Your Proof Fails? Testing Helps to Find the Reason

Guillaume Petiot[1], Nikolai Kosmatov[1], Bernard Botella[1],
Alain Giorgetti[2], and Jacques Julliand[2]

[1] CEA, LIST, Software Reliability Laboratory, PC 174, 91191 Gif-sur-Yvette France
firstname.lastname@cea.fr
[2] FEMTO-ST/DISC, University of Franche-Comté, 25030 Besançon Cedex France
firstname.lastname@femto-st.fr

Abstract. Applying deductive verification to formally prove that a program respects its formal specification is a very complex and time-consuming task due in particular to the lack of feedback in case of proof failures. Along with a non-compliance between the code and its specification (due to an error in at least one of them), possible reasons of a proof failure include a missing or too weak specification for a called function or a loop, and lack of time or simply incapacity of the prover to finish a particular proof. This work proposes a complete methodology where test generation helps to identify the reason of a proof failure and to exhibit a counterexample clearly illustrating the issue. We define the categories of proof failures, introduce two subcategories of contract weaknesses (single and global ones), and examine their properties. We describe how to transform a formally specified C program into C code suitable for testing, and illustrate the benefits of the method on comprehensive examples. The method has been implemented in STADY, a plugin of the software analysis platform FRAMA-C. Initial experiments show that detecting non-compliances and contract weaknesses allows to precisely diagnose most proof failures.

1 Introduction

Among formal verification techniques, *deductive verification* consists in establishing a rigorous mathematical proof that a given program meets its specification. When no confusion is possible, one also says that deductive verification consists in "proving a program". It requires that the program comes with a formal specification, usually given in special comments called *annotations,* including function contracts (with pre- and postconditions) and loop contracts (with loop variants and invariants). The *weakest precondition calculus* proposed by Dijkstra [19] reduces any deductive verification problem to establishing the validity of first-order formulas called *verification conditions*.

In modular deductive verification of a function f calling another function g, the roles of the pre- and postconditions of f and of the callee g are dual. The precondition of f is assumed and its postcondition must be proved, while at any call to g in f, the precondition of g must be proved before the call and its postcondition is assumed after the call. The situation for a function f with one call to g is presented in Fig. 1. An arrow in this figure informally indicates that its initial point provides a hypothesis for a proof of its final point. For instance, the precondition Pre_f of f and the postcondition $Post_g$

of g provide hypotheses for a proof of the postcondition $Post_f$ of f. The called function g is proved separately.

To reflect the fact that some contracts become hypotheses during deductive verification of f we use the term *subcontracts for* f to designate contracts of called functions and loops in f.

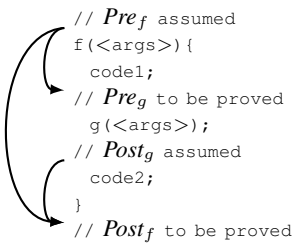

Fig. 1: Proof of f that calls g

Motivation. One of the most important difficulties in deductive verification is the manual processing of proof failures by the verification engineer since proof failures may have several causes. Indeed, a failure to prove Pre_g in Fig. 1 may be due to a *non-compliance* of the code to the specification: either an error in the code `code1`, or a wrong formalization of the requirements in the specification Pre_f or Pre_g itself. The verification can also remain inconclusive because of a *prover incapacity* to finish a particular proof within allocated time. In many cases, it is extremely difficult for the verification engineer to decide how to proceed: either suspect a non-compliance and look for an error in the code or check the specification, or suspect a prover incapacity, give up automatic proof and try to achieve an interactive proof with a proof assistant (like Coq [41]).

A failure to prove the postcondition $Post_f$ (cf. Fig. 1) is even more complex to analyze: along with a prover incapacity or a non-compliance due to errors in the pieces of code `code1` and `code2` or to an incorrect specification Pre_f or $Post_f$, the failure can also result from a *too weak* postcondition $Post_g$ of g, that does not fully express the intended behavior of g. Notice that in this last case, the proof of g can still be successful. However, the current automated tools for program proving do not provide a sufficiently precise indication on the reason of the proof failure. Some advanced tools produce a counterexample extracted from the underlying solver that cannot precisely indicate if the verification engineer should look for a non-compliance, or strengthen subcontracts (and which one of them), or consider adding additional lemmas or using interactive proof. So the verification engineer must basically consider all possible reasons one after another, and maybe initiate a very costly interactive proof. For a loop, the situation is similar, and offers an additional challenge: to prove the invariant preservation, whose failure can be due to several reasons as well.

The motivation of this work is twofold. First, we want to provide the verification engineer with a more precise feedback indicating the reason of each proof failure. Second, we look for a counterexample that either confirms the non-compliance and demonstrates that the unproven predicate can indeed fail on a test datum, or confirms a subcontract weakness showing on a test datum which subcontract is insufficient.

Approach and goals. The diagnosis of proof failures based on a counterexample generated by a prover can be imprecise since from the prover's point of view, the code of callees and loops in f is replaced by the corresponding subcontracts. To make this diagnosis more precise, one should take into account their code as well as their contracts. A recent study [42] proposed to use function inlining and loop unrolling (cf. Sec. 6). We propose an alternative approach: to use advanced test generation techniques in order to diagnose proof failures and produce counterexamples. Their usage requires

a translation of the annotated C program into an executable C code suitable for testing. Previous work suggested several comprehensive debugging scenarios relying on test generation only in the case of non-compliances [38], and proposed a rule-based formalization of annotation translation for that purpose [37]. The cases of subcontract weakness remained undetected and indistinguishable from a prover incapacity.

The overall goal of the present work is to provide a complete methodology for a more precise diagnosis of proof failures in all cases, to implement it and to evaluate it in practice. The proposed method is composed of two steps. The first step looks for a non-compliance. If none is found, the second step looks for a subcontract weakness. We propose a new classification of subcontract weaknesses into *single* (due to a single too weak subcontract) and *global* (possibly related to several subcontracts), and investigate their relative properties. Another goal is to make this method automatic and suitable for a non-expert verification engineer.

The contributions of this paper include:

- a classification of proof failures into three categories: *non-compliance* (NC), *subcontract weakness* (SW) and *prover incapacity*,
- a definition and comparative analysis of *global* and *single* subcontract weaknesses,
- a new program transformation for diagnosis of subcontract weaknesses,
- a complete testing-based methodology for diagnosis of proof failures and generation of counterexamples, suggesting possible actions for each category, illustrated on several comprehensive examples,
- an implementation of the proposed solution in a tool called STADY[3], and
- experiments showing its capability to diagnose proof failures.

Paper outline. Sec. 2 presents the tools used in this work and an illustrative example. Sec. 3 defines the categories of proof failures and counterexamples, and presents program transformations for their identification. The complete methodology for the diagnosis of proof failures is presented in Sec. 4. Our implementation and experiments are described in Sec. 5. Finally, Sec. 6 and 7 present some related work and a conclusion.

2 FRAMA-C Toolset and Illustrating Example

This work is realized in the context of FRAMA-C [31], a platform dedicated to analysis of C code that includes various analyzers in separate plugins. The WP plugin performs weakest precondition calculus for deductive verification of C programs. Various automatic SMT solvers can be used to prove the verification conditions generated by WP. In this work we use ALT-ERGO 0.99.1 and CVC3 2.4.1. To express properties over C programs, FRAMA-C offers the behavioral specification language ACSL [4, 31]. Any analyzer can both add ACSL annotations to be verified by other ones, and notify other plugins about its own analysis results by changing an annotation status.

