Published in Formal Aspects of Computing (2018)
https://doi.org/10.1007/s00165-018-0456-4

How Testing Helps to Diagnose Proof Failures

Guillaume Petiot[1,2], 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
[2]FEMTO-ST Institute, Univ. of Bourgogne Franche-Comté, CNRS, 25030 Besançon Cedex, France

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 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 C program formally specified in an executable specification language 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 [Dij76] 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. 1a. 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.

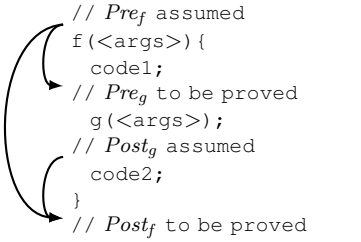
To reflect the fact that some contracts become hypotheses during deductive verification of f we use the term *subcontracts for* f to designate contracts of loops and called functions in f.

Correspondence and offprint requests to: CEA, List, Software Reliability Laboratory, PC 174, 91191 Gif-sur-Yvette Cedex, France. Email: nikolai.kosmatov@cea.fr

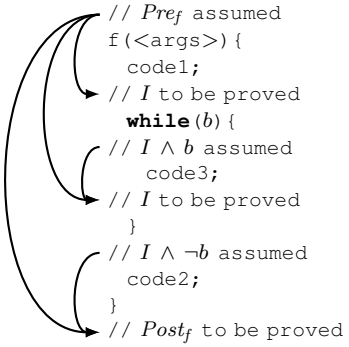
Motivation. One of the most important difficulties in deductive verification is the manual processing of proof failures by the verification engineer since proof failures may have several causes. Indeed, a failure to prove Pre_g in Fig. 1a may be due to a *non-compliance* of the code to the specification: 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 [Coq18, BC04]).

A failure to prove the postcondition $Post_f$ (cf. Fig. 1a) 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 but such a counterexample 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 (cf. Fig. 1b), 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.



(a) f contains a call to function g



(b) f contains a loop

Fig. 1: Verification of a function f with a function call or a loop

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.[1] To make this diagnosis more precise, one should take into account their code as well as their contracts. A study [TFNM13] proposed to use function inlining and loop unrolling (cf. Sec. 8). We propose an alternative approach: to use test case generation techniques based on Dynamic Symbolic Execution (DSE) in order to diagnose proof failures and produce counterexamples. Their usage requires code transformation translating the annotated C program into an executable C code suitable for testing. The application of test generation to the translated program in order to produce counterexamples will also be referred to as *(testing-based) counterexample synthesis*. Previous work suggested several comprehensive debugging scenarios relying on counterexample synthesis only in the case of non-compliances [PKGJ14], and proposed a rule-based formalization of annotation translation for that purpose [PBJ+14]. The cases of subcontract weakness remained undetected and indistinguishable from a prover incapacity.

The overall goal of the present work is to provide a 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.

[1] Some program provers like KEY [BHS07] can replace callees either by code or by contract. For loops, however, it can only be possible to unroll a loop a finite number of times.

The contributions of this paper include:

- a classification of proof failures into three categories: *non-compliance* (NC), *subcontract weakness* (SW) and *prover incapacity*, illustrated by several program examples,
- a definition and comparative analysis of *global* and *single* subcontract weaknesses,
- a complete description of program transformation techniques for diagnosis of non-compliances and subcontract weaknesses for all main kinds of specification clauses,
- a testing-based methodology for diagnosis of proof failures and generation of counterexamples, suggesting possible actions for each category, illustrated on several comprehensive examples,
- adaptive techniques for non-compliance and subcontract weakness detection,
- an implementation of the proposed solution in a tool called STADY[2], and
- experiments showing its capability to diagnose proof failures.

This paper is an extended version of an earlier work [PKB+16] that has been enriched by an extension of the method for support of yet unproven loop contracts, nested loops, loop variants and loop invariants. These extensions have been implemented in the STADY tool and evaluated on a larger set of programs. The present paper also discusses adaptive detection strategies that have been partially implemented. It also includes several additional examples illustrating various kinds of proof failures and gives a better informal view of them.

Paper outline. Sec. 2 presents the tools used in this work. Sec. 3 informally introduces the categories of proof failures and illustrates them with examples. Sec. 4 and 5 present program transformations for the classification of proof failures by category and the synthesis of counterexamples. The methodology for the diagnosis of proof failures is presented in Sec. 6. Our implementation and experiments are described in Sec. 7. Finally, Sec. 8 and 9 present some related work and a conclusion.

2. FRAMA-C Toolset

This work is realized in the context of FRAMA-C [KKP+15], a platform dedicated to analysis of C code that includes various analyzers in separate plugins. The WP plugin performs deductive verification of C programs by means of a weakest precondition calculus. Various automatic SMT solvers can be used to prove the verification conditions (VCs) generated by WP. In this work we use FRAMA-C SILICON, ALT-ERGO 1.01 and CVC3 2.4.1. To express properties over C programs, FRAMA-C offers the behavioral specification language ACSL [BCF+17, KKP+15]. 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.

We use the general term of a *contract* to designate the set of ACSL annotations describing a loop or a function. A function contract is composed of pre- and postconditions including E-ACSL clauses **`requires`**, **`assigns`** and **`ensures`** (see lines 1–3 of Fig. 3 for an example). A loop contract is composed of **`loop invariant`**, **`loop assigns`** and **`loop variant`** clauses (see lines 8–13 of Fig. 3 for an example).

For combinations with dynamic analysis, FRAMA-C also supports E-ACSL [DKS13, Sig12], a large executable subset of ACSL suitable for *Runtime Assertion Checking* (RAC). The purpose of runtime assertion checking is to evaluate each of the annotations encountered during the program execution for a given test datum (i.e. given values of input variables). 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 and memory-related constructs (e.g. on validity and initialization). E-ACSL does not include ACSL features that cannot be evaluated at runtime, such as unbounded quantifications, lemmas (which usually express non-executable mathematical properties) or axiomatics (non-executable by nature) [Sig12]. It comes with an instrumentation-based translating plugin, called E-ACSL2C [KPS13, JKS15], that transforms annotations into additional C code in order to evaluate them at runtime and report failures.

Since the purpose of this work is to combine static analysis (deductive verification) with dynamic analysis (testing-based counterexample synthesis), it will be convenient to use E-ACSL to express program specifications. However, the spec-to-code translation performed by the E-ACSL2C tool for runtime assertion checking (where the program is run with concrete inputs) is not suitable for counterexample synthesis (where the program is executed symbolically for unknown inputs). Indeed, the code produced by E-ACSL2C relies on complex external libraries (e.g. to handle memory-related annotations and unbounded integer arithmetic of E-ACSL) and cannot assume the precondition of the function

[2] See also `https://github.com/gpetiot/Frama-C-StaDy`.

under verification or another annotation, whereas the code produced for counterexample synthesis can efficiently rely on the underlying symbolic execution engine and constraint solver for these purposes. This explains the need for a dedicated spec-to-code translation mechanism.

For counterexample synthesis, this work relies on PATHCRAWLER [WMMR05, BDH+09, Kos15], a Dynamic Symbolic Execution (DSE) testing tool. PATHCRAWLER is based on a specific constraint solver, called COLIBRI, that implements advanced features such as floating-point and modular integer arithmetic. Symbolic execution makes it possible to support several features essential to this work. PATHCRAWLER supports assumptions, that is, additional hypotheses introduced into the code in the form of a builtin function call `fassume(cond)`. Whenever symbolic execution traverses such an assumption, the condition `cond` is added into the set of constraints. It results in generating only test inputs that satisfy this condition at the corresponding program point. PATHCRAWLER also supports the assignment of a non-deterministic value to some variable `x`, denoted in this paper by `x=Nondet()`. It can be seen as assigning to `x` an additional symbolic input at the corresponding program point (possibly taking a different value each time when this assignment is traversed). It is usually followed by an assumption of the form `fassume(cond);` that can constrain `x` and other variables. In this case, the solver tries to generate a suitable new value for `x` satisfying the required constraints. PATHCRAWLER also supports assertions of the form `fassert(cond);` which report a failure and exit the program whenever the given condition `cond` is not satisfied. Notice that such an assertion adds an additional conditional statement to evaluate `cond`. Finally, PATHCRAWLER offers dedicated builtin support for unbounded integer arithmetic of ACSL annotations (thanks to their support in COLIBRI as well) so that test generation does not need to handle the additional complexity of external libraries required by E-ACSL2C to treat unbounded integers during runtime assertion checking (see [PBJ+14] for more detail).

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. On the class of programs it supports, PATHCRAWLER is also *relatively complete* in the following sense: given a program with a finite number of paths and sufficient time, the tool will exhaustively explore all feasible paths of the program. In this case, the absence of a test for some test objective means that the test objective is infeasible (i.e. impossible to activate). This is due to the fact that the tool does not approximate path constraints [BDH+09, Sec. 3.1]. Of course, given only a finite (bounded) time, the tool can time out without generating a test for a given test objective.

3. Categories of Proof Failures Illustrated by Examples

In this section we describe various kinds of proof failures that can occur during the proof of an annotated program, and illustrate them using several examples of C programs. We start by introducing the notions of modular and non-modular vision of a program.

Modular and Non-modular Vision. Suppose a function under verification f contains one loop or one function call (see Fig.1). In deductive verification, during the proof of the postcondition of f, the code of the loop or called function is replaced by the corresponding contract. We say that the deductive verification tool has a *modular vision* of the function under verification: the code of the loop or called function is ignored by the tool, while their contracts are taken into account instead. The contracts of loops and called functions in f are referred to as subcontracts for f. The proof that the loops and called functions respect the corresponding subcontracts leads to separate VCs and is conducted separately.

On the other hand, for a given test datum, RAC checks every annotation reached by the program execution. RAC has a *non-modular vision* of the program, where the code of loops and called functions is executed without replacing them by the corresponding subcontracts.

