Deriving tests from failed proofs: experiments and results

Li Huang, Bertrand Meyer

Constructor Institute
{li.huang, bm}@constructor.org

Abstract. Ongoing work[1] has produced a general technique for turning failed program-proving attempts, which are largely useless to the programmer, into directly exploitable tests. The approach is embodied in the Proof2Test[2] framework, making use of the AutoProof[3] program-proving technology for Eiffel. The present report, intended as a complement to the article, provides the detailed results and analysis of the application of the approach and tools to a number of representative examples.

Keywords: Program verification, Counterexample, AutoProof, Proof2Test, Eiffel

1 Introduction and experiment setting

Program verification can take advantage of both proofs and tests. In current work, described in a separate article (see footnote 1) , we have developed the Proof2Test approach which derives a useful failing test from an essentially useless failed proof. The Proof2Test framework is part of a general verification framework for Eiffel, including in particular AutoProof for proofs and AutoTest for tests. The present discussion serves as a complement to the cited article by providing detailed results for a set of experiments applying Proof2Test to various programs, and analyzing the results.

The experiment ran on a Windows 11 machine with a 2.1 GHz Intel 12-Core 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.

[1] Li Huang, Bertrand Meyer, *A Failed Proof Can Yield a Useful Test*, submitted for publication, 2023, draft available at https://arxiv.org/abs/2208.09873
[2] https://github.com/huangl223/Proof2Test
[3] http://autoproof.sit.org/

2 Examples with numeric computations

2.1 ACCOUNT

ACCOUNT, whose implementation is shown below, is a class that describes the behaviors of bank accounts; it includes a set of features representing basic operations on bank account: `deposit` (line 51), `withdraw` (line 64), and `transfer` (line 77). Fig.1 shows the verification result of this version of ACCOUNT, which suggests a complete functional correctness. To demonstrate how Proof2Test can generate tests from proof failures, different faults are injected into the correct version; this results in 7 faulty variants of the ACCOUNT class, which will be discussed as follows.

```
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
16             balance_set: balance = 0
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)
39          -- Set 'credit_limit' to 'limit'.
40          require
41              limit_not_positive: limit <= 0
42              limit_valid: limit <= balance
43          do
44              credit_limit := limit
45          ensure
46              modify_field (["credit_limit", "closed"], Current)
47              credit_limit_set: credit_limit = limit
48          end
49
50
51      deposit (amount: INTEGER)
52          -- Deposit 'amount' in this account.
53          require
54              amount >= 0
55          do
56              balance := balance + amount
57          ensure
58              modify_field (["balance", "closed"], Current)
59              balance_increased: balance >= old balance
60              balance_set: balance = old balance + amount
61          end
62
63
64      withdraw (amount: INTEGER)
65          -- Withdraw 'amount' from this account.
66          require
67              amount_not_negative: amount >= 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 <= 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 ≥ 0
83              amount_available: amount ≤ available_amount
84              other ≠ Current
85          do
86              withdraw (amount)
87              other.deposit (amount)
88          ensure
89              modify_field (["balance", "closed"], [Current, other])
90              withdrawal_made: balance = old balance − amount
91              deposit_made: other.balance = old other.balance + amount
92          end
93
94  invariant
95      credit_limit_not_positive: credit_limit ≤ 0
96      balance_non_negative: balance − credit_limit ≥ 0
97  end
```

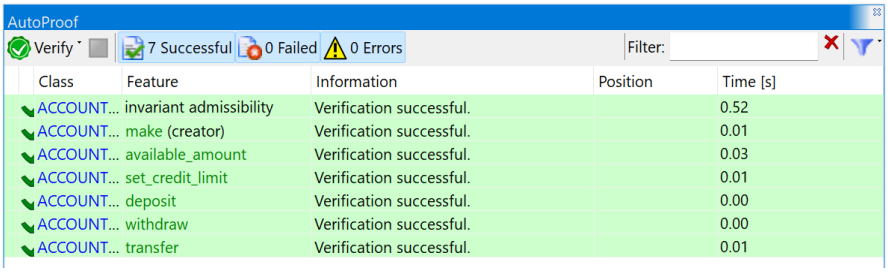
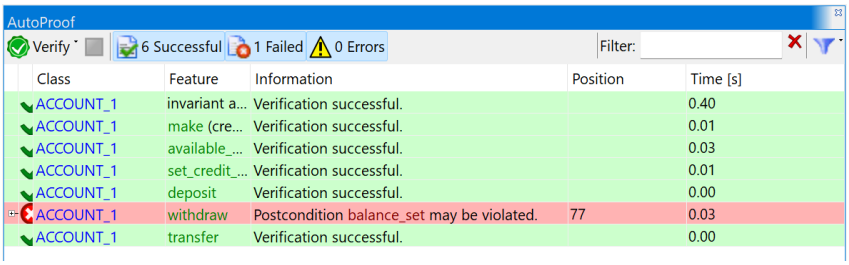


Fig. 1. Verification result of ACCOUNT in AutoProof: all routines are verified successfully (highlighted with green), which indicates that implementations of those routines are correct with respect to their specifications.

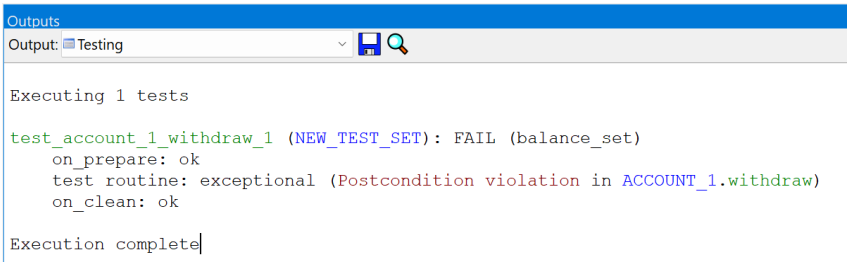
Variant 1 of ACCOUNT

- 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 Fig. 2(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: Fig. 3 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 Fig. 2(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.



(a)



(b)

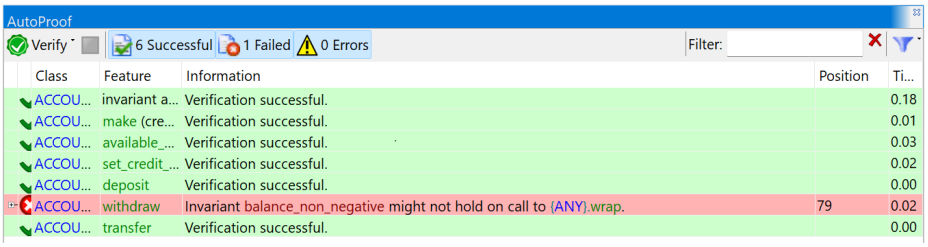
Fig. 2. (a) Verification result of `ACCOUNT_1` in AutoProof; (b) Testing result of `test_ACCOUNT_1_withdraw_1` in AutoTest

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

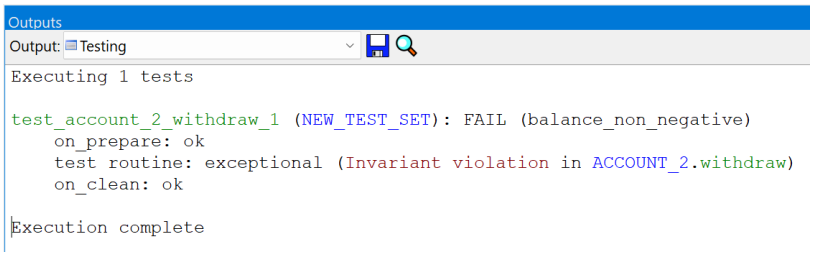
Fig. 3. Test from the failed proof of *balance_set*

Variant 2 of ACCOUNT

- Fault injection: at line 68, remove the precondition *amount_available* of `withdraw`.
- Resulting failure: as shown in Fig. 4(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
- Test generation time: 0.225 sec
- Resulting test case: Fig. 5 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 Fig. 4(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.



(a)



(b)

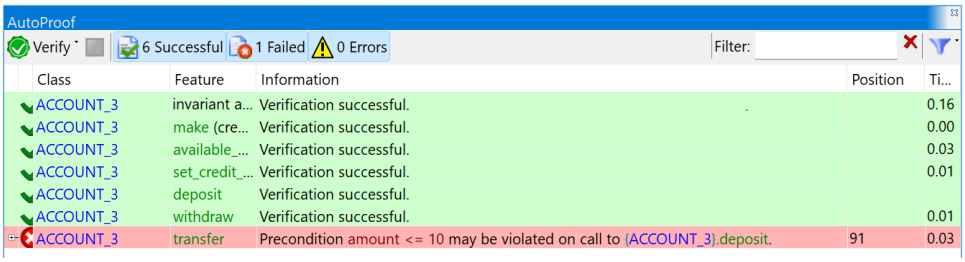
Fig. 4. (a) Verification result of `ACCOUNT_2` in AutoProof; (b) Testing result of `test_ACCOUNT_2_withdraw_1` in AutoTest

```
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
11    amount := 1
12    current_object.withdraw (amount)
13  end
```

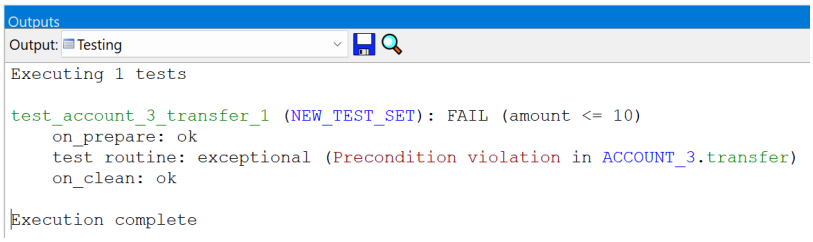
Fig. 5. Test case from failed proof of *balance_non_negative*

Variant 3 of ACCOUNT

- Fault injection: after line 54, add a precondition `amount` $\leq$ `10` for `deposit` to strengthen the precondition.
- Resulting failure: as shown in Fig. 6(a), the injected fault results in a failure of `transfer` — it does not satisfy the new precondition `amount` $\leq$ `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: Fig. 7 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` $\leq$ `10` of `deposit` (same failure in the proof), as shown in Fig. 6(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.



(a)



(b)

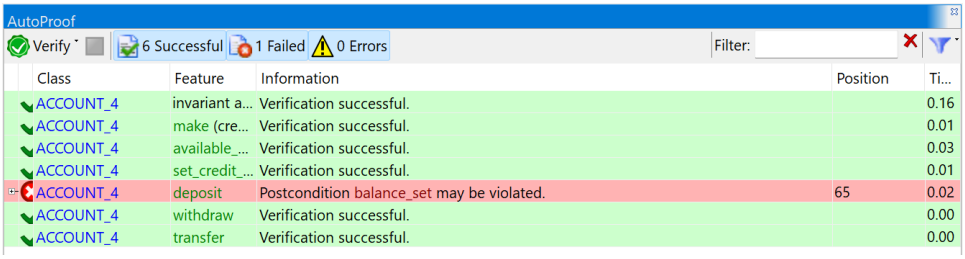
Fig. 6. (a) Verification result of `ACCOUNT_3` in AutoProof; (b) Testing result of `test_ACCOUNT_3_withdraw_1` in AutoTest

```
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
9       {P_INTERNAL}.set_integer_field_ ("balance", current_object, (-214748
            3599))
10        -- current_object.balance = (-2147483599)
11      {P_INTERNAL}.set_integer_field_ ("credit_limit", current_object, (-2
            147483632))
12        -- current_object.credit_limit = (-2147483632)
13      amount := 33
14      {P_INTERNAL}.set_integer_field_ ("balance", other, 7719)
15        -- other.balance = 7719
16      {P_INTERNAL}.set_integer_field_ ("credit_limit", other, (-2147481211
            ))
17        -- other.credit_limit = (-2147481211)
18      current_object.transfer (amount, other)
19    end
```

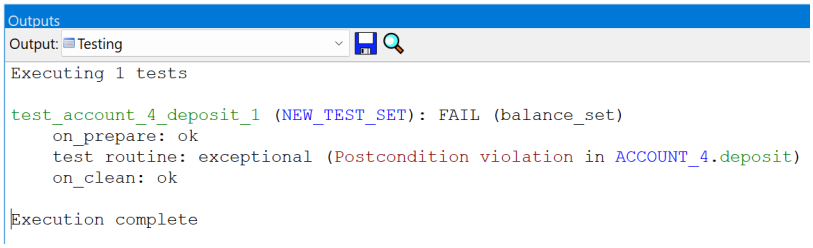
Fig. 7. Test case generated by Proof2Test

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 Fig. 8(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: Fig. 9 shows the test case from Proof2Test, which calls `deposit` with input — `balance` = `28101`, `credit_limit` = `0`, `amount` = `1`.
- Testings result: as shown in Fig. 8(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.



(a)



(b)

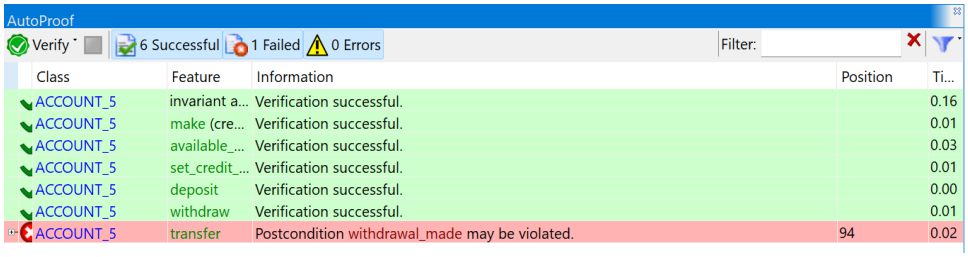
Fig. 8. (a) Verification result of `ACCOUNT_4` in AutoProof; (b) Testing result of `test_ACCOUNT_4_withdraw_1` in AutoTest

```
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)
8         -- current_object.balance = 28101
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
```

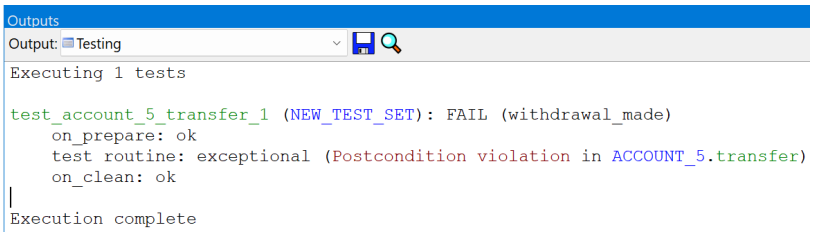
Fig. 9. Test case from failed proof of *balance_set*

Variant 5 of ACCOUNT

- Fault injection: at line 87, remove the precondition `other≠Current` of `transfer`.
- Resulting failure: as shown in Fig. 10(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
- Resulting test case: Fig. 11 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 Fig. 10(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.



(a)



(b)

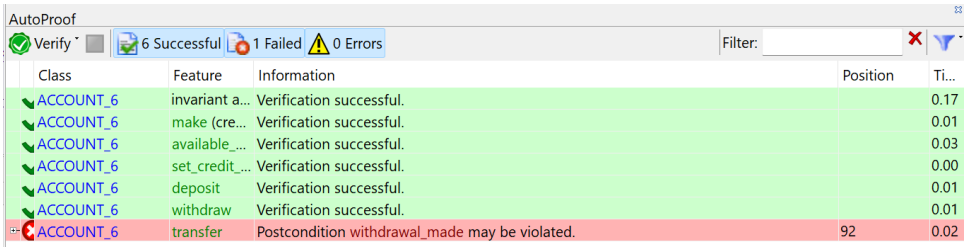
Fig. 10. (a) Verification result of `ACCOUNT_5` in AutoProof; (b) Testing result of `test_ACCOUNT_5_withdraw_1` in AutoTest

```
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, (-214748
            1210))
9         -- current_object.balance = (-2147481210)
10      {P_INTERNAL}.set_integer_field_ ("credit_limit", current_object, (-2
            147482752))
11        -- current_object.credit_limit = (-2147482752)
12      amount := 1542
13      other := current_object
14      current_object.transfer (amount, other)
15    end
```

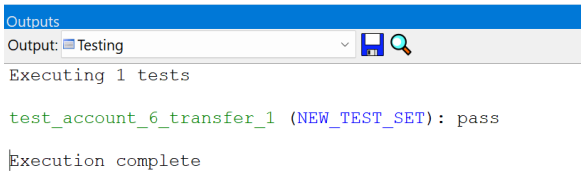
Fig. 11. Test case from failed proof of *withdrawal_made*

Variant 6 of ACCOUNT

- Fault injection: at line 73, remove the postcondition *balance_set* of `withdraw`.
- Resulting failure: as shown in Fig. 12(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 its client routine `transfer`, the prover is not able to establish the postcondition related to the functionality of `withdraw`.
- Proof time: 0.253 sec
- Test generation time: 0.214 sec
- Resulting test case: Fig. 13 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 Fig. 12(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.



(a)



(b)

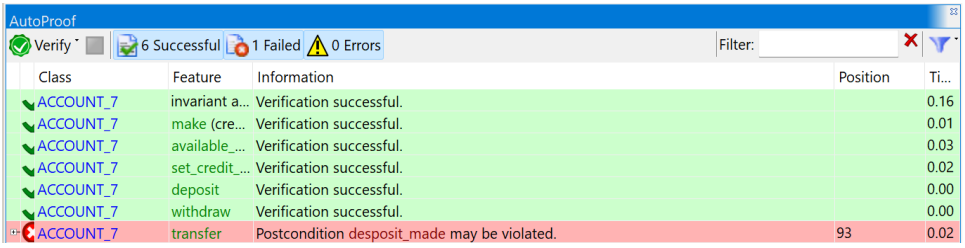
Fig. 12. (a) Verification result of `ACCOUNT_6` in AutoProof; (b) Testing result of `test_ACCOUNT_6_withdraw_1` in AutoTest

```
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
9       {P_INTERNAL}.set_integer_field_ ("balance", current_object, (-214747
            5928))
10        -- current_object.balance = (-2147475928)
11      {P_INTERNAL}.set_integer_field_ ("credit_limit", current_object, (-2
            147475929))
12        -- current_object.credit_limit = (-2147475929)
13      amount := 0
14      {P_INTERNAL}.set_integer_field_ ("balance", other, 0)
15        -- other.balance = 0
16      {P_INTERNAL}.set_integer_field_ ("credit_limit", other, 0)
17        -- other.credit_limit = 0
18      current_object.transfer (amount, other)
19    end
```

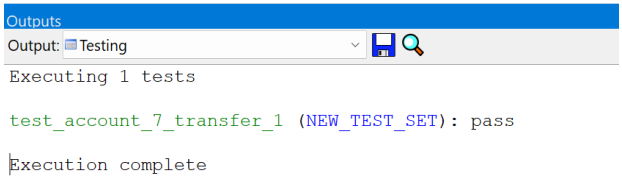
Fig. 13. Test case generated by Proof2Test

Variant 7 of ACCOUNT

- Fault injection: at line 60, remove the postcondition *balance_set* of `deposit`.
- Resulting failure: the injected fault, as shown in Fig. 14(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: Fig. 15 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 Fig. 14(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`.



