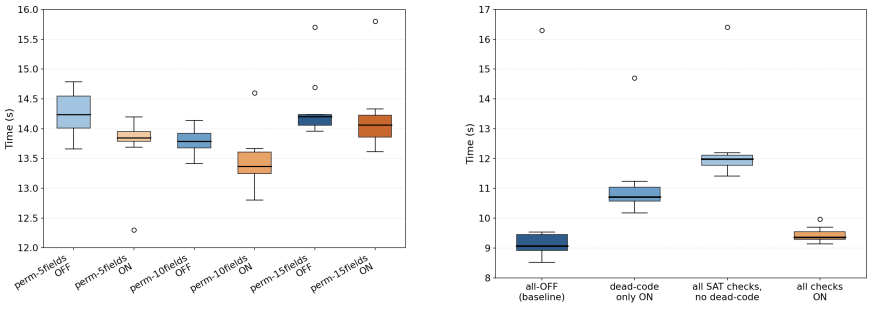
5.1 System Testing Approach

To test the correctness of our implementation, we ran the implementation on an extensive test suite. Each test is a complete PVL, Java, or C program run end-to-end through the full pipeline — parsing, rewrite passes, Viper, Z3 — with only the final verification outcome inspected. In total, the suite contains 128 tests. We built the test suite in a systematic manner: every smoke-test kind has its own group of tests, each containing both positive cases (a checker must fire) and negative cases (a checker must stay silent), so both missed detections and false positives are caught, see Table 1 in the appendix. Within each category, there are concrete scenarios: different contract shapes, nesting depths, combinations of assumptions, permissions and so on. This ensures that findings are not specific to one narrow program pattern. The suite includes several examples from Blanchard et al. [2], such as Listing 1, confirming VerCors's checks fire on the same inconsistencies as the Frama-C implementation.

5.2 Performance Evaluation

We also measure the overhead on performance caused by inserting the smoke tests. Overhead is measured as the increase in wall-clock verification time when the passes are enabled relative to a baseline with all passes disabled. Because JVM startup and parser caches alone account for 12–15 seconds of wall-clock time on short programs, backend-only time is extracted via the `--dev-time-backend` flag to isolate the cost of the passes themselves.

The benchmark suite contains 81 programs across eleven categories and is structured in two layers. First, each checker is evaluated individually using synthetic programs that each target a single feature, such as sequential and nested loops, branch-heavy methods, parallel blocks, lock invariants, or postconditions. These programs are parametrised by size to observe how overhead scales as programs grow. Second, all checkers are enabled together on 4 realistic combined programs that mix loops, branches, lock invariants, and postconditions, reflecting how the tool is used in practice. All experiments were run on an Intel Core Ultra



(a) For lock invariants (b) On an event-dispatcher program

Fig. 2: Backend verification time measurements (n=10 runs per configuration)

7 258V with 16 GB of RAM, running Ubuntu 24.04 under WSL2 on Windows, with OpenJDK 17.0.18.

The evaluation shows a clear split: dead-code checking is the main source of overhead and can cause severe slowdowns under certain patterns. In particular, if a method contains many branches, then the dead code check will instrument each branch entry with a **refute** `false`, causing a significant slowdown. However, we observed that the extra verification time does not grow linearly with the extra traces that need to be checked, as Silicon reuses proof states whenever possible.

Each of the four satisfiability checks was stress-tested individually: post-condition checks were run on examples ranging from simple returns to `forall` quantifiers and 30-method classes; invariant satisfiability checks were used on loop and par-block invariants with arithmetic conjuncts, nested `forall` expressions, function-call chains up to depth 3, and 1–10 field permissions, and on lock invariants with 5–15 field permissions across five classes and five call sites; and precondition smoke tests were run on the same programs' structured preconditions. Each checker was tested on matching satisfiable/unsatisfiable pairs. None produced a consistent overhead above the measurement noise floor of roughly ± 1 s. For example, Fig. 2a shows the lock invariant results: across 5, 10, and 15 field permissions the ON and OFF distributions overlap entirely, with no systematic shift in either direction. Several ON configurations ran marginally faster than their OFF baseline. In practice, all four SAT checks can be left enabled without measurable cost, even as program complexity scales.

The most surprising result was that running all checks at the same time costs *less* than running them one by one (Fig. 2b). Four real size programs were tested on this. Figure 2b shows the impact of the smoke tests on the verification time of one these programs. They all showed similar patterns. The exact cause requires further investigation, but our current hypothesis is that this is caused *proof witness sharing*. The dead-code check and the SAT-based checks both ask Z3 whether the same path conditions are satisfiable. A satisfying assignment found while proving one may serve directly as a witness for the other, so Z3 effectively solves the shared problem once rather than twice. Practically, the

combined overhead is bounded by the most expensive individual check rather than their sum, making it safe to leave all checks enabled in practice.

6 Related Work

As mentioned above, the smoke tests implemented in this paper are inspired by the work on smoke tests for Frama-C [2]. Other deductive verifiers also implemented multiple smoke tests.

OpenJML [4], a verifier for Java programs annotated with JML, as far as we know takes a similar approach to Frama-C's implementation. It checks a method's contract first, then, if that succeeds, runs a second pass confirming that the method's exit and each individual assertion can be reached.

Dafny is a programming language with verification built directly into it, rather than having it added as an external annotation layer. It detects the vacuous specifications passively: it inspects which assumptions the SMT solver actually used to close a proof, and if the goal itself was never needed, the proof relied on a contradiction [10].

