RESEARCH ARTICLE A failed proof can yield a useful test

Li Huang | Bertrand Meyer

Chair of Software Engineering, Constructor Institute, Schaffhausen, Switzerland

Correspondence

Li Huang, Chair of Software Engineering, Constructor Institute, Rheinweg 9, 8200 Schaffhausen, Switzerland. Email: li.huang@sit.org A successful automated program proof is, in software verification, the ultimate triumph. In practice, however, the road to such success is paved with many failed proof attempts. Unlike a failed test, which provides concrete evidence of an actual bug in the program, a failed proof leaves the programmer in the dark. Can we instead learn something useful from it?

The work reported here takes advantage of the rich information that some automatic provers internally collect about the program when attempting a proof. If the proof fails, the Proof2Test tool presented in this article uses the counterexample generated by the prover (specifically, the SMT solver underlying the Boogie tool used in the AutoProof system to perform correctness proofs of contract-equipped Eiffel programs) to produce a failed test, which provides the programmer with immediately exploitable information to correct the program. The discussion presents Proof2Test and the application of the ideas and tool to a collection of representative examples.

KEYWORDS:

Program verification, Eiffel, AutoProof, AutoTest, Counterexample

1 | INTRODUCTION

We usually seek success, but failures can be useful too. In software verification, for example, a failed *test* provides the programmer with concrete, immediately actionable information about a bug, helping to fix it.

A failed attempt at a *proof*, on the other hand, seems at first not to help much; it does not even demonstrate that there is a bug (and even less what that bug might be), as it could simply reflect a problem with the proof process.

The work reported here takes advantage of the power of a modern prover to make a failed proof useful too, by deriving a failed test from it. Examples handled so far seem to confirm the potential of the approach.

1.1 | Tests and proofs

This work combines tests and proofs, a combination whose benefits was not always self-evident. In fact, tests and proofs have long been considered the warring siblings of software verification. They are characterized by dual benefits and limitations:

- Tests have, in their favor, their concreteness and relative ease of preparation. They face, however, a fundamental limitation: successful tests, regardless of how many of them, do not guarantee the correctness of software.
- Proofs do hold that promise of correctness, *when they succeed*. In practice, however, the path towards a proof is arduous, involving many intermediate steps at which the proof fails and it is hard to know what needs to be done, such as changing the program, changing the specification, or finding a way to overcome a limitation of the proof tool.

	Proof	Test
Success	🖒 Program is correct	Program works for <i>one</i> more case
Failure	\mathbf{v} Inconclusive, don't know what's wrong	Derogram is incorrect

TABLE 1 The different meanings of failure and success for proofs and for tests

Can the two techniques help each other by combining their respective advantages? This article proposes a specific answer, exploiting the observation that *failure* has different consequences for proofs and for tests. Table 1 shows the difference. D denotes the interesting cases, those in which we can conclusively deduce a useful property:

- Top-left entry: proof succeeds (great news, the program is correct).
- Bottom-right entry: test fails (program is incorrect, we have found a bug, less triumphal news but still concrete and useful).

The other two cases, marked \mathbb{Q} , are disappointing since they offer little practical value:

- A successful test (top right) does not tell us that the program is correct in the general case. It is just one drop of correctness in the ocean of possible cases.
- A failed proof (bottom left) does not tell us whether the problem is with the program, the specification, or the limitations of the proof tool itself.

This last case, a failed proof, shaded in the table, is particularly frustrating. While a mechanically-supported program proof is the most exciting prospect in the search for program correctness, the daily practice of attempting to prove programs consists of getting notices of proof failure and attempting to remove the cause for the failure. It is not so different psychologically from the practice of *debugging* a program, except that it is a static rather than dynamic form of debugging, using only the program text and no executions.

Being static, this process lacks the immediacy and concreteness of a test, which (when it fails, as in the bottom-right entry) gives the programmer a direct example of an input that causes the program to violate its specification. Programmers can exploit such a test case as an immediately understandable indication of what needs to be fixed. In contrast, a failed proof just fails, giving you little insight as to what is actually wrong — without even guaranteeing that the fault lies with the program (as noted, it could be in the specification, or we might just have hit a limitation of the proof technology). The downside is that the tester (a term meant here to denote a person, not a tool) has to devise the failing test case, often a difficult endeavor for a program that has already achieved a first level of approximate correctness.

Proof tools, on the other hand, accumulate considerable knowledge about the program, including when they fail to prove it. The work reported in this article consists of using that knowledge to go automatically from the disappointing bottom-left case to the directly useful bottom-right case, a failed test, with its exploitable example of program incorrectness.

1.2 | Proof2Test

The tool presented in this article, Proof2Test, automatically generates failing tests from failed proofs. The underlying prover is AutoProof^[1,2], the verifier for checking functional correctness of Eiffel^[3] programs using axiomatic (Hoare-style) semantics. The functional correctness of Eiffel programs is specified by *contracts* (such as pre- and postconditions, class and loop invariants). When a proof fails, AutoProof's back-end prover (the Z3 solver^[4], used through Boogie^[5,6]) will produce a counterexample. Proof2Test exploits the information in the counterexample to generate a concrete test case. The generated test case is directly exploitable and executable in AutoTest^[7,8], a testing tool for Eiffel programs based on run-time assertion checking.

Since AutoProof and AutoTest use the same contracts for program analysis and they are both integrated in the same development environment (Eiffel IDE), executing the test produced by Proof2Test from the failed proof can reproduce an analogous test failure in AutoTest: the execution will raise the run-time violation of the same failed contract. The general aim is to combine the benefits of both static and dynamic forms of verification: proofs (performed by AutoProof) and tests (performed by AutoTest).

The source code for Proof2Test and examples, as well as complementary material, are available in a GitHub repository¹. The repository contains a detailed readme.md file explaining how to run the examples and reproduce our results (or try new examples).

1.3 | Structure of the discussion

To give the reader a general understanding of the approach before introducing the theory, Section 2 illustrates an example session of using Proof2Test. Section 3 introduces the fundamental technologies used in the Proof2Test verification framework. Section 4 describes the details of the implementation of Proof2Test. Section 5 demonstrates how Proof2Test helps diagnose failed proofs. Section 6 evaluates the applicability of Proof2Test through a series of examples. After a review of related work in Section 7, Section 8 describes limitations of the current state of the work and Section 9 concludes an analysis of limitations and ongoing work to overcome them.

For readers interested in detailed results, the Appendix covers the full data for two of the sample classes; it is extracted from a technical report^[9], complementing the present article by covering all samples.

2 | AN EXAMPLE SESSION WITH PROOF2TEST

Before exploring the principles and technology behind the Proof2Test tool, we look at the practical use of the tool on a representative example (Figure 1).

The intent of the max function in class MAX_IN_ARRAY is to return into Result the maximum element of an integer array a of size a.count. The two postcondition clauses in lines 26 and 27 (labeled *result_is_maximum* and *result_in_array*) specify this intent: every element of the array should be less than or equal to Result; and at least one element should be equal to Result.

When we try to verify the max function using AutoProof, verification fails and AutoProof returns the error message "Postcondition *result_is_maximum* (line 26) might be violated". Such a generic message tells us that the prover cannot establish the postcondition, but does not enable us to find out why. We are left with hypotheses, including: incorrect loop initialization (line 10); incorrect loop exit condition (line 16); incorrect instruction in the loop body (lines 18 - 21); incorrect loop invariant (lines 12 - 14); or limitations of the prover itself (in other words, it might be that the program is correct and AutoProof is unable to prove it).

What a programmer typically would like to see in such a case is not a general negative result, stating that the prover cannot establish a property, but a concrete, specific result, showing that a certain input breaks the specification. In other words, a failing test. Producing such a test is the purpose of Proof2Test. Figure 2 shows the generated test, which includes the following sequence of steps:

- Create an instance current_object of class MAX_IN_ARRAY (line 7).
- Create an integer array a and fill it with values 0 at position 1 and 8 at index 2 (lines 8 9).
- Call the erroneous function max on current_object with a as argument (line 10).

Running this Proof2Test-generated test in AutoTest (the test framework for Eiffel) leads to a run-time failure where the problematic postcondition *result_is_maximum* is violated. To explore the reason, we may step through the call current_object.max (a):

- 1. The precondition (line 5 in Figure 1) evaluates to True with a.count = 2.
- 2. After the loop initialization (line 10), Result = 0 and i = 2.
- 3. All the loop invariants (lines 12 14) are evaluated as True with i = 2, a.count = 2, a[1] = 0, a[2] = 8, Result = 0.
- 4. The exit condition of the loop (line 16) evaluates to True with a.count = 2 and i = 2, forcing the loop to terminate.
- 5. Postcondition *result_is_maximum* (line 26) evaluates to False with a [1] = 0, a [2] = 8 and Result = 0, which causes AutoTest to report a run-time exception of postcondition violation.

¹https://github.com/huangl223/Proof2Test

4

```
1
    class MAX_IN_ARRAY
 2
    feature
 3
         max (a: SIMPLE_ARRAY [INTEGER]): INTEGER
 4
                 require
 5
                         a.count > 0
 6
                 local
 7
                         i: INTEGER
 8
                 do
 9
                         from
10
                             Result := a [1]; i := 2
11
                         invariant
12
                             2 \le i and i \le a.count + 1
13
                             \forall j: 1 |..| (i - 1) | a [j] \le \text{Result}
14
                             \exists j: 1 | ... | (i - 1) | a [j] = Result
15
                         until
16
                              i = a.count
17
                        loop
18
                              if a [i] > Result then
19
                                     Result := a [i]
20
                              end
21
                             i:= i + 1
22
                         variant
23
                              a.count - i
24
                         end
25
                 ensure
26
                         result_is_maximum: ∀ j: 1 |..| a.count | a [j] <= Result
27
                         result_in_array: \exists j: 1 | .. | a.count | a [j] = Result
28
                 end
29
    end
```

FIGURE 1 MAX_IN_ARRAY is a class with a function max which finds the maximum element of an integer array a; the type of a is SIMPLE_ARRAY, a library class providing simple array features

This execution of the generated test reveals the problem: on the array created by the test, which holds its maximum value (8) in the last element (at index 2 in a), the loop terminates too early, preventing the program from getting to that value.

This scenario and its result provide the programmer with immediate evidence (not available from the failed proof attempt) of what is wrong with the program version of Figure 1. Any competent programmer will see right away what needs to be done to eliminate the error: update the exit condition i = a.count (line 16) to permit one more loop iteration, changing it to i > a.count.

This example illustrates how tests and proofs can be complementary techniques: while a successful proof conclusively shows that the program satisfies the given specification, a failed proof does not by itself tell us what is wrong with the program; in that case a test can bring the concrete evidence making it possible to proceed with the development process. The remaining sections explain the technology behind this approach.

3 | TECHNOLOGY STACK

Fundamental technologies used in Proof2Test include the Eiffel programming language, its libraries, the EiffelStudio IDE (development environment including compiler and AutoTest test framework) and two analysis tools, AutoProof (static) and AutoTest (dynamic).

```
1
    test_MAX_IN_ARRAY_max_1
 2
        local
 3
            current_object: MAX_IN_ARRAY
            a: SIMPLE_ARRAY [INTEGER]
 4
 5
            max_result: INTEGER
 6
        do
 7
            create current_object.default_create
 8
            create a.make (0)
 9
            a.force (0, 1); a.force (8, 2)
10
            max_result := current_object.max (a)
11
        end
```

FIGURE 2 A test case of max with input argument a[1] = 0, a[2] = 8

3.1 | Eiffel

The Eiffel object-oriented design and programming language^[3,10] natively supports the Design by Contract^[11] methodology. An Eiffel program consists of a set of classes. A class represents a set of run-time objects characterized by the features available on them. Figure 3 shows an Eiffel class representing bank accounts, devised for demonstration purposes and intentionally seeded with errors.

A class contains two kinds of features: *attributes* that represent data items associated with instances of the class, such as balance and credit_limit; *routines* representing operations applicable to these instances, including make, available_amount, deposit, withdraw, transfer. Routines are further divided into procedures (routines that have no returned value) and functions (routines that return a value). Here, available_amount is a function returning an integer (denoted by a special variable Result), and the other routines are procedures.

A class becomes a *client* of ACCOUNT by declaring one or more entities (attributes, local variables, formal arguments...) of type ACCOUNT, as in a: ACCOUNT.

The value of an reference entity is void initially and can be attached to an object by using the create instruction or through assignment of a non-void value. For example, create a.make(-100) creates a new object whose credit_limit is -100 and attaches it to the entity a.

Programmers can specify the properties of classes and features by equipping them with contracts of the following kinds:

- Precondition (require): property that *clients* must satisfy whenever they call a routine; the precondition of make (line 6), for example, requires the value of limit to be negative or 0.
- Postcondition (ensure): property that the routine (the *supplier*) guarantees on routine exit, assuming the precondition and termination; for example, the postcondition of deposit states that the value of balance at the exit of the routine will be greater than its entry value (old balance).
- Class invariant (invariant): constraint that must be satisfied by instances of the class after object creation and after every qualified call x.r (args) to a routine of the class; for example, the class invariant of ACCOUNT (line 58) constrains the value of available_amount to be always non-negative.
- Loop invariant (invariant): property that the loop guarantees after initialization and every iteration; for example, the loop invariant in max (lines 12 14 in Figure 1) specifies what the loop has achieved (a part of the final goal) at an intermediate iteration.
- Loop variant (variant): an integer measure that remains non-negative and decreases at each loop iteration, ensuring that the loop eventually terminates; the loop variant of the loop in max is a.count i (line 23 in Figure 1).