(a)



(b)

Fig. 14. (a) Verification result of `ACCOUNT_7` in AutoProof; (b) Testing result of `test_ACCOUNT_7_withdraw_1` in AutoTest

```
1   test_ACCOUNT_7_transfer_1
2     local
3       current_object: ACCOUNT_7
4       amount: INTEGER_32
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
11      {P_INTERNAL}.set_integer_field_ ("credit_limit", current_object, 0)
12        -- current_object.credit_limit = 0
13      amount := 0
14      {P_INTERNAL}.set_integer_field_ ("balance", other, 0)
15        -- other.balance = 0
16      {P_INTERNAL}.set_integer_field_ ("credit_limit", other, (-7720))
17        -- other.credit_limit = (-7720)
18      current_object.transfer (amount, other)
19    end
```

Fig. 15. Test case from the failed proof of *deposit_made*

2.2 CLOCK

CLOCK class implements a digital clock counting seconds, minutes, and hours. The version of CLOCK displaying below is verified successfully. The experiment includes 8 different variants of CLOCK class, with different faults injected based on the verified version.

```
1  class
2      CLOCK
3
4  create
5      make
6
7  feature {NONE} -- Initialization
8      make
9          note
10             status: creator
11         do
12             hours := 0
13             minutes := 0
14             seconds := 0
15         ensure
16             modify_model (["hours", "minutes", "seconds"], Current)
17             initialized: hours = 0 and minutes = 0 and seconds = 0
18         end
19
20 feature -- Access
21
22     hours: INTEGER
23             -- Hours of clock.
24
25     minutes: INTEGER
26             -- Minutes of clock.
27
28     seconds: INTEGER
29             -- Seconds of clock.
30
31 feature -- Element change
32
33     set_hours (a_value: INTEGER)
34             -- Set 'hours' to 'a_value'.
35         require
36             valid_hours: 0 ≤ a_value and a_value <24
37         do
38             hours := a_value
39         ensure
40             hours_set: hours = a_value
```

```
41              modify_model (“hours”, Current)
42          end
43
44      set_minutes (a_value: INTEGER)
45              -- Set ‘minutes’ to ‘a_value’.
46          require
47              valid_minutes: 0 ≤ a_value and a_value <60
48          do
49              minutes := a_value
50          ensure
51              minutes_set: minutes = a_value
52              modify_model (“minutes”, Current)
53          end
54
55      set_seconds (a_value: INTEGER)
56              -- Set ‘seconds’ to ‘a_value’.
57          require
58              valid_seconds: 0 ≤ a_value and a_value <60
59          do
60              seconds := a_value
61          ensure
62              seconds_set: seconds = a_value
63              modify_model (“seconds”, Current)
64          end
65
66  feature -- Basic operations
67
68      increase_hours
69              -- Increase ‘hours’ by one.
70          note
71              explicit: wrapping
72          do
73              if hours = 23 then
74                  set_hours (0)
75              else
76                  set_hours (hours + 1)
77              end
78          ensure
79              hours_increased: hours = (old hours + 1) \\ 24
80              modify_model (“hours”, Current)
81          end
82
83      increase_minutes
84              -- Increase ‘minutes’ by one.
85          note
```

```
86              explicit: wrapping
87          do
88              if minutes <59 then
89                  set_minutes (minutes + 1)
90              else
91                  set_minutes (0)
92                  increase_hours
93              end
94          ensure
95              hours_increased: old minutes =59 implies
96                                                  hours =(old hours +
                                                        1) \\ 24
97              hours_unchanged: old minutes <59 implies
98                                                  hours =old hours
99              minutes_increased: minutes =(old minutes + 1) \\ 60
100             modify_model (["minutes", "hours"], Current)
101         end
102
103     increase_seconds
104             -- Increase 'seconds' by one.
105         note
106             explicit: wrapping
107         do
108             if seconds ≥ 59 then
109                 set_seconds (0)
110                 increase_minutes
111             else
112                 set_seconds (seconds + 1)
113             end
114         ensure
115             hours_increased: old seconds = 59 and old minutes =59
                    implies
116                                 hours =(old hours + 1) \\ 24
117             hours_unchanged: old seconds < 59 or old minutes <59 implies
118                                 hours =old hours
119             minutes_increased: old seconds = 59 implies
120                                 minutes =(old minutes + 1) \\ 60
121             minutes_unchanged: old seconds < 59 implies
122                                 minutes =old minutes
123             seonds_inreased: seconds =(old seconds + 1) \\ 60
124             modify_model (["seconds", "minutes", "hours"], Current)
125         end
126
127 invariant
128     hours_valid: 0 ≤ hours and hours ≤ 23
```

```
129        minutes_valid: 0 ≤ minutes and minutes ≤ 59
130        seconds_valid: 0 ≤ seconds and seconds ≤ 59
131    end
```

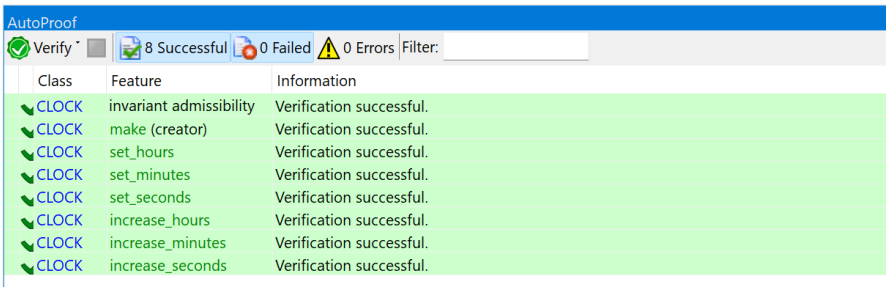
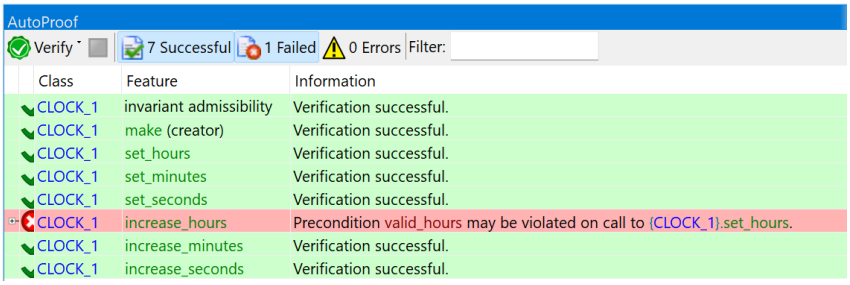


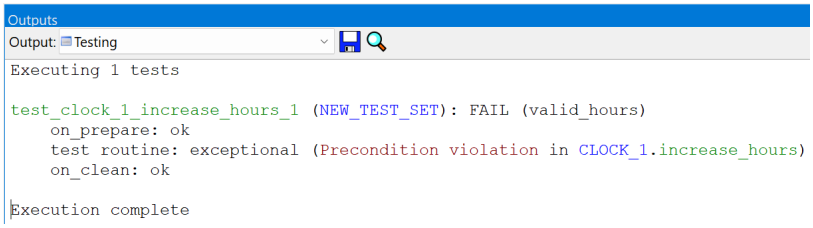
Fig. 16. Proof result of `CLOCK` in AutoProof

Variant 1 of `CLOCK`

- Fault injection: at line 73, in the `increase_hours` procedure, change the condition of the then branch from "`hours = 23`" into "`hours = 24`".
- Resulting failure: as shown in Fig. 17(a), the injected fault results in the violation of precondition `valid_hours` when calling `set_hours` from `increase_hours`.
- Cause of the failure: incorrect implementation of the routine body.
- Proof time: 0.253 sec
- Test generation time: 0.245 sec
- Resulting test case: Fig. 18 shows the test from Proof2Test, which calls `increase_hours` with input — `hours = 23`, `minutes = 39`, `seconds = 38`.
- Testings result: as shown in Fig. 17(b), execution of the test case raises an exception of precondition violation of `valid_hours` in `increase_hours`, which corresponds to the same failure in the proof.
- Comment: this test is useful to understand the fault in the program; the values of the test input incorporates a specific scenario, from which the execution of program will go to the same contract violation as in the proof.



(a)



(b)

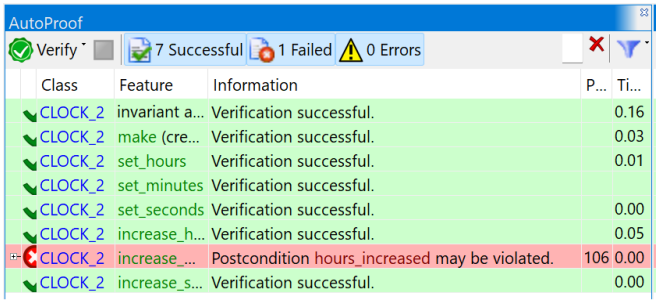
Fig. 17. (a) Verification result of `CLOCK_1` in AutoProof; (b) Testing result of `test_CLOCK_1_increase_hours_1` in AutoTest

```
1   test_CLOCK_1_increase_hours_1
2     local
3       current_object: CLOCK_1
4     do
5       create current_object.make
6       {P_INTERNAL}.set_integer_field_ ("hours", current_object, 23)
7                 -- current_object.hours = 23
8       {P_INTERNAL}.set_integer_field_ ("minutes", current_object, 39)
9                 -- current_object.minutes = 39
10      {P_INTERNAL}.set_integer_field_ ("seconds", current_object, 38)
11                -- current_object.seconds = 38
12      current_object.increase_hours
13    end
```

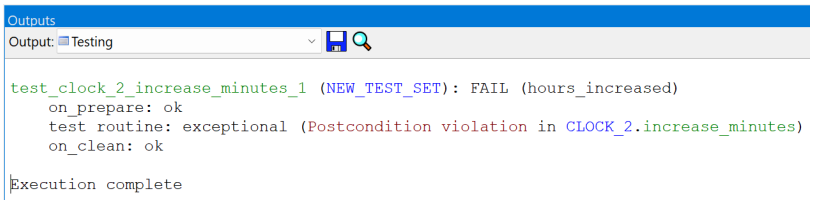
Fig. 18. Test case from failed proof of `valid_hours`

Variant 2 of `CLOCK`

- Fault injection: at line 92, remove the call `increase_hours` in the body of `increase_minutes`.
- Resulting failure: as shown in Fig. 19(a), the postcondition `hours_increased` is not satisfied in the proof of `increase_minutes`.
- Cause of the failure: incorrect implementation of the routine body.
- Proof time: 0.251 sec
- Test generation time: 0.231 sec
- Resulting test case: Fig. 20 shows the test case from Proof2Test, which calls `increase_minutes` with input — `hours` = 0, `minutes` = 59, `seconds` = 0.
- Testings result: as shown in Fig. 19(b), execution of the test case raises an exception of postcondition violation of `hours_increased`, which corresponds to the same failure in the proof.
- Comment: this test is useful as its execution shows a specific trace of how the postcondition `hours_increased` is violated.



(a)



(b)

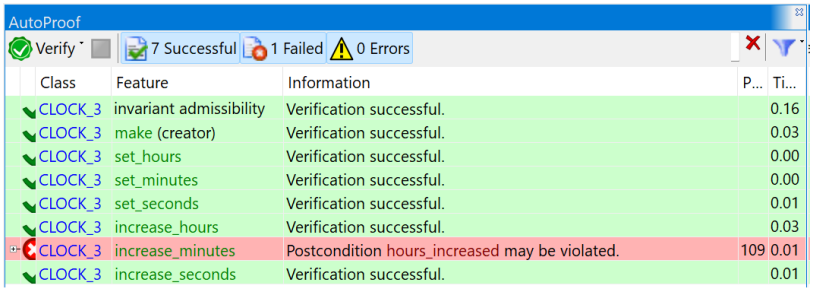
Fig. 19. (a) Verification result of `CLOCK_2` in AutoProof; (b) Testing result of `test_CLOCK_2_increase_minutes_1` in AutoTest

```
1   test_CLOCK_2_increase_minutes_1
2     local
3       current_object: CLOCK_2
4     do
5       create current_object.make
6       {P_INTERNAL}.set_integer_field_ ("hours", current_object, 0)
7         -- current_object.hours = 0
8       {P_INTERNAL}.set_integer_field_ ("minutes", current_object, 59)
9         -- current_object.minutes = 59
10      {P_INTERNAL}.set_integer_field_ ("seconds", current_object, 0)
11        -- current_object.seconds = 0
12      current_object.increase_minutes
13    end
```

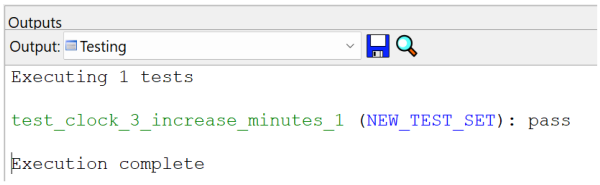
Fig. 20. Test case from failed proof of `hour_increased`

Variant 3 of CLOCK

- Fault injection: at line 79, remove the postcondition of `hours_increased` in the `increased_hours` procedure.
- Resulting failure: as shown in Fig. 21(a), the injected fault leads to violation of postcondition `hours_increased` when verifying `increase_minutes`.
- Cause of the failure: the postcondition of `increase_hours` is too weak to express the full functionality of the routine.
- Proof time: 0.263 sec
- Test generation time: 0.218 sec
- Resulting test case: Fig. 22 shows the test case from Proof2Test, which calls `increase_hours` with input — `hours` = 6, `minutes` = 59, `seconds` = 59.
- Testings result: as shown in Fig. 21(b), execution of the test case does not raise any exception; this is due to the same reason as discussed previously in variant 6 of ACCOUNT: the reasoning of the postcondition `hours_increased` of `increase_hours` relies on the postcondition of `increase_hours`, which is too weak to allow the prover to establish the postcondition `hours_increased`.



(a)



(b)

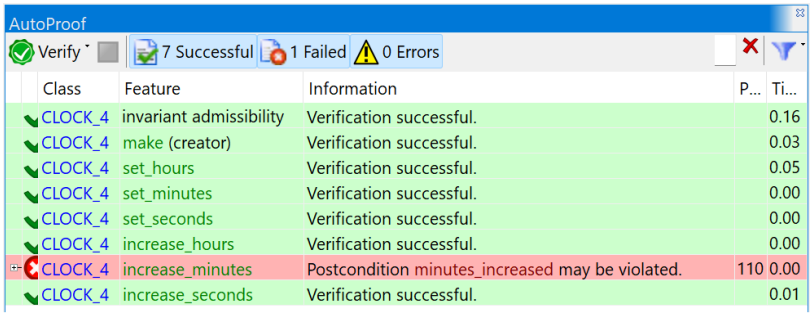
Fig. 21. (a) Verification result of `CLOCK_3` in AutoProof; (b) Testing result of `test_test_CLOCK_3_increase_minutes_1` in AutoTest

```
1   test_CLOCK_3_increase_minutes_1
2     local
3       current_object: CLOCK_3
4     do
5       create current_object.make
6       {P_INTERNAL}.set_integer_field_ ("hours", current_object, 6)
7         -- current_object.hours = 6
8       {P_INTERNAL}.set_integer_field_ ("minutes", current_object, 59)
9         -- current_object.minutes = 59
10      {P_INTERNAL}.set_integer_field_ ("seconds", current_object, 59)
11        -- current_object.seconds = 59
12      current_object.increase_minutes
13    end
```

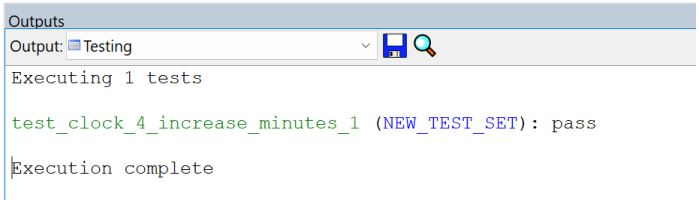
Fig. 22. Test case from the failed proof of `hours_increased`

Variant 4 of `CLOCK`

- Fault injection: at line 51, remove the postcondition of `minutes_set` in the `set_minutes` procedure.
- Resulting failure: as shown in Fig. 23(a), the postcondition `minutes_increased` is not satisfied when verifying the `increase_minutes` procedure.
- Cause of the failure: the postcondition of supplier routine `set_minutes` is too weak to represent its full functionality.
- Proof time: 0.259 sec
- Test generation time: 0.222 sec
- Resulting test case: Fig. 24 shows the test case from Proof2Test, which calls `increase_hours` with input — `hours` = 23, `minutes` = 58, `seconds` = 0.
- Testings result: as shown in Fig. 23(b), execution of the test case does not raise any exception; this is due to the similar reason as discussed previously in Variant 3 of `CLOCK`.



(a)



(b)

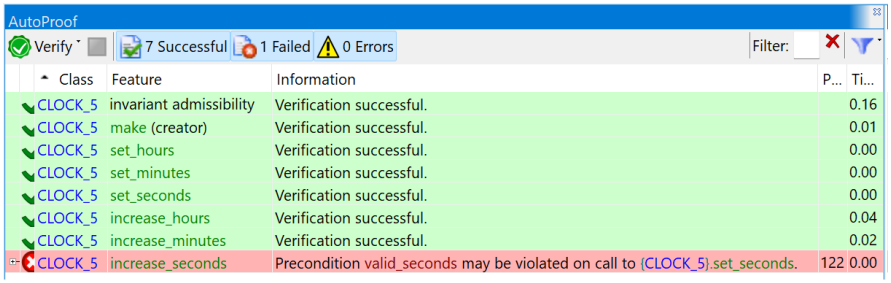
Fig. 23. (a) Verification result of `CLOCK_4` in AutoProof; (b) Testing result of `test_CLOCK_4_increase_minutes_1` in AutoTest

```
1   test_CLOCK_4_increase_minutes_1
2     local
3       current_object: CLOCK_4
4     do
5       create current_object.make
6       {P_INTERNAL}.set_integer_field_ ("hours", current_object, 23)
7         -- current_object.hours = 23
8       {P_INTERNAL}.set_integer_field_ ("minutes", current_object, 58)
9         -- current_object.minutes = 58
10      {P_INTERNAL}.set_integer_field_ ("seconds", current_object, 0)
11        -- current_object.seconds = 0
12      current_object.increase_minutes
13    end
```

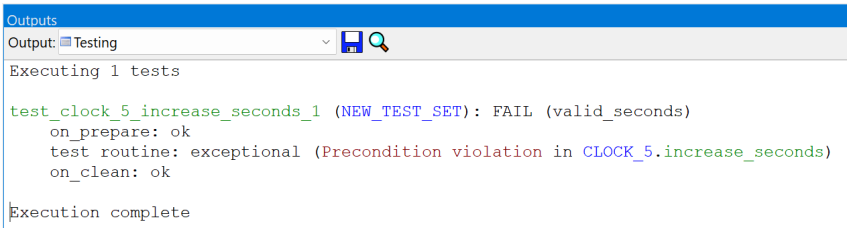
Fig. 24. Test case from failed proof of `minutes_increased`

Variant 5 of CLOCK

- Fault injection: at line 108, change the condition of the then branch from "`seconds`≥`59`" into "`seconds`>`59`".
- Resulting failure: as shown in Fig. 25(a), the injected fault leads to the violation of the precondition `valid_seconds` of `set_seconds` when it is called from the procedure `increase_seconds`.
- Cause of the failure: incorrect implementation of the routine body of `increase_seconds`.
- Proof time: 0.241 sec
- Test generation time: 0.201 sec
- Resulting test case: Fig. 26 shows the test case from Proof2Test, which calls `increase_seconds` with input — `hours` = `15`, `minutes` = `58`, `seconds` = `59`.
- Testings result: as shown in Fig. 25(b), execution of the test case raises an exception of precondition violation of `valid_seconds`, which is the same as the proof failure.
- Comment: this test is useful (the values in the test input are meaningful) as its execution shows a specific path that leads to the same contract violation as in the proof.