For combinations with dynamic analysis, FRAMA-C also supports E-ACSL [18, 40], a rich executable subset of ACSL suitable for *runtime assertion checking*. E-ACSL can express function contracts (pre/postconditions, guarded behaviors, completeness and disjointness of behaviors), assertions and loop contracts (variants and invariants). It supports quantifications over bounded intervals of integers, mathematical integers

[3] See also http://gpetiot.github.io/stady.html.

and memory-related constructs (e.g. on validity and initialization). It comes with an instrumentation-based translating plugin, called E-ACSL2C [33, 30], that allows to evaluate annotations at runtime and report failures. The C code generated by E-ACSL2C is inadequate[4] for test generation, which creates the need for a dedicated translation tool.

For test generation, this work relies on PATHCRAWLER [43, 6, 32], a Dynamic Symbolic Execution (DSE) testing tool. It is based on a specific constraint solver, COLIBRI, that implements advanced features such as floating-point and modular integer arithmetic. PATHCRAWLER provides coverage strategies like *all-paths* (all feasible paths) and *k-path* (feasible paths with at most k consecutive loop iterations). It is *sound*, meaning that each test case activates the test objective for which it was generated. This is verified by concrete execution. PATHCRAWLER is also *complete* in the following sense: if the tool manages to explore all feasible paths of the program, then the absence of a test for some test objective means that the test objective is infeasible (i.e. impossible to activate), since the tool does not approximate path constraints [6, Sec. 3.1].

Example. To illustrate various kinds of proof failures, let us consider the example of C program in Fig. 2 coming from [23]. It implements an algorithm proposed in [3, page 235] that sequentially generates *Restricted Growth Functions* (RGF). A function $a : \{0, \ldots, n-1\} \rightarrow \{0, ..., n-1\}$ is an *RGF of size* $n > 0$ if $a(0) = 0$ and $a(k) \leq a(k-1) + 1$ for any $1 \leq k \leq n-1$ (that is, the growth of $a(k)$ w.r.t. the previous step is at most 1). It is defined by the ACSL predicate `is_rgf` on lines 1–2 of Fig. 2, where the RGF a is represented by the C array of its values. For convenience of the reader, some ACSL notations are replaced by mathematical symbols (e.g. keywords **`\exists`**, **`\forall`** and `integer` are respectively denoted by $\exists$, $\forall$ and $\mathbb{Z}$).

Fig. 2 shows a main function `f` and an auxiliary function `g`. The precondition of `f` states that `a` is a valid array of size `n>0` (lines 22–23) and must be an RGF (line 24). The postcondition states that the function is only allowed to modify the values of array `a` except the first one `a[0]` (line 25), and that the generated array `a` is still an RGF (line 26). Moreover, this (simplified) contract also states that if the function returns 1 then the first modified value in RGF `a` has increased (lines 27–30). Here **`\at`**`(a[j],`**`Pre`**`)` denotes the value of `a[j]` in the **`Pre`** state, i.e. before the function is executed.

We focus now on the body of the function `f` in Fig. 2. The loop on lines 36–37 goes through the array from right to left to find the rightmost non-increasing element, that is, the maximal array index `i` such that `a[i]` $\leq$`a[i-1]`. If such an index `i` is found, the function increments `a[i]` (line 40) and fills out the rest of the array with zeros (call to `g`, line 41). The loop contract (lines 33–35) specifies the interval of values of the loop variable, the variable that the loop can modify as well as a loop variant that is used to ensure the termination of the loop. The loop variant expression must be non-negative whenever an iteration starts, and must strictly decrease after each iteration.

The function `g` is used to fill the array with zeros to the right of index `i`. In addition to size and validity constraints (lines 7–8), its precondition requires that the elements of `a` up to index `i` form an RGF (lines 9–10). The function is allowed to modify the

[4] E-ACSL2C relies on complex external libraries (e.g. to handle memory-related annotations and unbounded integer arithmetic of E-ACSL) and does not assume the precondition of the function under verification, whereas the translation for test generation can efficiently rely on the underlying test generator or constraint solver for these purposes [37].

```
1  /*@ predicate is_rgf(int *a, ℤ n) =
2    a[0] == 0 ∧ ∀ ℤ i; 1 ≤ i < n ⇒
         (0 ≤ a[i] ≤ a[i-1]+1); */
3
4  /*@ lemma max_rgf: ∀ int* a; ∀ ℤ n;
5    is_rgf(a, n) ⇒ (∀ ℤ i; 0 ≤ i < n ⇒
         a[i] ≤ i); */
6
7  /*@ requires n > 0;
8      requires \valid(a+(0..n-1));
9      requires 1 ≤ i ≤ n-1;
10     requires is_rgf(a,i+1);
11     assigns a[i+1..n-1];
12     ensures is_rgf(a,n); */
13 void g(int a[], int n, int i) {
14  int k;
15  /*@ loop invariant i+1 ≤ k ≤ n;
16      loop invariant is_rgf(a,k);
17      loop assigns k, a[i+1..n-1];
18      loop variant n-k; */
19  for (k = i+1; k < n; k++) a[k] = 0;
20 }
21
22 /*@ requires n > 0;
23     requires \valid(a+(0..n-1));
24     requires is_rgf(a,n);
25     assigns a[1..n-1];
26     ensures is_rgf(a,n);
27     ensures \result == 1 ⇒
28      ∃ ℤ j; 0 ≤ j < n ∧
29       (\at(a[j],Pre) < a[j] ∧
30        ∀ ℤ k; 0 ≤ k < j ⇒
              \at(a[k],Pre) == a[k]); */
31 int f(int a[], int n) {
32  int i,k;
33  /*@ loop invariant 0 ≤ i ≤ n-1;
34      loop assigns i;
35      loop variant i; */
36  for (i = n-1; i ≥ 1; i--)
37   if (a[i] ≤ a[i-1]) { break; }
38  if (i == 0) { return 0; } // Last RGF.
39  //@ assert a[i]+1 ≤ 2147483647;
40  a[i] = a[i] + 1;
41  g(a,n,i);
42  /*@ assert ∀ ℤ l; 0 ≤ l < i ⇒
         \at(a[l],Pre) == a[l]; */
43  return 1;
44 }
```

Fig. 2: Successor function for restricted growth functions (RGF)

elements of `a` starting from the index `i+1` (line 11) and generates an RGF (line 12). The loop invariants indicate the value interval of the loop variable `k` (line 15), and state that the property `is_rgf` is satisfied up to `k` (line 16). This invariant allows a deductive verification tool to deduce the postcondition. The annotation **`loop assigns`** (line 17) says that the only values the loop can change are `k` and the elements of `a` starting from the index `i+1`. The term `n-k` is a variant of the loop (line 18).

The ACSL lemma on lines 4–5 states that if an array is an RGF, then each of its elements is at most equal to its index. Its proof requires induction and cannot be performed by WP, which uses it to ensure the absence of overflow at line 40 (stated on line 39).

The functions of Fig. 2 can be fully proved using WP. Suppose now this example contains one of the following four mistakes: the verification engineer *either* forgets to specify the precondition on line 24, *or* writes the wrong assignment `a[i]=a[i]+2;` on line 40, *or* puts a too general clause **`loop assigns`** `i,a[1..n-1];` on line 34, *or* forgets to provide the lemma on lines 4–5. In each of these four cases, the proof fails (for the precondition of `g` on line 41 and/or the assertion on line 39) *for different reasons*. In fact, the code and specification are not compliant only in the first two cases, while the third failure is due to a too weak subcontract, and the last one comes from a prover incapacity. This work proposes a complete testing-based methodology to automatically distinguish the three reasons and suggest suitable actions in each case.