Contracts embedded in the code allow programmers to open the way to both static analysis of the program, with AutoProof, and dynamic analysis based on run-time assertion checking, with AutoTest.

6

```
1
    class ACCOUNT create
 2
        make
 3
                                                              34
                                                                  withdraw (amount: INTEGER)
    feature
 4
        make (limit: INTEGER)
                                                              35
                                                                       -- Withdraw 'amount' from this bank account.
                 -- Initialize with credit limit 'limit'36
 5
                                                                       require
             require
                                                              37
 6
                                                                           \texttt{amount} \geq 0
                                                              38
                                                                           amount < available_amount
 7
                 limit \leq 0
 8
                                                              39
             do
                                                                       do
                                                              40
 9
                 balance := 0; credit_limit := limit
                                                                           balance := balance + amount
10
                                                              41
             ensure
                                                                       ensure
                 balance_set: balance = 0
                                                              42
                                                                           balance_set: balance = old balance - amount
11
                                                              43
12
                 credit_limit_set: credit_limit = limit
                                                                       end
                                                              44
13
        end
14
                                                              45
                                                                  transfer (amount: INTEGER; other: ACCOUNT)
                                                                       -- Transfer 'amount' to 'other'.
        balance: INTEGER
                                                              46
15
                                                              47
16
             -- Balance of this account.
                                                                       require
                                                              48
17
                                                                           amount \ge 0
        credit_limit: INTEGER
                                                              49
18
                                                                           amount \leq available_amount
             -- Credit limit of this account.
                                                              50
19
                                                                       do
                                                              51
20
                                                                           balance := balance - amount
        available_amount: INTEGER
                                                              52
21
                                                                           other.deposit (amount)
             -- Amount available on this account.
                                                              53
22
                                                                       ensure
23
                                                              54
                                                                           withdrawal_made: balance = old balance -
             do
24
                 Result := balance - credit_limit
                                                                                amount
                                                                           deposit_made: other.balance = old other.
25
             end
                                                              55
26
                                                                                balance + amount
27
        deposit (amount: INTEGER)
                                                              56
                                                                       end
             -- Deposit 'amount' into this account.
                                                              57
28
29
             do
                                                              58
                                                                  invariant
                                                              59
30
                 balance := balance + amount
                                                                       available_amount \geq 0
31
             ensure
                                                              60
                                                                  end
32
                 balance increased: balance > old balance
33
             end
```

FIGURE 3 A class (with bugs) intended to implement the behavior of bank accounts

3.2 | AutoTest

AutoTest^[7,8] is an automatic contract-based testing tool: it uses the contracts present in Eiffel programs as test oracles and monitors their validity during execution. A test case in AutoTest is an argument-less procedure which calls a "target routine" (the routine to be tested) in a certain context (arguments and other objects). The context can be created manually, as in traditional testing environments, but AutoTest can also automatically create it, using sophisticated algorithms. Figure 2 test_MAX_IN_ARRAY_max_1 shows a test case with target routine max (part of the "target class" MAX_IN_ARRAY). A test case usually consists of the following components:

- Declaration of input variables of the target routine.
- Creation of fresh objects for the declared variables.
- Instantiation of the objects with concrete values.
- Invocation of the target routine using the instantiated variables.

The execution of a test case in AutoTest performs run-time assertion checking for the target routine: AutoTest executes the target routine through a qualified call² to the routine; during the execution of the routine call, AutoTest evaluates each of the contracts in the routine; if any assertion is violated (evaluated to False), AutoTest terminates the execution and reports an exception of type "contract violation". The test leads to one of the following verdicts:

- 1. Passes (the execution of the test case satisfies all specifications).
- 2. Fails with a contract violation of the target routine.
- 3. Remains unresolved: the violation is not the target routine's fault. The most important case in this category occurs when AutoTest creates an input state that does not satisfy the target routine's precondition. (AutoTest includes optimization strategies to avoid this case, which teaches nothing about the correctness of the software, although it might suggest that the routine's precondition is too strict.)

AutoTest as described here is a test execution and management framework. AutoTest also includes automatic test case generation mechanisms, not used in the present work.

3.3 | AutoProof

AutoProof^[1,2] checks the correctness of Eiffel programs against their functional specifications (contracts). Verification in AutoProof covers a wide range of properties including whether:

- The precondition of a routine holds at the time of call.
- The postcondition of a routine holds on routine exit.
- The initialization of a loop ensures the invariant.
- That initialization leaves the variant non-negative.
- The body of a loop preserves the loop invariant (leaves it true if it was true before).
- That body leaves the variant non-negative (if it was non-negative before).
- It decreases the variant.
- A creation procedure (constructor) of a class produces an object that satisfies the class invariant.
- A qualified call x.f (...) finds the target object (the object attached to x) in a consistent state. AutoProof integrates two mechanisms to verify class invariants: *ownership*^[12] and *semantic collaboration*^[13]. A new approach^[14], avoiding any need for programmer annotations, is currently being implemented to replace them.

AutoProof relies on Boogie and Z3 as explained next.

3.4 | Boogie and Z3

When verifying an Eiffel program, AutoProof translates it into a Boogie program, then relies on the Boogie verifier to transform this program into verification conditions for the SMT solver (Z3). The underlying theory is Dijkstra's weakest-precondition calculus^[15].

The verification condition for a routine r of body b, precondition P and postcondition Q, is of the form

 $P \implies (b \operatorname{wp} Q) \qquad [VC]$

where b wp Q, per Dijkstra's calculus, is the *weakest precondition* of b for Q, meaning the weakest possible assertion such that b, started in a state satisfying P, will terminate in a state satisfying Q. The "weak/strong" terminology for logical formulas refers to implication; more precisely, saying that P' is weaker than or equal to P (and P stronger than to equal to P') simply

²As a reminder, a qualified call is of the form x.r(args), on a target object identified by x, as opposed to an unqualified call r(args) applying to the current object.

expresses that $P \Rightarrow P'$. (Equivalently, if we identify *P* and *P'* with the set of states that satisfy the respective assertions, then "*P* is stronger than *P'*" means $P \subset P'$.) Then the verification condition [VC] expresses that the precondition *P* of the routine is strong enough to guarantee that the routine's execution will yield the postcondition *Q* (since *P* is at least as strong as the weakest possible assertion, *b* wp *Q*, guaranteeing this result).

If property [VC] is satisfied at the Boogie/Z3 level, the Eiffel routine is correct (with respect to P and Q).

The technique used by an SMT solver to prove a property such as [VC] is indirect: the solver tries to *satisfy* (the "S" in "SMT") a given logical formula — or to determine that such an assignment does not exist. So the formula on which Z3 will work is not [VC] itself but its negation [NVC]:

 $\neg (P \implies (b \operatorname{wp} Q)) \qquad [NVC]$

The proof succeeds if *no variable assignment* satisfies [NVC]; in other words, it is impossible to falsify the verification condition. In the present work, we are interested in the case in which the proof does *not* succeed: the solver does find a variable assignment satisfying [NVC] and hence violating [VC]. The solver has authoritatively found that the program is buggy, by *disproving* [VC]:

it found a set of variable values that causes the routine not to produce a final state satisfying *Q*. If the goal is not only to give the programmer a simple success/failure notification about the proof, but to help the programmer in the failure case, it is important that such a disproof be *constructive*: it actually identifies a *failing test case*. That test case is not, however, directly usable by the programmer. Instead, it is buried in internal Z3 information and SMT-solving notation. It is the task of Proof2Test to extract the relevant information and turn the counterexample into an actual test case that the

programmer can understand, in terms of the original programming language (Eiffel), and run. Section 4 will present the details of the counterexample and test generation process in Proof2Test. Listing 1 shows how the information appears internally (slightly simplified for the purposes of presentation). The notation is

the SMT-LIB format^[16], common to Z3 and other SMT solvers. The goal is to express [NVC] on line 42. The previous lines define the environment in the form of declarations of types or "sorts" (lines 1-2), functions and constants³ (3-19), assertions expressing typing properties (20 - 23) and assertions expressing verification conditions (24 - 42). The final line, check-sat, directs the solver to check satisfaction of the conditions.

- 1 (declare-sort T@U 0)
- 2 (declare-sort T@T 0)
- 3 (declare-fun type (T@U) T@T)
- 4 (declare-fun type_of (T@U) T@U)
- 5 (declare-fun intType() T@T)
- 6 (declare-fun TypeType () T@T)
- 7 (declare-fun FieldType (T@T) T@T)
- 8 (declare-fun ACCOUNT () TQU)
- 9 (declare-fun balance () T@U)
- 10 (declare-fun credit_limit() T@U)
- 11 (declare-fun Current () TCU)
- 12 (declare-fun amount () Int)
- 13 (declare-fun Heap@O() T@U)
- 14 (declare-fun Heap@1() T@U)
- 15 (declare-fun Succ (T@U T@U) Bool)
- 16 (declare-fun value () Int)
- 17 (declare-fun Select (T@U T@U T@U) T@U)
- 18 (declare-fun Store (TQU TQU TQU TQU) TQU)
- 19 (declare-fun fun.available_amount (T@U T@U) Int)
- 20 (assert (= (type ACCOUNT) TypeType))
- 21 (assert (= (type balance) (FieldType intType)))
- 22 (assert (= (type credit_limit) (FieldType intType)))
- 23 (assert (= (type_of Current) ACCOUNT))
- 24 (assert (let (

³Constants are treated like functions with no arguments, declared, like other functions, in declare-fun clauses.

25	(available_amount (fun.available_amount Heap@O Current))
26	(old_balance (Select Heap@O Current balance))
27	(current_balance (Select Heap@1 Current balance))
28	(state_transition (Succ Heap@0 Heap@1))
29	<pre>(update_balance (= Heap@1 (Store Heap@0 Current balance value)))</pre>
30)
31	(let (
32	$(pre (and (\geq amount 0) (\leq amount available_amount)))$
33	$(wp (\Rightarrow$
34	(= value (+ old_balance amount))
35	(and
36	update_balance state_transition
37	(= current_balance (- old_balance amount))
38)
39)
40)
41)
42	$(not (\Rightarrow pre wp))$
43)
44)
45)
46	(check-sat)

Listing 1: SMT encodings for verifying balance_set

The SMT-LIB syntax is parenthesis-based in the Lisp style, meaning in particular that function application is in prefix form: (f x) denotes the result of applying a function f to an argument x, which in more usual notation would be written f(x). In particular:

- (\Rightarrow x y) means x \Rightarrow y (x implies y) in ordinary mathematical notation.
- (let a b c) (to be read as "let a be defined as b in c") has the value of c, with every occurrence of a replaced by b.

The final condition (not (\Rightarrow pre wp)) on line 42 (in more usual notation: \neg (pre \Rightarrow wp)) corresponds to [NVC], where wp is the the weakest precondition. The preceding lines build up auxiliary elements culminating in this definition:

- Lines 1 2 define the basic *sorts*. "Sort" is to the modeling language, SMT-LIB, what "type" is to the programming language being modeled; as a result, a sort can describe both values and types from that programming language. Two fundamental sorts used in models generated by Boogie are T@U (line 1), describing values in the target programming language, and T@T, describing types in that language. The 0 in both declarations indicate the arity (number of arguments, here zero).
- Line 3 specifies that the function type, which yields the type of a value, maps a value (T@U) to a type (T@T).
- The type_of function (line 4) relates an object reference to the object's class. For example: Current⁴ is of type ACCOUNT (line 23); amount, an integer argument of withdraw, is declared as an integer constant (line 12); balance and credit_limit are two integer fields and defined as *instances* of a composite type FieldType intType (lines 21 22).
- Modeling the execution semantics of an object-oriented program requires modeling the behavior of the *heap* (where the objects are stored) through a sequence of constants prefixed with Heap@. In this example, there are two states during execution of withdraw: before and after the assignment balance := balance + amount, represented by Heap@0 and Heap@1 (lines 13 14). Succ (line 15) specifies the relation between two successive states. The auxiliary variable value (line 16) will contain the intermediate result of the assignment the result of balance + amount (line 34).

⁴Current represents the active object in the current execution context, similar to this in Java.

- Select and Store (lines 17 18) are functions for retrieving and updating the values of data fields. Select H obj fd will give the value in field fd of object obj in heap H. Store H obj fd new is a new store obtained by replacing that field value by new.
- Line 32 defines the precondition pre, rephrased in SMT-LIB from the original Eiffel precondition clause (require).
- Lines 24 30 introduce auxiliary variables.
- Lines 33 40 define wp, as the weakest precondition of the routine *balance_set* for its given two-clause postcondition (36 37).

4 | PROOF2TEST IMPLEMENTATION

Based on the technologies described in the previous section, Proof2Test automatically generates tests from failed proofs. This section presents the overall workflow of Proof2Test, then the details of its implementation.

4.1 | Overview of the Proof2Test process

Figure 4 outlines the workflow of Proof2Test. The inputs are a Boogie program (ap.bpl), generated by AutoProof from the input Eiffel program, and an SMT model file (ce.model), containing counterexamples from the Z3 solver. The output is a test script (test.e) in the form of an Eiffel class. Proof2Test goes through three steps to construct a test case:

(1) Collect the relevant context information from the Boogie program, including names and types of the input arguments of the failed routine r, as well as the attributes of the classes involved in r.

2 Extract an input vector (a sequence of values of *r*'s input arguments) from a counterexample.

③ Write a test case to the output file test.e based on the extracted context information and input vector.