Consider the C program of Fig. 2a where f is the function under verification. It calls another function g. The postconditions of g and f on lines 2 and 5 state that the variable x is increased at least by 1 in g and at least by 2 in f. Lines 3 and 6 specify that x is the only variable that can change its value after each call. For the input value $x = 0$, in non-modular vision of the call to g, the value of x after this call is equal to 1. In modular vision of the call to g, the new value satisfies $x \geq 1$. Similarly, for the program of Fig. 2b where the only modified statement is the assignment on line 4, the resulting value of x is equal to 2 in non-modular vision of the call to g. In modular vision, the resulting value of x satisfies $x \geq 1$ again. Notice that the property $x \geq 1$ is the only information on x the program prover has during the proof of f after the call to g. In both examples, the proof of f fails for the postcondition on line 5, while g is successfully proved. (For simplicity, we ignore arithmetic overflows in this example.)

```
1 int x;
2 /*@ ensures x ≥ \old(x)+1; // Proved
3     assigns x; */
4 void g() { x=x+1; }
5 /*@ ensures x ≥ \old(x)+2; // Proof failure
6     assigns x; */
7 void f() {     // If x = 0 here, then after the call to g:
8   g();        // x ≥ 1 in modular vision of g,
9 }             // x = 1 in non-modular vision of g
```

(a) Proof failure for line 5 caused by a non-compliance

```
1 int x;
2 /*@ ensures x ≥ \old(x)+1; // Proved
3     assigns x; */
4 void g() { x=x+2; }
5 /*@ ensures x ≥ \old(x)+2; // Proof failure
6     assigns x; */
7 void f() {     // If x = 0 here, then after the call to g:
8   g();        // x ≥ 1 in modular vision of g,
9 }             // x = 2 in non-modular vision of g
```

(b) Proof failure for line 5 caused by a subcontract weakness of g

Fig. 2. Toy examples of non-compliance and subconstract weakness (where for simplicity, overflows are ignored)

Deductive Verification and Counterexamples. A deductive verification tool (also referred to as a program prover) usually transforms the verification problem into several *Verification Conditions* (VCs) and reports which ones are not proved. As in other specification languages, for convenience of the users, an ACSL clause (such as pre- and postcondition, loop invariant, assertion) containing a conjunction of several properties can be split into several clauses of the same kind, written one after another. For instance, the postcondition **ensures** `P1 ∧ P2;` is equivalent to the sequence **ensures** `P1;` **ensures** `P2;` of two clauses. In such cases, the VCs are generated accordingly, that is, a separate VC for each clause.

The WP plugin of FRAMA-C generates VCs of the following kinds[3] (each of which can lead to a proof failure):

- a postcondition holds (for the function under verification),
- an assertion holds (at the corresponding program point),
- a loop invariant initially holds (i.e. the loop invariant of a loop is satisfied before the very first loop iteration),
- a loop invariant is preserved (i.e. if the loop invariant of a loop holds before a loop iteration, it also holds after it),
- a loop variant is non-negative before each iteration of the loop (that is, when execution enters the loop body),
- a loop variant decreases (i.e. the value of the variant after an iteration of the loop is strictly smaller than before this iteration),
- a precondition of a callee holds just before the call.

We propose to use dynamic analysis to help the verification engineer to analyze and fix proof failures. As dynamic analysis, we consider test generation on a transformed version of the initial program, aiming at generating a test datum (called *counterexample*) that illustrates the failure of the annotated program. This counterexample synthesis is implemented in two stages: a transformation of the annotated program into an *instrumented program* by translating annotations into code, and an application of a Dynamic Symbolic Execution tool on this instrumented program to find a counterexample. By nature, this method is in general incomplete since the set of program paths can be infinite or too large, or too complex (for example, for the underlying solver to solve the path constraints within a reasonable time).

Categories of Proof Failures. We distinguish three categories of proof failures:

- a *non-compliance* (NC) between program code and specification,
- a *subcontract weakness* (SW),
- a *prover incapacity.*

A *non-compliance* occurs when (concrete) program execution of some test datum respecting the precondition of the function under verification leads to a failure of some annotation. This failure can be detected by runtime assertion checking that corresponds to non-modular vision of all callees and loops. Such a test datum is called a *non-compliance counterexample* (NCCE). For example, for the program of Fig. 2a, the input $x = 0$ is a non-compliance counterexample: its execution in non-modular vision leads to a resulting value $x = 1$ that does not satisfy the postcondition of f (cf. line 5).

A *subcontract weakness* occcurs when the contracts of some loop(s) or called function(s) are too weak to deduce an annotation, though this proof failure cannot be explained by a non-compliance. In other words, there is a counterexample in modular vision of the corresponding function calls or loops that is not a counterexample in non-modular

[3] Other deductive verification tools may structure these properties in a slightly different way.

vision. This needs to be explained in more detail. In modular vision, several executions (possibly with different outputs) can be considered as valid executions of the same test datum. Indeed, in general a test datum cannot be executed concretely in a deterministic way in modular vision since some subcontracts can be satisfied for several output values of variables they are allowed to modify. But it can be executed symbolically, and one value can be chosen for each variable potentially modified by callees or loops considered in modular vision. We call such values *subcontract outputs*. For a called function, the returned value is a subcontract output as well. Their choice makes it possible to consider concrete execution of other parts of code that are not replaced by subcontracts. For instance, for the input $x = 0$, any value satisfying $x \geq 1$ after the call to g can be part of a valid execution in modular vision of f for Fig. 2b. We denote by nondet_x^i the subcontract output for x after the i-th subcontract ($i \geq 1$) traversed by program execution (in our example, after the call to g). If there is only one traversed subcontract, we omit the upper index and simply write nondet_x. To illustrate the proof failure of the postcondition of f, the subcontract output nondet_x of x after the call to g should be 1. Taking a greater value, say $\mathrm{nondet}_x = 2$, does not provoke a failure of the postcondition of f. Thus the input value $x = 0$ with the subcontract output $\mathrm{nondet}_x = 1$ after the call to g illustrates the subcontract weakness of g for the postcondition of f. Notice that this is not a non-compliance counterexample: in non-modular vision, the test input $x = 0$ leads to an output $x = 2$ that respects the postcondition of f.

Strictly speaking, a complete counterexample in modular vision includes a test datum V and subcontract outputs each time a subcontract is traversed by the execution, such that (i) the chosen execution of V in modular vision leads to an annotation failure, and (ii) the execution of V in non-modular vision does not fail. We call it a *subcontract weakness counterexample* (SWCE). For simplicity of definition of an SWCE, we sometimes give only the test datum V and omit subcontract outputs in this paper[4].

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. For instance, the input value $x = 0$ with the subcontract output $\mathrm{nondet}_x^1 = 1$ after the call to g is not considered as a subcontract weakness counterexample for function f for Fig. 2a since $x = 0$ is a non-compliance counterexample.

Remark 2. To describe executions in non-modular vision and detect subcontract weakness counterexamples, it is necessary to know subcontract outputs (i.e. which variables can be modified). For subcontract weakness detection, we assume that every subcontract for f contains a `(loop) assigns` clause. Such a clause defines the list of variables (surviving at the end of the subcontract) that can change their values after the corresponding function call or loop. Requiring such clauses is not a strong limitation since such clauses are anyway necessary to prove any nontrivial code.

Finally, in some cases, the prover can be unable to deduce an annotation while it does follow mathematically from the assumptions, and there exist neither non-compliance counterexamples nor subcontract weakness counterexamples. We call this case a *prover incapacity*. It can happen for properties with non-linear arithmetics, requiring reasoning by induction or additional lemmas, etc. Such cases were very frequent a few years ago and become less common for many simple programs today thanks to a very significant progress made by the modern SMT solvers. Unfortunately, they cannot be fully eliminated because of prover incompleteness.

The remainder of this section illustrates various proof failures with three simple examples of annotated C programs: integer square root (Sec. 3.1), binary search (Sec. 3.2) and a longer and less classic example of restricted growth functions (Sec. 3.3). The examples are presented in increasing order of complexity. The failures are shown for slightly modified versions of the examples, where we tried to cover a wide range of errors and omissions (incorrect expressions, wrong comparison operations, wrong or incomplete annotations, annotation omission, etc.). Non-compliances and subcontract weaknesses are illustrated for all examples. An example of prover incapacity is given in Sec. 3.3. 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 $\mathbb{Z}$).

3.1. Example 1: Multiplication-Free Integer Square Root

The program in Fig. 3. computes in the local variable r the integer square root of a given non-negative integer n, that is, a non-negative integer value x such that $x^2 \leq n < (x+1)^2$. The variables y and z respectively store r^2 and

[4] of course, they are always reported by the STADY tool and are very useful for a detailed analysis of the proof failure.

```
1  /*@ requires 0 ≤ n ≤ 10000;      // (S1) With requires \true; "Invariant initially holds" fails
2      ensures \result*\result ≤ n < (\result+1)*(\result+1);
3      assigns \nothing; */
4  int sqrt(int n) {
5    int r = n;
6    int y = n*n;
7    int z = -2*n+1;                // (S2) With z = 2*n+1; "Invariant initially holds" fails
8  /*@ loop invariant 0 ≤ r ≤ n ∧  // (S10) With r ≤ n "Variant␣non␣negative" fails
9        y == r*r ∧
10       n < (r+1)*(r+1) ∧          // (S7) Without line 10 "Postcond.holds" fails
11       z == -2*r+1;               // (S3) With z == 2*r+1; "Inv.init.holds/Inv.preserved" fail
12       loop assigns r, y, z;      // (S5) Without the condition on line 11 "Inv.preserved" fails
13       loop variant r; */         // (S9) With loop variant r-n; "Variant␣is␣non-negative" fails
14   while(y > n) {                 // (S6) With while(y>n+1) "Postcond.holds" fails
15     y = y+z;                     // (S4) With y = y-z; "Inv.preserved" fails
16     z = z+2;
17     r = r-1;
18   }
19   return r;                      // (S8) With return r-1; "Postcond.holds" fails
20 }
```

Fig. 3. Integer square root by decrementation

Version	Line changes in Fig. 3		Proof status: Proved (✓) or Failure (?) with failing annot.	Category of proof failure
	Line	Modified clause/statement		
S_0	no changes		✓	Proved
S_1	1	`requires \true;`	? (inv. init. holds)	nc
S_2	7	`int z = 2*n+1;`	? (inv. init. holds)	nc
S_3	11	`z == 2*r+1;`	? (inv. init. holds, inv. preserved)	nc
S_4	15	`y = y-z;`	? (inv. preserved)	nc
S_5	11	`\true;`	? (inv. preserved)	sw
S_6	14	`while(y > n+1)`	? (postcond.)	nc
S_7	10	⟨empty⟩	? (postcond.)	sw
S_8	19	`return r-1;`	? (postcond.)	nc
S_9	13	`loop variant r-n;`	? (variant non-negativity)	nc
S_{10}	8	`loop invariant r≤n ∧`	? (variant non-negativity)	sw

Fig. 4. Proof failures for different versions of the integer square root example given in Fig. 3

the difference $(r-1)^2 - r^2$. This implementation uses them to avoid slower multiplications (other than those by 2 efficiently executed by bitshifts). The program initially over-approximates r with n. Then it decrements r until the inequality $y \leq n$ becomes satisfied. Thus, the function returns the greatest integer r such that $r^2 \leq n$. Line 1 specifies the precondition and line 2 its postcondition. Line 3 indicates that all (non local) variables should keep the same values after the function call as before. Lines 8–11 define a loop invariant. Line 12 indicates which variables may be modified by the loop, while line 13 defines a loop variant. A *loop variant* is an integer expression that must be non-negative whenever a loop iteration starts, and must strictly decrease at each iteration, thus allowing the deductive verification tool to deduce the termination of the loop after a finite number of steps.