(a)



(b)

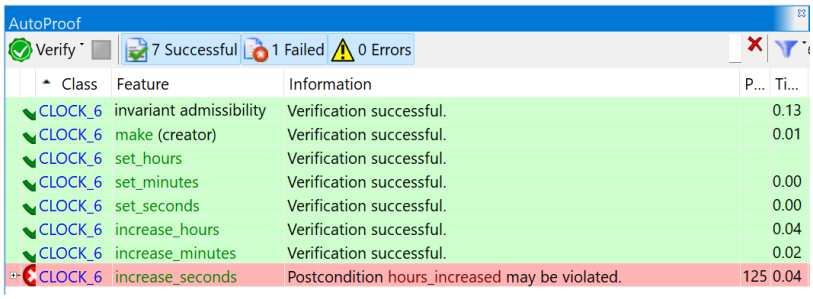
Fig. 25. (a) Verification result of `CLOCK_5` in AutoProof; (b) Testing result of `test_CLOCK_5_increase_seconds_1` in AutoTest

```
1   test_CLOCK_5_increase_seconds_1
2     local
3       current_object: CLOCK_5
4     do
5       create current_object.make
6       {P_INTERNAL}.set_integer_field_ ("hours", current_object, 15)
7         -- current_object.hours = 15
8       {P_INTERNAL}.set_integer_field_ ("minutes", current_object, 58)
9         -- current_object.minutes = 58
10      {P_INTERNAL}.set_integer_field_ ("seconds", current_object, 59)
11        -- current_object.seconds = 59
12      current_object.increase_seconds
13    end
```

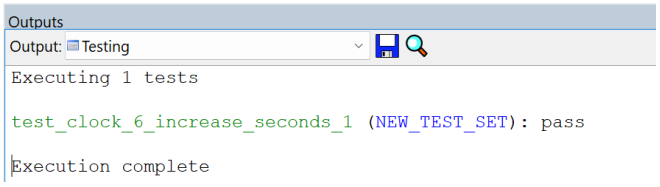
Fig. 26. Test case from failed proof of `valid_seconds`

Variant 6 of `CLOCK`

- Fault injection: at line 95, remove the postcondition `hours_increased` of `increase_minutes` procedure.
- Resulting failure: as shown in Fig. 27(a), the injected fault results in the violation of the postcondition `hours_increased` of `increase_seconds`.
- Cause of the failure: the postcondition of the routine `increase_minutes` is too weak to represent its full functionality.
- Proof time: 0.243 sec
- Test generation time: 0.219 sec
- Resulting test case: Fig. 28 shows the test case from Proof2Test, which calls `increase_hours` with input — `hours` = 22, `minutes` = 59, `seconds` = 59.
- Testings result: as shown in Fig. 27(b), execution of the test case does not raise any exception this is due to the same reason as discussed previously in Variant 3 and Variant 4 of `CLOCK`.
-



(a)



(b)

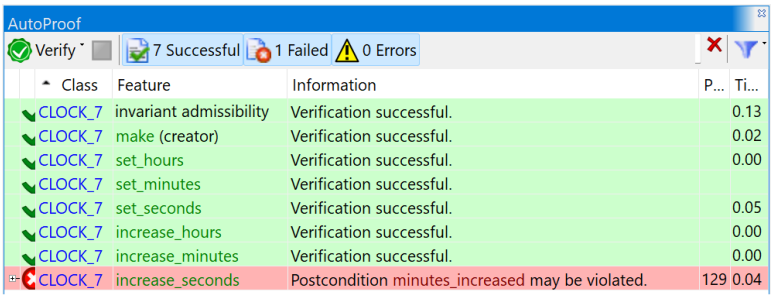
Fig. 27. (a) Verification result of `CLOCK_6` in AutoProof; (b) Testing result of `test_CLOCK_6_increase_seconds_1` in AutoTest

```
1   test_CLOCK_6_increase_seconds_1
2     local
3       current_object: CLOCK_6
4     do
5       create current_object.make
6       {P_INTERNAL}.set_integer_field_ ("hours", current_object, 22)
7         -- current_object.hours = 22
8       {P_INTERNAL}.set_integer_field_ ("minutes", current_object, 59)
9         -- current_object.minutes = 59
10      {P_INTERNAL}.set_integer_field_ ("seconds", current_object, 59)
11        -- current_object.seconds = 59
12      current_object.increase_seconds
13    end
```

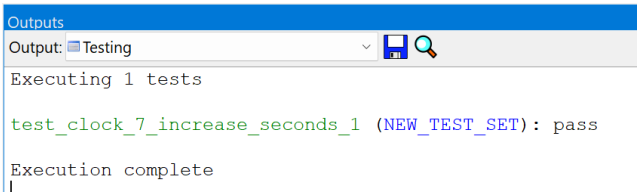
Fig. 28. Test case from failed proof of `hours_increased`

Variant 7 of CLOCK

- Fault injection: at line 99, remove the postcondition of `minutes_increased` of `increase_minutes` procedure.
- Resulting failure: as shown in Fig. 29(a), the postcondition `hours_increased` is violated.
- Cause of the failure: the postcondition of the routine `increase_minutes` is too weak to represent its full functionality.
- Proof time: 0.248 sec
- Test generation time: 0.201 sec
- Resulting test case: Fig. 30 shows the test case from Proof2Test, which calls `increase_hours` with input — `hours` = `0`, `minutes` = `0`, `seconds` = `59`.
- Testings result: as shown in Fig. 29(b), execution of the test case does not raise any exception; this is due to the same reason as discussed in Variant 3, 4 and 6.



(a)



(b)

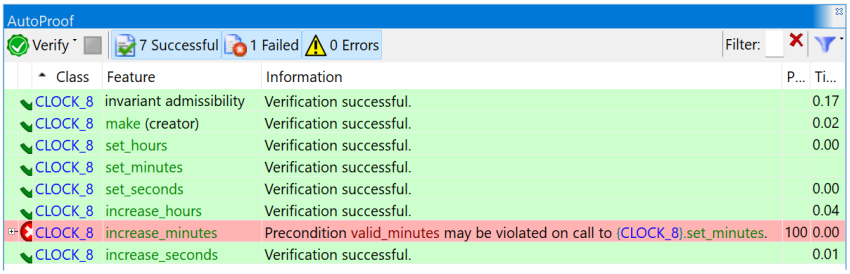
Fig. 29. (a) Verification result of `CLOCK_7` in AutoProof; (b) Testing result of `test_CLOCK_7_increase_seconds_1` in AutoTest

```
1   test_CLOCK_7_increase_seconds_1
2     local
3       current_object: CLOCK_7
4     do
5       create current_object.make
6       {P_INTERNAL}.set_integer_field_ ("hours", current_object, 0)
7         -- current_object.hours = 0
8       {P_INTERNAL}.set_integer_field_ ("minutes", current_object, 0)
9         -- current_object.minutes = 0
10      {P_INTERNAL}.set_integer_field_ ("seconds", current_object, 59)
11        -- current_object.seconds = 59
12      current_object.increase_seconds
13    end
```

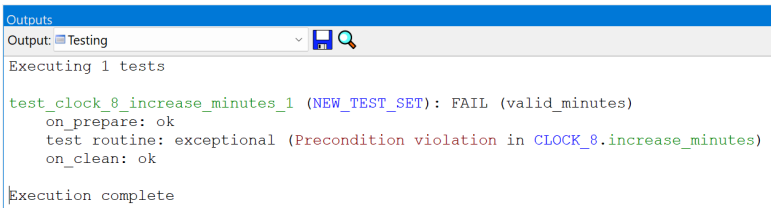
Fig. 30. Test case from failed proof of `minutes_increased`

Variant 8 of CLOCK

- Fault injection: at line 88, change the condition of the then branch from "`minutes`$<$`59`" into "`minutes`$\leq$`59`" in the `increase_minutes` procedure.
- Resulting failure: as shown in Fig. 31(a), the injected fault leads to the violation of precondition `valid_minutes` of the routine `set_minutes` when calling it from `increase_minutes`.
- Cause of the failure: incorrect implementation of routine body of `increase_minutes`.
- Proof time: 0.245 sec
- Test generation time: 0.192 sec
- Resulting test case: Fig. 32 shows the test case, which calls `increase_hours` with input — `hours` $=$ `14`, `minutes` $=$ `59`, `seconds` $=$ `39`.
- Testings result: as shown in Fig. 31(b) execution of the test case raises an exception of precondition violation of `valid_minutes`, which corresponds to the same proof failure.
- Comment: this variant is similar to Variant 5; the test is useful as its execution shows a specific path that leads to the same contract violation as in the proof.



(a)



(b)

Fig. 31. (a) Verification result of `CLOCK_8` in AutoProof; (b) Testing result of `test_CLOCK_8_increase_seconds_1` in AutoTest

```
test_CLOCK_8_increase_minutes_1
  local
    current_object: CLOCK_8
  do
    create current_object.make
    {P_INTERNAL}.set_integer_field_ ("hours", current_object, 14)
      -- current_object.hours = 14
    {P_INTERNAL}.set_integer_field_ ("minutes", current_object, 59)
      -- current_object.minutes = 59
    {P_INTERNAL}.set_integer_field_ ("seconds", current_object, 39)
      -- current_object.seconds = 39
    current_object.increase_minutes
  end
```

Fig. 32. Test case from failed proof of `valid_minutes`

2.3 HEATER

HEATER class, as shown below, implements a heater, which is automatically turned on/off based on the relation between the current temperature and the desired temperature. Fig. 33 displays the verification result of HEATER, which indicates that the implementation is correct with respect to its specification. In the experiment, 4 variants of faulty HEATER class are derived from this correct version, which will be discussed below.

```
class
    HEATER

create
    make

feature
    make
            -- By default, desired temperature is 20degree, deviation
                is 2 and heater is off
        do
            desired_temp := 20
            is_on := False
        ensure
            default_condition: desired_temp = 20 and is_on = False
        end

feature

    temperature: INTEGER
            -- Current temperature
```

```
21
22      desired_temp: INTEGER
23              -- Temperature defined by the user
24
25      is_on: BOOLEAN
26              -- Is heater turned on?
27
28      Deviation: INTEGER = 2
29              -- Deviation for turning on/off the heater
30
31  feature
32
33      set_temperature (a_value: INTEGER)
34              -- Set the 'temperature' to 'a_value'
35          do
36              temperature := a_value
37          end
38
39      set_desired_temperature (value: INTEGER)
40              -- Set the 'desired_temp' to 'value'
41          require
42              valid_value: value >= 10 and value <= 100
43          do
44              desired_temp := value
45          ensure
46              temperature_set: desired_temp = value
47          end
48
49      turn_on_off
50              -- Turn on or turn off the heater automatically based on
                    the current temperature
51          require
52              desired_temp_valid: desired_temp >= 10 and desired_temp <= 10
                    0
53          do
54              if is_on then
55                  if temperature > desired_temp + deviation then
56                      is_on := False
57                  end
58              else
59                  if temperature < desired_temp - deviation then
60                      is_on := True
61                  end
62              end
63          ensure
```

```
64              heater_is_turned_off: old (is_on and temperature >
                    desired_temp + deviation) implies (not is_on)
65              heater_remains_on: old (is_on and temperature ≤
                    desired_temp + deviation) implies is_on
66              heater_is_turned_on: old (not is_on and temperature <
                    desired_temp − deviation) implies is_on
67              heater_remains_off: old (not is_on and temperature ≥
                    desired_temp − deviation) implies (not is_on)
68          end
69
70  invariant
71      desired_temp_in_bound: desired_temp ≥ 10 and desired_temp ≤ 100
72
73  end
```

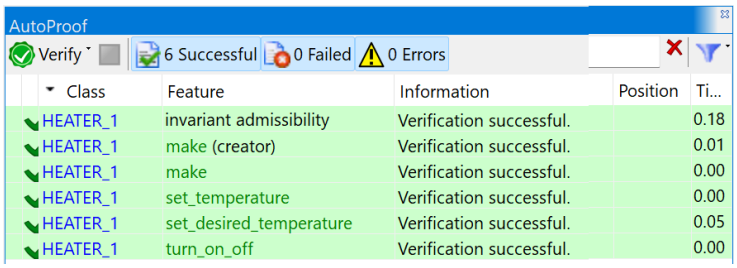
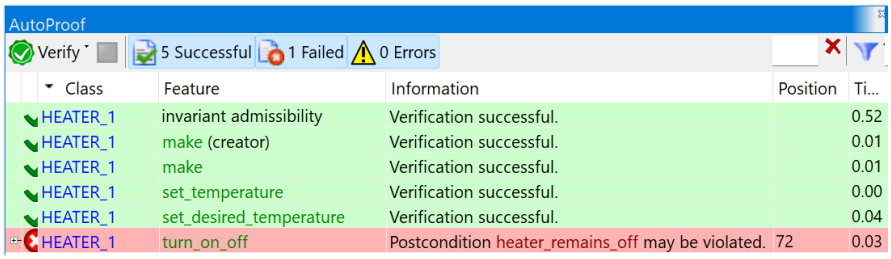


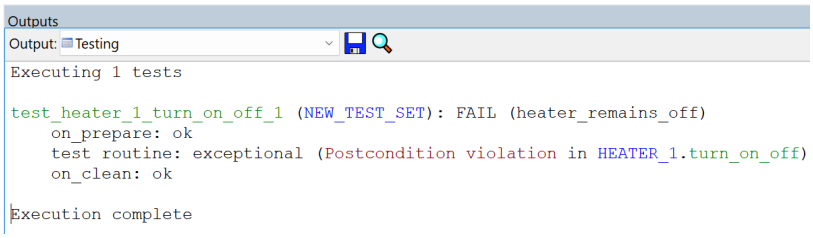
Fig. 33. Proof result of HEATER in AutoProof

Variant 1 of HEATER

- Fault injection: at line 59, change the condition of the then branch from "`temperature`< `desired_temp`−`deviation`" into "`temperature`≤ `desired_temp` −`deviation`".
- Resulting failure: as shown in Fig. 34(a), the injected faults results in the violation of the postcondition `heater_remains_off` of the procedure `turn_on_off`.
- Cause of the failure: incorrect implementation of the routine body of `turn_on_off`.
- Proof time: 0.615 sec
- Test generation time: 0.207 sec
- Resulting test case: Fig. 35 shows the test case from Proof2Test, which calls `turn_on_off` with input — `desired_temp` = 10, `is_on` = `false`, `temperature` = 8.
- Testings result: as shown in Fig. 34(b), execution of the test case raises an exception of postcondition violation of `heater_remains_off`, which corresponds to the same proof failure.
- Comment: the test is useful as its execution shows a specific path along which the same contract violation in the proof would be raised.



(a)



(b)

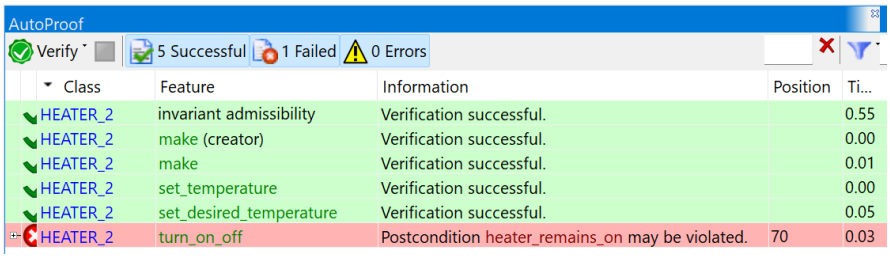
Fig. 34. (a) Verification result of `HEATER_1` in AutoProof; (b) Testing result of `test_HEATER_1_turn_on_off_1` in AutoTest

```
1   test_HEATER_1_turn_on_off_1
2     local
3       current_object: HEATER_1
4     do
5       create current_object.make
6       {P_INTERNAL}.set_integer_field_ ("desired_temp", current_object, 10)
7         -- current_object.desired_temp = 10
8       {P_INTERNAL}.set_boolean_field_ ("is_on", current_object, false)
9         -- current_object.is_on = false
10      {P_INTERNAL}.set_integer_field_ ("temperature", current_object, 8)
11        -- current_object.temperature = 8
12      current_object.turn_on_off
13    end
```

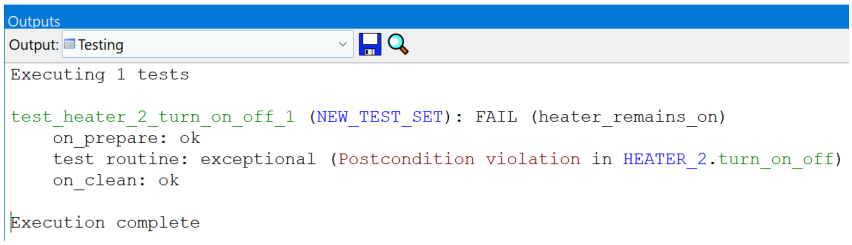
Fig. 35. Test case generated from failed proof of `heater_remains_off`

Variant 2 of HEATER

- Fault injection: at line 55, in the body of `turn_on_off`, change the condition of the then branch from "`temperature` > `desired_temp+deviation`" into "`temperature` ≥ `desired_temp+deviation`".
- Resulting failure: as shown in Fig. 36(a), the injected fault results in the violation of postcondition `heater_remains_on` of `turn_on_off` procedure.
- Cause of the failure: incorrect implementation of the routine body of `turn_on_off`.
- Proof time: 0.642 sec
- Test generation time: 0.213 sec
- Resulting test case: Fig. 37 shows the test case from Proof2Test, which calls `turn_on_off` with input — `desired_temp` = 48, `is_on` = `true`, `temperature` = 50.
- Testings result: as shown in Fig. 36(b), execution of the test case raises an exception of postcondition violation of `heater_remains_on`, which corresponds to the same proof failure.
- Comment: similar to Variant 1 of HEATER, the test is useful as its execution shows a specific path along which the same contract violation in the proof would be raised.



(a)



(b)

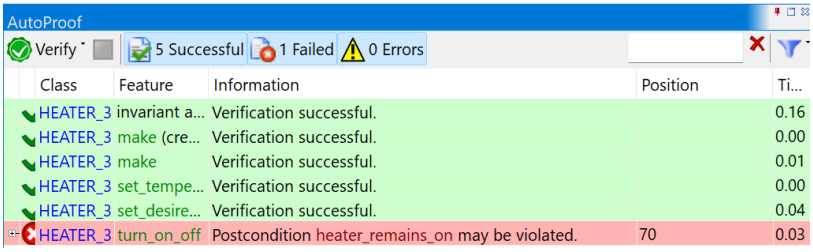
Fig. 36. (a) Verification result of `HEATER_2` in AutoProof; (b) Testing result of `test_HEATER_2_turn_on_off_1` in AutoTest

```
1   test_HEATER_2_turn_on_off_1
2     local
3       current_object: HEATER_2
4     do
5       create current_object.make
6       {P_INTERNAL}.set_integer_field_ ("desired_temp", current_object, 48)
7         -- current_object.desired_temp = 48
8       {P_INTERNAL}.set_boolean_field_ ("is_on", current_object, true)
9         -- current_object.is_on = true
10      {P_INTERNAL}.set_integer_field_ ("temperature", current_object, 50)
11        -- current_object.temperature = 50
12      current_object.turn_on_off
13    end
```

Fig. 37. Test case from failed proof of `heater_remains_on`

Variant 3 of HEATER

- Fault injection: at line 55, in the body of `turn_on_off`, change the condition of the then branch from "`temperature> desired_temp+deviation`" into "`temperature> desired_temp−deviation`".
- Resulting failure: as shown in Fig. 38(a), the injected fault results in the violation of postcondition `heater_remains_on` of the procedure `turn_on_off`.
- Cause of the failure: incorrect implementation of the routine body of `turn_on_off`.
- Proof time: 0.250 sec
- Test generation time: 0.212 sec
- Resulting test case: Fig. 39 shows the test case from Proof2Test, which calls `turn_on_off` with input — `desired_temp` = 45, `is_on` = `true`, `temperature` = 44.
- Testings result: as shown in Fig. 38(b), execution of the test case raises an exception of postcondition violation of `heater_remains_on`, which corresponds to the same proof failure.
- Comment: similar to Variant 1 and 2 of HEATER, the test is useful as its execution shows a specific path along which the same contract violation in the proof would be raised.