FIGURE 4 Proof2Test workflow

A proof failure in AutoProof corresponds to a contract violation for an erroneous routine of a class, which results in a counterexample. In the case of multiple proof failures, the resulting model file (ce.model) contains the same number of counterexamples. Proof2Test parses each counterexample and generates test cases respectively.

A **counterexample** is an execution trace (a sequence of program states) of the erroneous routine r, at the end of which the program reaches a failed state (violating a contract element of r). Proof2Test parses the model to obtain the trace's input vector, from which it produces a test case for AutoTest (Section 3.2).

4.2 | Extraction of input vector from counterexample

In the example of Figure 3, the routine withdraw is incorrect since its body does not ensure the postcondition *balance_set* (line 42). AutoProof consequently fails to prove the routine correct by obtaining, through Boogie, a counterexample from Z3. Figure 5 shows that counterexample (key parts only, slightly simplified).

1	$\texttt{amount} \rightarrow 1$	11	$\operatorname{Succ} \rightarrow \{$
2	$\texttt{Current} \rightarrow \texttt{T@U!val!18}$	12	$T@U!val!17 T@U!val!22 \rightarrow True $
3	Heap@O \rightarrow T@U!val!17	13	Store \rightarrow {
4	Heap@1 \rightarrow T@U!val!22	14	T@U!val!17 T@U!val!18 T@U!val!7 11 \rightarrow T@U!val!22 }
5	$ACCOUNT \rightarrow T@U!val!6$	15	$\texttt{Select} \rightarrow \{$
6	balance \rightarrow T@U!val!7	16	T@U!val!17 T@U!val!18 T@U!val!7 \rightarrow 10
7	$credit_limit \rightarrow T@U!val!8$	17	T@U!val!17 T@U!val!18 T@U!val!8 \rightarrow (- 20)
8	value $\rightarrow 11$	18	$\texttt{T@U!val!22 T@U!val!18 T@U!val!7 \rightarrow 11}$
9	$\texttt{type_of} \rightarrow \{$	19	$\texttt{fun.available_amount} \rightarrow \{$
10	$T@U!val!18 \rightarrow T@U!val!6$ }	20	$T@U!val!17 T@U!val!18 \rightarrow 30$

FIGURE 5 Counterexample of the proof failure of balance_set

In this example the path to a contract violation goes through only two states, called Heap@0 and Heap@1 (as defined in lines 3 and 4). The execution trace is not given explicitly but can be inferred from the differences between Heap@0 and Heap@1. Formally, the counterexample in Figure 5 is a sequence of definitions of name-value associations, in the form name \rightarrow value. Basic values, other than integers, are of the form T@U!val!n denoting an abstract location n. For example, the value of amount in the counterexample is 1 (line 1) and balance is stored at abstract location 7.

The initial and final heaps, Heap@0 and Heap@1, are associated (lines 3 and 4) with locations 17 and 22. To infer the execution trace, it suffices to look at the definitions starting with line 11, which show the differences between the contents of these two heaps by referring to T@U!val!17 and T@U!val!22.

The preceding lines define variable values (for example, balance, credit_limit).

For a function f of n arguments, the specification of the function's value in the counterexample takes the form

 $\mathtt{a}_1 \mathtt{a}_2 \dots \mathtt{a}_n \to \mathtt{x}$

 $\mathtt{b}_1 \ \mathtt{b}_2 \ ... \ \mathtt{b}_n \to \mathtt{y}$

•••

to mean that $f(a_1, a_2, ..., a_n) = x$ etc. For example lines 15 to 18 give the state of both the initial heap (T@U!val!17) and the final heap (T@U!val!22) by specifying the Select function. (As seen in Section 3.4, Select (H, obj, fd) is the value in field fd of object obj in heap H.) This specification yields the values, in either or both heaps, of the balance and credit_limit fields (T@U!val!7 and T@U!val!8) for the Current object (T@U!val!18). Lines 13 and 14 define a state change, in the form of an update of the Store function: change the balance field (T@U!val!7) of the current object (T@U!val!18) in Heap@O so that it will have the value 11 in Heap@1. In programming language terms this would be written just balance := 11 (where balance denotes Current.balance).

To construct the input vector of withdraw from the model, it suffices to obtain the following information:

- 1. The initial state of the Current object (the target object of the qualified call), which includes the initial values of the two data fields balance and credit_limit.
- 2. The value of the only argument amount.

On can obtain the value of amount directly (line 1), and the values of balance and credit_limit from their symbolic addresses and the Select function (16 - 17). The resulting input vector for withdraw is: Current.balance = 10, Current.credit_limit = -20, amount = 1.

4.3 | Construction of test cases based on input vectors

Once it has obtained the input vector, Proof2Test starts constructing a test case in the form of a qualified call, here current_object.withdraw(amount). The input vector contains the values of current_object and argument amount before the call.

Figure 6 shows the generated test case, which includes four components:

```
test_ACCOUNT_withdraw_1
 1
 2
      local
 3
            current_object: ACCOUNT; amount: INTEGER
 4
      do
 5
          create current_object.make(0)
 6
          {INTERNAL}.set_integer_field (balance, current_object, 10)
          {INTERNAL}.set_integer_field (credit_limit, current_object, (- 20))
 7
 8
          amount := 1
9
          current_object.withdraw(amount)
10
      end
```

FIGURE 6 Test case generated from the proof failure of withdraw

- 1. Local declaration of the target object (current_object) and input argument (amount) (line 3).
- 2. Creation of the target object (line 5).
- 3. Initialization of the target object and argument (lines 6 8). Since Eiffel is a strongly typed language, setting individual fields of arbitrary objects requires using low-level mechanisms from the library class INTERNAL, such as set_integer_field.
- 4. Qualified call to withdraw (line 9).

Alg	gorithm 1 Generate a single test case based on data collec	ted from a counterexample
1:	Input: target class c, target routine r, input vector v	
2:	Output: a test case t	
3:	<pre>t := name_of_test_case(c, r)</pre>	Construct the name for the test case
4:	t_d := declare_variable(current_object)	
5:	<pre>t_c := create_reference(current_object)</pre>	Declare and create current_object
6:	t_i := empty_string	
7:	across input arguments of r as a loop	
8:	$t_d := t_d + declare_variable(a)$	Declare each argument
9:	if a is a variable of primitive type then	
10:	<pre>value := get_value(a, v)</pre>	Get the value of a from v
11:	<pre>t_i := t_i + assignment(a, value)</pre>	Use assignments to instantiate primitive-type arguments
12:	else	
13:	<pre>t_c := t_c + creation_of_reference(a)</pre>	Create objects for reference arguments
14:	<pre>t_i := t_i + instantiate_reference(a, v)</pre>	Instantiate reference arguments according to v
15:	end	
16:	end	
17:	if r is a function then	
18:	<pre>res := name_of_result_variable(r)</pre>	
19:	$t_d := t_d + declare_variable(res)$	Construct a variable res to fetch r's result
20:	end	
21:	$t := t + t_d + t_c + t_i$	Combine the clauses of declaration, creation and instantiation
22:	<pre>t := t + call_routine(current_object, r, res)</pre>	Call r with its arguments
	-	-

The test generation process applies Algorithm 1. The result of the algorithm is the code of a test case t, which includes three components: declaration clauses (t_d), creation clauses (t_c) and instructions instantiating variables (t_i).

The algorithm loops over all the input arguments (lines 7 - 15) of r and instantiates each of them according to their types: arguments of primitive types (INTEGER, BOOLEAN, CHARACTER and REAL) are instantiated through direct assignments (line 10) to the corresponding values in the input vector (for example, the integer argument amount is instantiated as amount := 1); arguments of reference types are instantiated by applying the instantiate_reference procedure (see the details in Algorithm 2) based on the input vector. Finally, the algorithm builds a call of the target routine (lines 16 - 21) with the instantiated arguments.

Alg	orithm 2 instantiate_reference: instantiate a variable o	f reference type
1:	Input: object reference o, input vector v	
2:	Output: text t	
3:	if o has an instantiated alias then	
4:	alias := get_alias_of_reference(o)	
5:	<pre>t:=t+assignment(o, alias)</pre>	Construct an assignment o := alias
6:	elseif o is a variable of container type then	
7:	<pre>size := get_size_of_container(o, v)</pre>	Get o's size
8:	<pre>items := get_item_of_container(o, v)</pre>	Get o's elements
9:	from	
10:	i := 1	
11:	until	
12:	$i \leq size$	
13:	loop	
14:	<pre>t:=t+force_clause(o, items[i])</pre>	Use force routine to insert each of o's elements
15:	i := i + 1	
16:	end	
17:	else	
18:	across each field of o as f loop	
19:	if f is a variable of primitive type then	
20:	<pre>value := get_value(f, v)</pre>	
21:	<pre>t := set_field_clause(f, value)</pre>	Use set_type_field function to instantiate f
22:	else	
23:	<pre>t := instantiate_reference(f, v)</pre>	
24:	end	
25:	end	
26:	end	

Algorithm 2 instantiates an object reference o based on the input vector v:

- Checks whether o has an alias that has been instantiated earlier.
- If so, construct a direct assignment to the earliest alias (lines 4 5).
- If o is an object of a container type (such as ARRAY and SEQUENCE), instantiates each of its fields (lines 6 16).
- If it is instead an object of a non-container reference type, instantiate its fields transitively: assign to fields of primitive types (such as balance and credit_limit) their corresponding values in the input vector (lines 20 21); for fields of reference types, apply the procedure recursively (line 23).

5 | COMBINING PROOFS AND TESTS: THE PROCESS

As noted in the introduction, tests and proofs have often been considered distinct, incompatible or competing approaches to software verification. The assumption behind the present work is that it is better to view them as complementary.

5.1 | General description



FIGURE 7 Program verification with the assistance of Proof2Test.

The verification process combining both approaches, thanks to AutoProof, AutoTest and Proof2Test, is the following, illustrated by Figure 7:

- Step A: Attempt the verification with AutoProof. If the verification succeeds, the process stops. The remaining steps assume the verification failed. Figure 8 shows an example in which AutoProof is not able to prove some properties (although it does succeed with some others).
- Step B: Run Proof2Test. The remaining steps assume Proof2Test is able to generate a failing test (if not, Proof2Test is not helpful in this case).
- Step C: Run the test.
- Step D: use the test to attempt to correct the bug using standard testing and debugging techniques. The fix can be a change to the code or to the contract (or occasionally both).
- Repeat the process, attempting to prove the corrected code, until the proof succeeds.

AutoProof			×***
🐼 Verify 🔭 🔳	🛃 3 Successful [👌 3 Failed <u> 0</u> Errors	Filter: 🗙 🗡 -
	Class	Feature	Information
\checkmark	ACCOUNT	make (creator)	Verification successful.
\checkmark	ACCOUNT	invariant admissibility	Verification successful.
\checkmark	ACCOUNT	available_amount	Verification successful.
+- <mark>(3)</mark>	ACCOUNT	deposit	Postcondition balance_increased may be violated.
- <mark>8</mark>	ACCOUNT	withdraw	Postcondition balance_set may be violated.
😢	ACCOUNT	transfer	Postcondition deposit_made may be violated.
			Postcondition withdrawal_made may be violated.

FIGURE 8 Verification result of ACCOUNT in AutoProof, with some properties proved and others not

When Proof2Test is not able to generate a failing tests (the workflow in Figure 7 terminates with "No suggestion"), there are two possible reasons:

- the generated tests are passed, mostly due to the weakness of the specifications (loop invariants or postconditions of a supplier routine) of the target routine; Section 6 contains a more detailed discussion of such cases.
- the program contains the data types or program constructs that are not supported by Proof2Test; in this case, the test generation either terminates in an abnormal state (indicating a limitation of Proof2Test) or terminates normally but the resulting tests would be incorrect and thus not compilable.

5.2 | Running the verification process: an example

Here is the application of the above process to the ACCOUNT class from Section 3.1.

Step A: run AutoProof on the class; Figure 8 already showed the result, which displays AutoProof's inability to establish four postconditions: *balance_increased* in the deposit routine, *balance_set* in withdraw, *withdrawal_made* and *deposit_made* in transfer. In this case, the model file (ce.model) contains four counterexample models, each of which corresponds to a proof failure (postcondition).

Step B: use Proof2Test to generate test cases, shown in Figure 9, from the four counterexamples. In the order of proof failures:

- test_ACCOUNT_deposit_1 (lines 1-10) corresponds to deposit's postcondition *balance_increased*; it first instantiates current_object by setting the values of the balance and credit_limit to 38 and -62 (lines 6-7) and then sets the value of the input argument amount to -5 (line 8); finally it invokes a qualified call of deposit (the target routine) with the instantiated argument (line 9).
- test_ACCOUNT_withdraw_1 (lines 12 21) corresponds to the proof failure of withdraw; similar to the test case of deposit, it instantiates current_object's two fields, balance and credit_limit, to 10 and -20, and sets the value of amount to 1, followed by a call to withdraw.
- test_ACCOUNT_transfer_1 (lines 23 35) corresponds to the postcondition *deposit_made* of transfer; it first instantiates the state of current_object (lines 29 30), and then initializes the two arguments of transfer, amount (line 31) and other (line 32 33); it then calls transfer using the two arguments.
- test_ACCOUNT_transfer_2 (lines 37 47) corresponds to the proof failure of postcondition *withdrawal_made* of transfer; it follows the same structure of test_ACCOUNT_transfer_1: it starts with the initialization of current_object and arguments, followed by a call to transfer; additionally, as other and current_object are aliases (they have the same symbolic value in the counterexample model), instead of instantiating other using the set_type_field routines, it directly assign other with current_object (line 45).