To simplify this example, in addition to stating that input value n is non-negative, we limit it by $10\,000$ to avoid arithmetic overflows. This is done to simplify the presentation and is not a limitation of the proposed method: the absence of arithmetic overflows can be treated by STADY as any other assertions. Indeed, WP can automatically insert assertions stating the absence of arithmetic overflows, and then tries to prove them. If the proof of such an assertion fails, STADY can be used to diagnose the proof failure (as we illustrate for one assertion of this kind in Sec. 3.3).

Let us illustrate some cases of proof failures using modified versions of the annotated program of Fig. 3. Fig. 4 gives the considered versions with the modified lines, their proof status with some failing annotations and category of proof failure (if any). The initial version S_0 presented in Fig. 3 can be completely proved using FRAMA-C/WP. Each of the other versions contains exactly one modification.

3.1.1. Failure to prove that the loop invariant initially holds

If we

replace the precondition on line 1 of Fig. 3 by a trivial precondition **`requires \true;`** (S_1)

then the loop invariant (in particular, the property `0 ≤ r`) cannot be shown to hold before the first iteration of the loop. Alternatively, suppose that

the assignment `z = -2*n+1` *on line 7 of Fig. 3 is replaced by the assignment* `z = 2*n+1`. (S_2)

Then the loop invariant `z == -2*r+1` is not true before the loop, therefore, the proof that the loop invariant initially holds fails. On the other hand, in these two cases the proof that the loop invariant is preserved succeeds. Last, suppose that

the part `z == -2*r+1` *of the loop invariant on line 11 of Fig. 3 is replaced by* `z == 2*r+1`. (S_3)

Then the loop invariant is neither initially true, nor preserved.

In these cases the precondition, the code before the loop and the loop invariant are not compliant, so at least one of them must be modified. The postcondition is still established in all three cases.

Since no function calls or other loops appear before the loop in this program, we cannot illustrate the case of their contract weakness in this case, but such cases are of course possible in general.

3.1.2. Failure to prove that the loop invariant is preserved

Suppose that

the assignment `y = y+z` *on line 15 of Fig. 3 is replaced by* `y = y-z`. (S_4)

Then the proof that the invariant is preserved fails, in particular, because the loop body does not preserve the property `y == r*r`. This failure reveals a non-compliance between the loop body and the invariant. Suppose now that

the line 11 of Fig. 3 is empty, i.e. the part `z ==-2*r+1` *of the loop invariant is not provided.* (S_5)

Then the proof that the invariant is preserved fails as well. Here, the loop invariant $0 \leq r \leq n \wedge y = r^2 \wedge n < (r+1)^2$ is actually satisfied before the loop and after each loop iteration (and in particular, runtime assertion checking will not detect a failure for any test data). This failure reveals a weakness of the loop invariant that is not sufficient to establish the proof of its preservation. On the other hand, in both cases the proofs that the invariant initially holds and that the postcondition is established succeed.

3.1.3. Failure to prove that the postcondition holds

Suppose that

the loop condition `y > n` *on line 14 of Fig. 3 is replaced by* `y > n+1`. (S_6)

Then the proof that the postcondition holds fails because of a non-compliance between code and specification. Indeed, in this case the loop can exit too early, as soon as $r^2 \leq n + 1$ becomes satisfied. For instance, for input value $n = 3$, the value $r = 2$ will be returned instead of 1.

Suppose now that

the line 10 of Fig. 3 is empty, i.e. the part `n <(r+1)*(r+1)` *of the loop invariant is not provided.* (S_7)

Then the postcondition is not proved because the loop invariant is too weak. It is satisfied before the loop and after each loop iteration, but is not sufficient to establish the postcondition. Such a weakness of a loop invariant can be very difficult to distinguish from other failures.

Let us now assume that

the return statement **`return`** `r;` *on line 19 of Fig. 3 is replaced by* **`return`** `r-1;` (S_8)

Again, the proof of the postcondition of the function fails because after exiting the loop, the loop condition $y \leq n$ with the invariant $y = r^2$ imply $r^2 \leq n$, thus after **`return`** `r-1;` we have `(`**`\result`**`+1)*(`**`\result`**`+1) ≤ n` instead of the expected postcondition `n < (`**`\result`**`+1)*(`**`\result`**`+1)`.

The cases of weakness and non-compliance of a loop invariant can be very difficult to distinguish. Finally, an error in the postcondition itself can also lead to a proof failure.

```
1  /*@ requires 1 ≤ n ≤ 10000 ∧ \valid(t+(0..n-1));
2      requires ∀ ℤ i, j; 0 ≤ i < j < n ⇒ t[i] ≤ t[j]; // (B3) Without line 2 "Inv.preserved" fails
3      ensures -1 ≤ \result ≤ n-1;
4      ensures ∀ ℤ i; 0 ≤ i ≤ \result ⇒ t[i] ≤ x;
5      ensures ∀ ℤ i; \result < i < n ⇒ t[i] > x;
6      assigns \nothing; */
7  int binary_search(int t[], int n, int x) {
8    int L = -1, R = n-1;            // L..R is the search range
9    /*@ loop invariant -1 ≤ L ≤ R ≤ n-1;
10       loop invariant ∀ ℤ i; 0 ≤ i ≤ L ⇒ t[i] ≤ x;   // (B4) Without lines 10 and/or 11:
11       loop invariant ∀ ℤ i; R < i < n ⇒ t[i] > x;   //      ..."Postcond.holds" fails
12       loop assigns L, R;          // (B6) With loop assigns L,R,t[0]; "Inv.pres./assigns" fail
13       loop variant R-L; */        // (B1) With loop variant n-R; "Variant␣decreases" fails
14   while(L < R) {
15       int m = (L+R+1)/2;          // (B2) With m = (L+R)/2; "Variant␣decreases" fails
16     if(t[m] > x)
17       R = m-1;
18     else
19       L = m;
20   }
21   return L;
22 }
```

Fig. 5. Binary search of an element in a sorted array

Version	Line changes in Fig. 5		Proof status: Proved (✓) or Failure (?) with failing annot.	Category of proof failure
	Line	Modified clause/statement		
B_0	no changes		✓	Proved
B_1	13	`loop variant n-R;`	? (variant decreases)	nc
B_2	15	`int m = (L+R)/2;`	? (variant decreases)	nc
B_3	2	⟨empty⟩	? (inv.preserved)	nc
B_4	10–11	⟨empty⟩	? (postcond. fails)	sw
B_5	12	`loop assigns L;`	? (loop assigns fails)	nc
B_6	12	`loop assigns L,R,t[0];`	? (inv.pres.,assigns fail)	sw

Fig. 6. Proof failures for different versions of the binary search example given in Fig. 5

3.1.4. Failure to prove that the loop variant is non-negative

It is possible that the expression given as a loop variant does not allow to prove loop termination even when all other annotations (loop invariant, postcondition, etc.) are proved. For example, suppose we

replace the variant on line 13 of Fig. 3 by $r - n$. (S_9)

Then the proof that the variant is non-negative each time the loop enters a new iteration fails because the loop invariant becomes negative already at the second iteration. This proof failure is an example of non-compliance between code and specification.

Now assume that

the condition $0 \leq r \leq n$ *of the loop invariant on line 8 of Fig. 3 is replaced by* $r \leq n$. (S_{10})

In this case, the same proof failure occurs for a different reason: the loop invariant is too weak to deduce that the loop variant is non negative. In this case, runtime assertion checking will not detect any failure. Notice that all other VCs are proved in both cases.

Examples of failures to prove that the loop variant decreases are given in Sec. 3.2.

3.2. Example 2: Binary Search

The second example shown in Fig. 5 is a program that performs binary search of a given element x in a given array t of size n. The array is supposed to be sorted in increasing order. This version returns a value k that gives the index of the rightmost array element $t[k]$ such that $t[k] \leq x$, and $k = -1$ if x is strictly smaller than all elements of t. The

program maintains the range $L..R$ the searched index k can belong to, and reduces this range by half at every step of the loop on lines 14–20. The middle index m is computed (line 15) and the value of t at index m is compared to x to update the range (lines 16–19). As soon as the search range is reduced to one element (i.e. $L = R$), it contains the required value that is returned. The precondition indicates that the input array t is a valid array of positive size n (line 1) and that it is sorted (line 2). The ACSL predicate `\valid(t+(0..n-1))`, which is an equivalent form for `\valid(&t[0..n-1])`, states that the array elements `t[0]`, ..., `t[n-1]` referred by the indicated range of pointers can be safely read and written. To simplify the example and avoid considering overflows, we assume that $n \leq 10\,000$. The postcondition is defined on lines 3–5. Line 6 specifies that the global memory state (that is, non local variables) should keep the same values after the function execution as before it.

We will use this example to illustrate proof failures related to loop variants (some of which cannot be illustrated by the example of Fig. 3) and `(loop) assigns` clauses. We also consider failures caused by two very common errors, systematically done by the majority of students trying to specify and prove this example for the first time. These modified versions are summarized in Fig. 6. The initial version (B_0) presented in Fig. 5 is completely proved by FRAMA-C/WP.

3.2.1. Failure to prove that the loop variant decreases

For a successful proof of termination of a loop, in addition to being non negative each time when the loop enters a new iteration (cf. Sec. 3.1), the loop variant should strictly decrease. Let us suppose that

the loop variant $R - L$ on line 13 of Fig. 5 is replaced with $n - R$. (B_1)

This loop variant candidate does not necessarily decrease since a loop iteration modifies either L or R. Even if the loop actually terminates in this case, the deductive verification tool cannot prove it.

Another example of a common programming error inducing a similar proof failure occurs if

the assignment $m = (L + R + 1)/2$ on line 15 of Fig. 5 is replaced with $m = (L + R)/2$. (B_2)

In this case the program does not necessarily terminate, and the variant $R - L$ does not strictly decrease at each iteration. Indeed, when $R = L+1$, the value of m becomes $m = (L+L+1)/2 = L$. If $x \geq t[L]$ then the assignment $L = m$ on line 19 does not change the value of L and the loop variant remains unchanged as well. This mistake is revealed for example for $n = 2$, $x = 1$ and $t[0] = t[1] = 0$, because after the first iteration, we have $L = 0$ and $R = 1$, so the loop variant expression is $R - L = 1$, and after the second iteration the values remain the same: $L = 0$, $R = 1$ and $R - L = 1$.

3.2.2. Failures related to two common errors

A common error often made by junior verification engineers is to omit the precondition that the array is sorted. Let us suppose that

the precondition on line 2 of Fig. 5 is not provided. (B_3)

In this case, the proof that the loop invariant (lines 10 and 11) is preserved fails. The analysis of the failure becomes easier with a counterexample, for instance, for input data $n = 2$, $x = 0$, $t[0] = 10$ and $t[1] = -10$, we have after the first loop iteration $m = 0$, $L = -1$ and $R = 0$ so the loop invariant of line 11 fails after this iteration since $t[1] > x$ does not hold.