(a)

(b)

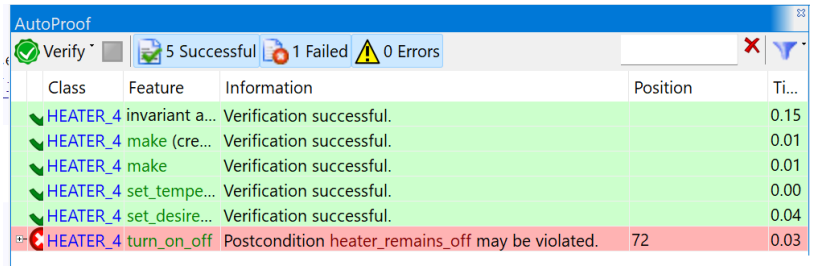
Fig. 38. (a) Verification result of `HEATER_3` in AutoProof; (b) Testing result of `test_HEATER_3_turn_on_off_1` in AutoTest

```
1   test_HEATER_3_turn_on_off_1
2     local
3       current_object: HEATER_3
4     do
5       create current_object.make
6       {P_INTERNAL}.set_integer_field_ ("desired_temp", current_object, 45)
7         -- current_object.desired_temp = 45
8       {P_INTERNAL}.set_boolean_field_ ("is_on", current_object, true)
9         -- current_object.is_on = true
10      {P_INTERNAL}.set_integer_field_ ("temperature", current_object, 44)
11        -- current_object.temperature = 44
12      current_object.turn_on_off
13    end
```

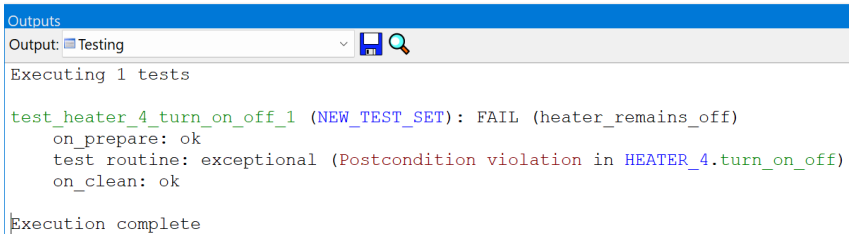
Fig. 39. Test case from failed proof of `heater_remains_on`

Variant 4 of HEATER

- Fault injection: at line 59, change the condition of the then branch from "`temperature< desired_temp−deviation`" into "`temperature< desired_temp +deviation`".
- Resulting failure: as shown in Fig. 40(a), the injected fault leads to the violation of the postcondition `heater_remains_off` in the procedure `turn_on_off`.
- Cause of the failure: incorrect implementation of the routine body of `turn_on_off`.
- Proof time: 0.246 sec
- Test generation time: 0.181 sec
- Resulting test case: Fig. 41 shows the test case from Proof2Test, which calls `turn_on_off` with input — `desired_temp` = 85, `is_on` = `false`, `temperature` = 86.
- Testings result: as shown in Fig. 40(b), execution of the test case raises an exception of postcondition violation of `heater_remains_off`, which corresponds to the same proof failure.
- Comment: similar to the previous variants, the test is useful as its execution shows a specific path along which the same contract violation in the proof would be raised.



(a)



(b)

Fig. 40. (a) Verification result of `HEATER_4` in AutoProof; (b) Testing result of `test_HEATER_4_turn_on_off_1` in AutoTest

```
test_HEATER_4_turn_on_off_1
  local
    current_object: HEATER_4
  do
    create current_object.make
    {P_INTERNAL}.set_integer_field_ ("desired_temp", current_object, 85)
      -- current_object.desired_temp = 85
    {P_INTERNAL}.set_boolean_field_ ("is_on", current_object, false)
      -- current_object.is_on = false
    {P_INTERNAL}.set_integer_field_ ("temperature", current_object, 86)
      -- current_object.temperature = 86
    current_object.turn_on_off
  end
```

Fig. 41. Test case from failed proof of `heater_remains_off`

2.4 LAMP

The LAMP class, as presented below, implements a lamp that has a switch and a dimmer. Its light intensity has three levels: low, medium and high. When the lamp is turned on, its light intensity will be the same as its intensity before it was last turned off. Fig.42 shows the verification result: this version of LAMP is correctly verified. Based on the verified version, 4 faulty variants of LAMP class are created, which are discussed below.

```
class
    LAMP

feature
    light_intensity: INTEGER
        -- Light intensity of the lamp

    is_on: BOOLEAN
        -- Is the lamp on?

  previous_light_intensity: INTEGER
        -- Light intensity of the lamp before it was last turned off

    High_intensity: INTEGER = 100
        -- High light intensity

    Medium_intensity: INTEGER = 75
        -- Medium light intensity

    Low_intensity: INTEGER = 25
        -- Low light intensity
```

```
22
23      Zero_intensity: INTEGER = 0
24          -- Zero light intensity
25
26  feature
27
28      turn_on_off
29          -- Turn on the lamp if it is off; turn off the lamp if it is on
30          do
31              if not is_on then
32                  is_on := True
33                  if previous_light_intensity > 0then
34                      light_intensity := previous_light_intensity
35                  else
36                      light_intensity := Low_intensity
37                  end
38              else
39                  is_on := False
40                  previous_light_intensity := light_intensity
41                  light_intensity := Zero_intensity
42              end
43          ensure
44              turn_on_1: old (not is_on and previous_light_intensity > 0)
                    implies (is_on and light_intensity = old
                    previous_light_intensity)
45              turn_on_2: old (not is_on and previous_light_intensity = 0)
                    implies (is_on and light_intensity = Low_intensity)
46              turn_off: old is_on implies (not is_on and
                    previous_light_intensity = old light_intensity and
                    light_intensity = Zero_intensity)
47      end
48
49      adjust_light
50          -- Adjust the light intensity
51          require
52              lamp_is_on: is_on = True
53          do
54              if light_intensity = Low_intensity then
55                  light_intensity := Medium_intensity
56              elseif light_intensity = Medium_intensity then
57                  light_intensity := High_intensity
58              elseif light_intensity = High_intensity then
59                  light_intensity := Low_intensity
60              end
61          ensure
```

```
62              from_low_to_medium: old light_intensity = Low_intensity
                    implies light_intensity = Medium_intensity
63              from_medium_to_high: old light_intensity = Medium_intensity
                     implies light_intensity = High_intensity
64              from_high_to_low: old light_intensity = High_intensity
                    implies light_intensity = Low_intensity
65          end
66
67  invariant
68      value_of_light_intensity: light_intensity = Zero_intensity or
            light_intensity = Low_intensity or light_intensity =
            Medium_intensity or light_intensity = High_intensity
69      value_of_previous_intensity: previous_light_intensity =
            Zero_intensity or previous_light_intensity = Low_intensity or
            previous_light_intensity = Medium_intensity or
            previous_light_intensity = High_intensity
70      light_intensity_when_off: is_on = (light_intensity ≠ Zero_intensity)
71  end
```

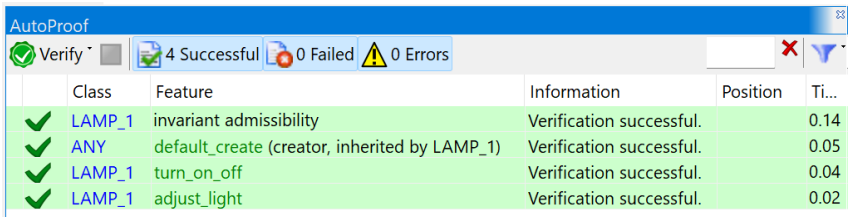


Fig. 42. Proof result of `LAMP` in AutoProof

Variant 1 of `LAMP`

- Fault injection: in the body of `turn_on_off`, switch the order of line 40 and line 41.
- Resulting failure: as shown in Fig. 43(a), the injected fault results in the violation of the postcondition `turn_off` of the procedure `turn_on_off`.
- Cause of the failure: incorrect implementation of the routine body of `turn_on_off`; the value of `light_intensity` should be stored into `previous_light_intensity` before being assigned to a new value.
- Proof time: 0.250 sec
- Test generation time: 0.194 sec
- Resulting test case: Fig. 44 shows the test case from Proof2Test, which calls `turn_on_off` with input — `is_on` = `true`, `light_intensity` = `75`, `previous_light_intensity` = `0`.

- Testings result: as shown in Fig. 43(b), execution of the test case raises an exception of postcondition violation of `turn_off`, which corresponds to the same proof failure.
- Comment: the test is useful as its execution shows a specific trace illustrating how the program goes to the same contract violation as in the proof.

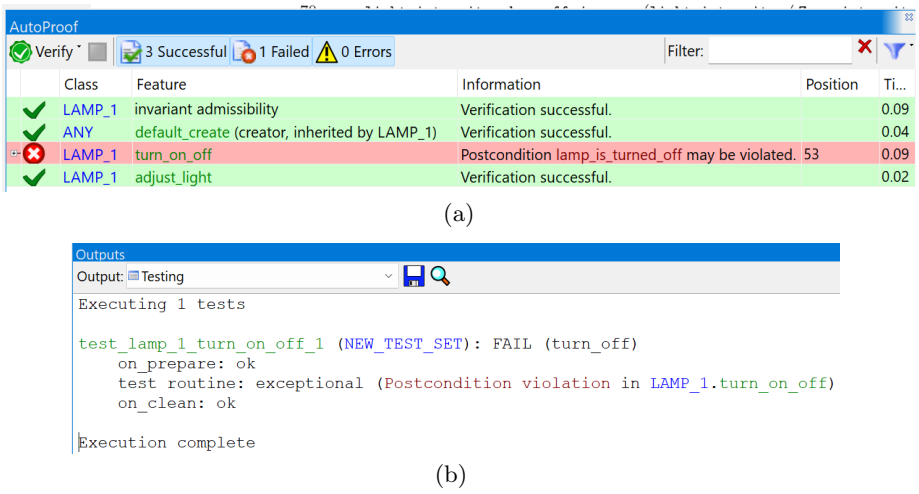


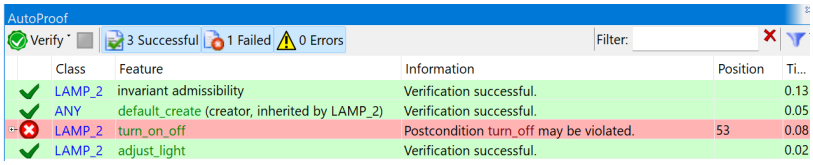
Fig. 43. (a) Verification result of `LAMP_1` in AutoProof; (b) Testing result of `test_LAMP_1_turn_on_off_1` in AutoTest

```
1   test_LAMP_1_turn_on_off_1
2     local
3       current_object: LAMP_1
4     do
5       create current_object
6       {P_INTERNAL}.set_boolean_field_ ("is_on", current_object, true)
7         -- current_object.is_on = true
8       {P_INTERNAL}.set_integer_field_ ("light_intensity", current_object,
            75)
9         -- current_object.light_intensity = 75
10      {P_INTERNAL}.set_integer_field_ ("previous_light_intensity",
            current_object, 0)
11        -- current_object.previous_light_intensity = 0
12      current_object.turn_on_off
13    end
```

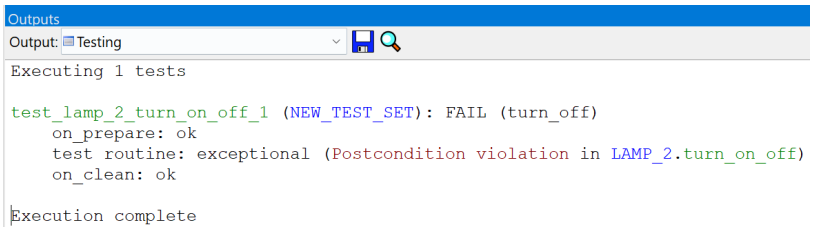
Fig. 44. Test case from failed proof of `lamp_is_turned_off`

Variant 2 of LAMP

- Fault injection: at line 40, remove the assignment `previous_light_intensity := light_intensity`.
- Resulting failure: as shown in Fig. 45(a), the injected fault leads to the violation of postcondition `turn_off` in the procedure `turn_on_off`.
- Cause of the failure: incorrect implementation of the body of `turn_on_off`; the postcondition `turn_off` requires that, the `previous_light_intensity` should store, when the light is turned off, the value of `light_intensity`; this is missing in the implementation.
- Proof time: 0.278 sec
- Test generation time: 0.205 sec
- Resulting test case: Fig. 46 shows the test case, which calls `turn_on_off` with input — `is_on` = `true`, `light_intensity` = `100`, `previous_light_intensity` = `0`.
- Testings result: as shown in Fig. 45(b), execution of the test case raises an exception of postcondition violation of `turn_off`, which corresponds to the same proof failure.
- Comment: the test is useful as its execution shows a specific trace illustrating how the program goes to the same contract violation as in the proof.



(a)



(b)

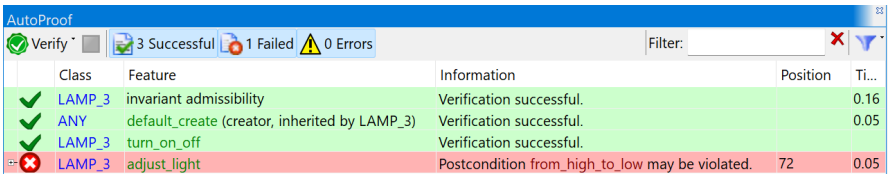
Fig. 45. (a) Verification result of `LAMP_2` in AutoProof; (b) Testing result of `test_LAMP_2_turn_on_off_1` in AutoTest

```
1   test_LAMP_2_turn_on_off_1
2     local
3       current_object: LAMP_2
4     do
5       create current_object
6       {P_INTERNAL}.set_boolean_field_ ("is_on", current_object, true)
7         -- current_object.is_on = true
8       {P_INTERNAL}.set_integer_field_ ("light_intensity", current_object,
            100)
9         -- current_object.light_intensity = 100
10      {P_INTERNAL}.set_integer_field_ ("previous_light_intensity",
            current_object, 0)
11        -- current_object.previous_light_intensity = 0
12      current_object.turn_on_off
13    end
```

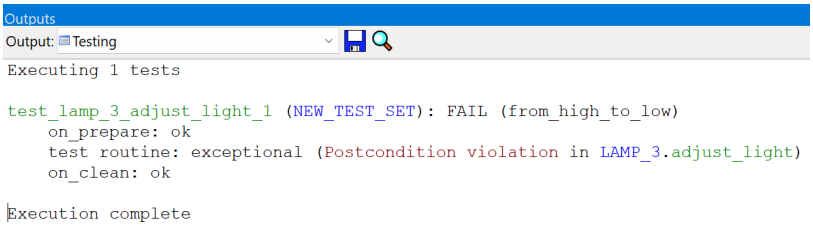
Fig. 46. Test case from failed proof of `turn_off`

Variant 3 of LAMP

- Fault injection: at line 59, in the body of `adjust_light`, change the right-hand side of the assignment from "`Low_intensity`" to "`Medium_intensity`".
- Resulting failure: as shown in Fig. 47(a), the injected fault causes the violation of postcondition `from_high_to_low` of the `adjust_light` routine.
- Cause of the failure: incorrect implementation of the routine body of `adjust_light`.
- Proof time: 0.265 sec
- Test generation time: 0.222 sec
- Resulting test case: Fig. 48 shows the test case from Proof2Test, which calls `adjust_light` with input — `is_on` = `true`, `light_intensity` = `100`, `previous_light_intensity` = `0`.
- Testings result: execution of the test case raises an exception of postcondition violation of `from_high_to_low`, as shown in Fig. 47(b), which corresponds to the same proof failure.
- Comment: the test is useful as its execution shows a specific trace illustrating how the program goes to the same contract violation as in the proof.



(a)



(b)

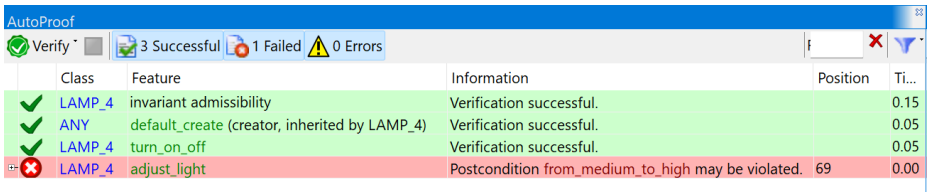
Fig. 47. (a) Verification result of `LAMP_3` in AutoProof; (b) Testing result of `test_LAMP_3_turn_on_off_1` in AutoTest

```
1   test_LAMP_3_adjust_light_1
2     local
3       current_object: LAMP_3
4     do
5       create current_object
6       {P_INTERNAL}.set_boolean_field_ ("is_on", current_object, true)
7         -- current_object.is_on = true
8       {P_INTERNAL}.set_integer_field_ ("light_intensity", current_object,
            100)
9         -- current_object.light_intensity = 100
10      {P_INTERNAL}.set_integer_field_ ("previous_light_intensity",
            current_object, 0)
11        -- current_object.previous_light_intensity = 0
12      current_object.adjust_light
13    end
```

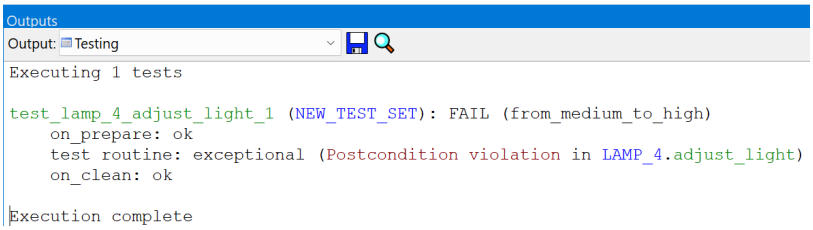
Fig. 48. Test case from failed proof of `from_high_to_low`

Variant 4 of LAMP

- Fault injection: at line 57, change the right-hand side of the assignment from "`High_intensity`" to "`Medium_intensity`".
- Resulting failure: as shown in Fig. 49(a), the postcondition `from_high_to_low` is violated.
- Cause of the failure: incorrect implementation.
- Proof time: 0.270 sec
- Test generation time: 0.208 sec
- Resulting test case: Fig. 50 shows the test case, which calls `adjust_light` with input — `is_on` = `true`, `light_intensity` = 75, `previous_light_intensity` = 100.
- Testings result: execution of the test case raises an exception of postcondition violation of `from_medium_to_high`, as shown in Fig. 49(b), which corresponds to the same proof failure.