Step C: exercise the test cases in AutoTest. Figure 10 shows the testing result: among the four test cases, test_ACCOUNT_transfer_1 passed and the other three test cases failed, raising the same postcondition violations as in the verification result (Figure 8).

Step D: to understand the causes of the contract violations, use the debugger from EiffelStudio to step through the three failed test cases: test_ACCOUNT_deposit_1, test_ACCOUNT_withdraw_1 and test_ACCOUNT_transfer_2. For each test case, move forward to the line of the respective call of the target routine and advance the execution of the target routine on one instruction at a time to observe how the program state (values of variables) evolves. Figure 11 shows the execution traces of the three target routines:

- deposit (Figure 11 (a)): initially the balance of the Current object is 38; after execution of the assignment, balance is set to 33 (the sum of its previous value and amount; the final state violates the postcondition *balance_increased* as the value of balance (33) is smaller than its old value (38). The execution trace reveals a flaw of deposit: depositing negative amount is permitted. To correct this error, a precondition should be added to require that the value of the input argument amount should be non-negative.
- withdraw (Figure 11 (b)): at the initial state, amount is 1, balance is 10, credit_limit is 20, available_amount (the difference between balance and credit limit) is 30; the two preconditions are satisfied; after executing the assignment, balance is set to 11 (the sum of its previous value and amount); at the final state, the value of old balance amount (9) is different with the value of balance (11), which violates the postcondition. This indicates the apparent bug: the routine implementation (addition operation) is inconsistent with what is expected in the postcondition (subtraction operation). Replacing the "+" with "-" is enough to correct the bug.
- transfer (Figure 11 (c)): initially, amount = 1 and available_amount is 51, which satisfies the two preconditions; after withdrawing amount from the Current account (balance := balance-amount), balance of the Current account is set to -3; moreover, since Current and other are aliases (they refer to the same object), balance of the other account is also set to -3; similarly, after depositing amount to the other, the balance in both other account and Current account are updated to -2; at the end state, the postcondition withdral_made is not satisfied. The execution trace shows

```
test_ACCOUNT_deposit_1
 1
 2
        local
 3
            current_object: ACCOUNT; amount: INTEGER
 4
        do
 5
            create current_object.make (0)
            {INTERNAL}.set_integer_field (balance, current_object, 38)
 6
 7
            {INTERNAL}.set_integer_field (credit_limit, current_object, (-62))
 8
            amount := (-5)
 0
            current_object.deposit (amount)
10
        end
11
12
    test_ACCOUNT_withdraw_1
13
      local
14
            current_object: ACCOUNT; amount: INTEGER
15
      do
16
          create current_object.make (0)
17
          {INTERNAL}.set_integer_field (balance, current_object, 10)
18
          {INTERNAL}.set_integer_field (credit_limit, current_object, (- 20))
19
          amount := 1
20
          current_object.withdraw(amount)
21
      end
22
23
    test_ACCOUNT_transfer_1
24
        local
25
            current_object, other: ACCOUNT; amount: INTEGER
26
        do
27
            create current_object.make (0)
28
            create other.make (0)
            \{INTERNAL\}.set_integer_field (balance, current_object, (-2))
29
30
            {INTERNAL}.set_integer_field (credit_limit, current_object, (-32))
31
            amount := 6
32
            {INTERNAL}.set_integer_field (balance, other, (-83))
33
            {INTERNAL}.set_integer_field (credit_limit, other, (-83))
34
            current_object.transfer (amount, other)
35
        end
36
    test_ACCOUNT_transfer_2
37
38
        local
39
            current_object, other: ACCOUNT; amount: INTEGER
40
        do
41
            create current_object.make (0)
42
            {INTERNAL}.set_integer_field (balance, current_object, (-2))
            {INTERNAL}.set_integer_field (credit_limit, current_object, (-53))
43
44
            amount := 1
45
            other := current_object
            current_object.transfer (amount, other)
46
47
        end
```



FIGURE 10 Testing results in AutoTest: "PASS" means running the test satisfies all assertions during execution; "FAIL" indicates the occurrence of a contract violation in the test run.



(c) Execution trace of transfer

FIGURE 11 Debugging failed tests in EiffelStudio: each block displays a program state (a list of variables and their values) after executing a designated statement (starting point of a blue line); the variable will be highlighted in red when its value is changed after executing the statement.

a erroneous scenario where the Current account is transferring money to itself. To exclude this particular case from the execution, we can add a precondition Current \neq other to limit that other should be an account other than Current.

These failed tests become part of the regression test suite of the target class ACCOUNT.

6 | EXPERIMENT AND EVALUATION

A preliminary evaluation of the usability of Proof2Test used the example programs of Table 2. Altogether, the experiment involved 49 editions of 9 Eiffel programs. The report includes not only the test results but also the test generation times for both the proofs and the tests.

The process is the following:

- Each edition results from injecting a single error into a correct program. The faults, injected manually, include:
 - 1. Switching between + and -, \leq and <, > and \geq .
 - 2. Removing a clause from a loop invariant, a precondition or a postcondition.
 - 3. Removing a statement in a routine body.
 - 4. Swapping two subsequent statements in a routine body.

In general, the decisions of which instruction or other clause to manipulate and how to mutate operators is random, with the goal of covering as many candidate areas as possible and creating faults that are similar to those made by programmers in practice.

- When verifying the modified program, AutoProof fails on a single assertion of one of the following seven kinds: postcondition violation at the end of a call; class invariant violation; precondition violation on entry to a call; loop invariant violated after loop initialization; loop invariant not maintained by loop body; loop variant not decreased; loop variant becoming negative.
- From each such failure, Proof2Test generates one test case.

Each row in Table 2 corresponds to an experimental task performed on a program variant, including three procedures: 1) verification of the program in AutoProof; 2) generation of test case in Proof2Test; 3) exercising the test case in AutoTest. The Size_P and Size_M columns provide the number of lines of the program and counterexample model. The "Proof time" column lists the time in seconds for obtaining a proof in AutoProof, and the "Test generation time" column the time Proof2Test takes to generate the test from the corresponding model. The last column shows the results of running the corresponding tests in AutoTest.

The examples include: 1) the class ACCOUNT introduced in Section 3.1; 2) a class CLOCK implementing a clock counting seconds, minutes, and hours; 3) a HEATER class implementing a heater, adjusting it state (on or off) based on the current temperature and user-defined temperature; 4) a class LAMP describing a lamp equipped with a switch (for switching the lamp on or off) and a dimmer (for adjusting the light intensity of the lamp); 5) a class BINARY_SEARCH implementing binary search; 6) a class LINEAR_SEARCH implementing linear search; 7) the class MAX_IN_ARRAY presented in Section 2; 8) a class SQUARE_ROOT for calculating two approximate square roots of a positive integer; 9) a class SUM_AND_MAX computing the maximum and the sum of the elements in an array. A.1 and A.2 detail the experiment results of ACCOUNT and LINEAR_SEARCH; the results for other examples can be found in the technical report^[9].

Among the 49 total test runs of the generated counterexamples, 37 failed but 12 runs passed. Ideally, we would like all counterexample to fail execution; the discrepancy reflects a difference of the semantics for proofs and tests. If a routine r calls a routine s:

- The proof of r relies only on the specification of s, independently of its implementation. In other words, proofs use modular semantics.
- A test of r, in particular an execution with run-time assertion monitoring under EiffelStudio and AutoTest, must execute r and hence must call s as it is, relying on its implementation. In other words, tests use global (non-modular) semantics.

As a consequence, assuming the prover (AutoProof) is sound, a test failure implies a proof failure but the reverse is not necessarily true: the counterexample generated by a failed proof (based on modular semantics) might not lead to a failure when we execute it (under non-modular semantics).

When such discrepancies arise in practice, they are typically due to a routine s with an implementation that is "correct" in some intuitive sense (it does what the programmer informally intended) but a specification that is not complete: it does not express all the relevant properties of the routine, which the prover would need to prove the correctness of the calling routine r.

Class	Size _P	Routine	Violated assertion	Failure type	Size _M	Proof time (s)	Test gen. time (s)	Test result
ACCOUNT	97	withdraw	balance_set	postcondition violation	658	0.247	0.241	Fail
			balance_non_negative	class invariant violation	620	0.275	0.225	Fail
		deposit	balance_set	postcondition violation	654	0.241	0.202	Fail
		transfer	amount ≤ 10	precondition violation	633	0.248	0.212	Fail
			withdrawal_made	postcondition violation	638	0.243	0.208	Fail
			withdrawal_made	postcondition violation	672	0.253	0.214	Pass
			deposit made	postcondition violation	660	0.244	0.223	Pass
	131	increase hours	valid hours	precondition violation	572	0.253	0.245	Fail
CLOCK		increase minutes	hours increased	postcondition violation	607	0.251	0.231	Fail
			hours increased	postcondition violation	641	0.263	0.218	Pass
			minutes increased	postcondition violation	606	0.259	0.222	Pass
		increase seconds	valid seconds	precondition violation	574	0.241	0.201	Fail
			hours increased	postcondition violation	647	0.243	0.219	Pass
			minutes increased	postcondition violation	625	0.248	0.201	Pass
			valid minutes	precondition violation	577	0.245	0.192	Fail
HEATER	73	turn on off	heater remains off	postcondition violation	633	0.615	0.207	Fail
	15	turn_on_on	heater remains on	postcondition violation	643	0.642	0.207	Fail
			heater remains on	postcondition violation	631	0.042	0.213	Fail
			heater remains off	postcondition violation	633	0.230	0.212	Fail
LAMD	71	turn on off	turn off	postcondition violation	728	0.240	0.101	Fail
	/1	turn_on_on	turn_off	postcondition violation	696	0.230	0.194	Fail
		adjust light	from high to low	postcondition violation	632	0.278	0.203	Fail
		aujust_ingiti	from medium to high	postcondition violation	624	0.203	0.222	Fail
	50	hinary saarah	nresent	postcondition violation	1242	0.270	0.208	Fail
BINARY_SEARCH	50	billary_search	present	invariant not maintained	1243	0.327	0.471	Fail Fail
			not_in_lower_part	invariant not maintained	1312	0.39	0.71	Fail Eail
			not_m_tower_part	Invariant not maintained	1370	0.393	0.440	Fail Eail
			-	variant not decreased	1570	0.323	0.398	Fail Eail
			not_m_upper_part	most and itian violation	1120	0.288	0.323	Fall Daga
			present	postcondition violation	1347	0.550	0.382	Pass
			not_in_iower_part	entry	1227	0.336	0.300	Fall
LINEAR_SEARCH	27	linear_search	result_in_bound	invariant violated on entry	955	0.709	0.332	Fail
			present	postcondition violation	977	0.283	0.373	Fail
			present	postcondition violation	975	0.280	0.344	Pass
			-	variant being negative	976	0.279	0.312	Fail
MAX_IN_ARRAY	33	max_in_array	max_so_far	invariant not maintained	1104	0.288	0.282	Fail
			is_maximum	postcondition violation	1059	0.284	0.313	Pass
			result_in_array	postcondition violation	1049	0.278	0.307	Pass
			is_maximum	postcondition violation	1057	0.286	0.323	Fail
			max_so_far	invariant violated on entry	992	0.290	0.303	Fail
			i_in_bounds	invariant violated on entry	1015	0.282	0.315	Fail
SQUARE_ROOT	38	square_root	-	variant not decreased	648	0.258	0.192	Fail
			valid_result	postcondition violation	723	0.257	0.205	Pass
			valid_result	invariant not maintained	644	0.308	0.181	Fail
			result_so_far	invariant not maintained	631	0.400	0.194	Fail
SUM_AND_MAX	36	sum_and_max	is_maximum	postcondition violation	1402	0.301	0.336	Pass
			is_maximum	postcondition violation	1405	0.321	0.333	Fail
			partial_sum_and_max	invariant not maintained	1178	0.307	0.333	Fail
			partial_sum_and_max	invariant violated on entry	1127	0.743	0.346	Fail
			sum_max_non_negative	invariant not maintained	1178	0.800	0.313	Fail
			sum_in_range	postcondition violation	1375	0.310	0.362	Fail
Total	556	14	49		44069	15.968	13.985	37/12

	TABLE 2 Test	generation	from fa	ailed p	roofs on	a co	llection	of	exampl	les
--	--------------	------------	---------	---------	----------	------	----------	----	--------	-----

The passing counterexample test runs are therefore useful, as the failing ones are, but in a different way: they usually alert the developers to the presence of a specification in need of improvement. (This case is the most common one. There always remains in principle the possibility of a mere limitation of the prover which – because of fundamental undecidability results — cannot be both sound and complete. However tempting it may be to blame the prover for proof failures, in practice this theoretical limitation seldom hits with a powerful prover such as the Boogie/Z3 combination for AutoProof: the culprit is almost always the programmer, not the tool.)

Incomplete specifications are indeed the reason for the 12 proof failures in the experiment. For example, failures in Variant 6 and 7 of ACCOUNT are due to weakness of the postconditions of the associated calling routines, and the failure in the Variant 3 of LINEAR_SEARCH to incomplete loop invariants.