3 Categories of Proof Failures and Counterexamples

Let P be a C program annotated in E-ACSL, and f the function under verification in P. Function f is assumed to be recursion-free. It may call other functions, let g denote any of them. A *test datum* V for f is a vector of values for all input variables of f. The *program path* activated by a test datum V, denoted π_V, is the sequence of program statements executed by the program on the test datum V. We use the general term of a *contract* to designate the set of E-ACSL annotations describing a loop or a function.

```
1  /*@ requires P1;
2      ensures P2; */
3  Type_g g(...) {
4    code1;
5  }
6  /*@ requires P5;
7      ensures P6; */
8  Type_f f(...) {
9    code2;
10   g(...);
11   //@ loop invariant P3;
12   while(b) {
13     code3;
14   }
15   code4;
16   //@ assert P4;
17   code5;
18 }
```

→

```
1  Type_g g(...) {
2    int pre_g; Spec2Code(P1, pre_g);
3    fassert(pre_g);
4    code1;
5    int post_g; Spec2Code(P2,post_g);
6    fassert(post_g);
7  }
8  Type_f f(...) {
9    int pre_f; Spec2Code(P5, pre_f);
10   fassume(pre_f);
11   code2;
12   g(...);
13   int inv1; Spec2Code(P3, inv1);
14   fassert(inv1);
15   while(b) {
16     code3;
17     int inv2; Spec2Code(P3, inv2);
18     fassert(inv2);
19   }
20   code4;
21   int asrt; Spec2Code(P4, asrt);
22   fassert(asrt);
23   code5;
24   int post_f; Spec2Code(P6,post_f);
25   fassert(post_f);
26 }
```

Fig. 3: (a) An annotated code, vs. (b) its translation in P^{NC} for $\mathfrak{D}^{\mathrm{NC}}$

A function contract is composed of pre- and postconditions including E-ACSL clauses **`requires`**, **`assigns`** and **`ensures`** (cf. lines 22–30 in Fig. 2). A loop contract is composed of **`loop invariant`**, **`loop variant`** and **`loop assigns`** clauses (cf. lines 15–18 in Fig. 2).

In Sec. 3.1, we define non-compliance and briefly recall the detection technique published in [37]. Sec. 3.2 is part of the original contribution of this paper, which introduces new categories of proof failures and a new detection technique.

3.1 Non-Compliance

Fig. 3 illustrates the translation of an annotated program P into another C program, denoted P^{NC}, on which we can apply test generation to produce test data violating some annotations at runtime. In Fig. 3, f is the function under verification and g is a called function. This translation is formally presented in [37]. P^{NC} checks all annotations of P in the corresponding program locations and reports any failure. For instance, the postcondition $Post_f$ of f is evaluated by the following code inserted at the end of the function f in P^{NC}:

```
int post_f; Spec2Code(Post_f, post_f); fassert(post_f);        (†)
```

For an E-ACSL predicate `P`, we denote by `Spec2Code(P, b)` the generated C code that evaluates the predicate `P` and assigns its validity status to the Boolean variable `b` (see [37] for details). The function call `fassert(b)` checks the condition `b` and reports the failure and exits whenever `b` is false. Similarly, preconditions and postconditions of a callee g are evaluated respectively before and after executing the function g. A loop invariant is checked before the loop (for being initially true) and after each loop iteration (for being preserved by the previous loop iteration). An assertion is checked at its location. To generate only test data that respect the precondition Pre_f of f, Pre_f is checked at the beginning of f by an inserted code similar to (†) except that `fassert` is replaced by `fassume` that assumes the given condition.

Definition 1 (Non-compliance). *We say that there is a* non-compliance *(*NC*) between code and specification in P if there exists a test datum V for f respecting its precondition, such that the execution of P^{NC} reports an annotation failure on V. In this case, we say that V is a* non-compliance counterexample *(*NCCE*).*

Test generation on the translated program P^{NC} can be used to generate NCCEs. We call this technique *Non-Compliance Detection*, denoted $\mathfrak{D}^{\mathrm{NC}}$. In this work we use the PATHCRAWLER test generator that will try to cover all program paths. Since the translation step added a branch for the false value of each annotation, PATHCRAWLER will try to cover at least one path where the annotation does not hold. (An optimization in PATHCRAWLER avoids covering the same `fassert` failure many times.) The $\mathfrak{D}^{\mathrm{NC}}$ step may have three outcomes. If an NCCE V has been found, it returns (nc, V, a) indicating the failing annotation a and recording the program path π_V activated by V on P^{NC}. Second, if it has managed to perform a complete exploration of all program paths without finding any NCCE, it returns no (cf. the discussion of completeness in Sec. 2). Otherwise, if only a partial exploration of program paths has been performed (due to a timeout, partial coverage criterion or any other limitation), it returns $\mathsf{?}$ (unknown).

3.2 Subcontract Weakness and Prover Incapacity

Following the modular verification approach, we assume that the called functions have been verified before the caller f. To simplify the presentation, we also assume that the loops preserve their loop invariants, and focus on other proof failures occurring during the modular verification of f.

More formally, a *non-imbricated* loop (resp. function, assertion) in f is a loop (resp. function called, assertion) in f lying outside any loop of f. A *subcontract for* f is the contract of some non-imbricated loop or function in f. A *non-imbricated annotation* in f is either a non-imbricated assertion or an annotation in a subcontract for f. For instance, the function f of Fig. 2 has two subcontracts: the contract of the called function g and the contract of the loop on lines 33–37. The contract of the loop in g on lines 15–19 is not a subcontract for f, but is a subcontract for g.

We focus on non-imbricated annotations in f and assume that all subcontracts for f are respected: the called functions in f respect their contracts, and the loops in f preserve their loop invariants and respect all imbricated annotations. Let c_f denote the contract of f, $\mathcal{C}$ the set of non-imbricated subcontracts for f, and $\mathcal{A}$ the set of all non-imbricated annotations in f and annotations of c_f. In other words, $\mathcal{A}$ contains the annotations included in the contracts $\mathcal{C} \cup \{c_f\}$ as well as the non-imbricated assertions in f. We also assume that every subcontract of f contains a (loop) assigns clause. This is not restrictive since such a clause is necessary to prove any nontrivial code.

Subcontract weakness. To apply testing for the contracts of called functions and loops in $\mathcal{C}$ instead of their code, we use a new program transformation of P producing another program P^{SW}. The code of all non-imbricated function calls and loops in f is replaced by the most general code respecting the corresponding subcontract as follows.

For the contract $c \in \mathcal{C}$ of a called function g in f, the program transformation (illustrated by Fig. 4) generates a new function `g_sw` with the same signature whose code simulates any possible behavior respecting the postcondition in c, and replaces all calls to g by a call to `g_sw`. First, `g_sw` allows any of the variables (or, more generally,

```
1  /*@ assigns k1,...,kN;
2    @ ensures P; */
3  Type_g g(...){ code1; }
4
5
6
7
8  Type_f f(...){ code2;
9    g(Args_g);
10   code3; }
```

→

```
1  Type_g g_sw(...){
2    k1=Nondet(); ... kN=Nondet();
3    Type_g ret = Nondet();
4    int post; Spec2Code(P, post);
5    fassume(post); return ret;
6  } //respects contract of g
7  Type_g g(...){ code1; }
8  Type_f f(...){ code2;
9    g_sw(Args_g);
10   code3; }
```

Fig. 4: (a) A contract $c \in \mathcal{C}$ of callee g in f, vs. (b) its translation for $\mathfrak{D}^{\mathrm{SW}}$

```
1 Type_f f(...){ code1;
2   /*@ loop assigns x1,...,xN;
3     @ loop invariant I; */
4   while(b){ code2; }
5   code3; }
```

→

```
1 Type_f f(...){ code1;
2   x1=Nondet(); ... xN=Nondet();
3   int inv1; Spec2Code(I, inv1);
4   fassume(inv1 && !b); //respects loop contract
5   code3; }
```

Fig. 5: (a) A contract $c \in \mathcal{C}$ of a loop in f, vs. (b) its translation for $\mathfrak{D}^{\mathrm{SW}}$

left-values) listed in the **assigns** clause of c to change its value (line 2 in Fig.4(b)). It can be done by assigning a non-deterministic value of the appropriate type using a dedicated function, denoted here by `Nondet()` (or simply by adding an array of fresh input variables and reading a different value for each use and each function invocation). If the return type of g is not **void**, another non-deterministic value is read for the returned value `ret` (line 3 in Fig.4(b)). Finally, the validity of the postcondition is evaluated (taking into account these new non-deterministic values) and assumed in order to consider only executions respecting the postcondition, and the function returns (lines 4–5 in Fig.4(b)).