(a)



(b)

Fig. 49. (a) Verification result of `LAMP_4` in AutoProof; (b) Testing result of `test_LAMP_4_turn_on_off_1` in AutoTest

```
1   test_LAMP_4_adjust_light_1
2     local
3       current_object: LAMP_4
4     do
5       create current_object
6       {P_INTERNAL}.set_boolean_field_ ("is_on", current_object, true)
7         -- current_object.is_on = true
8       {P_INTERNAL}.set_integer_field_ ("light_intensity", current_object,
            75)
9         -- current_object.light_intensity = 75
10      {P_INTERNAL}.set_integer_field_ ("previous_light_intensity",
            current_object, 100)
11        -- current_object.previous_light_intensity = 100
12      current_object.adjust_light
13    end
```

Fig. 50. Test case from failed proof of `from_medium_to_high`

3 Examples with loops

3.1 BINARY_SEARCH

The BINARY_SEARCH class, as shown below, implements the binary search algorithm, which aims to search a value in a sorted integer array by repeatedly dividing the search interval in half. Fig.51 shows the verification result of the class: the implementation is correct with respect to the specification. Based on the correct version, 6 variants of the class are derived and further discussed below.

```
1  class
2      BINARY_SEARCH
3
4  feature -- Binary search
5      binary_search(a: V_ARRAY [INTEGER]; value: INTEGER): INTEGER
6              -- Index of 'value' in 'a' using binary search. Return 0if
                   not found.
7              -- https://en.wikipedia.org/wiki/Binary_search_algorithm#
                   Iterative
8          note
9              status: impure
10         require
11             no_overflow: a.count <{INTEGER}.max_value
12             a_sorted: across 1|.. | a.count as i all
13                           across 1|.. | a.count as j all
14                               i ≤ j implies a.sequence[i] ≤ a.sequence[j]
                                     end end
15             a_size_limit: a.count > 0 and a.count ≤ 10
16             a_valid_bound: a.lower < a.upper and a.lower = 1
17         local
18             low, up, middle: INTEGER
19         do
20             from
21                 low := a.lower
22                 up := a.upper + 1
23                 Result := a.lower - 1
24             invariant
25                 low_and_up_range: a.lower ≤ low and low ≤ up and up ≤
                       a.upper + 1
26                 valid_bound: a.lower < a.upper
27                 result_range: Result = a.lower - 1 or a.lower ≤ Result
                       and Result ≤ a.upper
28                 not_in_lower_part: across 1|.. | (low - a.lower) as i all a
                       .sequence[i] <value end
29                 not_in_upper_part: across (up - a.lower + 1) |.. | a.
                       sequence.count as i all value < a.sequence[i] end
```

```
30                  found: (Result ≥ a.lower and Result ≤ a.upper) implies
                        (a.sequence[Result − a.lower + 1] = value)
31              until
32                  low ≥ up or Result ≥ a.lower
33              loop
34                  middle := low + ((up − low) // 2)
35                  if a[middle] < value then
36                      low := middle + 1
37                  elseif a[middle] > value then
38                      up := middle
39                  else
40                      Result := middle
41                  end
42              variant
43                  (a.upper − Result) + (up − low)
44              end
45          ensure
46              present: a.sequence.has (value) = (Result ≥ a.lower and
                    Result ≤ a.upper)
47              not_present: not a.sequence.has (value) = (Result = a.lower
                    − 1)
48              found_if_present: (Result ≥ a.lower and Result ≤ a.upper)
                    implies (a.sequence[Result − a.lower + 1] = value)
49          end
50  end
```

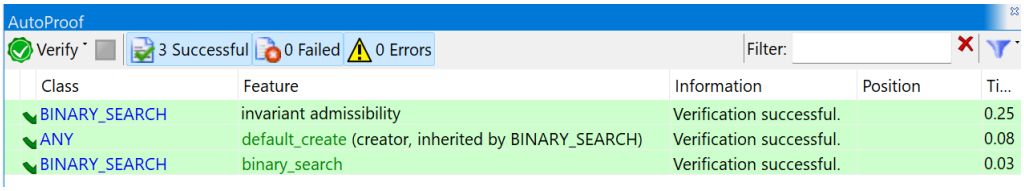
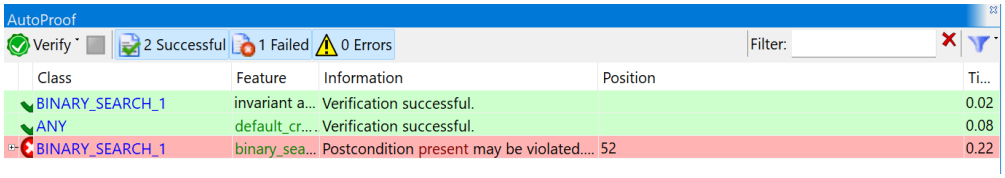


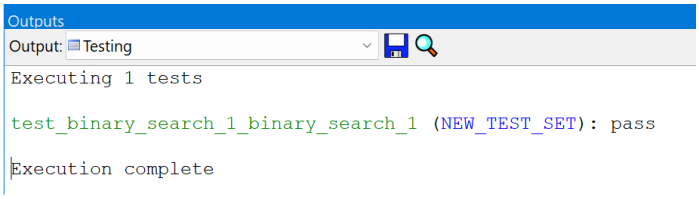
Fig. 51. Proof result of BINARY_SEARCH in AutoProof

Variant 1 of `BINARY_SEARCH`

- Fault injection: at line 30, remove the loop invariant `found`.
- Resulting failure: as shown in Fig. 52(a), the injected fault leads to the violation of the postcondition `present`.
- Cause of the failure: weakness/incompleteness of loop invariant.
- Proof time: 0.327 sec
- Test generation time: 0.471 sec
- Resulting test case: Fig. 53 shows the test case, which calls `binary_search` with input — `a` [1] = 0, `a` [2] = 2147457740, `value` = 2147457741.
- Testings result: as shown in Fig. 52(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 from) is not a real "counterexample" — it does not reveal the fault in the implementation and thus running the resulting test will not raise any exception; the passing test, however, indicates the weakness of the loop invariant.



(a)



(b)

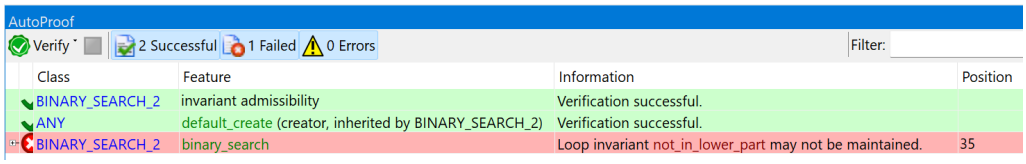
Fig. 52. (a) Verification result of `BINARY_SEARCH_1` in AutoProof; (b) Testing result of `test_BINARY_SEARCH_1_binary_search_1` in AutoTest

```
1   test_BINARY_SEARCH_1_binary_search_1
2     local
3       current_object: BINARY_SEARCH_1
4       a: V_ARRAY[INTEGER_32]
5       value: INTEGER_32
6       binary_search_result: INTEGER_32
7     do
8       create current_object
9       create a.make(1, 0)
10      a.force(0, 1)
11      a.force(2147457740, 2)
12
13      value := 2147457741
14      binary_search_result := current_object.binary_search (a, value)
15    end
```

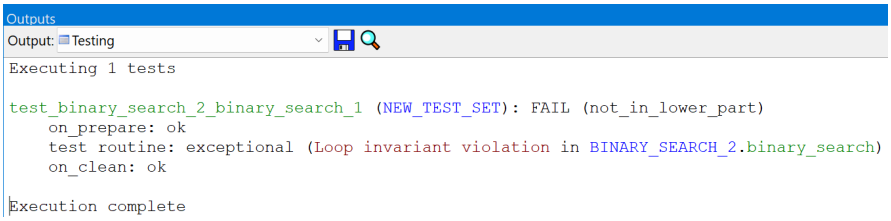
Fig. 53. Test case from failed proof of `present`

Variant 2 of BINARY_SEARCH

- Fault injection: at line 35, change the condition of the then branch from "`a [middle]<value`" into "`a [middle]≤value`".
- Resulting failure: as shown in Fig. 54(a), the injected fault results in the violation of loop invariant `not_in_lower_part`.
- Cause of the failure: incorrect implementation of loop body.
- Proof time: 0.395 sec
- Test generation time: 0.440 sec
- Resulting test case: Fig. 55 shows the test case from Proof2Test, which calls `binary_search` with the input array `a` and `value` instantiated with respect to the values in the counterexample.
- Testings result: as shown in Fig. 54(b), execution of the test case raises an exception of violation of loop invariant `not_in_lower_part`, which corresponds to the same proof failure.
- Comment: the test is useful as it reveals the fault in the program: the first element of the array has the same value as `value` (the value to search); when running the test, it is supposed that the loop ends at the first iteration — `value` is found at position 1 of the array; but due to the incorrect implementation of the routine, the iteration does not stop (the exit condition remains true after the first iteration) and continues for the second iteration, at which the loop invariant `not_in_lower_part` is evaluated to false (`a.sequence[1] = value`) and thus causes the exception.



(a)



(b)

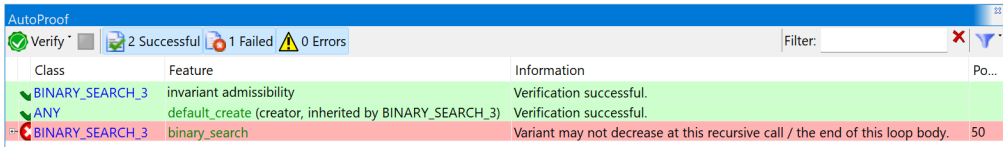
Fig. 54. (a) Verification result of `BINARY_SEARCH_2` in AutoProof; (b) Testing result of `test_BINARY_SEARCH_2_binary_search_1` in AutoTest

```
1   test_BINARY_SEARCH_2_binary_search_1
2     local
3       current_object: BINARY_SEARCH_2
4       a: V_ARRAY[INTEGER_32]
5       value: INTEGER_32
6       binary_search_result: INTEGER_32
7     do
8       create current_object
9       create a.make(1, 0)
10      a.force((−2147458457), 1)
11      a.force((−2147458457), 2)
12      a.force((−2147458457), 3)
13      a.force((−2147458457), 4)
14      a.force((−2147458457), 5)
15      a.force((−2147439710), 6)
16
17      value := (−2147458457)
18      binary_search_result := current_object.binary_search (a, value)
19    end
```

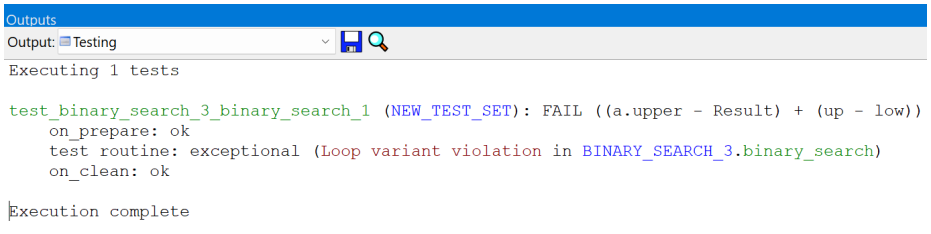
Fig. 55. Test case from failed proof of `not_in_lower_part`

Variant 3 of BINARY_SEARCH

- Fault injection: at line 36, change the assignment from "`low := middle + 1`" into "`low := middle`".
- Resulting failure: as shown in Fig. 56(a), the injected fault results in the violation that the variant of the loop does not decrease.
- Cause of the failure: incorrect implementation of loop body.
- Proof time: 0.325 sec
- Test generation time: 0.398 sec
- Resulting test case: Fig. 57 shows the test case from Proof2Test, which calls `binary_search` with input — `a` [1] = −22115, `a` [2] = −7979, `value` = 0.
- Testings result: as shown in Fig. 56(b), execution of the test case raises an exception of violation related to loop variant, which corresponds to the same proof failure.
- Comment: the test is useful as it reveals a bug in the program: initially, `low` = 1 and `upper` = 2; at the first iteration, the program assigns `middle` with 1 (at line 34); the condition of the then branch at line 35 is evaluated true,



(a)



(b)

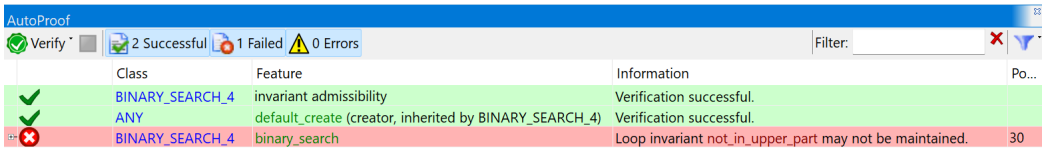
Fig. 56. (a) Verification result of `BINARY_SEARCH_3` in AutoProof; (b) Testing result of `test_BINARY_SEARCH_3_binary_search_1` in AutoTest

```
1   test_BINARY_SEARCH_3_binary_search_1
2     local
3       current_object: BINARY_SEARCH_3
4       a: V_ARRAY[INTEGER_32]
5       value: INTEGER_32
6       binary_search_result: INTEGER_32
7     do
8       create current_object
9       create a.make(1, 0)
10      a.force((−22115), 1)
11      a.force((−7979), 2)
12
13      value := 0
14      binary_search_result := current_object.binary_search (a, value)
15    end
```

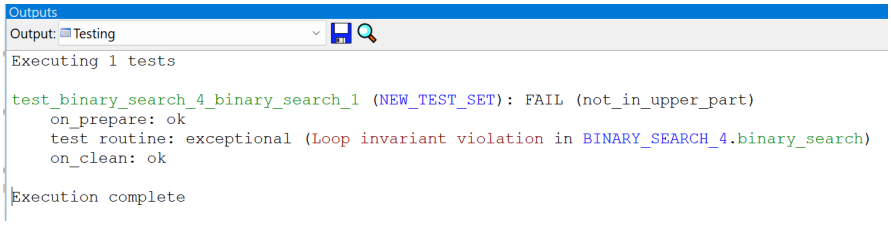
Fig. 57. Test case from failed proof of "variant not decrease"

Variant 4 of BINARY_SEARCH

- Fault injection: at line 37, change the condition of the `elseif` branch from "a[middle]>value" into "a[middle]≥value".
- Resulting failure: as shown in Fig. 58(a), the loop invariant of `not_in_upper_part` is violated.
- Cause of the failure: incorrect implementation of loop body.
- Proof time: 0.288 sec
- Test generation time: 0.323 sec
- Resulting test case: Fig. 59 shows the test case from Proof2Test, which calls `binary_search` with input — `a` [1] = 6, `value` = 6.
- Testings result: as shown in Fig. 58(b), execution of the test case raises an exception of violation of loop invariant `not_in_upper_part`, which corresponds to the same proof failure.
- Comment: this variant is similar to Variant 3; the test is useful as it shows a concrete trace that leads to the violation of the same contract in the failed proof.



(a)



(b)

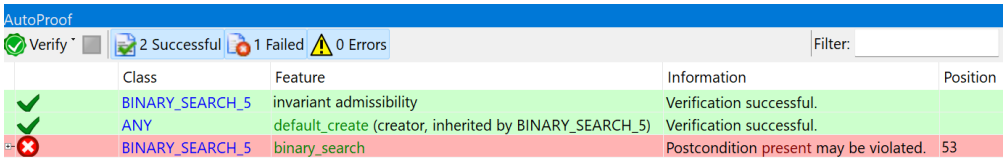
Fig. 58. (a) Verification result of `BINARY_SEARCH_4` in AutoProof; (b) Testing result of `test_BINARY_SEARCH_4_binary_search_1` in AutoTest

```
1   test_BINARY_SEARCH_4_binary_search_1
2     local
3       current_object: BINARY_SEARCH_4
4       a: SIMPLE_ARRAY[INTEGER_32]
5       value: INTEGER_32
6       binary_search_result: INTEGER_32
7     do
8       create current_object
9       create a.make_empty
10      a.force(6, 1)
11
12      value := 6
13      binary_search_result := current_object.binary_search (a, value)
14    end
```

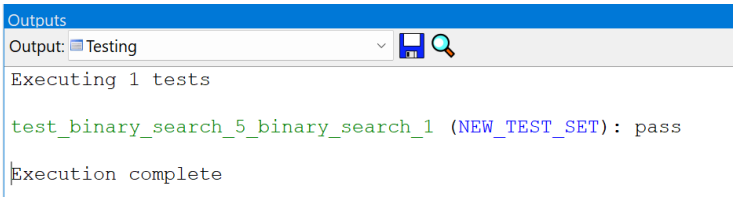
Fig. 59. Test case from the failed proof of `not_in_upper_part`

Variant 5 of `BINARY_SEARCH`

- Fault injection: at line 28, remove the loop invariant `not_in_lower_part`.
- Resulting failure: as shown in Fig. 60(a), the removal of the loop invariant causes the violation of postcondition of `present`.
- Cause of the failure: weakness/incompleteness of loop invariant.
- Proof time: 0.550 sec
- Test generation time: 0.382 sec
- Resulting test case: Fig. 61 shows the test case from Proof2Test, which calls `binary_search` with input extracted from the corresponding counterexample.
- Testings result: as shown in Fig. 60(b), execution of the test case does not raise any exception.
- Comment: this variant is similar to Variant 1; the passing test indicates that the proof failure is caused by the weakness of the auxiliary specification (loop invariant), not the implementation.



(a)



(b)

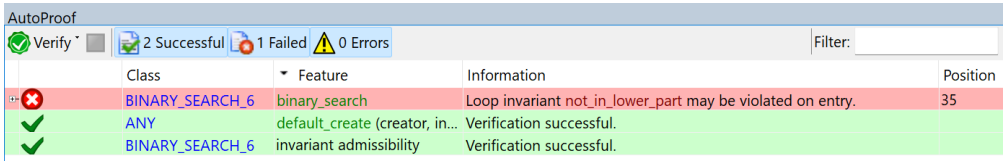
Fig. 60. (a) Verification result of `BINARY_SEARCH_5` in AutoProof; (b) Testing result of `test_BINARY_SEARCH_5_binary_search_1` in AutoTest

```
1   test_BINARY_SEARCH_5_binary_search_1
2     local
3       current_object: BINARY_SEARCH_5
4       a: V_ARRAY[INTEGER_32]
5       value: INTEGER_32
6       binary_search_result: INTEGER_32
7     do
8       create current_object
9       create a.make(1, 0)
10      a.force((−2147470858), 1)
11      a.force((−2147470858), 2)
12      a.force((−2147470858), 3)
13      a.force((−2147470858), 4)
14      a.force((−2147470858), 5)
15      a.force((−2147470858), 6)
16      a.force((−2147467706), 7)
17
18      value := (−2147467706)
19      binary_search_result := current_object.binary_search (a, value)
20    end
```

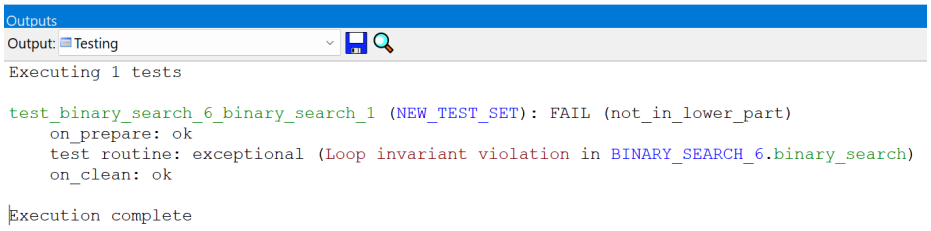
Fig. 61. Test case from failed proof of `present`

Variant 6 of BINARY_SEARCH

- Fault injection: at line 21, change the loop initialization "`low := a.lower`" into "`low := a.lower + 1`".
- Resulting failure: as shown in Fig. 62(a), the injected fault leads to the violation of the loop invariant `not_in_lower_part` at the entry of the loop (after loop initialization).
- Cause of the failure: incorrect implementation of loop initialization.
- Proof time: 0.356 sec
- Test generation time: 0.366 sec
- Resulting test case: Fig. 63 shows the test case from Proof2Test, which calls `binary_search` with input extracted from the corresponding counterexample.
- Testings result: as shown in Fig. 62(b), execution of the test case raises an exception of violation of the loop invariant `not_in_lower_part`, which is the same failed contract in the proof.
- 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.



(a)



(b)

Fig. 62. (a) Verification result of `BINARY_SEARCH_6` in AutoProof; (b) Testing result of `test_BINARY_SEARCH_6_binary_search_1` in AutoTest

```
1   test_BINARY_SEARCH_6_binary_search_1
2     local
3       current_object: BINARY_SEARCH_6
4       a: V_ARRAY[INTEGER_32]
5       value: INTEGER_32
6       binary_search_result: INTEGER_32
7     do
8       create current_object
9       create a.make(1, 0)
10      a.force(0, 1)
11      a.force(0, 2)
12      a.force(0, 3)
13      a.force(0, 4)
14      a.force(0, 5)
15      a.force(0, 6)
16      a.force(0, 7)
17      a.force(0, 8)
18
19      value := (-2147462410)
20      binary_search_result := current_object.binary_search (a, value)
21    end
```

Fig. 63. Test case from failed proof of `not_in_lower_part`

3.2 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). Fig.64 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.

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

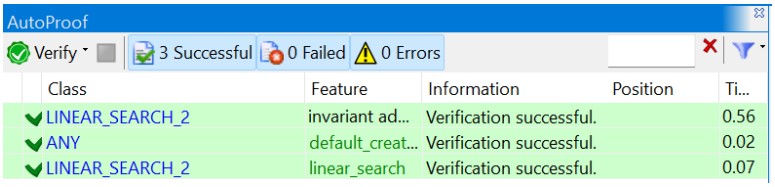
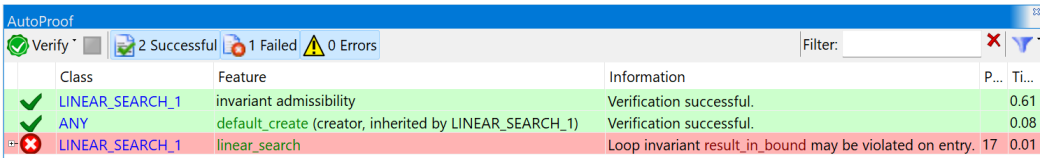


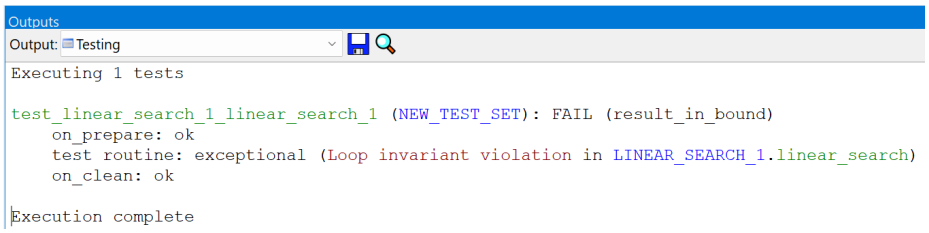
Fig. 64. Proof result of `LINEAR_SEARCH` in AutoProof

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 Fig. 65(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: Fig. 66 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 Fig. 65(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: similar to the Variant 6 of `BINARY_SEARCH`, 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.