For the 37 failures for which the tests fail, the causes can be categorized as follows:

- Postcondition violation due to incorrect implementations (examples in this category include the Variant 1, 4, 5 of ACCOUNT, and the Variant 2 of LINEAR_SEARCH).
- Precondition violation due to the inconsistency of specification between a routine and its calling routine; for example, the failure in Variant 3 of ACCOUNT is caused by the inconsistency of specification between the routine transfer and its calling routine deposit.
- Precondition violation of a calling routine due to a fault in its client routine; for example, the verification of Variant 1 of CLOCK (for the details of this variant, see the technical report^[9]) results in a failure of the precondition of a routine set_hours when calling it from another routine increase_hours.
- Violation of loop invariant at start of loop due to an incorrect implementation of loop initialization (see Variant 1 of LINEAR_SEARCH as an example).
- Loop invariant not maintained due to incorrect exit condition or faults in the loop body; examples in this category include Variant 4 of BINARY_SEARCH and Variant 1 of MAX_IN_ARRAY.
- Loop variant not decreased due to incorrect exit condition of the loop; an example in this category is Variant 1 of SQUARE_ROOT.
- Loop variant be negative due to the incorrectness of the loop variant; an example is the Variant 4 of LINEAR_SEARCH).

In most cases, the failing test is useful: executing it yields a specific trace illustrating how the program leads to the same contract violation that makes the proof fail. Even in cases such as Variant 2 and 5 of ACCOUNT in which the test input itself immediately demonstrates the problem, executing the automatically generated to allows programmers to step into the debugger and understand the issue in depth by going into step-by-step mode.

In all cases, the generated tests are important as regression tests: once the corresponding bug has been corrected, every test should become part of the project's regression suite, an essential tool for further development of the project.

7 | RELATED WORK

The idea of using counterexamples to generate test cases is not new, but it has been mostly applied to verification approaches using model checking of temporal-logic specifications^[17–20]. We are only aware of one existing attempt^[21] (building on work on using counterexamples for automatic program repair^[22]) to apply the idea in the context of Hoare-style verification; it exploits counterexamples produced by OpenJML^[23] (a verification tool for Java programs) to generate unit tests in JUnit^[24] format. The tool described in that article needs both to generate counter-examples from an SMT server and to generate, from the JML source, Java code instrumented to monitor some of the assertions (expressed as comments in the original code) at run-time. It then uses the concrete tests, extracted from counterexamples, to call the code. Proof2Test does not need any extra code generation since it uses inputs from counterexamples to call the erroneous routines, taking advantage of the built-in runtime assertion checking mechanism of EiffelStudio. We benefit here from the integration of contracts as a basic feature of the language and environment, rather than an add-on to a non-contract language such as Java.

KeyTestGen^[25,26] is a test generation tool based on the KeY, which is an automatic proof system for Java programs^[27] based on dynamic logic. The KeyTestGen makes use of the branch information (in the form of proof trees), generated during program

proofs, to construct path constraints with respect to different branches. It then applies an SMT solver to generate test data that satisfy those path constraints. As a result, the produced tests cases cover different program branches (paths). In contrast, Proof2Test aims to produce tests that can reproduce the proof failures at run time; it directly exploits counterexamples that are available in cases of proof failures and thus requires no extra SMT solving process.

In line with the objective of helping verification engineers understand the reasons of proof failures, many approaches have been proposed to provide a more user-friendly visualization of counterexample models: Claire et al.^[6] developed the Boogie Verification Debugger (BVD), which can interpret a counterexample model as a static execution trace (a sequence of abstract states). BVD has been applied in Dafny^[28] and VCC^[29]. David et al.^[30] transformed the models back into a counterexample trace comprehensible at the original source code level (SPARK) and display the trace using comments. Similarly, Stoll^[31] implemented a tool that translates the models into programs understandable at the Viper source code level^[32]. Chakarov et al.^[33] transformed SMT models to a format close to the Dafny syntax. Instead of providing a static diagnosis trace, the Proof2Test approach is able to present a "dynamic" trace, through execution of tests produced based on the models, which allows verification engineers to produce a program trace leading to a failing run-time state. Verification engineers can even use a debugger to step through the test, observing the faulty behaviors at their own pace. We find this approach more appropriate since every software developer is used to working with a debugger.

Another way of facilitating proof failures diagnosis is to generate of useful counterexamples: Polikarpova et al.^[34] developed a tool Boogaloo, which applied symbolic execution to generate counterexamples for failed proofs of Boogie programs. Boogaloo displays the resulting counterexamples in the form of valuations of relevant variables, similar to the intermediate result of the present work after the step "extraction of input data from the model" in Section 4. Likewise, Petiot et al.^[35,36] developed STADY that produces failing tests for the failed assertions using symbolic execution techniques. Their approach is also referred to testing-based counterexample synthesis: they first translated the original C program into programs suitable for testing (run-time assertion checking), and then applied symbolic execution to generate counterexamples (input of the failing tests) based on the translated program. Unlike that approach, Proof2Test exploits the original programs, that are inherently amenable to run-time assertion checking and thus requires no additional program transformation. Furthermore, our approach directly makes use of the counterexample models produced by the underlying provers and no extra counterexample generation process is performed. However, their approach is more fine-grained than ours as they can distinguish between specification weaknesses failures and prover limitations failures.

Other facilities for diagnosing proof failures: Müller et al.^[37] implemented a Visual Studio dynamic debugger plug-in for Spec#, to reproduce a failing execution from the viewpoint of the prover. This approach creates a variation of the original program for debugging, based on a modular verification semantics: the effect of the iteration of a loop is represented by the corresponding values in the counterexample. Tschannen et al.^[38] proposed a "two-step" approach to narrow down the reasons of proof failures: it compares the proof failures with those of its variant where called functions are inlined and loops are unrolled, which allows to discern failures caused by specification weaknesses and failures resulting from inconsistency between code and contract. However, inlining and unrolling are limited to a given number of nested calls and explicit iterations.

Some recent approaches share the present work's vision of combining static and dynamic techniques to make verification more usable. Julian et al.^[39] proposed a combination of AutoProof and AutoTest at a higher level, where the two tools were integrated in the Eiffel IDE to avail the complementarity of proof and testing. On one hand, for those programming features currently not supported by AutoProof (such as the code that relies on external precompiled C functions), testing can be used to check the correctness of routines. On the other hand, proof can be used to analyze the code that can not be tested (e.g., deferred functions that have no implementations). Collaborative verification^[40] is also based on the combination of testing and static verification, and on the explicit formalization of the restrictions of each tool used in the combination. The Proof2Test approach also complements the limitations of proof techniques with testing, with a particular purpose to improve user experience in understanding proof failures. However, it does not integrate the results of the two different techniques.

8 | LIMITATIONS

The results obtained in this study rely on a specific combination of technologies (section 3): Eiffel as the programming language, contract-equipped programs, and the AutoProof tool stack relying on Boogie, itself based on the Z3 SMT solver.

That last brick is the easiest to replace since Proof2Test relies not on the specifics of Z3 but on its SMT-LIB interface, which current SMT solvers generally share.

More generally, the setup assumes a Hoare-style verification framework (of which Boogie is but one example), and a language that supports the corresponding constructs (preconditions, postconditions, class invariants). Examples of such languages include the JML (Java Modeling Language)^[41] extension of Java, the Spark^[42] extension of Ada, and the Spec#^[43] extension of C#. We have not studied the possible application of the ideas to completely different verification frameworks, based for example on abstract interpretation or model checking.

The current version of Proof2Test is subject to the following limitations:

- It does not support the more advanced parts of the Eiffel system, in particular generic classes.Data structures are limited to arrays and sequences.
- It generates tests for individual routines (methods). There is no mechanism at this point to generate tests for an entire program.

These limitations will need to be removed for Proof2Test to be applicable to industrial-grade programs.

9 | CONCLUSION AND FUTURE WORK

The key assumption behind this work is that program proofs (static) and program tests (dynamic) are complementary rather than exclusive approaches. Software verification is hard; we should take advantage of all techniques that help. Proofs bring the absolute certainties that tests lack, but are abstract and hard to get right; tests cannot guarantee correctness but, when they fail, bring the concreteness of counterexamples, immediately understandable to the programmer and opening up the possibility of using well-understood debugging tools. Relying on the seamless integration of AutoProof and AutoTest in the Eiffel method and the EiffelStudio environment, Proof2Test attempts to leverage the benefits of both.

From this basis, work is proceeding in various directions:

- Cover the language constructs and types that (as noted in section 8) are not yet handled.
- Generate failing tests when the tests from Proof2Test are successful or non-executable.
- Make Proof2Test (currently a separate tool) a part of the EiffelStudio tool suite, at the same level of integration as AutoProof and AutoTest.
- Extend the scope of the work to include support for more intricate specifications, such as class invariants and assertions involving ghost states.
- On the theoretical side, develop a detailed classification of proof failures and possible fix actions with correspondence to their categories.
- On the empirical side, perform a systematic study of the benefits of the proposed techniques for verification non-experts.

As it stands, we believe that Proof2Test advances the prospect of an effective approach to software verification combining the power of modern proving and testing techniques.

Acknowledgments Alexander Kogtenkov and Alexandr Naumchev made important contributions to the discussions leading to this article. We are grateful to Filipp Mikoian and Manuel Oriol for useful discussions. We thank Amirfarhad Nilizadeh, one of the authors of the pioneering counterexample-based JML-related work cited earlier^[21], for useful comments on an early version. We are indebted to the reviewers of that early version for insightful comments that led to significant improvements of the article.

References

 Tschannen J, Furia CA, Nordio M, and Polikarpova N. Autoproof: Auto-active Functional Verification of Object-Oriented Programs. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Springer; 2015. p. 566–580.

- 2. AutoProof; Available from: http://comcom.csail.mit.edu/autoproof/.
- 3. Meyer B. Object-Oriented Software Construction, second edition. Prentice Hall; 1997.
- De Moura L, and Bjørner N. Z3: An Efficient SMT Solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Springer; 2008. p. 337–340.
- Barnett M, Chang BYE, DeLine R, Jacobs B, and Leino KRM. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: International Symposium on Formal Methods for Components and Objects. Springer; 2005. p. 364–387.
- 6. Le Goues C, Leino KRM, and Moskal M. The Boogie Verification Debugger. In: International Conference on Software Engineering and Formal Methods (SEFM). Springer; 2011. p. 407–414.
- 7. AutoTest;. Available from: https://www.eiffel.org/doc/eiffelstudio/Using_AutoTest.
- Wei Y, Gebhardt S, Meyer B, and Oriol M. Satisfying Test Preconditions Through Guided Object Selection. In: International Conference on Software Testing, Verification and Validation (ICST). IEEE; 2010. p. 303–312.
- 9. Huang L, and Meyer B. Deriving tests from failed proofs: experiments and results;. Available from: https://github.com/ huangl223/Proof2Test/blob/main/Deriving_tests_from_failed_proofs_experiments_and_results.pdf.
- 10. Meyer B. Touch of Class: Learning to Program Well with Objects and Contracts. Springer; 2016.
- 11. Meyer B. Applying "Design by Contract". Computer. 1992;25(10):40-51.
- Leino KRM, and Müller P. Object Invariants in Dynamic Contexts. In: European Conference on Object-Oriented Programming (ECOOP). Springer; 2004. p. 491–515.
- Polikarpova N, Tschannen J, Furia CA, and Meyer B. Flexible Invariants through Semantic Collaboration. In: International Symposium on Formal Methods (FM). Springer; 2014. p. 514–530.
- 14. Meyer B, Arkadova A, and Kogtenkov A. The Concept of Class Invariant in Object-Oriented Programming. arXiv (preprint of article submitted for publication). 2022;Available from: https://arxiv.org/abs/2109.06557.
- 15. Dijkstra EW. A Discipline of Programming. Prentice Hall; 1976.
- Barrett C, Stump A, Tinelli C, et al. The SMT-LIB Standard: Version 2.0. In: International Workshop on Satisfiability Modulo Theories. vol. 13; 2010. p. 14.
- 17. Beyer D, Chlipala AJ, Henzinger TA, Jhala R, and Majumdar R. Generating Tests from Counterexamples. In: International Conference on Software Engineering (ICSE). IEEE; 2004. p. 326–335.
- 18. Fantechi A, Gnesi S, and Maggiore A. Enhancing Test Coverage by Back-tracing Model-Checker counterexamples. Electronic Notes in Theoretical Computer Science (ENTCS). 2005;**116**:199–211.
- 19. Black PE. Modeling and Marshaling: Making Tests from Model Checker Counterexamples. In: Digital Avionics Systems Conference (DASC). vol. 1. IEEE; 2000. p. 1B3–1.
- 20. Beyer D, Dangl M, Lemberger T, and Tautschnig M. Tests from Witnesses. In: International Conference on Tests and Proofs (TAP). Springer; 2018. p. 3–23.
- Nilizadeh A, Calvo M, Leavens GT, and Cok DR. Generating Counterexamples in the Form of Unit Tests from Hoare-style Verification Attempts. In: International Conference on Formal Methods in Software Engineering (FormaliSE). IEEE; 2022. p. 124–128.
- Nilizadeh A, Calvo M, Leavens GT, and Le XBD. More Reliable Test Suites for Dynamic APR by Using Counterexamples. In: International Symposium on Software Reliability Engineering (ISSRE). IEEE; 2021. p. 208 – 219.
- Cok DR. JML and OpenJML for Java 16. In: International Workshop on Formal Techniques for Java-like Programs (FTfJP). ACM; 2021. p. 65–67.