Similarly, for the contract $c \in \mathcal{C}$ of a loop in f, the program transformation replaces the code of the loop by another code that simulates any possible behavior respecting c, that is, ensuring the "loop postcondition" $I \wedge \neg b$ after the loop, as shown in Fig. 5. In addition, the transformation treats in the same way as in P^{NC} all other annotations in $\mathcal{A}$: preconditions of called functions, initial loop invariant verifications and the pre- and postcondition of f (they are not shown in Fig. 4(b) and 5(b) but an example of such transformation is given in Fig. 3).

Definition 2 (Global subcontract weakness). *We say that P has a* global subcontract weakness *for f if there exists a test datum V for f respecting its precondition, such that the execution of P^{NC} does not report any annotation failure on V, while the execution of P^{SW} reports an annotation failure on V. In this case, we say that V is a* global subcontract weakness counterexample *(global* SWCE*) for the set of subcontracts $\mathcal{C}$.*

Remark 1. Notice that we do not consider the same counterexample as an NCCE and an SWCE. Indeed, even if it is arguable that some counterexamples may illustrate both a subcontract weakness and a non-compliance, we consider that non-compliances usually come from a direct conflict between the code and the specification and should be addressed first, while subcontract weaknesses are often more subtle and will be easier to address when non-compliances are eliminated.

Again, test generation can be applied on P^{SW} to generate global SWCE candidates. When it finds a test datum V such that P^{SW} fails on V, we use runtime assertion

```
1 int x;
2 /*@ ensures x ≥ \old(x)+1; assigns x;*/
3 void g1() { x=x+2; }
4 /*@ ensures x ≥ \old(x)+1; assigns x;*/
5 void g2() { x=x+2; }
6 /*@ ensures x ≥ \old(x)+1; assigns x;*/
7 void g3() { x=x+2; }
8 /*@ ensures x ≥ \old(x)+4; assigns x;*/
9 void f() { g1(); g2(); g3(); }
```

(a) Absence of single SWCEs for any subcontract does not imply absence of global SWCEs

```
1 int x;
2 /*@ ensures x ≥ \old(x)+1; assigns x;*/
3 void g1() { x=x+1; }
4 /*@ ensures x ≥ \old(x)+1; assigns x;*/
5 void g2() { x=x+1; }
6 /*@ ensures x ≥ \old(x)+1; assigns x;*/
7 void g3() { x=x+2; }
8 /*@ ensures x ≥ \old(x)+4; assigns x;*/
9 void f() { g1(); g2(); g3(); }
```

(b) Global SWCEs do not help to find precisely a too weak subcontract

Fig. 6: Two examples where the proof of f fails due to subcontract weaknesses

checking: if P^{NC} fails on V, then V is classified as an NCCE, otherwise V is a global SWCE (cf. Remark 1). We call this technique *Global Subcontract Weakness Detection* for the set of all subcontracts, denoted $\mathfrak{D}^{\mathrm{SW}}_{\mathrm{global}}$. The $\mathfrak{D}^{\mathrm{SW}}_{\mathrm{global}}$ step may have four outcomes. It returns (nc, V, a) if an NCCE V has been found for the failing annotation a, and (sw, V, a, C) if V has been finally classified as an SWCE, where a is the failing annotation and C is the set of subcontracts. The program path π_V activated by V and leading to the failure (on P^{NC} or P^{SW}) is recorded as well. If $\mathfrak{D}^{\mathrm{SW}}_{\mathrm{global}}$ has managed to perform a complete exploration of all program paths without finding a global SWCE, it returns no. Otherwise, if only a partial exploration of program paths has been performed it returns ? (unknown).

A global SWCE does not explicitly indicate which single subcontract $c \in \mathcal{C}$ is too weak (cf. Remark 2 below). To do so, we propose another program transformation of P into an instrumented program P^{SW}_c. It is done by replacing only one non-imbricated function call or loop by the most general code respecting the postcondition of the corresponding subcontract c (as indicated in Fig. 4 and 5) and transforming other annotations in $\mathcal{A}$ in the same way as in P^{NC}.

Definition 3 (Single subcontract weakness). *Let c be a subcontract for f. We say that c is a* too weak subcontract *(or has a* single subcontract weakness*) for f if there exists a test datum V for f respecting its precondition, such that the execution of P^{NC} does not report any annotation failure on V, while the execution of P^{SW}_c reports an annotation failure on V. In this case, we say that V is a* single subcontract weakness counterexample *(single* SWCE*) for the subcontract c in f.*

For any subcontract $c \in \mathcal{C}$, test generation can be separately applied on P^{SW}_c to generate single SWCE candidates. If such a test datum V is generated, it is checked on P^{NC} to classify it as an NCCE or a single SWCE (cf. Remark 1). This technique, applied for all subcontracts one after another until a first counterexample V is found, is called *Single Contract Weakness Detection,* and denoted $\mathfrak{D}^{\mathrm{SW}}_{\mathrm{single}}$. The $\mathfrak{D}^{\mathrm{SW}}_{\mathrm{single}}$ step may have three outcomes. It returns (nc, V, a) if an NCCE V has been found for a failing annotation a, and $(\mathsf{sw}, V, a, \{c\})$ if V has been finally classified as a single SWCE, where a is the failing annotation and c is the single too weak subcontract. The program path π_V activated by V and leading to the failure (on P^{NC} or P^{SW}_c) is recorded as well. Otherwise, it returns ? (unknown).

Global vs. single subcontract weaknesses. Even after an exhaustive path testing, the absence of a single SWCE for any subcontract c cannot ensure the absence of a global SWCE, as detailed in the following remark.

Remark 2. A proof failure can be due to the weakness of several subcontracts, while no single one of them is too weak. In other words, the absence of single SWCEs does not imply the absence of global SWCEs. When a single SWCE exists, it can indicate a single too weak subcontract more precisely than a global SWCE.

Indeed, consider the example in Fig. 6a, where the proof of the postcondition of f fails. If we apply $\mathfrak{D}^{\mathrm{SW}}_{\mathrm{single}}$ to any of the subcontracts, we always have `x ≥\old(x)+5` at the end of f (we add 1 to x by executing the translated subcontract, and add 2 twice by executing the other two functions' code), so the postcondition of `f` holds and no weakness is detected. If we run $\mathfrak{D}^{\mathrm{SW}}_{\mathrm{global}}$ to consider all subcontracts at once, we only get `x≥\old(x)+3` after executing the three subcontracts, and can exhibit a global SWCE.

On the other hand, running $\mathfrak{D}^{\mathrm{SW}}_{\mathrm{global}}$ produces a global SWCE that does not indicate which of the subcontracts is too weak, while $\mathfrak{D}^{\mathrm{SW}}_{\mathrm{single}}$ can sometimes be more precise. For Fig. 6b, since the three callees are replaced by their subcontracts for $\mathfrak{D}^{\mathrm{SW}}_{\mathrm{global}}$, it is impossible to find out which one is too weak. Counterexamples generated by a prover suffer from the same precision issue: taking into account all subcontracts instead of the corresponding code prevents from a precise identification of a single too weak subcontract. In this example $\mathfrak{D}^{\mathrm{SW}}_{\mathrm{single}}$ can be more precise, since only the replacement of the subcontract of `g3` also leads to a single SWCE: we can have `x ≥\old(x)+3` by executing `g1`, `g2` and the subcontract of `g3`, exhibiting the contract weakness of `g3`. Thus, the proposed $\mathfrak{D}^{\mathrm{SW}}_{\mathrm{single}}$ technique can provide the verification engineer with a more precise diagnosis than counterexamples extracted from a prover.