(a)



(b)

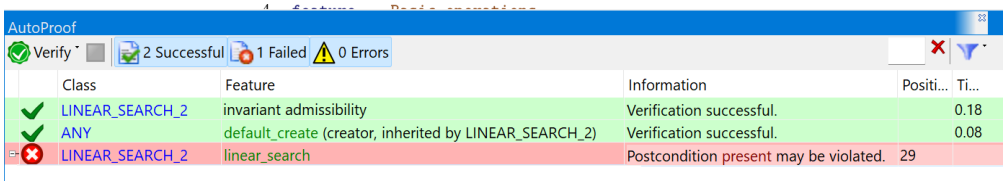
Fig. 65. (a) Verification result of `LINEAR_SEARCH_1` in AutoProof; (b) Testing result of `test_LINEAR_SEARCH_1_binary_search_1` in AutoTest

```
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
13      value := (-2147475929)
14      linear_search_result := current_object.linear_search (a, value)
15    end
```

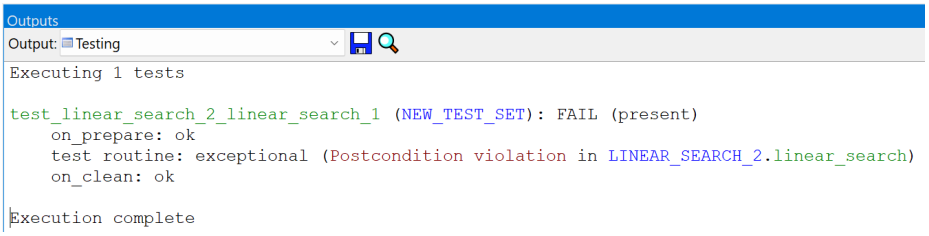
Fig. 66. Test case from failed proof of `result_in_bound`

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 Fig. 67(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: Fig. 68 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 Fig. 67(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 equality in the postcondition `present` — the left-hand part `a.sequence.has` (`value`) is false, as `value` does not match to any element of the input array `a`, while the right-hand part `Result`≤`a.count` is true (`Result` = 2 and `a.count` = 2).



(a)



(b)

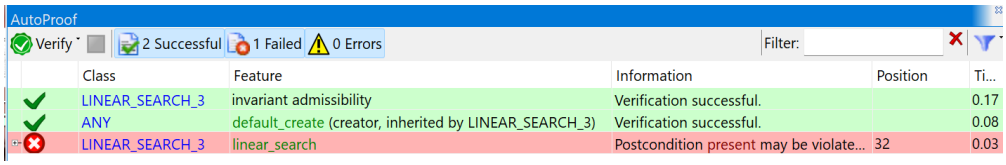
Fig. 67. (a) Verification result of `LINEAR_SEARCH_2` in AutoProof; (b) Testing result of `test_LINEAR_SEARCH_2_binary_search_1` in AutoTest

```
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      linear_search_result := current_object.linear_search (a, value)
15    end
```

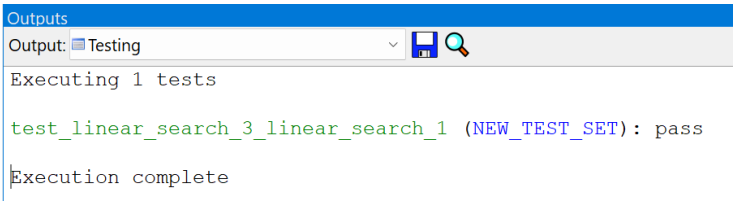
Fig. 68. Test case from failed proof of `present`

Variant 3 of LINEAR_SEARCH

- Fault injection: at line 13, remove the loop invariant `not_present_so_far`.
- Resulting failure: as shown in Fig. 69(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: Fig. 70 shows the test case from Proof2Test, which calls `linear_search` with input: `a`[1] = `0`, `a`[2] = `-2147462410`, `value` = `-2147462410`.
- Testings result: as shown in Fig. 69(b), execution of the test case does not raise any exception.
- Comment: this variant is similar to Variant 5 of BINARY_SEARCH; the passing test indicates that the proof failure is caused by the weakness of the auxiliary specification (loop invariant), not the implementation.



(a)



(b)

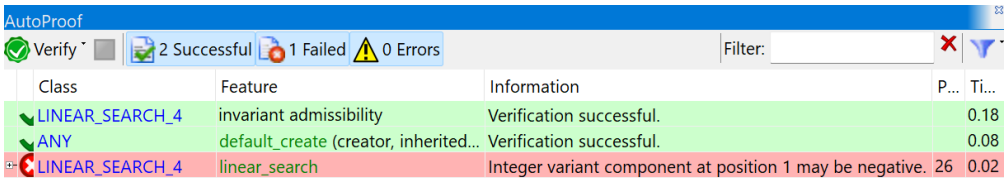
Fig. 69. (a) Verification result of `LINEAR_SEARCH_3` in AutoProof; (b) Testing result of `test_LINEAR_SEARCH_3_binary_search_1` in AutoTest

```
1   test_LINEAR_SEARCH_3_linear_search_1
2     local
3       current_object: LINEAR_SEARCH_3
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((−2147462410), 2)
12
13      value := (−2147462410)
14      linear_search_result := current_object.linear_search (a, value)
15    end
```

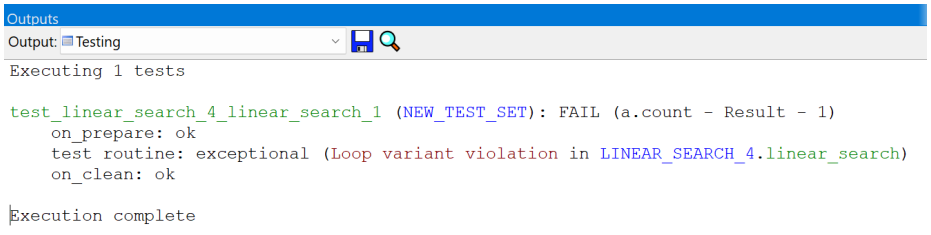
Fig. 70. Test case from failed proof of `present`

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 Fig. 71(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: Fig. 72 shows the test case from Proof2Test, which calls `linear_search` with input extracted from the corresponding counterexample.
- Testings result: as shown in Fig. 71(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.



(a)



(b)

Fig. 71. (a) Verification result of `LINEAR_SEARCH_4` in AutoProof; (b) Testing result of `test_LINEAR_SEARCH_4_binary_search_1` in AutoTest

```
test_LINEAR_SEARCH_4_linear_search_1
  local
    current_object: LINEAR_SEARCH_4
    a: SIMPLE_ARRAY[INTEGER_32]
    value: INTEGER_32
    linear_search_result: INTEGER_32
  do
    create current_object
    create a.make_empty
    a.force(0, 1)
    a.force(0, 2)
    a.force(0, 3)
    a.force(0, 4)
    a.force((−2147482506), 5)

    value := (−2147482505)
    linear_search_result := current_object.linear_search (a, value)
  end
```

Fig. 72. Test case from failed proof of "variant may be negative"

3.3 MAX_IN_ARRAY

The MAX_IN_ARRAY class, as presented below, implements an algorithm that computes the maximum element of an integer array a. Fig.73 shows the verification result of the class, which suggests a complete functional correctness. 6 variants of the class are generated by injecting different faults in the correct version, which will be discussed below.

```
class
    MAX_IN_ARRAY

feature -- Basic operations
    max_in_array (a: SIMPLE_ARRAY [INTEGER]): INTEGER
            -- Find the maximum element of 'a'.
        require
            array_not_empty: a.count > 0
        local
            i: INTEGER
        do
            Result := a [1]
            from
                i := 2
            invariant
                i_in_bounds: 2 ≤ i and i ≤ a.count + 1
                max_so_far: across 1|.. | (i − 1) as c all a.sequence [c]
                    ≤ Result end
```

```
18                  result_in_array: across 1|.. | (i − 1) as c some a.sequence [
                        c] = Result end
19              until
20                  i = a.count + 1
21              loop
22                  if a [i] > Result then
23                      Result := a [i]
24                  end
25                  i := i + 1
26                  variant
27                      a.count − i
28              end
29          ensure
30              is_maximum: across 1|.. | a.count as c all a.sequence [c] ≤
                    Result end
31              result_in_array: across 1|.. | a.count as c some a.sequence [c] =
                    Result end
32          end
33  end
```

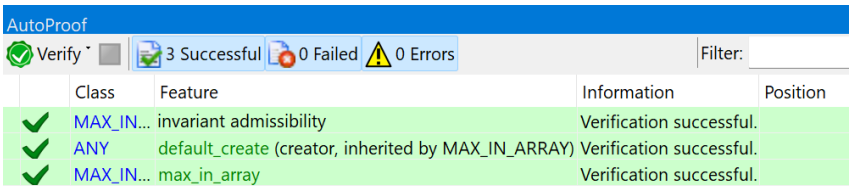
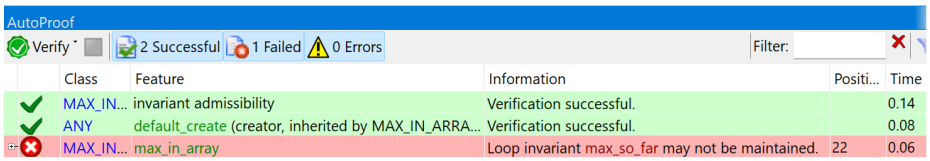


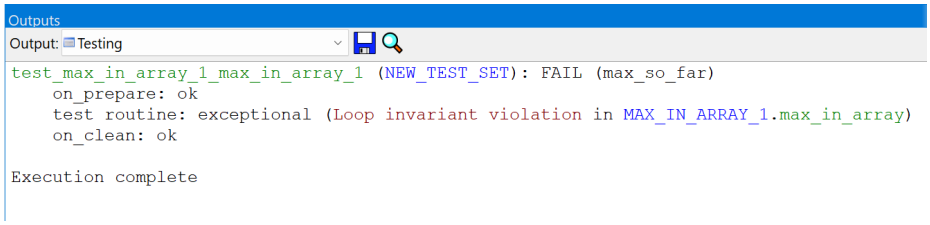
Fig. 73. Proof result of `MAX_IN_ARRAY` in AutoProof

Variant 1 of `MAX_IN_ARRAY`

- Fault injection: at line 22, change the condition of the then branch into "`a [i] <Result`".
- Resulting failure: as shown in Fig. 74(a), the loop invariant `max_so_far` is violated during the iteration of the loop.
- Cause of the failure: incorrect implementation of the loop body.
- Proof time: 0.288 sec
- Test generation time: 0.282 sec
- Resulting test case: Fig. 75 shows the test case, which calls `max_in_array` with input extracted from the corresponding counterexample.
- Testings result: as shown in Fig. 76, execution of the test case raise an exception of violating the loop invariant `max_so_far`, which corresponds to the same failure in the verification.
- Comment: the test is useful as it is able to show a concrete trace that leads to the contract violation; the values in the test input, however, is not that meaningful, as any other valid test input will have the same effect.



(a)



(b)

Fig. 74. (a) Verification result of `MAX_IN_ARRAY_1` in AutoProof; (b) Testing result of `test_MAX_IN_ARRAY_1_binary_search_1` in AutoTest

```
1   test_MAX_IN_ARRAY_1_max_in_array_1
2     local
3       current_object: MAX_IN_ARRAY_1
4       a: SIMPLE_ARRAY[INTEGER_32]
5       max_in_array_result: INTEGER_32
6     do
7       create current_object
8       create a.make_empty
9       a.force((−2147451757), 1)
10      a.force((−2147451757), 2)
11      a.force((−26641), 3)
12      a.force((−23245), 4)
13      a.force((−23245), 5)
14
15      max_in_array_result := current_object.max_in_array (a)
16    end
```

Fig. 75. Test case from failed proof of `max_so_far`

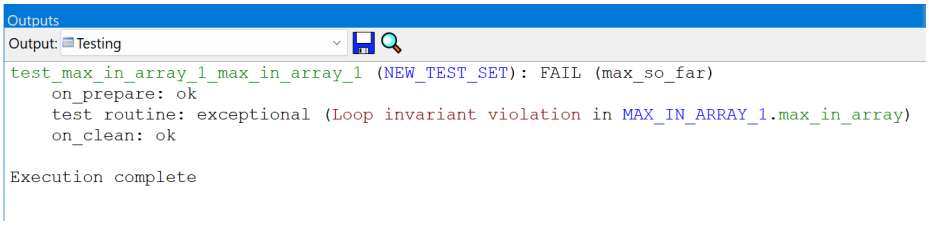
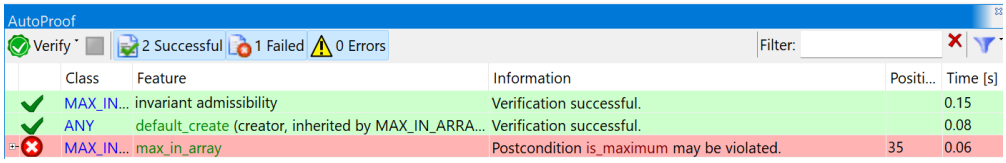


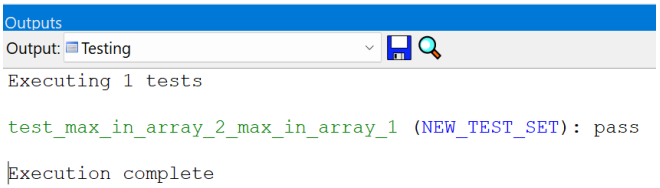
Fig. 76. Testing result of `test_MAX_IN_ARRAY_1_max_in_array_1` in AutoTest

Variant 2 of `MAX_IN_ARRAY`

- Fault injection: at line 17, remove the loop invariant `max_so_far`.
- Resulting failure: as shown in Fig. 77(a), the removal of the loop invariant results in the violation of postcondition *is_maximum*.
- Cause of the failure: weakness/incompleteness of loop invariant.
- Proof time: 0.284 sec
- Test generation time: 0.313 sec
- Resulting test case: Fig. 78 shows the test case from Proof2Test, which calls `max_in_array` with input extracted from the corresponding counterexample.
- Testings result: as shown in Fig. 77(b), execution of the test case passes all the specifications.
- Comment: this variant is similar to Variant 5 of `BINARY_SEARCH`; the passing test indicates that the proof failure is caused by the weakness of the auxiliary specification (loop invariant), not the implementation.



(a)



(b)

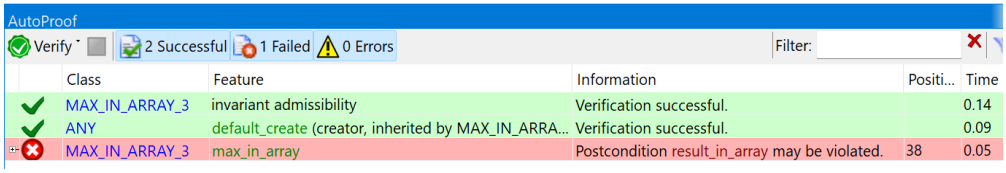
Fig. 77. (a) Verification result of `MAX_IN_ARRAY_2` in AutoProof; (b) Testing result of `test_MAX_IN_ARRAY_2_max_in_array_1` in AutoTest

```
1   test_MAX_IN_ARRAY_2_max_in_array_1
2     local
3       current_object: MAX_IN_ARRAY_2
4       a: SIMPLE_ARRAY [INTEGER_32]
5       max_in_array_result: INTEGER_32
6     do
7       create current_object
8       create a.make_empty
9       a.force ((−2147455548), 1)
10      a.force ((−2147455548), 2)
11      a.force (11798, 3)
12      a.force (0, 4)
13
14      max_in_array_result := current_object.max_in_array (a)
15    end
```

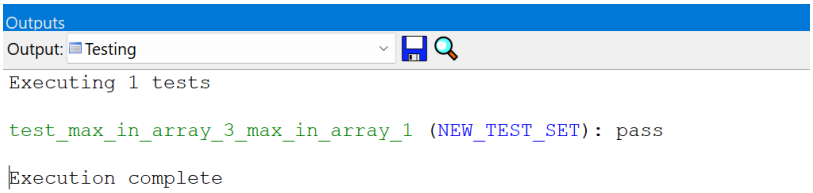
Fig. 78. Test case from failed proof of *is_maximum*

Variant 3 of MAX_IN_ARRAY

- Fault injection: at line 18, remove the loop invariant *result_in_array*.
- Resulting failure: as shown in Fig. 79(a), the removal of the loop invariant leads to the violation of the postcondition *result_in_array*.
- Cause of the failure: weakness/incompleteness of loop invariant.
- Proof time: 0.278 sec
- Test generation time: 0.307 sec
- Resulting test case: Fig. 80 shows the test case from Proof2Test, which calls `max_in_array` with input extracted from the corresponding counterexample.
- Testings result: as shown in Fig. 79(b), execution of the test case passes all the specifications.
- Comment: similar to Variant 2, the passing test indicates that the proof failure is caused by the weakness of the loop invariant.



(a)



(b)

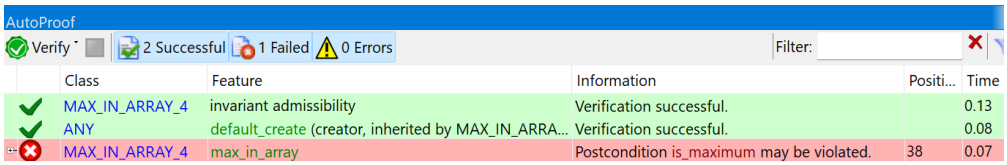
Fig. 79. (a) Verification result of `MAX_IN_ARRAY_3` in AutoProof; (b) Testing result of `test_MAX_IN_ARRAY_3_max_in_array_1` in AutoTest

```
1   test_MAX_IN_ARRAY_3_max_in_array_1
2     local
3       current_object: MAX_IN_ARRAY_3
4       a: SIMPLE_ARRAY[INTEGER_32]
5       max_in_array_result: INTEGER_32
6     do
7       create current_object
8       create a.make_empty
9       a.force(0, 1)
10      a.force(0, 2)
11
12      max_in_array_result := current_object.max_in_array (a)
13    end
```

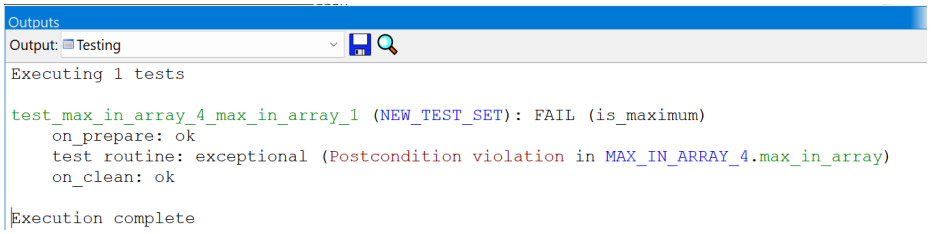
Fig. 80. Test case from failed proof of *result_in_array*

Variant 4 of `MAX_IN_ARRAY`

- Fault injection: at line 20, change the exit condition of the loop from "`i = a.count + 1`" into "`i = a.count`".
- Resulting failure: as shown in Fig. 81(a), the injected faults causes the violation of the postcondition *is_maximum*.
- Cause of the failure: incorrect exit condition of the loop.
- Proof time: 0.286 sec
- Test generation time: 0.323 sec
- Resulting test case: Fig. 82 shows the test case from Proof2Test, which calls `max_in_array` with input extracted from the corresponding counterexample.
- Testings result: as shown in Fig. 81(b), execution of the test case raises an exception of postcondition violation of *is_maximum*, which corresponds to the same proof failure.
- Comment: this test is useful as it shows a specific scenario from which the program will go to a failing state, violating the same failed contract in the proof; during the execution of the test, after going through 2 iterations, the program terminates with `Result = 0`; this reveals the program fault: the program terminates too early to reach the actual maximum value of `a` (the third element of `a`).