- 24. Cheon Y, and Leavens GT. A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. In: European Conference on Object-Oriented Programming (ECOOP). Springer; 2002. p. 231–255.
- Engel C, and H\u00e4hnle R. Generating unit tests from formal proofs. In: Tests and Proofs: First International Conference, TAP 2007, Zurich, Switzerland, February 12-13, 2007. Revised Papers 1. Springer; 2007. p. 169–188.
- 26. Ahrendt W, Gladisch C, and Herda M. Proof-based test case generation. Deductive Software Verification–The KeY Book: From Theory to Practice. 2016;p. 415–451.
- 27. Ahrendt W, Beckert B, Bruns D, Bubel R, Gladisch C, Grebing S, et al. The KeY platform for verification and analysis of Java programs. In: Verified Software: Theories, Tools and Experiments: 6th International Conference, VSTTE 2014, Vienna, Austria, July 17-18, 2014, Revised Selected Papers 6. Springer; 2014. p. 55–71.
- Leino KRM. Dafny: An Automatic Program Verifier for Functional Correctness. In: International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR). Springer; 2010. p. 348–370.
- Cohen E, Dahlweid M, Hillebrand M, Leinenbach D, Moskal M, Santen T, et al. VCC: A Practical System for Verifying Concurrent C. In: International Conference on Theorem Proving in Higher Order Logics (TPHOLs). Springer; 2009. p. 23–42.
- Hauzar D, Marché C, and Moy Y. Counterexamples from Proof Failures in SPARK. In: International Conference on Software Engineering and Formal Methods (SEFM). Springer; 2016. p. 215–233.
- 31. Stoll C. SMT Models for Verification Debugging (Master thesis). ETH Zurich; 2019.
- 32. Müller P, Schwerhoff M, and Summers AJ. Viper: A Verification Infrastructure for Permission-based Reasoning. In: International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI). Springer; 2016. p. 41–62.
- 33. Chakarov A, Fedchin A, Rakamarić Z, and Rungta N. Better Counterexamples for Dafny. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Springer; 2022. p. 404–411.
- 34. Polikarpova N, Furia CA, and West S. To Run What No One Has Run Before: Executing an Intermediate Verification Language. In: International Conference on Runtime Verification (RV). Springer; 2013. p. 251–268.
- 35. Petiot G, Kosmatov N, Botella B, Giorgetti A, and Julliand J. How Testing Helps to Diagnose Proof Failures. Formal Aspects of Computing (FAC). 2018;**30**(6):629–657.
- Petiot G, Kosmatov N, Botella B, Giorgetti A, and Julliand J. Your Proof Fails? Testing Helps to Find the Reason. In: International Conference on Tests and Proofs (TAP). Springer; 2016. p. 130–150.
- Müller P, and Ruskiewicz JN. Using Debuggers to Understand Failed Verification Attempts. In: International Symposium on Formal Methods (FM). Springer; 2011. p. 73–87.
- Tschannen J, Furia CA, Nordio M, and Meyer B. Program Checking with Less Hassle. In: Working Conference on Verified Software: Theories, Tools, and Experiments. Springer; 2013. p. 149–169.
- Tschannen J, Furia CA, Nordio M, and Meyer B. Usable Verification of Object-Oriented Programs by Combining Static and Dynamic Techniques. In: International Conference on Software Engineering and Formal Methods (SEFM). Springer; 2011. p. 382–398.
- Christakis M, Müller P, and Wüstholz V. Collaborative Verification and Testing with Explicit Assumptions. In: International Symposium on Formal Methods (FM). Springer; 2012. p. 132–146.
- Leavens GT, Baker AL, and Ruby C. JML: a Java modeling language. In: Formal Underpinnings of Java Workshop (at OOPSLA'98). Citeseer; 1998. p. 404–420.
- 42. Carré B, and Garnsworthy J. SPARK—an annotated Ada subset for safety-critical programming. In: Proceedings of the conference on TRI-ADA'90; 1990. p. 392–402.

APPENDIX: DETAILED RESULTS FOR SELECTED PROGRAMS

The detailed results and analysis of applying Proof2Test to the sample classes listed in the article appear in a technical report^[9]. The material in this Appendix, extracted from that report, covers classes ACCOUNT and LINEAR_SEARCH.

All runs took place on a Windows 11 machine with a 2.1 GHz Intel 12-Core processor and 32 GB of memory. AutoProof or Proof2Test was the only computationally-intensive process running during the experiments. Version numbers for the underlying techology are: EiffelStudio 22.05; Boogie 2.11.1.0; Z3 4.8.14. On average, AutoProof ran for 0.326 seconds for each program; Proof2Test ran for 0.285 each test generation, with each test generation run producing one test case, from a single counterexample model.

A.1 Results for class ACCOUNT

Below shows a correct version of the ACCOUNT class, which includes a set of features representing basic operations on bank account: deposit (line 51), withdraw (line 64), and transfer (line 77). Figure 12 displays its verification result, which suggests a complete functional correctness. To demonstrate how Proof2Test can generate tests from proof failures, different faults are introduced into the correct version, resulting in 5 faulty variants.

```
1
    class
2
         ACCOUNT
3
4
    create
5
         make
6
7
    feature {NONE} -- Initialization
8
         make
9
              -- Initialize empty account.
10
             note
11
                  status: creator
12
             do
13
                  balance := 0
14
                  credit_limit := 0
15
              ensure
                  balance_set: balance = 0
16
17
                  credit_limit_set: credit_limit = 0
18
             end
19
20
    feature -- Access
21
22
         balance: INTEGER
23
              -- Balance of this account.
24
25
         credit_limit: INTEGER
26
              -- Credit limit of this account.
27
28
         available_amount: INTEGER
29
              -- Amount available on this account.
30
             note
31
                  status: functional
32
             do
33
                  Result := balance - credit_limit
34
             end
35
```

```
36
    feature -- Basic operations
37
38
         set_credit_limit (limit: INTEGER)
              -- Set 'credit_limit' to 'limit'.
39
              require
40
                  limit_not_positive: limit \leq 0
41
42
                  limit_valid: limit ≤ balance
43
              do
                  credit_limit := limit
44
45
              ensure
                  modify_field ([''credit_limit'', ''closed''], Current)
46
                  credit_limit_set: credit_limit = limit
47
48
              end
49
50
         deposit (amount: INTEGER)
51
              -- Deposit 'amount' in this account.
52
53
              require
                   \texttt{amount} \geq 0
54
55
              do
56
                  balance := balance + amount
57
              ensure
                  modify_field ([''balance'', ''closed''], Current)
58
                  balance\_increased; balance \geq old balance
59
                  balance_set: balance = old balance + amount
60
61
              end
62
63
         withdraw (amount: INTEGER)
64
              -- Withdraw 'amount' from this account.
65
              require
66
67
                   amount_not_negative: amount \ge 0
68
                   amount_available: amount < available_amount
69
              do
70
                  balance := balance - amount
71
              ensure
72
                  modify_field (["balance", "closed"], Current)
73
                  balance_set: balance = old balance - amount
74
                  balance_decrease: balance \leq old balance
75
              end
76
77
         transfer (amount: INTEGER; other: ACCOUNT_1)
78
              -- Transfer 'amount' from this account to 'other'.
79
              note
80
                  explicit: wrapping
81
              require
82
                   amount_not_negative: amount \ge 0
83
                   amount_available: amount < available_amount
84
                   other \neq Current
85
              do
86
                  withdraw (amount)
```

87	other.deposit (amount)
88	ensure
89	<pre>modify_field ([''balance'', ''closed''], [Current, other])</pre>
90	withdrawal_made: balance = old balance - amount
91	<pre>deposit_made: other.balance = old other.balance + amount</pre>
92	end
93	
94	invariant
95	credit_limit_not_positive: credit_limit ≤ 0
96	$balance_non_negative: balance - credit_limit \geq 0$
97	end

Aut	utoProof								
Ø	Verify 🔭 📃	🛃 7 Successful 脑 0 Faile	d 🕂 0 Errors	Filter:	× .				
	Class	Feature	Information	Position	Time [s]				
	ACCOUNT	invariant admissibility	Verification successful.		0.52				
	ACCOUNT	make (creator)	Verification successful.		0.01				
	ACCOUNT	available_amount	Verification successful.		0.03				
	ACCOUNT	set_credit_limit	Verification successful.		0.01				
	ACCOUNT	deposit	Verification successful.		0.00				
	ACCOUNT	withdraw	Verification successful.		0.00				
	ACCOUNT	transfer	Verification successful.		0.01				

FIGURE 12 Verification result of ACCOUNT in AutoProof: all routines are verified successfully (highlighted with green), which indicates that implementations of those routines are correct with respect to their specifications.

Variant 1 of ACCOUNT

- Fault injection: at line 73, change the postcondition *balance_set* from "balance = old balance amount" into "balance = old balance + amount".
- Resulting failure: as shown in Figure 13(a), the fault results in a violation of postcondition *balance_set* of the withdraw procedure.
- Cause of the failure: the implementation of withdraw (which *deduces* balance by amount) and specification (which requires the *increment* of balance by amount) is inconsistent.
- Proof time: 0.247 sec
- Test generation time: 0.241 sec
- Resulting test case: Figure 14 shows the test case generated from the failure it calls withdraw with input balance = 11797, credit_limit = -1, and amount = 11798;
- Testings result: running the test case, as shown in Figure 13(b), raises an exception of violation of *balance_set*, which maps to the same failure in AutoProof.
- Comment: the value of the test input does not contain specific meaning to the failure; as the failure is caused by inconsistency between implementation and specification, executing the withdraw procedure with any valid test input (which satisfies the precondition) would raise the same contract violation as in the proof.

Aut	oProof				83
Ø	Verify 🔭 📰 🛃 6 Su	uccessful 👔	1 Failed 🕂 0 Errors	Filter:	× 1.
	Class	Feature	Information	Position	Time [s]
	ACCOUNT_1	invariant a	Verification successful.		0.40
	ACCOUNT_1	make (cre	Verification successful.		0.01
	ACCOUNT_1	available	Verification successful.		0.03
	ACCOUNT_1	set_credit	Verification successful.		0.01
	ACCOUNT_1	deposit	Verification successful.		0.00
•-(ACCOUNT_1	withdraw	Postcondition balance_set may be violated.	77	0.03
	ACCOUNT_1	transfer	Verification successful.		0.00



(b)

FIGURE 13 (a) Verification result of ACCOUNT_1 in AutoProof; (b) Testing result of test_ACCOUNT_1_withdraw_1 in AutoTest

```
1
      test_ACCOUNT_1_withdraw_1
2
        local
3
           current_object: ACCOUNT_1
           amount: INTEGER_32
4
5
        do
6
           create current_object.make
7
           {P_INTERNAL}.set_integer_field_ ("balance", current_object, 11797)
             -- current_object.balance = 11797
8
9
           {P_INTERNAL}.set_integer_field_ (''credit_limit'', current_object, (-1))
             -- current_object.credit_limit = (-1)
10
           amount := 11798
11
12
           current_object.withdraw (amount)
13
         end
```

FIGURE 14 Test from the failed proof of balance_set

Variant 2 of ACCOUNT

- Fault injection: at line 68, remove the precondition amount_available of withdraw.
- Resulting failure: as shown in Figure 15(a), the class invariant balance_non_negative (line 96), which states that the balance (represented by balance amount) should not be negative, is violated. (Note that a class invariant which is supposed to hold at the entry and exit of every routine.)
- Cause of the failure: the precondition of withdraw is too weak; there should be a precondition to constrain the amount permitted in a withdrawal operation.
- Proof time: 0.275 sec

Αι	utoProof							
	Verify 🔭 📃	×	Y					
	Class	Feature	Information	Position	Ti.,			
,	ACCOU	invariant a	Verification successful.		0.1			
,	ACCOU	make (cre	Verification successful.		0.0			
,	ACCOU	available	Verification successful.		0.0			
,	ACCOU	set_credit	Verification successful.		0.0			
,	ACCOU	deposit	Verification successful.		0.0			
- -	ACCOU	withdraw	Invariant balance_non_negative might not hold on call to (ANY).wrap.	79	0.0			
	ACCOU	transfer	Verification successful.		0.0			



(b)

FIGURE 15 (a) Verification result of ACCOUNT_2 in AutoProof; (b) Testing result of test_ACCOUNT_2_withdraw_1 in AutoTest

```
1
        test_ACCOUNT_2_withdraw_1
2
        local
3
           current_object: ACCOUNT_2
4
           amount: INTEGER_32
5
        do
6
           create current_object.make
7
           {P_INTERNAL}.set_integer_field_ ("balance", current_object, 0)
8
             -- current_object.balance =0
9
           {P_INTERNAL}.set_integer_field_ ("credit_limit", current_object, 0)
10
             -- current_object.credit_limit =0
           amount := 1
11
12
           current_object.withdraw (amount)
13
         end
```

FIGURE 16 Test case from failed proof of balance_non_negative

- Test generation time: 0.225 sec
- Resulting test case: Figure 16 shows the test case from Proof2Test, which calls withdraw with input balance = 0, credit_limit = 0, amount = 1.
- Testings result: running the test case in raises the failure of invariant balance_non_negative (same failed contract in the proof) as shown in Figure 15(b).
- Comment: the test input itself immediately demonstrates the problem when there is no money left in the account, withdrawal operation should be forbidden; the executable test, however, is still useful in this case: programmers can still choose run the test and switch to the debugging mode to see how it fails step by step; the test can also be a part of the test suite for regression testing in later development stages.