For the VerCors verifier itself, we recently added support to prove consistency of axiomatic extensions [9]. This allows developers to add support for new axiomatic data types, without accidentally introducing inconsistencies. In contrast with the approach in this paper, which is targeting mistakes by the specification writers, those consistency proofs support the tool developers.

7 Conclusions

This paper showed how several smoke tests that were earlier defined for Frama-C have been implemented for VerCors. VerCors already checked for precondition satisfiability, but now it also checks for dead code, postcondition satisfiability and invariant satisfiability (for loop, lock, and atomic-block invariants). We discussed how the implementation was adjusted to VerCors, where each smoke test is implemented by a single rewrite pass, instrumenting the internal AST. The implementation was evaluated in a systematic manner. We also measured performance overhead, where we saw that dead-code detection can add exponential overhead when independent branches are written as separate `if` statements rather than as a chain[3], while the satisfiability checks add no measurable cost individually, and even less when run together.

All evaluation here used Viper's Silicon backend exclusively. It is future work to check whether the findings on performance overhead are the same when we use the Carbon backend.

[3] The effects of different branch structures are discussed more thoroughly in the thesis this paper is based on: https://purl.utwente.nl/essays/110823

References

1. Armborst, L., Bos, P., van den Haak, L.B., Huisman, M., Rubbens, R., Sakar, Ö., Tasche, P.: The vercors verifier: A progress report. In: Gurfinkel, A., Ganesh, V. (eds.) Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part II. Lecture Notes in Computer Science, vol. 14682, pp. 3–18. Springer (2024), https://doi.org/10.1007/978-3-031-65630-9_1
2. Blanchard, A., Correnson, L., Djoudi, A., Kosmatov, N.: No smoke without fire: Detecting specification inconsistencies with Frama-C/WP. In: Huisman, M., Howar, F. (eds.) Tests and Proofs - 18th International Conference, TAP 2024, co-located with FM 2024, Milan, Italy, September 2024, Proceedings. Lecture Notes in Computer Science, vol. 15153, pp. 65–83. Springer (2024). https://doi.org/10.1007/978-3-031-72044-4_4
3. Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.J.: Permission accounting in separation logic. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 259–270. POPL '05, Association for Computing Machinery, New York, NY, USA (2005). https://doi.org/10.1145/1040305.1040327
4. Cok, D.R.: OpenJML: JML for Java 7 by extending OpenJDK. In: Bobaru, M.G., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6617, pp. 472–479. Springer (2011). https://doi.org/10.1007/978-3-642-20398-5_35
5. Hähnle, R., Huisman, M.: Deductive software verification: From pen-and-paper proofs to industrial tools. In: Steffen, B., Woeginger, G.J. (eds.) Computing and Software Science - State of the Art and Perspectives, Lecture Notes in Computer Science, vol. 10000, pp. 345–373. Springer (2019). https://doi.org/10.1007/978-3-319-91908-9_18
6. Kosmatov, N., Prevosto, V., Signoles, J. (eds.): Guide to Software Verification with Frama-C. Computer Science Foundations and Applied Logic, Springer Cham (2024). https://doi.org/10.1007/978-3-031-55608-1
7. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
8. Müller, P., Schwerhoff, M., Summers, A.J.: Viper - a verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) Verification, Model Checking, and Abstract Interpretation. VMCAI. Springer Berlin Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_2
9. Putti, E., Stekelenburg, A.: Consistency proofs for axiomatic extensions: translating VerCors's axiomatic data types into Isabelle. In: Formal Methods for Industrial Critical Systems (FMICS 2026) (2026), to appear
10. Tomb, A.: Identifying specification problems in dafny programs. Dafny Blog (2023), https://dafny.org/blog/2023/10/27/proof-dependencies/, accessed June 2026
11. VerCors team: The VerCors verifier tutorial, https://vercors.ewi.utwente.nl/wiki/

A Description of Test Suite

Below we indicate the different system test categories for the smoke-test suite. There were 128 tests in total.

Table 1: Test suite categories and descriptions

Category	**What is tested**
Dead code — branch conditions (10)	Branches made unreachable by contradictory or self-contradicting guard conditions
Dead code — state narrowing (11)	Code made unreachable by a preceding **assume** or **inhale** statement that narrows the symbolic state to a contradiction
Dead code — parallel blocks (5)	Parallel block bodies made unreachable by an empty iteration range or a contradictory **context**/**context_everywhere** clause
Dead code — atomic blocks (5)	Atomic block branches and bodies made unreachable by the inhaled invariant, including re-entrant permission duplication
Post-loop dead code (10)	Code after a loop made unreachable by the loop invariant combined with the negated exit condition
Loop invariant satisfiability (15)	Unsatisfiable loop invariants, including those containing function calls and permission clauses
Block (parallel) invariant satisfiability (5)	Unsatisfiable `par` block invariants, including permission-duplicating and nested cases
Lock invariant satisfiability (10)	Unsatisfiable intrinsic lock invariants, independent of lock/unlock usage, and lock/unlock sites made dead by a satisfiable invariant
Postcondition satisfiability (31)	Self-contradicting postconditions, including function calls, permissions, and `\forall` quantifiers
Combined scenarios (6)	Programs where multiple checkers fire simultaneously
Cascade suppression (12)	Nested dead regions where only the outermost cause is reported
Intentional assert false (8)	Deliberate unreachable markers that must not produce spurious `deadBranch` warnings