(a)



(b)

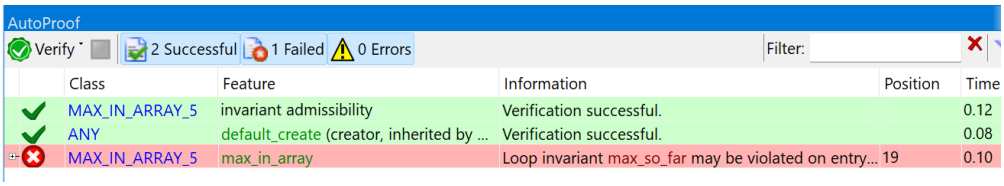
Fig. 81. (a) Verification result of `MAX_IN_ARRAY_4` in AutoProof; (b) Testing result of `test_MAX_IN_ARRAY_4_max_in_array_1` in AutoTest

```
1   test_MAX_IN_ARRAY_4_max_in_array_1
2     local
3       current_object: MAX_IN_ARRAY_4
4       a: SIMPLE_ARRAY [INTEGER_32]
5       max_in_array_result: INTEGER_32
6     do
7       create current_object
8       create a.make_empty
9       a.force (0, 1)
10      a.force (0, 2)
11      a.force (28101, 3)
12
13      max_in_array_result := current_object.max_in_array (a)
14    end
```

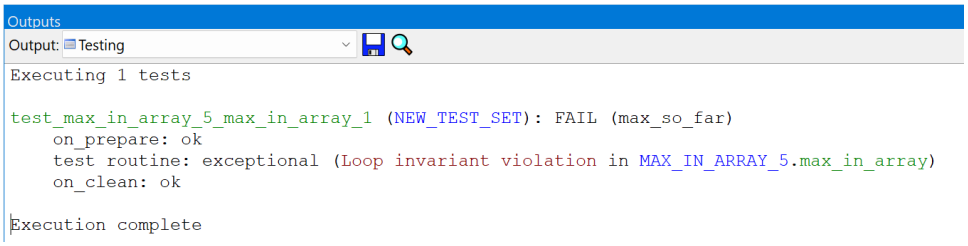
Fig. 82. Test case from failed proof of *is_maximum*

Variant 5 of MAX_IN_ARRAY

- Fault injection: at line 12, change the statement from "`Result` := `a` [1]" into "`Result` := 0".
- Resulting failure: as shown in Fig. 83(a), the injected fault causes the violation of the loop invariant `max_so_far` on the entry.
- Cause of the failure: incorrect implementation in the code snippet before the loop.
- Proof time: 0.290 sec
- Test generation time: 0.303 sec
- Resulting test case: Fig. 84 shows the test case from Proof2Test, which calls `max_in_array` with input extracted from the corresponding counterexample.
- Testings result: as shown in Fig. 83(b), execution of the test case raises an exception of postcondition violation of `max_so_far`, which corresponds to the same proof failure.
- Comment: the test is useful as it is able to show a concrete trace that leads to the contract violation; the values in the test input, however, is not that meaningful, as any other valid test input will have the same effect.



(a)



(b)

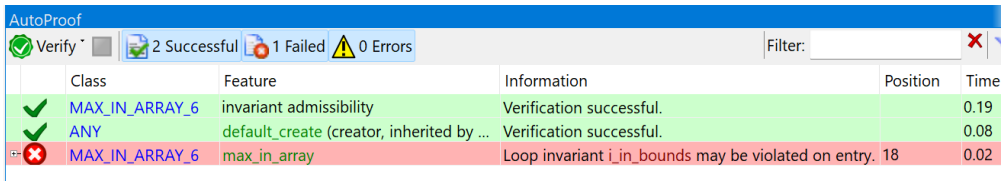
Fig. 83. (a) Verification result of `MAX_IN_ARRAY_5` in AutoProof; (b) Testing result of `test_MAX_IN_ARRAY_5_max_in_array_1` in AutoTest

```
1   test_MAX_IN_ARRAY_5_max_in_array_1
2     local
3       current_object: MAX_IN_ARRAY_5
4       a: SIMPLE_ARRAY [INTEGER_32]
5       max_in_array_result: INTEGER_32
6     do
7       create current_object
8       create a.make_empty
9       a.force (8856, 1)
10      a.force (8856, 2)
11      a.force (8856, 3)
12      a.force (8856, 4)
13
14      max_in_array_result := current_object.max_in_array (a)
15    end
```

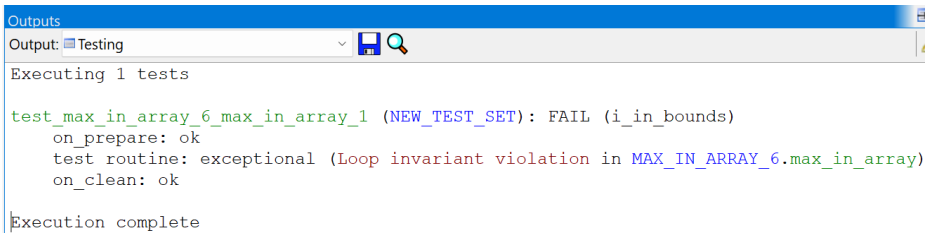
Fig. 84. Test case from failed proof of `max_so_far`

Variant 6 of `MAX_IN_ARRAY`

- Fault injection: at line 14, change the code from "`i := 2`" into "`i := 1`".
- Resulting failure: as shown in Fig. 85(a), the injected fault causes the violation of the loop invariant `i_in_bounds` at the entry of the loop (after loop initialization).
- Cause of the failure: incorrect implementation of the loop initialization.
- Proof time: 0.282 sec
- Test generation time: 0.315 sec
- Resulting test case: Fig. 86 shows the test case, which calls `max_in_array` with input extracted from the corresponding counterexample.
- Testings result: as shown in Fig. 85(b), execution of the test case raises an exception of loop invariant of `i_in_bounds`, which corresponds to the same proof failure.
- Comment: similar to Variant 5, the test is useful as it is able to show a concrete trace that leads to the contract violation; the values in the test input, however, is not that meaningful, as any other valid test input will have the same effect.



(a)



(b)

Fig. 85. (a) Verification result of `MAX_IN_ARRAY_6` in AutoProof; (b) Testing result of `test_MAX_IN_ARRAY_6_max_in_array_1` in AutoTest

```
test_MAX_IN_ARRAY_6_max_in_array_1
  local
    current_object: MAX_IN_ARRAY_6
    a: SIMPLE_ARRAY[INTEGER_32]
    max_in_array_result: INTEGER_32
  do
    create current_object
    create a.make_empty
    a.force((−2147471851), 1)
    a.force((−2147471851), 2)

    max_in_array_result := current_object.max_in_array (a)
  end
```

Fig. 86. Test case from failed proof of `i_in_bounds`

3.4 SQUARE_ROOT

The `SQUARE_ROOT` class, as shown below, calculates two approximate square roots `x` and `y` of a given positive integer `n`: the value of `n` falls between x^2 and y^2; if `n` is perfect square, then $y = x$ and $x^2 = n$, otherwise $y = x + 1$. Fig. 87 presents the verification result of the class, which indicates a full functional correctness. By injecting different faults in the correct version, 4 variants of `SQUARE_ROOT` are derived and discussed below.

```
class
    SQUARE_ROOT
feature
    square_root (n: INTEGER): TUPLE [x: INTEGER; y: INTEGER]
            -- 'x' and 'y' are two approximate square roots of 'n'
        require
            valid_n: n ≥ 0
        local
            x1, x2, mid: INTEGER
        do
            from
                x1 := 0
                x2 := n
            invariant
                valid_result: (x1 = x2 and x1 * x1 = n) or (x1 < x2 and x1
                    * x1 < n and x2 * x2 ≥ n)
            until
                x2 − x1 ≤ 1 or x1 = x2
            loop
                mid := (x1 + x2) // 2-- integer division
                if mid * mid = n then
```

```
21                      x1 := mid
22                      x2 := mid
23                  else
24                      if mid * mid < n then
25                          x1 := mid
26                      else
27                          x2 := mid
28                      end
29                  end
30              variant
31                  x2 - x1
32              end
33              Result := [x1, x2]
34          ensure
35              valid_result: (Result.x = Result.y and Result.x * Result.x = n
                    )
36                  or (Result.x + 1= Result.y and Result.x * Result.x <n
                        and Result.y * Result.y ≥ n)
37          end
38  end
```

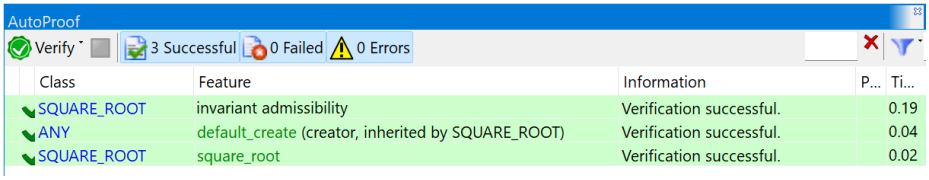
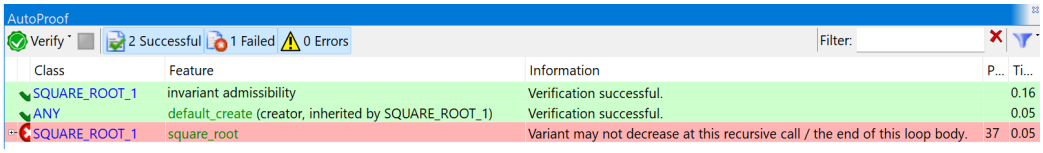


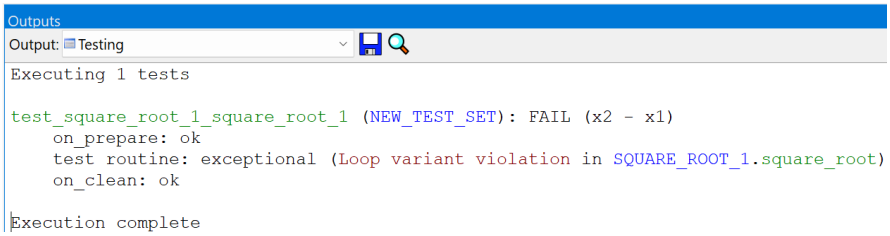
Fig. 87. Proof result of SQUARE_ROOT in AutoProof

Variant 1 of SQUARE_ROOT

- Fault injection: at line 17, change the left part of the exit condition from "x2− x1≤1" into "x2− x1<1".
- Resulting failure: as shown in Fig. 88(a), the injected fault leads to the failure that the loop variant is not decreased during the loop iteration.
- Cause of the failure: incorrect exit condition of the loop.
- Proof time: 0.258 sec
- Test generation time: 0.192 sec
- Resulting test case: Fig. 89 shows the test case from Proof2Test, which calls square_root with input argument n = 10.
- Testings result: as shown in Fig. 88(b), execution of the test case raises an exception related to loop variant, which corresponds to the same proof failure.
- Comment: the test is useful as it is able to show how the value of variant varies at each iteration.



(a)



(b)

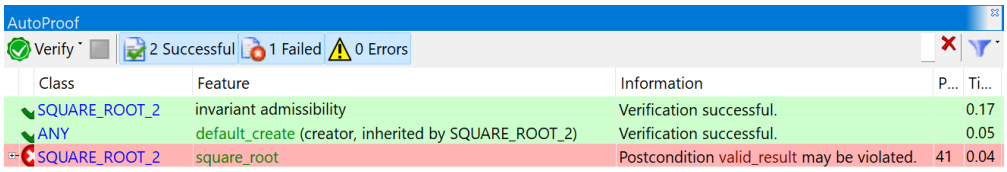
Fig. 88. (a) Verification result of SQUARE_ROOT_1 in AutoProof; (b) Testing result of test_SQUARE_ROOT_1_square_root_1 in AutoTest

```
1   test_SQUARE_ROOT_1_square_root_1
2     local
3       current_object: SQUARE_ROOT_1
4       n: INTEGER_32
5       square_root_result: TUPLE[INTEGER_32,INTEGER_32]
6     do
7       create current_object
8       n := 10
9       square_root_result := current_object.square_root (n)
10    end
```

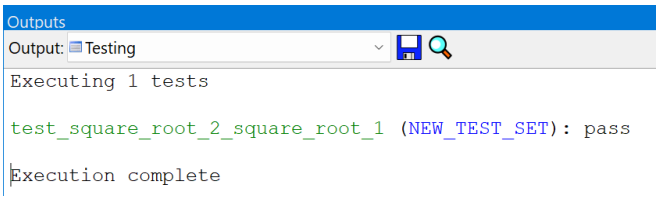
Fig. 89. Test case from failed proof of "variant may not decrease"

Variant 2 of SQUARE_ROOT

- Fault injection: at line 15, remove the loop invariant `valid_result`.
- Resulting failure: as shown in Fig. 90(a), the injected fault leads to the violation of postcondition `valid_result`.
- Cause of the failure: weakness/incompleteness of loop invariant.
- Proof time: 0.257 sec
- Test generation time: 0.205 sec
- Resulting test case: Fig. 91 shows the test case from Proof2Test, which calls `square_root` with input argument `n = 0`.
- Testings result: as shown in Fig. 90(b), execution of the test case does not raise any exception.
- Comment: similar to Variant 2 of MAX_IN_ARRAY, the passing test indicates that the proof failure is caused by the weakness of the loop invariant.



(a)



(b)

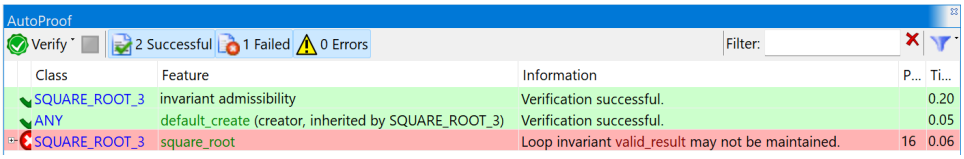
Fig. 90. (a) Verification result of `SQUARE_ROOT_2` in AutoProof; (b) Testing result of `test_SQUARE_ROOT_2_square_root_1` in AutoTest

```
1   test_SQUARE_ROOT_2_square_root_1
2     local
3       current_object: SQUARE_ROOT_2
4       n: INTEGER_32
5       square_root_result: TUPLE[INTEGER_32,INTEGER_32]
6     do
7       create current_object
8       n := 0
9       square_root_result := current_object.square_root (n)
10    end
```

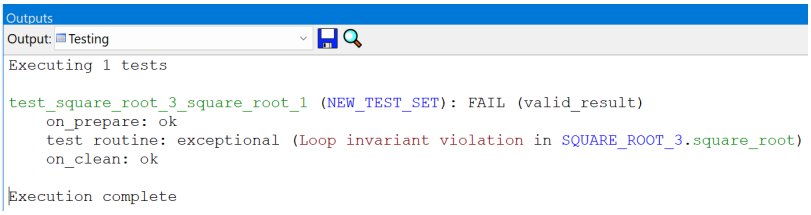
Fig. 91. Test case from failed proof of `valid_result`

Variant 3 of SQUARE_ROOT

- Fault injection: change the condition of the then branch at line 20 from "`mid * mid = n`" into "`mid * mid ≠ n`".
- Resulting failure: as shown in Fig. 92(a), the injected fault results in the violation of the loop invariant `result_so_far`.
- Cause of the failure: incorrect implementation of the loop body.
- Proof time: 0.308 sec
- Test generation time: 0.181 sec
- Resulting test case: Fig. 93 shows the test case from Proof2Test, which calls `square_root` with input argument `n = 3`.
- Testings result: as shown in Fig. 92(b), execution of the test case raise an exception of violation of loop invariant `valid_result`, which corresponds to the same proof failure.
- Comment: the test is useful as it is able to show a concrete trace that leads to the same contract violation as in the proof; during the execution of the test, initially, `x1= 0` and `x2= 3`; at the first iteration, the program assigns `mid` with 1 (line 19); the condition of the then branch is true (`mid * mid ≠ n`) and the program assigns both `x1` and `x2` with the value of `mid` (line 21 and 22); at the beginning of the second iteration, the loop invariant `valid_result` is evaluated as false (`x1= x2` is true but `x1*x1=n` is false).



(a)



(b)

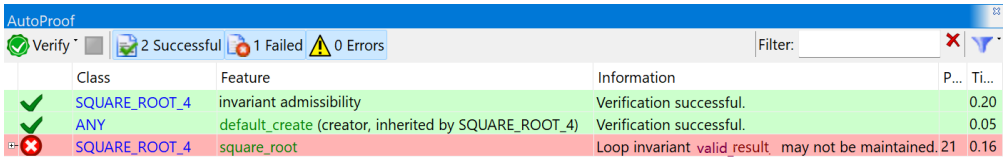
Fig. 92. (a) Verification result of `SQUARE_ROOT_3` in AutoProof; (b) Testing result of `test_SQUARE_ROOT_3_square_root_1` in AutoTest

```
1   test_SQUARE_ROOT_3_square_root_1
2     local
3       current_object: SQUARE_ROOT_3
4       n: INTEGER_32
5       square_root_result: TUPLE [INTEGER_32,INTEGER_32]
6     do
7       create current_object
8       n := 3
9       square_root_result := current_object.square_root (n)
10    end
```

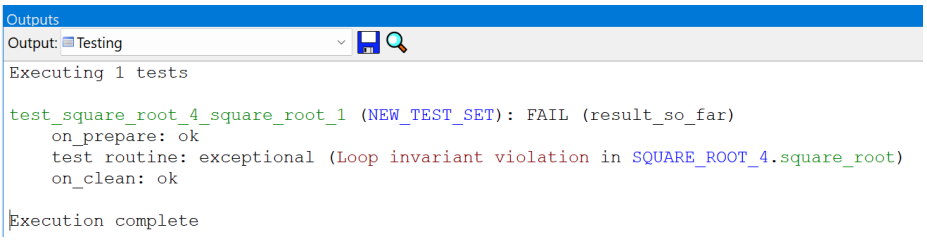
Fig. 93. Test case from failed proof of `valid_result`

Variant 4 of SQUARE_ROOT

- Fault injection: at line 24, change the condition of the then branch from "`mid * mid <n`" into "`mid * mid > n`".
- Resulting failure: as shown in Fig. 94(a), the loop invariant `valid_result` is not satisfied during the loop iteration.
- Cause of the failure: incorrect implementation of the loop body.
- Proof time: 0.400 sec
- Test generation time: 0.194 sec
- Resulting test case: Fig. 95 shows the test case from Proof2Test, which calls `square_root` with input argument `n = 11`.
- Testings result: as shown in Fig. 94(b), execution of the test case raise an exception of violation of loop invariant `valid_result`, which corresponds to the same proof failure.
- Comment: the test is useful as it is able to show a concrete trace that leads to the same contract violation as in the proof; during the execution of the test, initially, `x1= 0` and `x2= 11`; at the first iteration, the program assigns `mid` with 5 (line 19); the condition of the `else` branch is true (`mid * mid ≠ n`); the condition of the `then` branch at line 24 is true (`mid * mid > n`), thus the program assigns `x1` with the value of `mid` (line 25); at the beginning of the second iteration, the loop invariant `valid_result` is evaluated as false (in the right-hand side of the `or` clause, `x1<x2` is true but `x1*x1<n` is false), causing an exception of the invariant violation.