Au	toProof				8
۲	Verify * 📰 🛃 6 Su	ıccessful 诸	1 Failed 🔥 0 Errors Filter:	×	V
	Class	Feature	Information	Position	Ti
	ACCOUNT_3	invariant a	Verification successful.		0.16
	ACCOUNT_3	make (cre	Verification successful.		0.00
	ACCOUNT_3	available	Verification successful.		0.03
	ACCOUNT_3	set_credit	Verification successful.		0.01
	ACCOUNT_3	deposit	Verification successful.		
	ACCOUNT_3	withdraw	Verification successful.		0.01
-	ACCOUNT_3	transfer	Precondition amount <= 10 may be violated on call to {ACCOUNT_3}.deposit.	91	0.03

Outputs		
Output: Testing	~ 📙 🔍	
Executing 1 tests		
<pre>test_account_3_transfer_ on_prepare: ok test routine: except on_clean: ok</pre>	1 (NEW_TEST_SET): FAIL	(amount <= 10) olation in ACCOUNT_3.transfer)
Execution complete		

(b)

FIGURE 17 (a) Verification result of ACCOUNT_3 in AutoProof; (b) Testing result of test_ACCOUNT_3_withdraw_1 in AutoTest

Variant 3 of ACCOUNT

- Fault injection: after line 54, add a precondition $\texttt{amount} \leq 10$ for deposit to strengthen the precondition.
- Resulting failure: as shown in Figure 17(a), the injected fault results in a failure of transfer it does not satisfy the new precondition amount ≤ 10 when calling deposit.
- Cause of the failure: the inconsistency of specification between a supplier routine deposit and its client routine transfer: when the precondition of a routine is changed, its client routine should be changed accordingly. In this example, the upper limit of transfer should be consistent with the upper limit of deposit. In other words, the amount of money in a transfer operation should not exceed the maximum amount that is permitted in a deposit operation.
- Proof time: 0.248 sec
- Test generation time: 0.212 sec
- Resulting test case: Figure 18 shows the test case from Proof2Test, which calls transfer with input Current.balance = -2147483599, Current.credit_limit = -2147483632, amount = 33, other.balance = 7719, other.credit_limit = -2147481211.
- Testings result: running the test case in raises the failure of precondition violation amount ≤ 10 of deposit (same failure in the proof), as shown in Figure 17(b).
- Comment: during the execution of the test, when the program calls other.deposit from transfer, the input for deposit is amount = 33, which violates the precondition of deposit and demonstrates the problem.

```
1
      test_ACCOUNT_3_transfer_1
2
        local
3
           current_object: ACCOUNT_3
4
           amount: INTEGER_32
5
           other: ACCOUNT_3
6
        do
7
           create current_object.make
8
           create other.make
           {P_INTERNAL}.set_integer_field_ (''balance'', current_object, (-2147483599))
9
             -- current_object.balance = (-2147483599)
10
           {P_INTERNAL}.set_integer_field_ (''credit_limit'', current_object, (-2147483632))
11
             -- current_object.credit_limit = (-2147483632)
12
13
           amount := 33
14
           {P_INTERNAL}.set_integer_field_ ("balance", other, 7719)
             -- other.balance =7719
15
           {P_INTERNAL}.set_integer_field_ (''credit_limit'', other, (-2147481211))
16
             -- other.credit_limit = (-2147481211)
17
           current_object.transfer (amount, other)
18
19
         end
```

FIGURE 18 Test case generated by Proof2Test

Variant 4 of ACCOUNT

- Fault injection: at line 56, change the body of deposit from "balance := balance + amount" into "balance := balance amount".
- Resulting failure: as shown in Figure 19(a), the postcondition balance_set is violated.
- Cause of the failure: this failure is similar to the failure in Variant 1, which results from the inconsistency between the implementation of deposit and its postcondition.
- Proof time: 0.241 sec
- Test generation time: 0.202 sec
- Resulting test case: Figure 20 shows the test case from Proof2Test, which calls deposit with input balance = 28101, credit_limit = 0, amount = 1.
- Testings result: as shown in Figure 19(b), running the test raises an exception of postcondition violation of *balance_set*, which corresponds to the same failure in the proof.
- Comment: this Variant of ACCOUNT is similar to Variant 1; the values in the test input does not contain any specific meaning; running deposit with any valid test input would lead to the same contract violation.

,	AutoProof					
1	🚫 Verify * 🔳 🛃 6 Su	ccessful 🚡	1 Failed 🕂 0 Errors	Filter:	×	V
	Class	Feature	Information		Position	Ti
	ACCOUNT_4	invariant a	Verification successful.			0.16
	ACCOUNT_4	make (cre	Verification successful.			0.01
	ACCOUNT_4	available	Verification successful.			0.03
	ACCOUNT_4	set_credit	Verification successful.			0.01
	■CACCOUNT_4	deposit	Postcondition balance_set may be violated.		65	0.02
	ACCOUNT_4	withdraw	Verification successful.			0.00
	ACCOUNT_4	transfer	Verification successful.			0.00
	■ CACCOUNT_4 ACCOUNT_4 ACCOUNT_4 ACCOUNT_4	deposit withdraw transfer	Postcondition balance_set may be violated. Verification successful. Verification successful.		65	0.02 0.00 0.00

Outputs	
Output: Testing	
Executing 1 tests	
<pre>test_account_4_deposit_1 (! on_prepare: ok test routine: exception on_clean: ok</pre>	<pre>NEW_TEST_SET): FAIL (balance_set) nal (Postcondition violation in ACCOUNT_4.deposit)</pre>
Execution complete	

(b)

FIGURE 19 (a) Verification result of ACCOUNT_4 in AutoProof; (b) Testing result of test_ACCOUNT_4_withdraw_1 in AutoTest

```
1
      test_ACCOUNT_4_deposit_1
2
        local
3
          current_object: ACCOUNT_4
4
          amount: INTEGER_32
5
        do
6
          create current_object.make
7
           {P_INTERNAL}.set_integer_field_ (''balance'', current_object, 28101)
             -- current_object.balance = 28101
8
9
           {P_INTERNAL}.set_integer_field_ ("credit_limit", current_object, 0)
10
             -- current_object.credit_limit =0
11
          amount := 1
12
          current_object.deposit (amount)
13
        end
```

FIGURE 20 Test case from failed proof of balance_set

Variant 5 of ACCOUNT

- Fault injection: at line 87, remove the precondition other \neq Current of transfer.
- Resulting failure: as shown in Figure 21(a), the fault injection leads to the violation of postcondition *withdrawal_made* when verifying transfer.
- Cause of the failure: the precondition of transfer is too weak; it should exclude the case where an account transfers money to itself.
- Proof time: 0.243 sec
- Test generation time: 0.208 sec

I I

uto	Proof					
ک ۷	ʻerify ʻ 🔳 🛃 6 Su	iccessful 👔	1 Failed 🕂 0 Errors	Filter:	*	V
(Class	Feature	Information		Position	Ti
~/	ACCOUNT_5	invariant a	Verification successful.			0.16
~/	ACCOUNT_5	make (cre	Verification successful.			0.01
~/	ACCOUNT_5	available	Verification successful.			0.03
1	ACCOUNT_5	set_credit	Verification successful.			0.01
×/	ACCOUNT_5	deposit	Verification successful.			0.00
~/	ACCOUNT_5	withdraw	Verification successful.			0.01
-6/	ACCOUNT_5	transfer	Postcondition withdrawal_made may be violated.		94	0.02

(a)



FIGURE 21 (a) Verification result of ACCOUNT_5 in AutoProof; (b) Testing result of test_ACCOUNT_5_withdraw_1 in AutoTest

1	test_ACCOUNT_5_transfer_1
2	local
3	current_object: ACCOUNT_5
4	amount: INTEGER_32
5	other: ACCOUNT_5
6	do
7	create current_object.make
8	{P_INTERNAL}.set_integer_field_(``balance'', current_object,(-2147481210))
9	current_object.balance = (-2147481210)
10	{P_INTERNAL}.set_integer_field_(``credit_limit``, current_object,(-2147482752))
11	current_object.credit_limit = (-2147482752)
12	amount := 1542
13	other := current_object
14	current_object.transfer (amount, other)
15	end

FIGURE 22 Test case from failed proof of withdrawal_made

- Resulting test case: Figure 22 shows the test case, which calls transfer with input balance = -2147481210, credit_limit = -2147482752, amount = 1542, and other is an alias of Current (line 13).
- Testings result: as presented in Figure 21(b), running the test case raises the failure of violation of postcondition *withdrawal_made* of transfer, which is the same as the proof failure in AutoProof.

AutoProof						83
🧭 Verify 🔭 📰	🛃 6 Su	iccessful 🚡	1 Failed 🕂 0 Errors Fi	lter:	×	V.
Class		Feature	Information		Position	Ti
ACCOUNT	_6	invariant a	Verification successful.			0.17
💊 ACCOUNT	_6	make (cre	Verification successful.			0.01
💊 ACCOUNT	_6	available	Verification successful.			0.03
💊 ACCOUNT	_6	set_credit	Verification successful.			0.00
💊 ACCOUNT	_6	deposit	Verification successful.			0.01
ACCOUNT	_6	withdraw	Verification successful.			0.01
•• CACCOUNT	_6	transfer	Postcondition withdrawal_made may be violated.		92	0.02
		Outp	(a)			
		Outp	ut: Testing V			
		Exe	cuting 1 tests			
		tes	t_account_6_transfer_1 (NEW_TEST_SET): pass			
		Exe	cution complete			
			(b)			

FIGURE 23 (a) Verification result of ACCOUNT_6 in AutoProof; (b) Testing result of test_ACCOUNT_6_withdraw_1 in AutoTest

Variant 6 of ACCOUNT

- Fault injection: at line 73, remove the postcondition *balance_set* of withdraw.
- Resulting failure: as shown in Figure 23(a), the injected fault results in the violation of postcondition *withdrawal_made* when verifying transfer.
- Cause of the failure: the postcondition of withdraw is incomplete not strong enough to represent the functionality of withdraw; as a result, when reasoning about the correctness of its client routine transfer, the prover is not able to establish the postcondition withdrawal_made of transfer, which is related to the functionality of withdraw.
- Proof time: 0.253 sec
- Test generation time: 0.214 sec
- Resulting test case: Figure 24 shows the test case, which calls transfer with input Current.balance = -2147475928, Current.credit_limit = -2147475929, amount = 0, other.balance = 0, and other.credit_limit = 0
- Testings result: as shown in Figure 23(b), the execution of the test terminates with no contract violation; this is because when verifying a client routine, AutoProof uses the postconditions of the involved supplier routines, instead of their bodies, to represent their functional behaviors; in this example, as the postcondition of withdraw does not strong enough to express its functionality (balance should be deduced by amount), AutoProof fails to establish the corresponding postcondition withdrawal_made of transfer; in other words, the counterexample, from which the test input is extracted, is not a real "counterexample"; but the successfulness of the testing result reveals the weakness of specifications in the relevant routines.

```
1
      test_ACCOUNT_6_transfer_1
2
        local
3
           current_object: ACCOUNT_6
4
           amount: INTEGER_32
5
           other: ACCOUNT_6
6
        do
7
           create current_object.make
8
           create other.make
           {P_INTERNAL}.set_integer_field_ (''balance'', current_object, (-2147475928))
9
10
             -- current_object.balance = (-2147475928)
           {P_INTERNAL}.set_integer_field_(''credit_limit'', current_object, (-2147475929))
11
             -- current_object.credit_limit = (-2147475929)
12
13
           amount := 0
14
           {P_INTERNAL}.set_integer_field_ ("balance", other, 0)
             -- other.balance = 0
15
16
           {P_INTERNAL}.set_integer_field_ ("credit_limit", other, 0)
             -- other.credit_limit =0
17
           current_object.transfer (amount, other)
18
19
         end
```

FIGURE 24 Test case generated by Proof2Test

Variant 7 of ACCOUNT

- Fault injection: at line 60, remove the postcondition *balance_set* of deposit.
- Resulting failure: the injected fault, as shown in Figure 25(a), results in the violation of postcondition *deposit_made* when verifying transfer.
- Cause of the failure: similar to the previous failure (in Variant 6), this failure of transfer is due to the weakness of the postcondition of its supplier class deposit the postcondition is not strong enough to represent the functionality of deposit.
- Proof time: 0.244 sec
- Test generation time: 0.223 sec
- Resulting test case: Figure 26 shows the test from Proof2Test, which calls transfer with input Current.balance = 0, Current.credit_limit = 0, amount = 0, other.balance = 0, and other.credit_limit = -7720.
- Testings result: as shown in Figure 25(b), execution of the test case terminates with no exception raised; similar to the failure in Variant 6, as the postcondition of the supplier routine deposit is too weak to describe its functional behavior (balance should be increased by amount), AutoProof fails to establish the postcondition *deposit_made* of transfer, requiring that balance of the other object should be increased by amount.