Another common error is related to an incomplete loop invariant. Suppose

the loop invariants on lines 10–11 of Fig. 5 are omitted. (B_4)

In this case, the proof that the postcondition holds fails. In this case, the loop invariant is too weak to prove the postcondition.

3.2.3. Failures related to `(loop) assigns` clauses

The `assigns` clause (in a function contract) and the `loop assigns` clause (in a loop contract) define variables (or, more precisely, left-values[5]) that are allowed to have different values after the corresponding function or loop. These

[5] In C, left-values basically refer to objects whose address can be taken and, therefore, that have a location (e.g. variables, dereferenced pointers).

clauses provide a concise way to express that all other variables remain unchanged without having to list them. For a function contract, local variables should not be specified since only non local variables survive after the end of the function. For a loop, all potentially modified global and local variables (except local variables whose scope is entirely inside the loop body) should be specified since all variables that exist before and after the loop body can potentially change their values after a loop iteration. Let us illustrate by two examples the issues related to **`loop assigns`** clauses. (Similar issues occur for **`assigns`** clauses in functions contracts.)

The first issue is related to a too restrictive **`loop assigns`** clause. Suppose for instance that

the clause on line 12 of Fig. 5 is replaced with **`loop assigns`** `L`. (B_5)

Since this clause is too restrictive (it does not allow modification of R) the deductive verification tool reports a proof failure of the **`loop assigns`** clause. Indeed, this is a non-compliance between code and specification. Contrary to other cases, this failure is very explicit: the failing annotation is too restrictive. That is why in this work we do not seek to further diagnose the proof failures of (**`loop`**) **`assigns`** clauses as non-compliances since we consider that such proof failures are sufficiently explicit.

The second issue is related to a too permissive **`loop assigns`** clause. Let us suppose that the **`loop assigns`** clause is too general, for example, if

the clause on line 12 of Fig. 5 is replaced with **`loop assigns`** `L,R,t[0]`. (B_6)

In this case, the **`loop assigns`** clause (line 12) itself is proved, but the proofs of the **`assigns`** clause (line 6) and the invariant preservation fail. This is an example of a too weak subcontract: if the loop can modify `t[0]`, these properties cannot be proved any more. The first failure would not occur if some other parts of the function (and thus the **`assigns`** clause) were allowed to modify `t[0]`, that would make it even more difficult to understand the reason of the failure to prove that the loop invariant is preserved. Notice that in this example, all annotations are still satisfied in practice (and runtime assertion checking will not detect any failure). The feedback of the deductive verification tool is not sufficiently precise in this case, so we do diagnose weaknesses of (**`loop`**) **`assigns`** clauses in this work since they can lead to proof failures of various annotations. A counterexample illustrating that a value of `t[0]` can change after the loop according to the loop contract and thus contradict the loop invariant preservation can be very helpful to understand the issue.

3.3. Example 3: Restricted Growth Functions (RGF)

To illustrate various categories of proof failures on a more complex example, let us consider the C program in Fig. 7. It includes function calls and a lemma requiring proof by induction. It illustrates new proof failure cases, among which SW of a function contract and prover incapacity.

This example comes from a C library of generators of combinatorial structures specified with ACSL for deductive verification [GGP15]. The main function f is similar to the successor function `next_rgf` presented in the running example of this previous work [GGP15, Section 2.2]. The main difference is that its last loop is implemented here by the auxiliary function g, in order to illustrate modularity.

The successor function f modifies its input array a, whilst preserving an invariant on a (*invariance property*) and turning a into a greater array in lexicographic order (*progress property*).

In Combinatorics, a function $a : \{0, \ldots, n-1\} \to \{0, ..., n-1\}$ is a (particular case of) *restricted growth function (RGF) of size* $n > 0$ if $a(0) = 0$ and $0 \leq a(k) \leq 1 + a(k-1)$ for $1 \leq k \leq n-1$ (that is, the growth of $a(k)$ w.r.t. the previous value $a(k-1)$ is at most 1). The interested reader will find more detail in [MV13]. The invariant defined by the ACSL predicate `is_rgf` on lines 1–2 of Fig. 7 states that the C array a stores the values of an RGF.

The first two preconditions of f (lines 23–24) state that a is a valid array of size $n > 0$. The third precondition (line 25) states that a must be an RGF. The **`assigns`** clause (line 26) states that the function is only allowed to modify the values of array a except the first one $a[0]$. The first postcondition (line 27) states that the generated array a is still an RGF. Together with the precondition on line 25 it states the invariance property. The second postcondition (lines 28–31) is a key argument to prove the progress property, when the function returns 1: it states that the leftmost modified element of a has increased. 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. 7. The loop on lines 37–38 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 42) and fills out the rest of the array with zeros (call to g, line 43). The loop contract (lines 34–36) 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.

```
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++)
20     a[k] = 0;
21 }
22
23 /*@ requires n > 0;
24     requires \valid(a+(0..n-1));
25     requires is_rgf(a,n);
26     assigns a[1..n-1];
27     ensures is_rgf(a,n);
28     ensures \result == 1 ⇒
29       (∃ ℤ j; 0 ≤ j < n ∧
30         \at(a[j],Pre) < a[j] ∧
31         (∀ ℤ k; 0 ≤ k < j ⇒ \at(a[k],Pre) == a[k])); */
32 int f(int a[], int n) {
33   int i = n-1;
34   /*@ loop invariant 0 ≤ i ≤ n-1;
35       loop assigns i;
36       loop variant i; */
37   while (i ≥ 1 ∧ a[i] > a[i-1])
38     i--;
39   if (i == 0)  // Last RGF.
40     return 0;
41   //@ assert a[i]+1 ≤ 2147483647;
42   a[i] = a[i] + 1;
43   g(a,n,i);
44   /*@ assert ∀ ℤ l; 0 ≤ l < i ⇒ \at(a[l],Pre) == a[l]; */
45   return 1;
46 }
```

Fig. 7. Successor function for restricted growth functions (RGF)

Version	Line changes in Fig. 7		Proof status: Proved (✓) or Failure (?) with failing annot.	Category of proof failure
	Line	Modified clause/statement		
R_0	no changes		✓	Proved
R_1	25	`requires is_rgf(a,n);` deleted	? (precond. g fails)	nc
R_2	35	`loop assigns i, a[1..n-1];`	? (postcond. fails)	sw
R_3	4-5	Lemma `max_rgf` deleted	? (assert fails)	Prover incapacity
R_4	42	`a[i] = a[i]+2;`	? (precond. g fails)	nc

Fig. 8. Proof failures for different versions of the RGF example given in Fig. 7

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 (line 10). The function is allowed to modify the 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 code of Fig. 7 can be fully proven in WP. We illustrate the following proof failure cases with modified versions of this example summarized in Fig. 8.

3.3.1. NC *of a function contract*

If we

remove the precondition on line 25 of Fig. 7, (R_1)

the precondition of function g on line 10 fails at the call to g on line 43.

3.3.2. SW *of a loop contract (**`loop assigns`** clause) to prove the postcondition*

Suppose we

*replace **`loop assigns`** on line 35 of Fig. 7 by **`loop assigns`** `i,a[1..n-1]`.* (R_2)

Then there is a subcontract weakness to prove the postcondition of the loop.

3.3.3. Prover incapacity

The ACSL lemma `max_rgf` on lines 4–5 of Fig. 7 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 automatically by WP that uses this lemma to ensure the absence of overflow at line 42 (stated on line 41). If we

remove the lemma `max_rgf` on lines 4–5 in Fig. 7 (R_3)

the proof of the assertion fails due to the incapacity of the prover to make the adequate inductive reasoning. With the lemma on lines 4–5, the functions of Fig. 7 are completely proved using WP.

3.3.4. NC *of the precondition of a called function*

If we

replace the statement on line 42 of Fig. 7 by $a[i] = a[i] + 2$, (R_4)

there is a non-compliance of the precondition of g on line 10 for the call on line 43.

The examples of Sec. 3.1, 3.2 and 3.3 clearly demonstrate that the same proof failures can come from very different issues, and belong to different categories.

4. Non-Compliance

For the remainder of the paper, 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[6]. Function f 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.

In this section we define non-compliance more formally and briefly recall the non-compliance detection technique presented in [PBJ+14]. This technique first translates an annotated program P into another C program, denoted P^{NC}, and then applies test generation to produce test data violating some annotations at runtime. We present the intrumented program P^{NC} in Sec. 4.1, define non-compliance and non-compliance detection (denoted $\mathfrak{D}^{\mathrm{NC}}$) in Sec. 4.2, and discuss its adaptive version in Sec. 4.3.

[6] This assumption is not a theoretical limitation of the method: it is made for simplicity of presentation and because PATHCRAWLER currently does not support recursive and mutually recursive functions.

```
1  /*@ requires P1;
2      ensures Q1; */
3  Type_g g(...) {
4    code1;
5    return ...;
6  }
7  /*@ requires P2;
8      ensures Q2; */
9  Type_f f(...) {
10   code2;
11   g(...);
12   /*@ loop invariant I;
13       loop assigns x1, ..., xN;
14       loop variant E; */
15   while(b) {
16     code3;
17   }
18   code4;
19   //@ assert P4;
20   code5;
21   return ...;
22 }
```

→

```
1  Type_g g(...) {
2    // Check that the precondition of g holds
3    int pre_g; Spec2Code(P1, pre_g);
4    fassert(pre_g);
5    code1;
6    // Check that the postcondition of g holds
7    int post_g; Spec2Code(Q1,post_g);
8    fassert(post_g);
9    return ...;
10 }
11 Type_f f(...) {
12   // Assume the precondition of f
13   int pre_f; Spec2Code(P2, pre_f);
14   fassume(pre_f);
15   code2;
16   g(...);
17   // Check that the invariant initially holds
18   int inv1; Spec2Code(I, inv1);
19   fassert(inv1);
20   while(b) {
21     // Check that the variant is non-negative
22     int var1; Spec2Code(E ≥ 0, var1);
23     fassert(var1);
24     BegIter: code3;
25     // Check that the invariant is preserved
26     int inv2; Spec2Code(I, inv2);
27     fassert(inv2);
28     // Check that the variant decreases
29     int var2; Spec2Code(E <\at(E, BegIter), var2);
30     fassert(var2);
31   }
32   code4;
33   // Check that the assertion is true
34   int asrt; Spec2Code(P4, asrt);
35   fassert(asrt);
36   code5;
37   // Check that the postcondition of f holds
38   int post_f; Spec2Code(Q2,post_f);
39   fassert(post_f);
40   return ...;
41 }
```

Fig. 9. (a) An annotated code, (b) its translation in P^{NC} for non-compliance detection ($\mathfrak{D}^{\mathrm{NC}}$)

4.1. Instrumented Program P^{NC}

Fig. 9 illustrates the translation of an annotated C program P into an intrumented program P^{NC}. It shows the translation for function contracts, loop contracts and assertions, where f is the function under verification and g is a called function. For simplicity of presentation, we assume that the code in Fig. 9a does not contain control jump statements such as **`break`** or **`continue`**, and each function can have a unique **`return`** statement (if any) at the very end of its body[7]. As mentioned in Sec. 3.2, the `(loop) assigns` clauses are not considered during non-compliance detection because provers usually give a sufficiently clear feedback about them.