We define a combined *subcontract weakness detection* technique, denoted $\mathfrak{D}^{\mathrm{SW}}$, by applying $\mathfrak{D}^{\mathrm{SW}}_{\mathrm{single}}$ followed by $\mathfrak{D}^{\mathrm{SW}}_{\mathrm{global}}$ until the first counterexample is found. In other words, $\mathfrak{D}^{\mathrm{SW}}$ looks first for single, then for global subcontract weaknesses. $\mathfrak{D}^{\mathrm{SW}}$ may have the same four outcomes as $\mathfrak{D}^{\mathrm{SW}}_{\mathrm{global}}$. It allows us to be both precise (and indicate when possible a single subcontract being too weak), and complete (able to find global subcontract weaknesses even when there are no single ones).

Prover incapacity. When neither a non-compliance nor a global subcontract weakness exists, we cannot demonstrate that it is impossible to prove the property.

Definition 4 (Prover incapacity). *We say that a proof failure in P is due to a* prover incapacity *if for every test datum V for f respecting its precondition, neither the execution of P^{NC} nor that of P^{SW} reports any annotation failure on V. In other words, there is no* NCCE *and no global* SWCE *for P.*

4 Diagnosis of Proof Failures using Structural Testing

In this section, we present an overview of our method for diagnosis of proof failures using the detection techniques of Sec. 3, illustrate it on several examples and provide a comprehensive list of suggestions of actions for each category of proof failures.

The method. The proposed method is illustrated by Fig. 7. Suppose that the proof of the annotated program P fails for some non-imbricated annotation $a \in \mathcal{A}$. The first step tries to find a non-compliance using $\mathfrak{D}^{\mathrm{NC}}$. If such a non-compliance is found, it

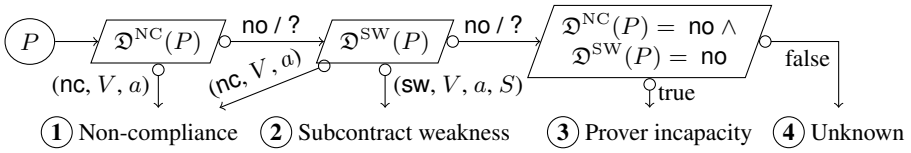

Fig. 7: Combined verification methodology in case of a proof failure on P

generates an NCCE (marked by (1) in Fig. 7) and classifies the proof failure as a non-compliance. If the first step cannot generate a counterexample, the $\mathfrak{D}^{\mathrm{SW}}$ step combines $\mathfrak{D}^{\mathrm{SW}}_{\mathrm{single}}$ and $\mathfrak{D}^{\mathrm{SW}}_{\mathrm{global}}$ and tries to generate single SWCEs, then global SWCEs, until the first counterexample is generated. It can be classified either as a non-compliance (1) (that is possible if path testing in $\mathfrak{D}^{\mathrm{NC}}$ was not exhaustive, cf. Remark 1 and Def. 2, 3) or a subcontract weakness (2). If no counterexample has been found, the last step checks the outcomes. If both $\mathfrak{D}^{\mathrm{NC}}$ and $\mathfrak{D}^{\mathrm{SW}}$ have returned no, that is, both $\mathfrak{D}^{\mathrm{NC}}$ and $\mathfrak{D}^{\mathrm{SW}}_{\mathrm{global}}$ have performed a complete path exploration without finding a counterexample, the proof failure is classified as a prover incapacity (3) (cf. Def. 4). Otherwise, it remains unclassified (4).

Fig. 8 illustrates the method on several variants of the illustrating example. It details the lines modified in the program of Fig. 2 to obtain the new variant, the intermediate results of deductive verification, $\mathfrak{D}^{\mathrm{NC}}$ and $\mathfrak{D}^{\mathrm{SW}}$, and the final outcome. The final outcome includes the proof failure category and, if any, the generated counterexample V, the recorded path π_V, the reported failing annotation a and a set of too weak subcontracts S. This outcome can be extremely helpful for the verification engineer. Suppose we try to prove in WP a modified version of the function f of Fig. 2 where the precondition at line 24 is missing (cf. #1 in Fig. 8). The proof of the precondition on line 10 (for the call of g on line 41) fails without indicating a precise reason. The $\mathfrak{D}^{\mathrm{NC}}$ step generates an NCCE (case (1)) where `is_rgf(a,n)` is clearly false due to `a[0]` being non-zero, and indicates the failing annotation (coming from line 10). That helps the verification engineer to understand and fix the issue.

Let us suppose now that the clause on line 34 has been erroneously written as follows: `loop assigns i, a[1..n-1];` (cf. #2 in Fig. 8). The loop on lines 36–37 still preserves its invariant. The $\mathfrak{D}^{\mathrm{NC}}$ step does not find any NCCE, as this modification did not introduce any non-compliance between the code and its specification. Thanks to the spec-to-code replacement shown in Fig. 5, $\mathfrak{D}^{\mathrm{SW}}_{\mathrm{single}}$ for the contract of this loop will detect a single subcontract weakness for the loop contract (case (2)), leading to a failure of the precondition of g (on line 10) for the call on line 41. With this indication, the verification engineer will try to strengthen the loop contract and find the issue.

Suppose now the lemma on lines 4–5 is missing (cf. #4 in Fig. 8). The proof of the assertion at line 39 of Fig. 2 (stating the absence of overflow at line 40) fails without giving a precise reason, since the prover does not perform the induction and cannot deduce the right bounds on `a[i]`. Neither $\mathfrak{D}^{\mathrm{NC}}$ nor $\mathfrak{D}^{\mathrm{SW}}$ produces a counterexample, and as the initial program has too many paths, their outcomes are ? (unknown) (case (4)). For such situations, we introduce the possibility to reduce the input domain for test generation by using a new ACSL clause **`typically`**. The verification engineer can insert the

#	Modified lines		Intermediate outcome			Final outcome of STADY
	Line	New (added) clause	Proof (failing annot.)	$\mathfrak{D}^{\mathrm{NC}}$	$\mathfrak{D}^{\mathrm{SW}}$	
0	–	–	✓	–	–	Proved
1	24	(deleted)	? (l.39, 41, 26)	nc	–	$V = \langle$ `n=1; a[0]=-214739` $\rangle$ is an NCCE
2	34	`loop assigns i,a[1..n-1];`	? (l.39, 41, 42, 26–30)	?	sw for l.33–34	$V = \langle$ `n=2; a[0]=0; a[1]=0;` $\mathbf{nondet}_{\texttt{a[1]=97157;}}$ $\mathrm{nondet}_{\texttt{i=0}} \rangle$ is an SWCE
3	4–5 after 22	(deleted) `typically n<5;` (added)	? (l.39)	no	no	Prover incapacity (for the program with reduced domain)
4	4–5	(deleted)	? (l.39)	?	?	Unknown

Fig. 8: Method results for different versions of the illustrating example

clause `typically n<5;` after the line 22 to reduce the array size for test generation (this clause is ignored by the proof). Running STADY now allows the tool to perform a complete exploration of all program paths (for `n<5`) both for $\mathfrak{D}^{\mathrm{NC}}$ and $\mathfrak{D}^{\mathrm{SW}}$ without finding a counterexample. STADY classifies the proof failure for the program with the reduced domain as a prover incapacity (case (3), cf. #3 in Fig. 8). That gives the verification engineer more confidence that the proof failure has the same reason on the initial program for bigger sizes `n`. She may now try an interactive proof or add additional lemmas or assertions, and does not waste her time looking for a bug or a subcontract weakness.