1

utoProof										8
🔊 Verify 🔭	🔳 🛃 6 Sı	uccessful 👔	1 Failed <u> 0</u> Errors				Filter:		×	V
Class		Feature	Information					Position		Ti
ACCOU	NT_7	invariant a	Verification successful.							0.16
ACCOUN	NT_7	make (cre	Verification successful.							0.01
ACCOU	NT_7	available	Verification successful.							0.03
ACCOU	NT_7	set_credit	Verification successful.							0.02
ACCOU	NT_7	deposit	Verification successful.							0.00
ACCOU	NT_7	withdraw	Verification successful.							0.00
	NT_7	transfer	Postcondition desposit_	made may be v	iolated.			93		0.02
		Outputs		(a)						
		Output:	Testing		~ 📙 🔍					
		Execu	ting 1 tests							
		test_	account_7_trans	fer_1 (NEW	_TEST_SET)	: pass				
		Execu	tion complete							

FIGURE 25 (a) Verification result of ACCOUNT_7 in AutoProof; (b) Testing result of test_ACCOUNT_7_withdraw_1 in AutoTest

(b)

```
1
      test_ACCOUNT_7_transfer_1
2
        local
3
           current_object: ACCOUNT_7
           amount: INTEGER_32
4
5
          other: ACCOUNT_7
6
        do
7
           create current_object.make
8
           create other.make
9
           {P_INTERNAL}.set_integer_field_ (''balance'', current_object, 0)
10
             -- current_object.balance =0
           {P_INTERNAL}.set_integer_field_ ("credit_limit", current_object, 0)
11
12
             -- current_object.credit_limit =0
           amount := 0
13
14
           {P_INTERNAL}.set_integer_field_ ("balance", other, 0)
15
             -- other.balance = 0
           {P_INTERNAL}.set_integer_field_ ('`credit_limit'', other, (-7720))
16
17
             -- other.credit_limit = (-7720)
18
           current_object.transfer (amount, other)
19
         end
```

FIGURE 26 Test case from the failed proof of deposit_made

A.2 Results for class LINEAR_SEARCH

The LINEAR_SEARCH class, which is displayed below, implements a function that returns the index of a given integer 'value' in an integer array 'a' using linear search starting from beginning of the array; if the 'value' is not found in 'a', the function returns the value "a.count + 1" (a.count represents the number of elements in a). Figure 27 shows the verification result of LINEAR_SEARCH, which indicates the complete correctness of its functionality. 4 variants of LINEAR_SEARCH are produced based on the correct version and are discussed below.

38

```
1
    class
 2
         LINEAR_SEARCH
 3
 4
    feature -- Basic operations
         linear_search (a: SIMPLE_ARRAY [INTEGER]; value: INTEGER): INTEGER
 5
 6
              require
 7
                   array_not_empty: a.count > 0
 8
              do
 9
                   from
10
                        Result := 1
11
                   invariant
                        result_in_bound: 1 \le \text{Result} and \text{Result} \le a.count + 1
12
13
                        not_present_so_far: across 1|.. | (Result -1) as i all a.sequence [i] \neq value end
14
                   until
                        Result = a.count + 1 or else a [Result] = value
15
16
                   loop
                        Result := Result + 1
17
18
                   variant
19
                        a.count - Result + 1
20
                   end
21
              ensure
22
                   result_in_bound: 1 \le \text{Result} and \text{Result} \le a.count + 1
23
                   present: a.sequence.has (value) = (Result \leq a.count)
24
                   found_if_present: (Result ≤ a.count) implies a.sequence [Result] = value
25
                   first_from_front: across 1|.. | (Result -1) as i all a.sequence [i] \neq value end
26
              end
27
    end
```

Au	utoProof				X
۲	🕽 Verify 🍯 📰 🛃 3 Successful 👔 () Failed <u>A</u> 0 Erro	ors	· 3	< 🛝 -
	Class	Feature	Information	Position	Ti
	VLINEAR_SEARCH_2	invariant ad	Verification successful.		0.56
	√ ANY	default_creat	Verification successful.		0.02
	VINEAR_SEARCH_2	linear_search	Verification successful.		0.07

FIGURE 27 Proof result of LINEAR_SEARCH in AutoProof

AutoPro	oof				8
🚫 Ver	ify 🔭 📰 🛃 2 Successfu	Il 🔥 1 Failed 🕂 0 Errors	Filter:	×	V
	Class	Feature	Information	P	Ti
\checkmark	LINEAR_SEARCH_1	invariant admissibility	Verification successful.		0.61
\checkmark	ANY	default_create (creator, inherited by LINEAR_SEARCH_1)	Verification successful.		0.08
-83	LINEAR_SEARCH_1	linear_search	Loop invariant result_in_bound may be violated on entry.	17	0.01
		(a)			
	Outputs				
	Output: 🖃 Testing	⊢ Q			
	Executing 1	tests			
<pre>test_linear_search_1_linear_search_1 (NEW_TEST_SET): FAIL (result_in_bound)</pre>					
Execution complete					
		(b)			

FIGURE 28 (a) Verification result of LINEAR_SEARCH_1 in AutoProof; (b) Testing result of test_LINEAR_SEARCH_1_linear_search_1 in AutoTest

Variant 1 of LINEAR_SEARCH

- Fault injection: at line 10, change the loop initialization from "Result := 1" into "Result := 0".
- Resulting failure: as shown in Figure 28(a), the injected fault leads to the violation of the loop invariant result_in_bound at the entry of the loop (after loop initialization).
- Cause of the failure: incorrect implementation of loop initialization.
- Proof time: 0.709 sec
- Test generation time: 0.332 sec
- Resulting test case: Figure 29 shows the test case from Proof2Test, which calls linear_search with input extracted from the corresponding counterexample: a[1] = 0, a[2] = 0, value = (-2147475929).
- Testings result: as shown in Figure 28(b), execution of the test case raises an exception of violation of loop invariant result_in_bound, which corresponds to the same proof failure.
- Comment: the test is useful as its execution demonstrates a specific case where the program goes to a failure state, violating the same contract as in the proof failure; the values in the test input, however, is not that meaningful to this failure, as running the program with any valid input would cause the same contract violation.

40

```
1
      test_LINEAR_SEARCH_1_linear_search_1
2
        local
3
           current_object: LINEAR_SEARCH_1
4
           a: SIMPLE_ARRAY [INTEGER_32]
5
           value: INTEGER_32
6
           linear_search_result: INTEGER_32
7
         do
8
           create current_object
9
           create a.make_empty
10
           a.force(0, 1)
11
           a.force(0, 2)
12
           value := (-2147475929)
13
14
           linear_search_result := current_object.linear_search (a, value)
15
         end
```

FIGURE 29 Test case from failed proof of result_in_bound

Variant 2 of LINEAR_SEARCH

- Fault injection: at line 12, change the left part of the exit condition from "Result = a.count + 1" into "Result = a.count".
- Resulting failure: as shown in Figure 30(a), the injected fault results in the violation of the postcondition present.
- Cause of the failure: incorrect exit condition (the condition for a loop to terminate).
- Proof time: 0.283 sec
- Test generation time: 0.373 sec
- Resulting test case: Figure 31 shows the test case, which calls linear_search with input extracted from the corresponding counterexample: a[1] = 0, a[2] = 0, value = (-2147475282)
- Testings result: as shown in Figure 30(b), execution of the test case raises an exception of violation postcondition present, which corresponds to the same proof failure.
- Comment: during the execution of the test, the program terminates after 1 iteration with Result = 2; this leads to the violation of the postcondition present in the equality expression, the left-hand segment a.sequence.has (value) is false, as value does not match to any element of the input array a, while the right-hand segment Result ≤ a.count is true (Result = 2 and a.count = 2).

	A first Designmentions			
roof				
rify 🔭 📰 🛃 2 Successf	ul 🔥 1 Failed 🕂 0 Errors		×	۲
Class	Feature	Information	Positi	Ti
LINEAR_SEARCH_2	invariant admissibility	Verification successful.		0.
ANY	default_create (creator, inherited by LINEAR_SEARCH_2)	Verification successful.		0.0
LINEAR_SEARCH_2	linear_search	Postcondition present may be violated.	29	
Outputs				
output lesting	•••••••••••••••••••••••••••••••••••••			
test_linear_sea on_prepare: test routin on_clean: co	arch_2_linear_search_1 (NEW_TEST_SET): FA ok he: exceptional (Postcondition violation ok	IL (present) in LINEAR_SEARCH_2.linear_se	arch)	
Execution compl	-+-			

FIGURE 30 (a) Verification result of LINEAR_SEARCH_2 in AutoProof; (b) Testing result of test_LINEAR_SEARCH_2_linear_search_1 in AutoTest

1	test_LINEAR_SEARCH_2_linear_search_1
2	local
3	current_object: LINEAR_SEARCH_2
4	a: SIMPLE_ARRAY [INTEGER_32]
5	value: INTEGER_32
6	linear_search_result: INTEGER_32
7	do
8	create current_object
9	create a.make_empty
10	a.force(0, 1)
11	a.force(0, 2)
12	
13	value := (-2147475282)
14	<pre>linear_search_result := current_object.linear_search (a, value)</pre>
15	end

FIGURE 31 Test case from failed proof of present

Variant 3 of LINEAR_SEARCH

- Fault injection: at line 13, remove the loop invariant not_present_so_far.
- Resulting failure: as shown in Figure 32(a), the injected fault causes the violation of the postcondition present.
- Cause of the failure: weakness/incompleteness of loop invariant.
- Proof time: 0.280 sec
- Test generation time: 0.344 sec
- Resulting test case: Figure 33 shows the test case from Proof2Test, which calls linear_search with input: a[1] = 0, a[2] = -2147462410, value = -2147462410.

AutoProof							8
Ӯ Verify `	· 🔳	Nuccessful 2	🚡 1 Failed 🕂 0 Errors	Filter:	×	J	r
	Class		Feature	Information	Position	Ti	
\checkmark	LINEA	AR_SEARCH_3	invariant admissibility	Verification successful.		0.1	17
\checkmark	ANY		default_create (creator, inherited by LINEAR_SEARCH_3)	Verification successful.		0.0	30
- 😢	LINEA	AR_SEARCH_3	linear_search	Postcondition present may be violate	32	0.0	03
		Outputs					
	Output: Testing Executing 1 tests test_linear_search_3_linear_search_1 (NEW_TEST_SET): pass Execution complete						
			(b)				

FIGURE 32 (a) Verification result of LINEAR_SEARCH_3 in AutoProof; (b) Testing result of test_LINEAR_SEARCH_3_binary_search_1 in AutoTest

```
1
      test_LINEAR_SEARCH_3_linear_search_1
2
        local
3
           current_object: LINEAR_SEARCH_3
           a: SIMPLE_ARRAY [INTEGER_32]
4
5
           value: INTEGER_32
           linear_search_result: INTEGER_32
6
7
         do
8
           create current_object
9
           create a.make_empty
10
           a.force(0, 1)
           a.force((-2147462410), 2)
11
12
13
           value := (-2147462410)
14
           linear_search_result := current_object.linear_search (a, value)
15
         end
```

FIGURE 33 Test case from failed proof of present

- Testings result: as shown in Figure 32(b), execution of the test case does not raise any exception.
- Comment: when trying to verify postcondition present, the prover uses the loop invariant, instead of the loop body, to represent the behaviors of the loop; if the loop invariant is not strong enough to express the functionality of the loop, as in this example, the prover is not able to establish the relevant postcondition; in this case, the counterexample (from which the test is extracted) is not a real "counterexample", as it does not reveal the fault in the implementation; the passing test indicates that the proof failure is caused by the weakness of the auxiliary specification (the loop invariant), not the implementation.

AutoProof												
🐼 Verify 🖢 🛃 2 Successful 🚡 1 Failed 🕂 0 Errors Filter:												
	Class	Feature	Information	Ρ	Ti							
	LINEAR_SEARCH_4	invariant admissibility	Verification successful.		0.18							
	ANY	default_create (creator, inherited	Verification successful.		80.0							
₽-	LINEAR_SEARCH_4	linear_search	Integer variant component at position 1 may be negative.	26	0.02							
(a)												
Ou	tput: 🗖 Testing	~ 📊 🔍										
Executing 1 tests												
<pre>test_linear_search_4_linear_search_1 (NEW_TEST_SET): FAIL (a.count - Result - 1) on_prepare: ok test routine: exceptional (Loop variant violation in LINEAR_SEARCH_4.linear_search) on_clean: ok</pre>												
Execution complete												

FIGURE 34 (a) Verification result of LINEAR_SEARCH_4 in AutoProof; (b) Testing result of test_LINEAR_SEARCH_4_binary_search_1 in AutoTest

(b)

Variant 4 of LINEAR_SEARCH

- Fault injection: change the loop variant at line 19 from "a.count Result + 1" into "a.count Result 1".
- Resulting failure: as shown in Figure 34(a), the injected faults leads to the violation that "the integer variant component at iteration 1 may be negative".
- Cause of the failure: incorrect loop variant.
- Proof time: 0.279 sec
- Test generation time: 0.312 sec
- Resulting test case: Figure 35 shows the test case from Proof2Test, which calls linear_search with input extracted from the corresponding counterexample.
- Testings result: as shown in Figure 34(b), execution of the test case raises an exception related to loop variant expression, which corresponds to the same failure in the verification.
- Comment: the test is useful as it is able to show how the value of variant varies at each iteration; the values in the test input, however, is not that meaningful, as any other valid test input will have the same effect.

```
1
      test_LINEAR_SEARCH_4_linear_search_1
2
        local
3
           current_object: LINEAR_SEARCH_4
           a: SIMPLE_ARRAY [INTEGER_32]
4
5
           value: INTEGER_32
6
           linear_search_result: INTEGER_32
7
        do
8
           create current_object
9
           create a.make_empty
10
           a.force(0, 1)
           a.force(0, 2)
11
           a.force(0, 3)
12
           a.force(0, 4)
13
           a.force((-2147482506), 5)
14
15
           value := (-2147482505)
16
           linear_search_result := current_object.linear_search (a, value)
17
18
        end
```

FIGURE 35 Test case from failed proof of "variant may be negative"