(a)



(b)

Fig. 94. (a) Verification result of `SQUARE_ROOT_4` in AutoProof; (b) Testing result of `test_SQUARE_ROOT_4_square_root_1` in AutoTest

```
1   test_SQUARE_ROOT_4_square_root_1
2     local
3       current_object: SQUARE_ROOT_4
4       n: INTEGER_32
5       square_root_result: TUPLE[INTEGER_32,INTEGER_32]
6     do
7       create current_object
8       n := 11
9       square_root_result := current_object.square_root (n)
10    end
```

Fig. 95. Test case from failed proof of `result_so_far`

3.5 SUM_AND_MAX

The SUM_AND_MAX class, as listed below, computes the sum and maximum of an integer array a. Fig.96 shows the verification result of the class, which indicates a complete functional correctness. 6 faulty variants of the class are derived by injecting different faults in the correct version, which will be discussed below.

```
1  class
2      SUM_AND_MAX
3  feature
4      sum_and_max (a: SIMPLE_ARRAY [INTEGER]): TUPLE [sum, max: INTEGER]
5              -- Calculate sum and maximum of array 'a'.
6          require
7              a_not_void: a ≠ Void
8              natural_numbers: across 1|.. | a.count as ai all a.sequence [ai]
                   ≥ 0 end
9              array_not_empty: a.count > 0
10         local
11             i: INTEGER
12             sum, max: INTEGER
13         do
14             from
15                 i := 2; max := a[1]; sum := a[1]
16             invariant
17                 i_in_range: 1 ≤ i and i ≤ a.count + 1
18                 sum_and_max_not_negative: sum ≥ 0 and max ≥ 0
19                 partial_sum_and_max: sum ≤ (i − 1) * max
20                 max_so_far: across 1|.. | (i − 1) as ai all max ≥ a.
                       sequence [ai]
21             until
22                 i > a.count
23             loop
24                 if a [i] > max then
25                     max := a [i]
26                 end
27                 sum := sum + a [i]
28                 i := i + 1
29             end
30             Result := [sum, max]
31         ensure
32             sum_in_range: Result.sum ≤ a.count * Result.max
33             is_maximum: across 1|.. | a.count as ai all Result.max ≥ a.
                   sequence [ai] end
34             modify()
35         end
36 end
```

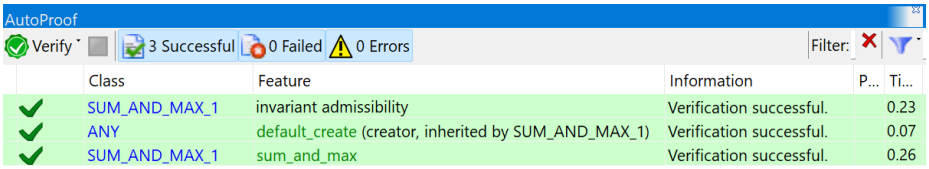
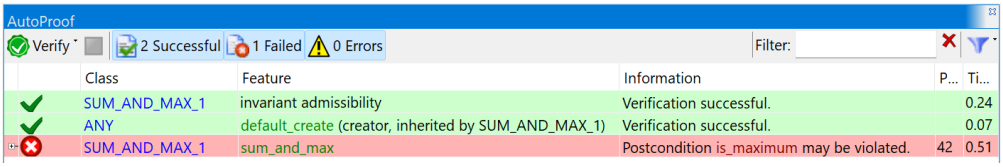


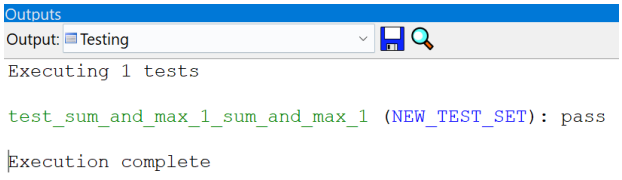
Fig. 96. Proof result of `SUM_AND_MAX` in AutoProof

Variant 1 of `SUM_AND_MAX`

- Fault injection: at line 20, remove the loop invariant `max_so_far`.
- Resulting failure: as shown in Fig. 97(a), the removal of the loop invariant causes the violation of postcondition *is_maximum*.
- Cause of the failure: weakness/incompleteness of the loop invariant.
- Proof time: 0.301 sec
- Test generation time: 0.336 sec
- Resulting test case: Fig. 98 shows the test case from Proof2Test, which calls `sum_and_max` with input argument `a`[1] = 2.
- Testings result: as shown in Fig. 97(b), execution of the test case does not raise any exception.
- Comment: similar to Variant 2 of `SQUARE_ROOT`, the passing test indicates that the proof failure is caused by the weakness of the loop invariant.



(a)



(b)

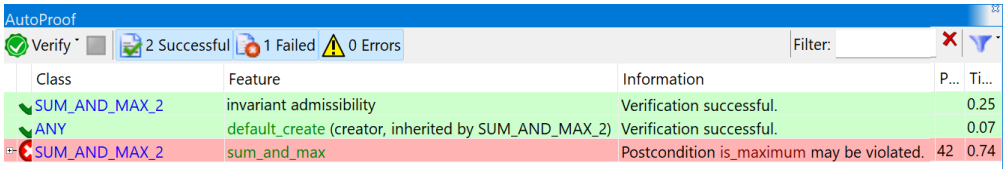
Fig. 97. (a) Verification result of `SUM_AND_MAX_1` in AutoProof; (b) Testing result of `test_SUM_AND_MAX_1_sum_and_max_1` in AutoTest

```
1   test_SUM_AND_MAX_1_sum_and_max_1
2     local
3       current_object: SUM_AND_MAX_1
4       a: SIMPLE_ARRAY[INTEGER_32]
5       sum_and_max_result: TUPLE[INTEGER_32,INTEGER_32]
6     do
7       create current_object
8       create a.make_empty
9       a.force(2, 1)
10
11      sum_and_max_result := current_object.sum_and_max (a)
12    end
```

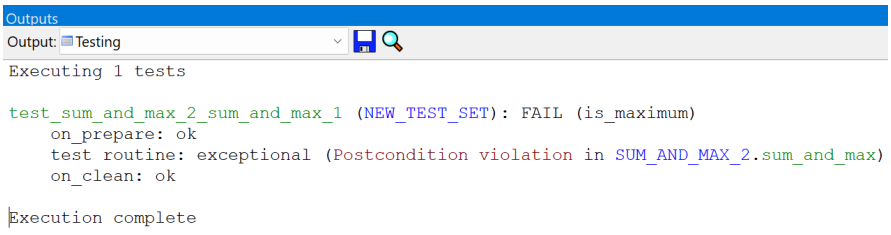
Fig. 98. Test case from failed proof of *is_maximum*

Variant 2 of SUM_AND_MAX

- Fault injection: at line 22, change the exit condition from "`i>a.count`" into "`i≥a.count`".
- Resulting failure: as shown in Fig. 99(a), the injected fault leads to the violation of the postcondition *is_maximum*.
- Cause of the failure: incorrect exit condition of the loop.
- Proof time: 0.321 sec
- Test generation time: 0.333 sec
- Resulting test case: Fig. 100 shows the test case, which calls `sum_and_max` with input argument `a[1] = 0`, `a[2]= 0`, `a[3]= 0`, `a[4] = 10`.
- Testings result: as shown in Fig. 99(b), execution of the test case raises an exception of the violation of postcondition *is_maximum*, which corresponds to the same proof failure.
- Comment: the test is useful as its execution displays an error trace that leads to the violation of the same failed contract in the proof; during the execution of the test, after 2 iteration, the loop terminates earlier due to the incorrect exit condition; thus the execution is not able reaching the actual maximum element — the last element of the array, which should be visited at the third iteration.



(a)



(b)

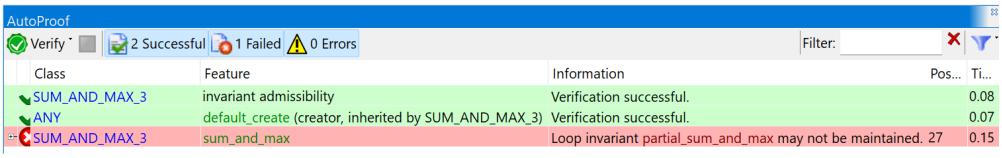
Fig. 99. (a) Verification result of `SUM_AND_MAX_2` in AutoProof; (b) Testing result of `test_SUM_AND_MAX_2_sum_and_max_1` in AutoTest

```
1   test_SUM_AND_MAX_2_sum_and_max_1
2     local
3       current_object: SUM_AND_MAX_2
4       a: SIMPLE_ARRAY[INTEGER_32]
5       sum_and_max_result: TUPLE[INTEGER_32,INTEGER_32]
6     do
7       create current_object
8       create a.make_empty
9       a.force(0, 1)
10      a.force(0, 2)
11      a.force(0, 3)
12      a.force(10, 4)
13
14      sum_and_max_result := current_object.sum_and_max (a)
15    end
```

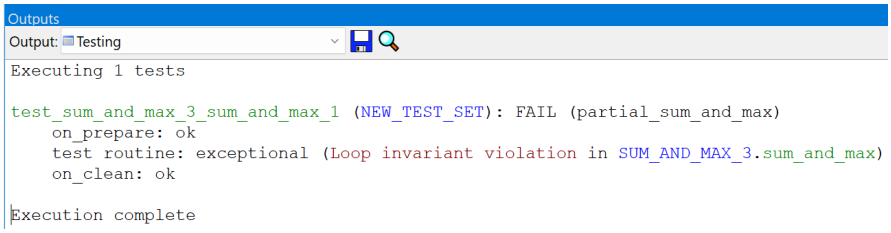
Fig. 100. Test case from failed proof of *is_maximum*

Variant 3 of `SUM_AND_MAX`

- Fault injection: at line 24, change the condition of the then branch from "`a [i]>max`" into "`a [i]<max`".
- Resulting failure: as shown in Fig. 101(a), the injected fault results in the violation of the loop invariant `partial_sum_and_max`.
- Cause of the failure: incorrect implementation of the loop body.
- Proof time: 0.307 sec
- Test generation time: 0.333 sec
- Resulting test case: Fig. 102 shows the test case from Proof2Test, which calls `sum_and_max` with input argument `a[1] = 0`, `a[2]= 5`.
- Testings result: as shown in Fig. 101(b), execution of the test case raises an exception of the violation of loop invariant `partial_sum_and_max`, which corresponds to the same proof failure.
- Comment: the test is useful as it is able to show how the program goes to the violation of the same failed contract as in the proof; during the execution of the test, after loop initialization (line 15), `max = 0` and `sum = 0`; at the first iteration, the program goes to the `else` branch (as `a [i]<max` is false); the program then updates `sum` to 5 (line 27); at this point, the loop invariant `partial_sum_and_max` is false with `sum = 5`, `i = 3` and `max = 0`, which causes the exception.



(a)



(b)

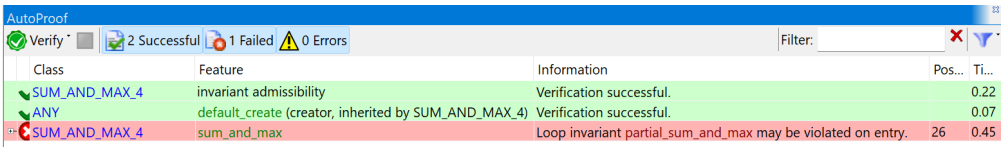
Fig. 101. (a) Verification result of `SUM_AND_MAX_3` in AutoProof; (b) Testing result of `test_SUM_AND_MAX_3_sum_and_max_1` in AutoTest

```
1   test_SUM_AND_MAX_3_sum_and_max_1
2     local
3       current_object: SUM_AND_MAX_3
4       a: SIMPLE_ARRAY[INTEGER_32]
5       sum_and_max_result: TUPLE[INTEGER_32,INTEGER_32]
6     do
7       create current_object
8       create a.make_empty
9       a.force(0, 1)
10      a.force(5, 2)
11
12      sum_and_max_result := current_object.sum_and_max (a)
13    end
```

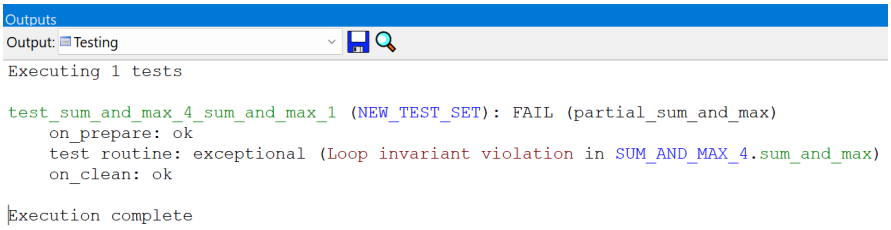
Fig. 102. Test case from failed proof of `partial_sum_and_max`

Variant 4 of SUM_AND_MAX

- Fault injection: at line 15, change the first segment of the loop initialization from "`i` := 2" into "`i` := 1".
- Resulting failure: as shown in Fig. 103(a), the mutation of the program leads to the violation of the loop invariant `partial_sum_and_max` at the entry of the loop.
- Cause of the failure: incorrect implementation of the loop initialization.
- Proof time: 0.743 sec
- Test generation time: 0.346 sec
- Resulting test case: Fig. 104 shows the test case from Proof2Test, which calls `sum_and_max` with input argument `a`[1] = 1, `a`[2] = 1.
- Testings result: as shown in Fig. 103(b), execution of the test case raises an exception of the violation of loop invariant `partial_sum_and_max`, which corresponds to the same proof failure.
- Comment: the test can demonstrate a specific scenario from which the program goes to the state where the same contract is violated as in the proof; during the execution of the test, after loop initialization (line 15), `i` = 1, `max` = 1 and `sum` = 1; at the first iteration, the program goes to the `else` branch (as `a` [`i`]>`max` is false); the program then updates `sum` to 2 (line 27); at this point, the loop invariant `partial_sum_and_max` is false with `sum` = 2, `i` = 2 and `max` = 1, which causes the exception. In this example, however, reading the test input itself may enable us to identify the fault: in the test input, all the elements in the array `a` have the same value, hence each element is the maximum and `sum` and `max` should have the following relation: `sum` = `a.count` * `max`, which is a special case of the loop invariant `partial_sum_and_max` (`sum`≤`i` * `max`); since this invariant does not hold, we might tend to suspect the correctness of `i` (ideally, `i` should be equal to `a.count`).



(a)



(b)

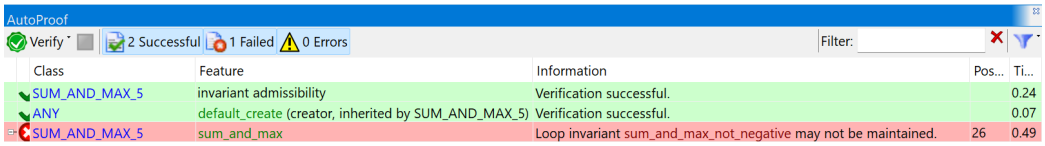
Fig. 103. (a) Verification result of `SUM_AND_MAX_4` in AutoProof; (b) Testing result of `test_SUM_AND_MAX_4_sum_and_max_1` in AutoTest

```
1   test_SUM_AND_MAX_4_sum_and_max_1
2     local
3       current_object: SUM_AND_MAX_4
4       a: SIMPLE_ARRAY[INTEGER_32]
5       sum_and_max_result: TUPLE[INTEGER_32,INTEGER_32]
6     do
7       create current_object
8       create a.make_empty
9       a.force(1, 1)
10      a.force(1, 2)
11
12      sum_and_max_result := current_object.sum_and_max (a)
13    end
```

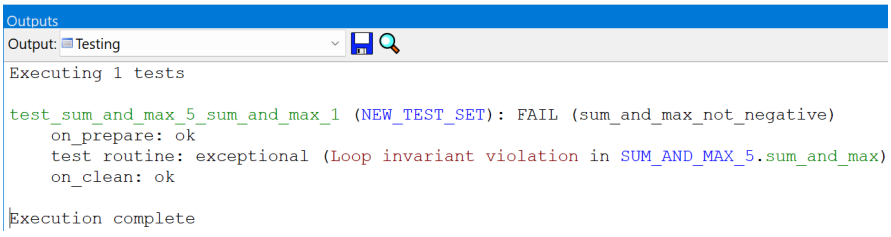
Fig. 104. Test case from failed proof of `partial_sum_and_max`

Variant 5 of `SUM_AND_MAX`

- Fault injection: at line 8, remove the precondition `natural_numbers`.
- Resulting failure: as shown in Fig. 105(a), the injected fault leads to the violation of the loop invariant `sum_and_max_not_negative`.
- Cause of the failure: weakness of precondition.
- Proof time: 0.800 sec
- Test generation time: 0.313 sec
- Resulting test case: Fig. 106 shows the test case of Proof2Test, which calls `sum_and_max` with input argument `a`[1] = 1, `a`[2] = 1.
- Testings result: as shown in Fig. 105(b), execution of the test case raises an exception of the violation of loop invariant `sum_and_max_not_negative`, which corresponds to the same proof failure.
- Comment: this test is useful as it can demonstrate how the same failed contract is violated during execution: after 3 iteration of the loop, the value of `sum` becomes negative and violates the loop invariant `sum_and_max_not_negative`.



(a)



(b)

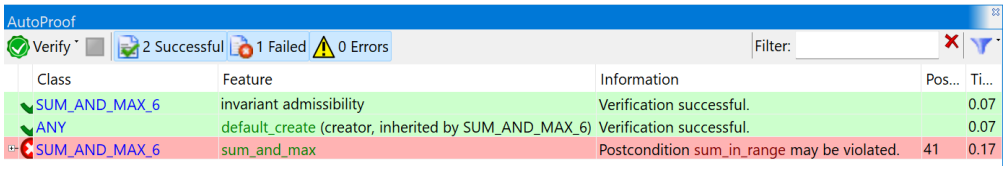
Fig. 105. (a) Verification result of `SUM_AND_MAX_5` in AutoProof; (b) Testing result of `test_SUM_AND_MAX_5_sum_and_max_1` in AutoTest

```
1   test_SUM_AND_MAX_5_sum_and_max_1
2     local
3       current_object: SUM_AND_MAX_5
4       a: SIMPLE_ARRAY[INTEGER_32]
5       sum_and_max_result: TUPLE[INTEGER_32,INTEGER_32]
6     do
7       create current_object
8       create a.make_empty
9       a.force(0, 1)
10      a.force(0, 2)
11      a.force(0, 3)
12      a.force((-10), 4)
13      a.force(4, 5)
14      a.force(4, 6)
15
16      sum_and_max_result := current_object.sum_and_max (a)
17    end
```

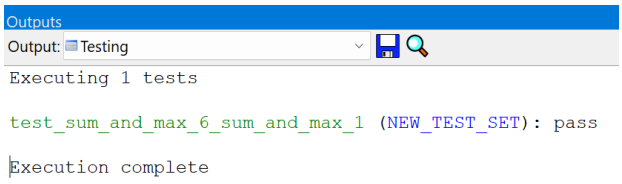
Fig. 106. Test case from failed proof of `sum_and_max_not_negative`

Variant 6 of SUM_AND_MAX

- Fault injection: at line 19, remove the loop invariant `partial_sum_and_max`.
- Resulting failure: as shown in Fig. 107(a), the injected fault results in the violation of the postcondition `sum_in_range`.
- Cause of the failure: weakness/incompleteness of loop invariant.
- Proof time: 0.310 sec
- Test generation time: 0.362 sec
- Resulting test case: Fig. 108 shows the test case from Proof2Test, which calls `sum_and_max` with input argument `a[1] = 0`.
- Testings result: as shown in Fig. 107(b), execution of the test case does not raise any exception.
- Comment: similar to Variant 1, the passing test indicates that the proof failure is caused by the weakness of the loop invariant.



(a)



(b)

Fig. 107. (a) Verification result of `SUM_AND_MAX_6` in AutoProof; (b) Testing result of `test_SUM_AND_MAX_6_sum_and_max_1` in AutoTest

```
1   test_SUM_AND_MAX_6_sum_and_max_1
2     local
3       current_object: SUM_AND_MAX_6
4       a: SIMPLE_ARRAY[INTEGER_32]
5       sum_and_max_result: TUPLE[INTEGER_32,INTEGER_32]
6     do
7       create current_object
8       create a.make_empty
9       a.force(0, 1)
10
11      sum_and_max_result := current_object.sum_and_max (a)
12    end
```

Fig. 108. Test case from failed proof of `sum_in_range`