Suggestions of actions. Based on the possible outcomes of the method (illustrated in Fig. 7), we are able to suggest the most suitable actions to help the verification engineer with the verification task. A reported *non-compliance* (nc, V, a) means that there is an inconsistency between the precondition, the annotation a and the code of the path π_V leading to a. Thanks to the counterexample, the user will understand the issue by *tracing the values of variables along* π_V, or exploring them in a dubugger [35]. In FRAMA-C, the execution on V can be conveniently explored using VALUE or PATHCRAWLER [31]. If an NCCE is generated, there is *no need to try an automatic or interactive proof, or look for a subcontract weakness* — it will not help.

A reported *subcontract weakness* (sw, V, a, S) for a set of subcontracts S means that at least one of them has to be strengthened. By Def. 2 and 3, the non-compliance is excluded here, that is, the execution of P^{NC} on V respects the annotation a. Thus the suggested action is to *strengthen the subcontract(s) of* S. In the case of a single subcontract weakness, S is a singleton so the suggestion is very precise and helpful to the user. Again, *trying interactive proof or writing additional assertions or lemmas will be useless* here since the property can obviously not be proved.

For a *prover incapacity,* the verification engineer may *add lemmas, assertions or hypotheses* that can help the theorem prover to succeed, or *try another theorem prover*, or *use a proof assistant* like COQ, even if it can be more complex and time-consuming.

Finally, when the verdict is *unknown,* i.e. test generation for $\mathfrak{D}^{\mathrm{NC}}$ and/or $\mathfrak{D}^{\mathrm{SW}}$ times out, the verification engineer may *strengthen the precondition* for test generation to reduce the input domain, or *extend the timeout* to give STADY more time to conclude.

		Proof				$\mathfrak{D}^{\mathrm{NC}}$				$\mathfrak{D}^{\mathrm{SW}}$				$\mathfrak{D}^{\mathrm{NC}} + \mathfrak{D}^{\mathrm{SW}}$		
	#mut	#✓	%	$t^{✓}$	$t^{?}$	#✗	%	$t^{✗}$	$t^{?}$	#✗	%	$t^{✗}$	$t^{?}$	%	t	#?
Total	928	80 /928	8.6			776 /848	91.5			48 /72	66.7			97.2		24
Max			20.8	$\leqslant 4.4$	$\leqslant 61.3$		96.2	$\leqslant 9.4$	$\leqslant 8.3$		100.0	$\leqslant 6.4$	$\leqslant 11.6$	100.0	$\leqslant 19.9$	
Mean			8	≈ 2.6	≈ 13.0		92.6	≈ 2.4	≈ 2.5		80.0	≈ 2.4	≈ 6.3	98.1	≈ 2.7	

Fig. 9: Summarized experiments of proof failure diagnosis for mutants with STADY

5 Implementation and Experiments

Implementation. The proposed method for diagnosis of proof failures has been implemented as a FRAMA-C plugin, named STADY. It relies on other plugins: WP [31] for deductive verification and PATHCRAWLER [6] for structural test generation. STADY currently supports a significant subset of the E-ACSL specification language, including `requires`, `ensures`, `behavior`, `assumes`, `loop invariant`, `loop variant` and `assert` clauses. Quantified predicates `\exists` and `\forall` and builtin terms as `\sum` or `\numof` are translated as loops (recall that E-ACSL allows only finite intervals of quantification). Logic functions and named predicates are treated by inlining. The `\old` and `\at(-,Pre)` constructs are treated by saving the initial values of formal parameters and global variables at the beginning of the function. Validity checks of pointers are partially supported due to the current limitation of the underlying test generator: we can only check the validity of input pointers and global arrays. The `assigns` clauses are considered only during the $\mathfrak{D}^{\mathrm{SW}}$ phase: we do not try to fix an incomplete `assigns` clause (with missing variables, leading to a non-compliance) because provers usually give a sufficiently clear feedback about that; but we do try to identify a too weak (i.e. too permissive) `assigns` clause since provers would report a failure elsewhere in this case. Inductive predicates, recursive functions and real numbers are not yet supported.

The research questions we address in our experiments are the following.

RQ1 Is STADY able to precisely diagnose most proof failures in C programs?
RQ2 What are the benefits of the $\mathfrak{D}^{\mathrm{SW}}$ step (in particular, with respect to $\mathfrak{D}^{\mathrm{NC}}$)?
RQ3 Is STADY able to generate NCCEs or SWCEs even with a partial testing coverage?
RQ4 Is STADY's execution time comparable to the time of an automatic proof?

Experimental protocol. The evaluation used 20 annotated programs from an independent benchmark [7], whose size varies from 35 to 100 lines of annotated C code. These programs manipulate arrays, they are fully specified in ACSL and their specification expresses non-trivial properties of C arrays. To evaluate the method presented in Sec. 4 and its implementation, we apply STADY on systematically generated altered versions[5] (or *mutants*) of correct C programs. Each mutant is obtained by performing a single modification (or *mutation*) on the initial program. The mutations include: a binary operator modification in the code or in the specification, a condition negation in the code, a relation modification in the specification, a predicate negation in the specification, a partial loop invariant or postcondition deletion in the specification. Such mutations model frequent errors in the code and specification (e.g. confusions between $+$ and $-$, $\leq$ and $<$, $\leq$ and $\geq$, a missing loop invariant, pre- or postcondition, etc.)

[5] Available at: https://github.com/gpetiot/StaDy/tree/master/TAP_2016/benchmark

that can lead to proof failures. In this study, we do not mutate the precondition of the function under verification, and restrict possible mutations on binary operators to avoid creating absurd expressions, in particular for pointer arithmetic.

The first step tries to prove each mutant using WP. In our experiments, each prover tries to prove each verification condition during at most 40 seconds. The proved mutants respect the specification and are classified as correct. Second, we apply the $\mathfrak{D}^{\mathrm{NC}}$ method on the remaining mutants. It classifies proof failures for some mutants as non-compliances and indicates a failing annotation. The third step applies the $\mathfrak{D}^{\mathrm{SW}}$ method on remaining mutants, classifies some of them as subcontract weaknesses and indicates a weak subcontract. If no counterexample has been found by the $\mathfrak{D}^{\mathrm{SW}}$, the mutant remains unclassified. The results are summarized in Fig. 9. The columns present the number of generated mutants, and the results of each of the three steps: the number (#) and ratio (%) of classified mutants, maximal and average execution time of the step over classified mutants ($t^{\checkmark}$ or $t^{\times}$) and over non-classified mutants ($t^{?}$) at this step. The ratios are computed with respect to the number of unclassified mutants remaining after the previous step. The $\mathfrak{D}^{\mathrm{NC}} + \mathfrak{D}^{\mathrm{SW}}$ columns sum up selected results after both $\mathfrak{D}^{\mathrm{NC}}$ and $\mathfrak{D}^{\mathrm{SW}}$ steps: the average and maximal time (t) are shown globally over all mutants. The time is computed until the proof is finished or until the first counterexample is generated. The final number of remaining unclassified mutants (#?) is given in the last column.

Experimental results. For the 20 considered programs, 928 mutants have been generated. 80 of them have been proved by WP. Among the 848 unproven mutants, $\mathfrak{D}^{\mathrm{NC}}$ has detected a non-compliance induced by the mutation in 776 mutants (91.5%), leaving 72 unclassified. Among them, $\mathfrak{D}^{\mathrm{SW}}$ has been able to exhibit a counterexample (either an NCCE or an SWCE) for 48 of them (66.7%), finally leaving 24 programs unclassified.

Regarding **RQ1**, STADY has found a precise reason of the proof failures and produced a counterexample in 824 of the 848 unproven mutants, i.e. classifying 97.2%. Exploring the benefits of detecting a prover incapacity requires to manually reduce the input domain, to try additional lemmas or an interactive proof, so it was not sufficiently investigated in this study (and probably requires another, non mutational approach).