For an E-ACSL predicate Q, we denote by $Spec2Code$`(`Q`, b)` the generated C code that evaluates the predicate Q and assigns its validity status to the Boolean variable `b` (see [PBJ+14] for details). For instance, if Q is the predicate

$\forall\ \mathbb{Z}$ `i; 0` $\leq$ `i <` ℓ `n` $\Rightarrow$ `t[i]` $\neq$ `0;`

then the (simplified) translation produced by $Spec2Code$`(`Q`, b)` can be the following:

```
int i;
for(b = 1, i = 0; i < n && b == 1; i++)
  if (t[i] == 0) b = 0;
```

[7] These assumptions are made for clarity and do not represent limitations of the method. Notice that FRAMA-C performs a code normalization step that facilitates the intrumentation in such cases in STADY.

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);
```

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 (cf. lines 3–4, 7–8 in Fig. 9b). A loop invariant is checked before the loop (for being initially true, cf. lines 18–19 in Fig. 9b) and after each loop iteration (for being preserved by the previous loop iteration, cf. lines 26–27 in Fig. 9b). An assertion is checked at its location (see lines 34–35 in Fig. 9b). To generate only test data that respect the precondition Pre_f of f, Pre_f is assumed at the beginning of f by the code (see an example lines 13–14 in Fig. 9b):

```
int pre_f; Spec2Code(Pre_f, pre_f); fassume(pre_f);
```

where `fassume` assumes the given condition (see lines 13–14 in Fig. 9b).

As in verification conditions, the translation for loop variants is aimed at detecting failures for two kinds of properties: loop variant is non-negative and strictly decreases. Lines 22–23 in Fig. 9b check that the variant E is non-negative at the beginning of each execution of the loop body `code3`. Then lines 29–30 check that the variant strictly decreases at each iteration of the loop by comparing its value at the beginning of the loop body (at label `BegIter` added on line 24) with its value at the end of the loop body.

4.2. Non-Compliance Detection $\mathfrak{D}^{\mathrm{NC}}$

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 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 relative 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 ? (unknown).

4.3. Adaptive Non-Compliance Detection

To reduce the number of sessions of test generation, the instrumentation for non-compliance detection of all required properties can be realized in the same instrumented program P^{NC}. However, a systematic instrumentation of all checks leads to a bigger number of program paths and is usually not necessary. We need to instrument and perform only those checks for which the proof fails. We call this approach *adaptive non-compliance detection*.

For example, the adaptive instrumentation does not add lines 18–19 when the proof that the invariant on line 11 in Fig. 9a initially holds succeeds. Similarly, lines 26–27 are useless when the invariant preservation is proved, while lines 34–35 can be omitted if the assertion on line 18 in Fig. 9a is proved.

For the called functions, the instrumentation for the postcondition is not necessary if the contract of the callee has been proved. Indeed, if the precondition of a callee g is verified and its contract is proved, then the postcondition holds after the call. For example, we can remove the lines 7–8 in Fig. 9b if g is proved because the precondition is guaranteed by lines 3–4.

5. Subcontract Weakness and Prover Incapacity

In this section we formally define subcontract weakness and prover incapacity, and introduce the subcontract weakness detection technique. This technique translates a given annotated program P into another C program, denoted P^{SW}, and then applies test generation to produce test data violating some annotations at runtime. We present the intrumented program P^{SW} in Sec. 5.1, define subcontract weaknesses and subcontract weakness detection (denoted $\mathfrak{D}^{\mathrm{SW}}$) in Sec. 5.2, present the adaptive detection in Sec. 5.3, discuss the differences between global and single subcontract weaknesses in Sec. 5.4, and define prover incapacity in Sec. 5.5.

```
/*@ requires P1;
    assigns y1,...,yN;
    ensures Q1; */
Type_g g(...){
  code1;
  return ...;
}
```

→

```
Type_g g_sw(...){
  // Check that the precondition of g holds
  int pre_g; Spec2Code(P1, pre_g);
  fassert(pre_g);
  // Simulate the body of g by contract
  y1=Nondet(); ... yN=Nondet();
  Type_g ret = Nondet(); // simulates returned value (if any)
  int post; Spec2Code(Q1, post);
  fassume(post); return ret;
}
```

```
Type_f f(...){
  code2;
  g(Args_g);
  code3;
  return ...;
}
```

```
Type_f f(...){
  code2;
  g_sw(Args_g);
  code3;
  return ...;
}
```

Fig. 10. (a) A contract $c \in \mathcal{C}$ of callee g in f, vs. (b) its translation in P^{SW} for subcontract weakness detection ($\mathfrak{D}^{\mathrm{SW}}$)

```
/*@ loop invariant I;
    loop assigns x1,...,xN;
    loop variant E; */
while(b) {
  code4;
}
```

→

```
// Check that the invariant initially holds
int inv1; Spec2Code(I, inv1);
fassert(inv1);
// Simulate a few first iterations by contract
x1=Nondet(); ... xN=Nondet();
int inv2; Spec2Code(I, inv2);
fassume(inv2);
// Execute an arbitrary iteration of the loop
if(b){
  // Check that the variant is non-negative
  BegIter: fassert(E ≥ 0);
  code4;
  // Check that the invariant is preserved
  int inv3; Spec2Code(I, inv3);
  fassert(inv3);
  // Check that the variant decreases
  fassert(E < \at(E,BegIter));
  // Do not continue the iteration if no issue found
  exit(0);
} // Here, b is false, as required after the loop
// The loop is fully simulated by contract
```

Fig. 11. (a) A contract $c \in \mathcal{C}$ of a loop in f, vs. (b) its translation in P^{SW} for subcontract weakness detection ($\mathfrak{D}^{\mathrm{SW}}$)

5.1. Instrumented Program P^{SW}

Following the modular verification approach, we assume that the called functions have been verified before the caller f. Unlike in [PKB+16], we do not assume here that the loop contracts are verified, and consider all (possibly nested) loops without restricting ourselves to weaknesses of the outer loops.

Let c_f denote the contract of f, $\mathcal{C}$ the set of subcontracts for f, and $\mathcal{A}$ the set of all annotations in f and annotations of c_f. In other words, $\mathcal{A}$ contains the annotations included in the set of contracts $\mathcal{C} \cup \{c_f\}$ as well as the assertions in f. We also assume that every subcontract for f contains a (**loop**) **assigns** clause (cf. Remark 2). As in Sec. 4.1, it will be convenient to assume that function f does not contain control jump statements such as **break** or **continue**, and each function can have a unique **return** statement (if any) at the very end of its body.

To apply testing for the contracts of loops and called functions in $\mathcal{C}$ instead of their code, we use a new program transformation of P producing another program P^{SW}. The goal is to replace the code of such a function call or loop by the most general code respecting the corresponding subcontract. Let us first illustrate the transformation for non-imbricated (or non-nested) function calls and loops in f, that is, function calls and loops that are not inside another loop in f.

Suppose that f contains a function call to g whose contract is $c_g \in \mathcal{C}$. The program transformation (illustrated by Fig. 10) generates a new function g_sw with the same signature whose code simulates any possible behavior

respecting the postcondition in c_g, and replaces all calls to g by a call to `g_sw`. First, `g_sw` checks the precondition of g (lines 3–4 in Fig. 10b). Then `g_sw` chooses values for subcontract outputs. It allows any of the variables (or, more generally, left-values) listed in the **`assigns`** clause of c_g to change its value (line 6 in Fig. 10b). It can be done by assigning a non-deterministic value of the appropriate type that is denoted here for simplicity by the dedicated function `Nondet` (cf. Sec. 2), 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 7 in Fig. 10b). 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 8–9 in Fig. 10b).

Suppose now that f contains a loop w whose contract is $c_w \in \mathcal{C}$ (see Fig. 11). The instrumented code P^{SW} should allow test generation to detect:

(i) a weakness of c_w to prove other annotations outside the considered loop w (and therefore reached after exiting w),
(ii) a weakness of c_w to prove properties required by the loop contract c_w at the end of an arbitrary loop iteration (that is, loop invariant is preserved, loop variant is non-negative and decreases), and
(iii) a weakness of c_w to prove other properties coming from the body `code4` of w in case it contains assertions, function calls or nested loops (i.e. an assertion in the body of loop w, precondition of a function called in the body of loop w, and the fact that the loop invariant of an inner nested loop w' inside loop w initially holds).

In the instrumented program presented in Fig. 11b, the detection of (i) is ensured by the program paths where lines 1–7 are followed by the empty else branch of the conditional statement on line 9 (i.e. the case b is not true). Indeed, in this case the generated code simulates a complete execution of the loop by the loop contract c_w. First, it checks that the loop invariant initially holds (lines 2–3 in Fig. 11b). Then it chooses values for subcontract outputs and assumes that the loop invariant is preserved (lines 5–7 in Fig. 11b). The paths going through the else branch of the conditional statement on line 9 ensure additionally that b is not true. Thus the "loop postcondition" $I \wedge \neg b$ after the loop is indeed ensured on line 21.

The detection of (ii) and (iii) is achieved by the program paths where lines 1–7 are followed by the then branch of the conditional statement on line 9 in Fig. 11b (i.e. the case b is true). In this case, lines 1–7 simulate a few iterations by the loop contract c_w, followed by an inlined iteration (on lines 10–17). The code of the iteration checks that the loop invariant is preserved (lines 14–15), that the loop variant is non-negative (line 11) and decreases (line 17). The label `BegIter` is added on line 11 to refer to the value of the variant E at the beginning of the iteration.

Notice that pieces of code `code2`, `code3` and `code4` in Fig. 10 and 11 are also transformed in P^{SW} in the same way: if they contain function calls or loops they are in turn replaced as shown in Fig. 10 and Fig. 11. In addition, the transformation treats in the same way as in P^{NC} all assertions in $\mathcal{A}$ (they are not shown in Fig. 10b and Fig. 11b but an example is given on lines 34–35 in Fig. 9b). This recursive transformation ensures that the program paths where lines 1–7 are followed by the then branch of the conditional statement on line 9 address (iii) and detect a weakness of c_w for other properties coming from the body of w. Since these paths are only used to detect weaknesses of c_w for annotations inside an arbitrary loop iteration, they are cut (line 19 in Fig. 11b) and do not continue to statements outside the loop body (already addressed by (i)).

In this way, the intrumented program P^{SW} creates a modular vision of f suitable for detection of weaknesses by test generation.

5.2. Subcontract Weakness Detection $\mathfrak{D}^{\mathrm{SW}}$

As explained in Remark 1, we do not consider the same counterexample as a non-compliance and subcontract weakness counterexample at the same time.

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} on V does not report any annotation failure, while the execution of P^{SW} on V for some suitable subcontract outputs reports an annotation failure. In this case, we say that V (with suitable subcontract outputs) is a *global subcontract weakness counterexample* (global SWCE) for the set of subcontracts $\mathcal{C}$.

Test generation can be applied on P^{SW} to generate global SWCE candidates. When it finds a test datum V (with some suitable subcontract outputs) such that P^{SW} fails on V, we use runtime assertion 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 $\mathfrak{D}^{\mathrm{SW}}_{\mathrm{global}}$. In the method described below in

Section 6, $\mathfrak{D}^{\mathrm{SW}}_{\mathrm{global}}$ will be applied after $\mathfrak{D}^{\mathrm{NC}}$ (but since $\mathfrak{D}^{\mathrm{NC}}$ could have timed out, $\mathfrak{D}^{\mathrm{SW}}_{\mathrm{global}}$ can still find a NCCE). 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}) and subcontract outputs (for a global SWCE) are 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 3 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 function call or loop by the most general code respecting the postcondition of the corresponding subcontract c (as indicated in Fig. 10 and 11). Other annotations in $\mathcal{A}$ are transformed in the same way as in P^{NC}. We say that the proof of an annotation a *relies* on a subcontract c of a function call or a loop if this subcontract becomes a hypothesis during the proof of a, that is, c is an assumption in the corresponding verification condition. Typically, if annotation a appears after a function call (or a loop), the proof of a assumes the contract of the called function (or the loop) since the prover uses a modular vision, where the code of the function (or the loop) is replaced by the subcontract (cf. Section 3).

Definition 3 (Single subcontract weakness). Let c be a subcontract for f, and a an annotation in f whose proof relies on c. We say that c is a *too weak subcontract* (or has a *single subcontract weakness*) for a in f if there exists a test datum V for f respecting its precondition, such that the execution of P^{NC} on V does not report a failure of annotation a, while the execution of P^{SW}_c on V for some suitable subcontract outputs reports a failure of annotation a. In this case, we say that V (with suitable subcontract outputs) is a *single subcontract weakness counterexample* (single SWCE) for the subcontract c with respect to annotation a 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) and subcontract outputs (for a single SWCE) are recorded as well. Otherwise, it returns ? (unknown).

5.3. Adaptive Subcontract Weakness Detection

As for $\mathfrak{D}^{\mathrm{NC}}$, to reduce the number of test generation sessions and the number of program paths in the instrumented programs, we propose an *adaptive subcontract weakness detection*. Indeed, checking annotations whose proof was successful is not necessary (in both $\mathfrak{D}^{\mathrm{SW}}_{\mathrm{global}}$ and $\mathfrak{D}^{\mathrm{SW}}_{\mathrm{single}}$ steps). Second, a subcontract c can obviously be too weak for an annotation a only if the proof of a relies on c (cf. Def. 3). Thus, it is useful to look for a single subcontract weakness of a subcontract c for an annotation a only if a is unproven and if the proof of a relies on c. We can therefore restrict the $\mathfrak{D}^{\mathrm{SW}}_{\mathrm{single}}$ to subcontracts c that may have an impact on an *unproven* annotation.

5.4. 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 3. 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. 12a, 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` $\geq$ `\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`$\geq$`\old(x)+3` after executing the three subcontracts, and can exhibit a global SWCE.

```
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. 12. Two examples where the proof of f fails due to subcontract weaknesses

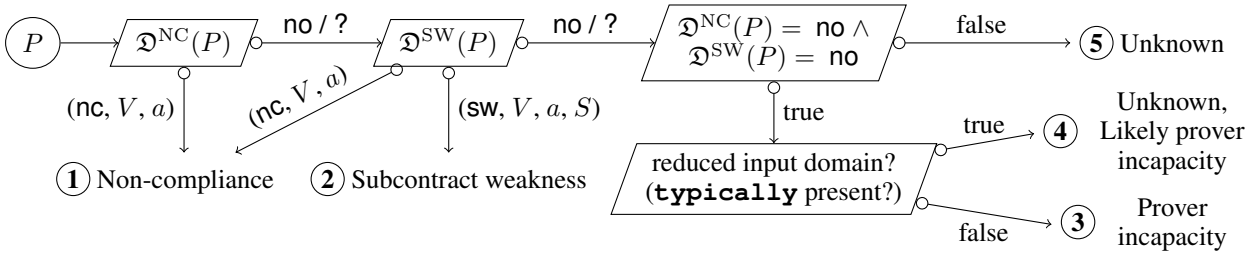


Fig. 13. Combined verification methodology in case of a proof failure on P

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. 12b, 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` $\geq$`\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 the subcontract weakness detection both to precisely indicate, when possible, a single too weak subcontract, and to be able to find global subcontract weaknesses even when there are no single ones.

5.5. 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} report any annotation failure on V. In other words, there is no NCCE and no global SWCE for P.

Notice that Definitions 1, 2, 3, 4 define theoretical notions of proof failure reasons. As program verification in general, their sound and complete detection is undecidable. The next section proposes our method for diagnosis of proof failures.

Version	Line changes in the code in Fig. 7	Intermediate outcome			Final outcome
		Proof (failing annot.)	$\mathfrak{D}^{\mathrm{NC}}$	$\mathfrak{D}^{\mathrm{SW}}$	
R_0	no changes	✓	–	–	Proved
R_1	Line 25 deleted	? (l. 27, 41 and 43)	nc	–	(1) V_1 is an NCCE
R_2	`loop assigns i, a[1..n-1];` on line 35	? (l. 41, 43, 44, 27 and 28–31)	?	sw on l. 37–38	(2) V_2 is an SWCE
R_3	Lemma deleted (lines 4-5)	? (line 41)	?	?	(5) Unknown
R'_3	Lemma deleted (lines 4-5), and `typically n < 5;` added after line 25	? (line 41)	no	no	(4) Likely prover incapacity (analysis with reduced domain)
R_4	`a[i] = a[i]+2;` on line 42	? (line 43)	nc	–	(1) V_4 is an NCCE

with $V_1 = \langle$`n=1; a[0]=-214739`$\rangle$, $V_2 = \langle$`n=2; a[0]=a[1]=0;` $\mathrm{nondet}_{\texttt{a[1]=97157}}$`;` $\mathrm{nondet}_{\texttt{i=0}}$ $\rangle$ and $V_4 = \langle$`n=2; a[0]=a[1]=0`$\rangle$.

Fig. 14. Method results for different versions of the RGF illustrating example

6. Diagnosis of Proof Failures using Structural Testing

In Section 6.1 we present an overview of the proposed method for diagnosis of proof failures using the detection techniques of Sections 4 and 5 and illustrate it on several examples. Section 6.2 provides a comprehensive list of suggestions of actions for each category of proof failures.

6.1. Method

The proposed method is illustrated by Fig. 13. Suppose that the proof of the annotated program P fails for some 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 generates an NCCE (case (1) in Fig. 13) 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 (case (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, we have two situations. If both $\mathfrak{D}^{\mathrm{NC}}$ and $\mathfrak{D}^{\mathrm{SW}}$ steps were performed on the complete input domain (without using a **`typically`** clause, which reduces the input domain for testing as detailed below), the proof failure is classified as a prover incapacity (case (3), cf. Def. 4). If the user reduced the input domain for testing to a smaller domain (using a **`typically`** clause), the result is still unknown, but it is likely that there is a prover incapacity for the initial program (case (4)). This choice of suggestion is based on the fact that for many programs a counterexample can be found already on a reduced input domain if the reduced domain is representative of the complete input domain. Otherwise, if at least one of $\mathfrak{D}^{\mathrm{NC}}$ and $\mathfrak{D}^{\mathrm{SW}}$ was inconclusive and returned ?, the proof failure remains unclassified (case (5)).

Fig. 14 illustrates the method on several versions, labeled by $R_0, \ldots, R_4$, of the RGF example of Fig. 7 described in Sec. 3.3. For each version, the second column details the lines modified in the intitial program (denoted R_0) of Fig. 7. Columns 3 to 5 report the intermediate results of deductive verification, non-compliance detection ($\mathfrak{D}^{\mathrm{NC}}$) and specification weakness detection ($\mathfrak{D}^{\mathrm{SW}}$). Proof failure is depicted by ? in Column 3, followed by the lines of the unproved annotations. The symbol '–' means that the corresponding detection is not necessary and thus not performed. Column 6 reports the proof failure category and, if any, the generated counterexample V of the final outcome, which also includes the recorded path π_V, the reported failing annotation a, subcontract outputs and a set of too weak subcontracts S (not shown in the figure).

For all versions that include the lemma `max_rgf` in Fig. 7, the deductive verification fails for the lemma because its proof requires inductive reasoning. The symbol ✓ in Column 3 for the version R_0 means that all other verification conditions of R_0 are automatically proved. The other versions (summarized in Fig.14) illustrate the cases (1) to (4) of the method and show how the final outcome can be helpful for the verification engineer.

In version R_1 we try to prove with WP a modified version of the function f of Fig. 7 where the precondition on line 25 is missing. There are three failing annotations: the postcondition of f on line 27, the assertion on line 41 and

the precondition of g for its call on line 43. The $\mathfrak{D}^{\mathrm{NC}}$ step generates the NCCE V_1 (case (1)) and indicates that the postcondition of f on line 27 fails. This is clearly due to `a[0]` being non-zero. That helps the verification engineer to understand and fix the issue.

In version R_2 we suppose that the clause on line 35 has been erroneously written as follows: **`loop assigns`** `i, a[1..n-1]`. The failing annotations are the postconditions of f on line 27 and 28–31, the assertions on lines 41 and 44 and the precondition of g for its call on line 43. $\mathfrak{D}^{\mathrm{NC}}$ times out and reports no counter-example. Since the array length n is not upper-bounded, the path exploration of the $\mathfrak{D}^{\mathrm{NC}}$ step is not complete because of a too large number of paths. The $\mathfrak{D}^{\mathrm{SW}}_{\mathrm{single}}$ step generates the SWCE V_2 for the loop contract (case (2)), with subcontract outputs $\mathrm{nondet}_{a[1]} = 97157$ and $\mathrm{nondet}_i = 0$ after the loop on lines 37-38, leading to a failure of the precondition of g (on line 10) for the call on line 43. It illustrates a subcontract weakness of the contract of this loop. The loop on lines 37–38 still preserves its invariant. With this indication, illustrated by an inconsistent new value of $a[1]$, the verification engineer will strengthen the loop contract.

In version R_3 we suppose that the lemma on lines 4–5 is missing. The proof of the assertion on line 41 of Fig. 7 (stating the absence of overflow on line 42) 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 (5)).