Regarding **RQ2**, $\mathfrak{D}^{\mathrm{NC}}$ alone diagnosed 776 of 848 unproven mutants (91.5%). $\mathfrak{D}^{\mathrm{SW}}$ diagnosed 48 of the 72 remaining mutants (66.7%) bringing a significant complementary contribution to a better understanding of reasons of many proof failures.

To address **RQ3**, we set a timeout for any test generation session to 5 seconds (i.e. one session for the $\mathfrak{D}^{\mathrm{NC}}$ step, and several sessions for $\mathfrak{D}^{\mathrm{SW}}$ steps), and limit the number of explored program paths using the *k-path* criterion (cf. Sec. 2) with $k = 4$. Both the session timeout and *k-path* heavily limit the testing coverage but STADY still detects 97.2% of faults in the generated programs. That demonstrates that the proposed method can efficiently classify proof failures and generate counterexamples even with a partial testing coverage and can therefore be used for programs where the total number of paths cannot be limited (e.g. by the `typically` clause).

Concerning **RQ4**, on the considered programs WP needs on average 2.6 sec. per mutant (at most 4.4 sec.) to prove a program, and spends 13.0 sec. on average (at most

61.3 sec.) when the proof fails. The total execution time of StaDy is comparable: it needs on average 2.7 sec. per unproven mutant (at most 19.9 sec.).

Summary. The experiments show that the proposed method can automatically classify a significant number of proof failures within an analysis time comparable to the time of an automatic proof and for programs for which only a partial testing coverage is possible. The $\mathfrak{D}^{\mathrm{SW}}$ technique offers an efficient complement to $\mathfrak{D}^{\mathrm{NC}}$ for a more complete and more precise diagnosis of proof failures.

6 Related Work

Assisting program verification and generation of counterexamples have been addressed in different research work (e.g. [2, 5, 8, 10, 13, 17, 20, 21, 28, 29, 34, 36, 39]). We detail below a few projects most closely related to the present work.

Understanding proof failures. When SMT solvers fail on some verification conditions and provide a counter-model to explain that failure, the counter-model can be turned into a counterexample for the program under verification. This non-trivial task is designed in [29] and implemented for SPARK, a subset of Ada targeted for formal verification. This static analysis is complementary to our combination of static and dynamic analyses. It would be useful to adapt it to C/ACSL programs. For C programs, SMT models are already exploited, for instance by the CBMC model checker [26].

A two-step verification in [42] compares the proof failures of an Eiffel program with those of its variant where called functions are inlined and loops are unrolled. It reports code and contract revision suggestions from this comparison. Inlining and unrolling are respectively limited to a given number of nested calls and explicit iterations. If that number is too small the semantics is lost and a warning of unsoundness is reported. A bigger number of inlinings often overpasses the capacity of the solver, while DSE, focusing on one path at a time, can be expected to be more efficient. Another benefit of DSE is the possibility to use concrete values (e.g. discovered in a previous execution) even when the constraints become very complex and the solver cannot generate a counterexample.

Dafny has also been recently extended with tools for diagnosing proof failures [12]. When the proof times out, an algorithm decomposes it and tries to diagnose on which part the user has to focus to prevent the timeout. Then, if the proof fails, following the approach we proposed in our previous work [37], a DSE tool is used to try to find counterexamples demonstrating non-compliance between program and specification. But, when no counterexample is found, the user must manually try to find the reason of the proof failure (with the Boogie Verification Debugger), whereas we extend the approach by further exploiting DSE to automatically identify subcontract weaknesses. The notions of global and single SW and their comparison are also new.

Proof tree analysis. More precision can be statically obtained by analyzing the unclosed branches of a proof tree. The work [24] is performed in the context of KeY and its verification calculus that applies deduction rules to a dynamic formula mixing a program and its specification. It proposes *falsifiability preservation checking* that helps to distinguish whether the branch failure comes from a programming error or from a contract weakness. However this technique can detect bugs only if contracts are strong enough. Moreover it is automatic only if a prover (typically, an SMT solver) can decide

the non-satisfiability of the first-order formula expressing the falsifiability preservation condition. The test generation proposed in [22] exploits the proof trees built by the KEY prover during a proof attempt. The relevance of generated tests depends on the quality of the provided specification, and it does not allow to distinguish non-compliances from specification weaknesses.

Combination of static and dynamic analysis. Static and dynamic analysis work better when used together, as in SYNERGY [27], its interprocedural and compositional extension in SMASH [25], the method SANTE [9] and the present method. Static analysis maintains an over-approximation that aims at verifying the correctness of the system, while dynamic analysis maintains an under-approximation trying to detect an error. Both abstractions help each other in a way similar to the counterexample guided abstraction refinement method (CEGAR) [16]. The work [10] combines symbolic execution, testing and automatic debugging, through the identification of counterexamples violating metamorphic relations for the program under test. The debugging builds a cause-effect chain to a failure, by analysis of some path conditions. Comparatively, our method focuses on deductive verification rather than on symbolic execution, and aims at verifying behavioral pre-post specifications rather than metamorphic relations.

Counterexamples for non-inductive invariants. Counterexamples can be generated to show that invariants proposed for transition systems are too strong or too weak [15]. Differences with our work are the focus on invariants, the formalism of transition systems, and the use of random testing (with QUICKCHECK).

Other verification feedbacks. Our goal was to find input data to illustrate proof failures. A complementary work [35] proposed to extend a runtime assertion checker to use it as a debugger to help the user understand complex counterexamples. For NC errors in the code, [11] proposed to analyze a trace formula to identify the fragments of code that can cause them. Our approach is complementary on two points. First, we detect either NC or SW errors. Second, we consider that the origin of an NC can be either in the code or in the specifications. Combining our method with such a localization of causes of NC errors, extended to specifications, would be another contribution.

Checking prover assumptions. Axioms are logic properties used as hypotheses by provers and thus usually not checked. Model-based testing applied to a computational model of an axiom can permit to detect errors in axioms and thus to maintain the soundness of the axiomatization [1]. This work is complementary to ours because it tackles the case of deductive verification trivially succeeding due to an invalid axiomatization, whereas we tackle the case of inconclusive deductive verification. [14] proposed to complete the results of static checkers with dynamic symbolic execution using PEX. The explicit assumptions used by the verifier (absence of overflows, non-aliasing, etc.) create new branches in the program's control flow graph which PEX tries to explore. This approach permits to detect errors out of the scope of the considered static checkers, but does not provide counterexamples in case of a specification weakness.

The present work continues previous efforts to facilitate deductive verification by generating counterexamples. We propose an original detection technique of three categories of proof failure that gives a more precise diagnosis than in the previous work using testing. That is due to dedicated detection methods for non-compliances and subcontract weaknesses, as well as the definition and detection of single and global

subcontract weaknesses. To the best of our knowledge, such a complete testing-based methodology, automatically providing the verification engineer with a precise feedback on proof failures was not studied, implemented and evaluated before.

The different techniques of assisting deductive verification (in particular, by generating counterexamples using solvers' counter-models or by test generation) being relatively recent and intrinsically incomplete, further work is still required to better compare them and understand in which cases which technique is more practical.

7 Conclusion and Future Work

We proposed a new approach to improve the user feedback in case of a proof failure. Our method relies on test generation and helps to decide whether the proof has failed or timed out due to a non-compliance (NC) between the code and the specification, a subcontract weakness (SW), or a prover weakness. This approach is based on a spec-to-code program transformation that produces an input program for the test generation tool. Our experiments show that our implementation – in a FRAMA-C plugin, STADY– was able to diagnose over 97% of unproven programs. In particular, the subcontract weakness detection ($\mathfrak{D}^{\mathrm{SW}}$) proposed in this paper was able to diagnose 66.7% of proof failures that remained unclassified after the non-compliance detection ($\mathfrak{D}^{\mathrm{NC}}$).