For such situations, we introduce the possibility to reduce the input domain for test generation by using a new ACSL clause **`typically`**. This clause is ignored by the proof. The verification engineer can insert the clause

`typically` `n < 5;`

at the end of line 25 to reduce the array size for test generation (version R'_3 in Fig. 14). 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 unknown, likely prover incapacity (case (4)). If the verification engineer is convinced that the original program behaves similarly to the program with input domain reduced by the **`typically`** clause, this diagnostic gives her more confidence that the proof failure has the same reason on the initial program, here 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.

In version R_4 we suppose that the statement on line 42 is `a[i] = a[i] + 2`. Then the proof fails for the precondition of g called on line 43 and $\mathfrak{D}^{\mathrm{NC}}$ finds the NCCE V_4 leading to `a[0] = 0` and `a[1] = 2` when calling g. These values contradict the precondition `is_rgf(a,n)` on line 10.

For lack of space, we do not detail the outcomes of StaDy for the examples of Sec. 3.1 and 3.2, but in all cases StaDy produces a diagnostic of the proof failure with a counterexample illustrating the issue.

6.2. Suggestions of actions

Based on the possible outcomes of the method (illustrated in Fig. 13), we are able to suggest the most suitable actions 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 debugger [MR11]. In Frama-C, the execution on V can be conveniently explored using Value or PathCrawler [KKP+15]. 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.

For an *unknown, likely prover incapacity* result, $\mathfrak{D}^{\mathrm{NC}}$ and $\mathfrak{D}^{\mathrm{SW}}$ steps were used on a reduced domain and detected no counteraxamples on it. If the reduced domain is representative of the initial program domain, the verification engineer can prioritize the prover incapacity reason of the failure and conduct the actions described above. She should however use this result with care: the method only suggests to seriously consider the prover incapacity reason, but cannot guarantee it. Counterexamples can still exist for a bigger domain if the reduced domain is too small or not representative and the program can have a different behavior on a bigger domain.

		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	2036	462 (/2036)	22.5			1347 (/1574)	85.6			157 (/227)	69.1			95.5		70
Max			36.1	3.4	39.4		100	1.75	1.84		100.0	4.23	42.0	100.0	43.5	
Mean			23.8	2.2	12.5		86.1	1.53	1.55		81.3	2.65	10.3	96.6	2.2	

Fig. 15. Summarized experiments of proof failure diagnosis for mutants with StaDy

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.

7. 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 [KKP+15] for deductive verification and PathCrawler [BDH+09] 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 such 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. (Mutually) recursive functions and recursive logic definitions are currently not supported in StaDy since such functions are not yet supported in PathCrawler, but the proposed method applies for them as well. 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. Other labels (different from **`Pre`**) in annotations and **`goto`** statements are not yet supported. 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`** and **`loop assigns`** clauses are considered only during the $\mathfrak{D}^{\mathrm{SW}}$ phase: we do not try to diagnose a non-compliant (**`loop`**) **`assigns`** clause by $\mathfrak{D}^{\mathrm{NC}}$ because provers give a clear feedback about such a non-compliance, but we do try to identify a too weak (i.e. too permissive) (**`loop`**) **`assigns`** clause since provers would report a failure elsewhere in this case (cf. examples B_5 and B_6 in Sec. 3.2). Inductive predicates, recursive functions and real numbers are not yet supported. Adaptive detection methods are currently supported for $\mathfrak{D}^{\mathrm{NC}}$ and $\mathfrak{D}^{\mathrm{SW}}_{\mathrm{global}}$. Adaptive target subcontract selection for $\mathfrak{D}^{\mathrm{SW}}_{\mathrm{single}}$ is not yet automatic, but can be done manually by a dedicated option (for one subcontract or any set of subcontracts). The investigation of other adaptive strategies for precisely identifying a minimal subset of too weak subcontracts and their implementation are left as future work.

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 $\mathfrak{D}^{\mathrm{NC}}$ step and the $\mathfrak{D}^{\mathrm{SW}}$ step?

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 26 correct annotated programs whose size varies from 35 to 100 lines of annotated C code. Among them, 20 originate from an independent benchmark [BG17], developed and maintained by Fraunhofer FOKUS, independently from the authors of the paper and the developers of Frama-C. 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. 6 and its implementation, we apply StaDy on systematically generated altered versions[8] (or *mutants*) obtained from the 26 correct program examples. 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.) that can lead to proof failures. In this study, we do not mutate the

[8] Available at: `https://github.com/gpetiot/StaDy/tree/master/FAC_2017/benchmark`

precondition of the function under verification, and restrict possible mutations on binary operators to avoid creating absurd expressions, in particular for pointer arithmetic.

Compared to an earlier experiment campaign [PKB+16], in order to evaluate the most recent extensions of the method and its implementation in StaDy, the present experiments are applied on 6 additional programs (those given in the paper as well as several examples with nested loops) and use several additional mutation operators (such as mutations in nested loops, partial omission of parts of a conjunctive annotation, mutations focused on the loop variant) resulting in a much richer set of mutants. We have also performed a manual validation of the results of StaDy on several dozens of selected examples, paying particular attention to programs with nested loops, whose support is a new contribution of this paper.

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 mutants proved by Wp are denoted by ✓, and the mutants for which a counterexample is generated by $\mathfrak{D}^{\mathrm{NC}}$ or $\mathfrak{D}^{\mathrm{SW}}$ are indicated by ✗. The results are summarized in Fig. 15. 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 (in sec.) of the step over classified mutants ($t^{✓}$ or $t^{✗}$) 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 unproven 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 26 considered programs, 2036 mutants have been generated. 462 of them have been proved by Wp. Among the 1574 unproven mutants, $\mathfrak{D}^{\mathrm{NC}}$ has detected a non-compliance induced by the mutation in 1347 mutants (85.6%), leaving 227 unclassified. Among them, $\mathfrak{D}^{\mathrm{SW}}$ has been able to exhibit a counterexample (either an NCCE or an SWCE) for 157 of them (69.1%), finally leaving 70 programs unclassified.

Regarding **RQ1**, StaDy has found a precise reason of the proof failures and produced a counterexample in 1504 of the 1574 unproven mutants, i.e. classifying 95.5% of them. 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 1347 of 1574 unproven mutants (85.6%). $\mathfrak{D}^{\mathrm{SW}}$ diagnosed 157 of the 227 remaining mutants (69.1%) 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 (including one session for the $\mathfrak{D}^{\mathrm{NC}}$ step, and possibly 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 95.5% 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 easily limited (e.g. by the `typically` clause).

Concerning **RQ4**, on the considered programs Wp needs on average 2.2 sec. per mutant (at most 3.4 sec.) to prove a program, and spends 12.5 sec. on average (at most 39.4 sec.) when the proof fails. The total execution time of StaDy is comparable: it needs on average 2.2 sec. per unproven mutant (at most 43.5 sec.). Most of this time is used by test generation performed by PathCrawler. The time required for the specification-to-code translation performed by StaDy is negligible.

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 test 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.

Other Case Studies. One particularly interesting example is the TimSort algorithm (in Java) studied using KeY in [dGRdB+15]. Applying StaDy on a partial C/ACSL implementation of the algorithm containing the erroneous function `mergeCollapse`, we were able to produce a concrete counterexample illustrating that two unproven post-conditions of `mergeCollapse` are due to non-compliances. In another project, StaDy was successfully applied to

generate counterexamples for automatically generated annotated programs that model relational properties over several function calls using self-composition [BKLG+18]. Despite a relative complexity of the generated (self-composed) programs, STADY was able to produce counterexamples for almost all considered programs.

Threats to Validity. As it is often the case in software verification studies, one major threat is related to the representativeness of results, i.e. their *external validity*. In our case, due to the nature of the problem, we are restricted to realistic annotated programs that cannot be generated automatically or extracted from existing databases of unspecified code. Therefore, to reduce this threat, we mainly used programs from an *independent* benchmark [BG17] created in order to illustrate on different examples the usage of the ACSL specification language for deductive verification with FRAMA-C. This benchmark is developed precisely to illustrate various aspects of specification and deductive verification with FRAMA-C. It has been maintained for several years by the Formal Methods group of the Fraunhofer FOKUS institute in Berlin, independently from the authors of the paper and the authors of FRAMA-C. It should be noted that different results can be obtained on a different set of programs.

Scalability of the results is another threat since we do not demonstrate their validity for functions of larger programs. Because of the modular approach of deductive verification, it can be argued that the proposed technique should only be applied on a unit level, separately for each function, since the verification engineer proves a program in this way. Indeed, in the current practice of deductive verification, it does not make sense to analyze proof failures for the whole module or application at the same time. For complex programs, the proposed method can therefore suffer from the same scalability issues as deductive verification in general.

Another scalability concern is related to the usage of structural test generation that can often time out without achieving a full coverage. To address this issue, we have specifically investigated the impact of a partial test coverage on the effectiveness of the method (cf. **RQ3** above) and proposed a convenient way to reduce the input domain (using a `typically` clause, an extension of ACSL).

Other threats can be due to the used measurements, i.e. *construct validity*. To reduce this threat, we used a careful measurement of results (including analysis time for each step and each mutant, their mean and maximal values, separately computed for classified and unclassified proof failures). One concern is producing realistic situations in which the verification engineer can need help in the analysis of proof failures. While the first users of STADY have appreciated its feedback, we have not yet had the opportunity to organize a fair evaluation with a representative group of users. Thus we have performed an extended set of experiments using simulation of errors by mutations as an alternative in the meanwhile. We have chosen a large subset of mutation operators (mutation in the code, mutation in an annotation, deletion of an annotation) that model frequent erroneous situations (incorrect code or annotations, incomplete specification) leading to proof failures. This approach is suitable for non-compliances and subcontract weaknesses, and certainly less suitable for the more subtle prover incapacity cases. The results should be confirmed later by a representative user study.

8. Related Work

Assisting program verification and generation of counterexamples have been addressed in different research work (e.g. [AAPW15, BN04, CDKM11, CTZ11, CEM14, CCFL13, DF12, DHT03, GKW+15, HMM16, KV09, Owr06, PW10]). 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 [HMM16] and implemented for SPARK, a subset of Ada targeted for formal verification. This static analysis does not require to restrict the specification language, as we do, but it is not guaranteed that the provided models are real counterexamples and when they are, it does not allow the user to distinguish non-compliances from specification weaknesses. It is complementary to our combination of static and dynamic analyses and 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 [GKL04].

A two-step verification in [TFNM13] 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. This approach allows to detect specification weaknesses. The difference to our approach is that after a proof failure, they need to be able to prove a program variant (for example, replacing by the unrolled loop a loop contract that may be too weak). In our case, we need to be able to find a counter-example for a different variant (in the case of a loop contract weakness, replacing all other subcontracts except this loop contract by the real code). In the two-step verification approach, 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 can overpass the capacity of the prover, while DSE, focusing on one path at a time, can be expected to be more efficient, but can suffer from a combinatorial explosion of the number of paths. 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 [CLMW16]. 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 [PBJ+14], 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.