One benefit of the proposed approach is the ability to provide the verification engineer with a precise reason and a counterexample that facilitate the processing of proof failures. Generated counterexamples illustrate the issue on concrete values and help to find out more easily why the proof fails. The method is completely automatic, relies on the existing specification and does not require any additional manual specification or instrumentation task. As a consequence, this method can be adopted by less experienced verification engineers and software developers.

While the complete method requires to have the source code of called functions, the global subcontract weakness detection ($\mathfrak{D}^{\mathrm{SW}}_{\mathrm{global}}$) remains applicable even without their source code. Another limitation is related to a potentially big number of program paths, which cannot be explored. However, our initial experiments show that in practice most proof failures can be automatically classified even after test generation with a partial test coverage, within a testing time comparable to the time of the proof attempt.

We are convinced that the proposed methodology facilitates the verification task and lowers the level of expertise required to conduct deductive verification, removing one of the major obstacles for its wider use in industry. Future work includes further evaluation of the proposed technique, a study of optimized combinations of $\mathfrak{D}^{\mathrm{NC}}$ and $\mathfrak{D}^{\mathrm{SW}}$ for subsets of annotations and subcontracts, experiments on a larger class of programs and a better support of E-ACSL constructs in our implementation. In the DEWI project, we apply STADY to verification of protocols of wireless sensor networks. An experimental comparison of STADY with the inlining-based technique of [42] is another work perspective that will require the implementation of that technique in FRAMA-C.

Acknowledgment. Part of the research work leading to these results has received funding for DEWI project (www.dewi-project.eu) from the ARTEMIS Joint Undertaking under grant agreement No. 621353. The authors thank the FRAMA-C and PATHCRAWLER teams for providing the tools and support. Special thanks to François Bobot, Loïc Correnson, Julien Signoles and Nicky Williams for many fruitful discussions, suggestions and advice.

References

1. Ahn, K.Y., Denney, E.: Testing first-order logic axioms in program verification. In: TAP (2010)
2. Arlt, S., Arenis, S.F., Podelski, A., Wehrle, M.: System testing and program verification. In: Softw. Eng. & Management (2015)
3. Arndt, J.: Matters Computational - Ideas, Algorithms, Source Code [The fxtbook] (2010), http://www.jjj.de
4. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, http://frama-c.com/acsl.html
5. Berghofer, S., Nipkow, T.: Random testing in Isabelle/HOL. In: SEFM (2004)
6. Botella, B., Delahaye, M., Hong Tuan Ha, S., Kosmatov, N., Mouy, P., Roger, M., Williams, N.: Automating structural testing of C programs: Experience with PathCrawler. In: AST (2009)
7. Burghardt, J., Gerlach, J., Lapawczyk, T.: ACSL by Example (2016), https://gitlab.fokus.fraunhofer.de/verification/open-acslbyexample/blob/master/ACSL-by-Example.pdf
8. Chamarthi, H.R., Dillinger, P.C., Kaufmann, M., Manolios, P.: Integrating testing and interactive theorem proving. In: ACL2 (2011)
9. Chebaro, O., Kosmatov, N., Giorgetti, A., Julliand, J.: Program slicing enhances a verification technique combining static and dynamic analysis. In: SAC (2012)
10. Chen, T.Y., Tse, T.H., Zhou, Z.Q.: Semi-proving: An integrated method for program proving, testing, and debugging. IEEE Transactions on Software Engineering (2011)
11. Christ, J., Ermis, E., Schäf, M., Wies, T.: Flow-sensitive fault localization. In: VMCAI (2013)
12. Christakis, M., Leino, K.R.M., Müller, P., Wüstholz, V.: Integrated environment for diagnosing verification errors. In: TACAS (2016)
13. Christakis, M., Emmisberger, P., Müller, P.: Dynamic test generation with static fields and initializers. In: RV (2014)
14. Christakis, M., Müller, P., Wüstholz, V.: Collaborative verification and testing with explicit assumptions. In: FM (2012)
15. Claessen, K., Svensson, H.: Finding counter examples in induction proofs. In: TAP (2008)
16. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM (2003)
17. Cousot, P., Cousot, R., Fähndrich, M., Logozzo, F.: Automatic inference of necessary preconditions. In: VMCAI (2013)
18. Delahaye, M., Kosmatov, N., Signoles, J.: Common specification language for static and dynamic analysis of C programs. In: SAC (2013)
19. Dijkstra, E.W.: A Discipline of Programming. In: Series in Automatic Computation, Prentice Hall, Englewood Cliffs (1976)
20. Dimitrova, R., Finkbeiner, B.: Counterexample-guided synthesis of observation predicates. In: FORMATS (2012)
21. Dybjer, P., Haiyan, Q., Takeyama, M.: Combining testing and proving in dependent type theory. In: TPHOLs (2003)
22. Engel, C., Hähnle, R.: Generating unit tests from formal proofs. In: TAP (2007)
23. Genestier, R., Giorgetti, A., Petiot, G.: Sequential generation of structured arrays and its deductive verification. In: TAP (2015)
24. Gladisch, C.: Could we have chosen a better loop invariant or method contract? In: TAP (2009)
25. Godefroid, P., Nori, A.V., Rajamani, S.K., Tetali, S.D.: Compositional may-must program analysis: unleashing the power of alternation. In: POPL (2010)

26. Groce, A., Kroening, D., Lerda, F.: Understanding counterexamples with explain. In: CAV (2004)
27. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: SYNERGY: A new algorithm for property checking. In: FSE (2006)
28. Guo, S., Kusano, M., Wang, C., Yang, Z., Gupta, A.: Assertion guided symbolic execution of multithreaded programs. In: ESEC/FSE (2015)
29. Hauzar, D., Marché, C., Moy, Y.: Counterexamples from proof failures in SPARK. In: SEFM (2016), to appear
30. Jakobsson, A., Kosmatov, N., Signoles, J.: Fast as a shadow, expressive as a tree: hybrid memory monitoring for C. In: SAC (2015)
31. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Asp. Comput. 27(3), 573–609 (2015), http://frama-c.com
32. Kosmatov, N.: Online version of PathCrawler. (2010–2015), http://pathcrawler-online.com/
33. Kosmatov, N., Petiot, G., Signoles, J.: An optimized memory monitoring for runtime assertion checking of C programs. In: RV (2013)
34. Kovács, L., Voronkov, A.: Finding loop invariants for programs over arrays using a theorem prover. In: FASE (2009)
35. Müller, P., Ruskiewicz, J.N.: Using debuggers to understand failed verification attempts. In: FM (2011)
36. Owre, S.: Random testing in PVS. In: AFM (2006)
37. Petiot, G., Botella, B., Julliand, J., Kosmatov, N., Signoles, J.: Instrumentation of annotated C programs for test generation. In: SCAM (2014)
38. Petiot, G., Kosmatov, N., Giorgetti, A., Julliand, J.: How test generation helps software specification and deductive verification in Frama-C. In: TAP (2014)
39. Podelski, A., Wies, T.: Counterexample-guided focus. In: POPL (2010)
40. Signoles, J.: E-ACSL: Executable ANSI/ISO C Specification Language, http://frama-c.com/download/e-acsl/e-acsl.pdf
41. The Coq Development Team: The Coq proof assistant. http://coq.inria.fr
42. Tschannen, J., Furia, C.A., Nordio, M., Meyer, B.: Program checking with less hassle. In: Verified Software: Theories, Tools, Experiments (2014)
43. Williams, N., Marre, B., Mouy, P., Roger, M.: PathCrawler: automatic generation of path tests by combining static and dynamic analysis. In: EDCC (2005)