Notice that our method assumes that the program (or its part over which the proposed method should be applied to diagnose a proof failure) is annotated in an executable specification language. While this assumption is not a limitation for most C programs, it can make the proposed method unsuitable for object-oriented programs if the properties to be verified use a (non-executable) quantification over all objects.

Proof tree analysis. More precision can be statically obtained by analyzing the unclosed branches of a proof tree. The work [Gla09] 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 [EH07] 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 [GHK+06], its interprocedural and compositional extension in SMASH [GNRT10], the method SANTE [CKGJ12] 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) [CGJ+03]. The work [CTZ11] 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 [CS08]. 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 [MR11] 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, [CESW13] 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 [AD10]. 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. [CMW12] 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 simplify 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 subcon-

tract weaknesses, as well as the definition and detection of single and global subcontract weaknesses. To the best of our knowledge, such a 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 test generation or using counter-models produced by solvers) 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.

9. 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 95% of unproven programs. In particular, the non-compliance detection ($\mathfrak{D}^{\mathrm{NC}}$) was able to diagnose 85% of the unproven programs and the subcontract weakness detection ($\mathfrak{D}^{\mathrm{SW}}$) was able to diagnose 67.4% of the remaining proof failures.

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. 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 fully 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 whole 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.

Future work includes further evaluation of the proposed technique, experiments on a larger class of programs, as well as a better support of E-ACSL constructs and of the adaptive subcontract weakness detection in our implementation. An interesting direction is to study other optimized combinations of $\mathfrak{D}^{\mathrm{NC}}$ and $\mathfrak{D}^{\mathrm{SW}}$ for subsets of annotations and subcontracts. For instance, for unproven properties depending on several subcontracts, it can be useful to precisely identify a minimal set of too weak subcontracts. In this paper we have proposed a strategy that first tries to detect a subcontract weakness for a single subcontract ($\mathfrak{D}^{\mathrm{SW}}_{\mathrm{single}}$), considering them one after another, then for the set of all subcontracts ($\mathfrak{D}^{\mathrm{SW}}_{\mathrm{global}}$). Other strategies could try to identify a minimal set of too weak subcontracts, e.g., by considering smaller and smaller subsets of subcontracts when a global subcontract weakness counterexample is found.

An experimental comparison of STADY with a prover-based inlining technique (like in [TFNM13] or in the KEY tool [BHS07]) is another perspective that will require the implementation of that technique in FRAMA-C. The ongoing effort to support recursive functions in PATHCRAWLER will allow their support in STADY. A rigorous formalization of the proposed categories of proof failures and diagnosis techniques is also a future work direction, that can require to formalize both the deductive verification and test generation tools. Finally, organizing a user study in a real-life setting would be desirable to precisely evaluate the benefits of the proposed techniques for the users.

Acknowledgment. The authors thank the FRAMA-C and PATHCRAWLER teams for providing the tools and support. Special thanks to Jean-Luc Baril, François Bobot, Loïc Correnson, Julien Signoles, Vincent Vajnovszki and Nicky Williams for many fruitful discussions, suggestions and advice. Many thanks to the anonymous referees for their very helpful comments.

References

[AAPW15] Stephan Arlt, Sergio Feo Arenis, Andreas Podelski, and Martin Wehrle. System testing and program verification. In *Softw. Eng. & Management*, volume 239 of *LNI*, pages 71–72. GI, 2015.

[AD10] Ki Yung Ahn and Ewen Denney. Testing first-order logic axioms in program verification. In *TAP*, volume 6143 of *LNCS*, pages 22–37. Springer, 2010.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development; Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[BCF+17] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language*, 2017. `http://frama-c.com/acsl.html`.

[BDH+09] Bernard Botella, Mickaël Delahaye, Stéphane Hong Tuan Ha, Nikolai Kosmatov, Patricia Mouy, Muriel Roger, and Nicky Williams. Automating structural testing of C programs: Experience with PathCrawler. In *AST*, pages 70–78. IEEE Computer Society, 2009.

[BG17] Jochen Burghardt and Jens Gerlach. ACSL by example, 2017. Published online: `https://github.com/fraunhoferfokus/acsl-by-example`.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer, Heidelberg, 2007.

[BKLG+18] Lionel Blatter, Nikolai Kosmatov, Pascale Le Gall, Virgile Prevosto, and Guillaume Petiot. Static and dynamic verification of relational properties on self-composed C code. In *TAP*, LNCS. Springer, 2018. To appear.

[BN04] Stefan Berghofer and Tobias Nipkow. Random testing in Isabelle/HOL. In *SEFM*, pages 230–239. IEEE Computer Society, 2004.

[CCFL13] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. Automatic inference of necessary preconditions. In *VMCAI*, volume 7737 of *LNCS*, pages 128–148. Springer, 2013.

[CDKM11] Harsh Raju Chamarthi, Peter C. Dillinger, Matt Kaufmann, and Panagiotis Manolios. Integrating testing and interactive theorem proving. In *ACL2*, volume 70 of *EPTCS*, pages 4–19, 2011.

[CEM14] Maria Christakis, Patrick Emmisberger, and Peter Müller. Dynamic test generation with static fields and initializers. In *RV*, volume 8734 of *LNCS*, pages 269–284. Springer, 2014.

[CESW13] Jürgen Christ, Evren Ermis, Martin Schäf, and Thomas Wies. Flow-sensitive fault localization. In *VMCAI*, volume 7737 of *LNCS*, pages 189–208. Springer, 2013.

[CGJ+03] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[CKGJ12] Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. Program slicing enhances a verification technique combining static and dynamic analysis. In *SAC*, pages 1284–1291. ACM, 2012.

[CLMW16] Maria Christakis, K. Rustan M. Leino, Peter Müller, and Valentin Wüstholz. Integrated environment for diagnosing verification errors. In *TACAS*, volume 9636 of *LNCS*, pages 424–441. Springer, 2016.

[CMW12] Maria Christakis, Peter Müller, and Valentin Wüstholz. Collaborative verification and testing with explicit assumptions. In *FM*, volume 7436 of *LNCS*, pages 132–146. Springer, 2012.

[Coq18] Coq Development Team. *The Coq Proof Assistant Reference Manual*, 2018. `http://coq.inria.fr/`.

[CS08] Koen Claessen and Hans Svensson. Finding counter examples in induction proofs. In *TAP*, volume 4966 of *LNCS*, pages 48–65. Springer, 2008.

[CTZ11] Tsong Yueh Chen, T. H. Tse, and Zhiquan Zhou. Semi-proving: An integrated method for program proving, testing, and debugging. *IEEE Transactions on Software Engineering*, 37(1):109–125, 2011.

[DF12] Rayna Dimitrova and Bernd Finkbeiner. Counterexample-guided synthesis of observation predicates. In *FORMATS*, volume 7595 of *LNCS*, pages 107–122. Springer, 2012.

[dGRdB+15] Stijn de Gouw, Jurriaan Rot, Frank S. de Boer, Richard Bubel, and Reiner Hähnle. OpenJDK's Java.utils.Collection.sort() is broken: The good, the bad and the worst case. In *CAV*, volume 9206 of *LNCS*, pages 273–289. Springer, 2015.

[DHT03] Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. Combining testing and proving in dependent type theory. In *TPHOLs*, volume 2758 of *LNCS*, pages 188–203. Springer, 2003.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. In: Series in Automatic Computation. Prentice Hall, Englewood Cliffs, 1976.

[DKS13] Mickaël Delahaye, Nikolai Kosmatov, and Julien Signoles. Common specification language for static and dynamic analysis of C programs. In *SAC*, pages 1230–1235. ACM, 2013.

[EH07] Christian Engel and Reiner Hähnle. Generating unit tests from formal proofs. In *TAP*, volume 4454 of *LNCS*, pages 169–188. Springer, 2007.

[GGP15] Richard Genestier, Alain Giorgetti, and Guillaume Petiot. Sequential generation of structured arrays and its deductive verification. In *TAP*, volume 9154 of *LNCS*, pages 109–128. Springer, 2015.

[GHK+06] Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. SYNERGY: A new algorithm for property checking. In *FSE*, pages 117–127. ACM, 2006.

[GKL04] Alex Groce, Daniel Kroening, and Flavio Lerda. Understanding counterexamples with explain. In *CAV*, volume 3114 of *LNCS*, pages 453–456. Springer, 2004.

[GKW+15] Shengjian Guo, Markus Kusano, Chao Wang, Zijiang Yang, and Aarti Gupta. Assertion guided symbolic execution of multithreaded programs. In *ESEC/FSE*, pages 854–865. ACM, 2015.

[Gla09] Christoph Gladisch. Could we have chosen a better loop invariant or method contract? In *TAP*, volume 5668 of *LNCS*, pages 74–89. Springer, 2009.

[GNRT10] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai Deep Tetali. Compositional may-must program analysis: unleashing the power of alternation. In *POPL*, pages 43–56. ACM, 2010.

[HMM16] David Hauzar, Claude Marché, and Yannick Moy. Counterexamples from proof failures in SPARK. In *SEFM*, volume 9763 of *LNCS*, pages 215–233. Springer, 2016.

[JKS15] Arvid Jakobsson, Nikolai Kosmatov, and Julien Signoles. Fast as a shadow, expressive as a tree: hybrid memory monitoring for C. In *SAC*, pages 1765–1772. ACM, 2015.

[KKP+15] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. *Formal Asp. Comput.*, 27(3):573–609, 2015.

[Kos15] Nikolai Kosmatov. Online version of PathCrawler., 2010–2015. http://pathcrawler-online.com/.

[KPS13] Nikolai Kosmatov, Guillaume Petiot, and Julien Signoles. An optimized memory monitoring for runtime assertion checking of C programs. In *RV*, volume 8174 of *LNCS*, pages 328–333. Springer, 2013.

[KV09] Laura Kovács and Andrei Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *FASE*, volume 5503 of *LNCS*, pages 470–485. Springer, 2009.

[MR11] Peter Müller and Joseph N. Ruskiewicz. Using debuggers to understand failed verification attempts. In *FM*, volume 6664 of *LNCS*, pages 73–87. Springer, 2011.

[MV13] Toufik Mansour and Vincent Vajnovszki. Efficient generation of restricted growth words. *Information Processing Letters*, 113(17):613 – 616, 2013.

[Owr06] Sam Owre. Random testing in PVS. In *Workshop on Automated Formal Methods (AFM)*, 2006.

[PBJ+14] Guillaume Petiot, Bernard Botella, Jacques Julliand, Nikolai Kosmatov, and Julien Signoles. Instrumentation of annotated C programs for test generation. In *SCAM*, pages 105–114. IEEE Computer Society, 2014.

[PKB+16] Guillaume Petiot, Nikolai Kosmatov, Bernard Botella, Alain Giorgetti, and Jacques Julliand. Your proof fails? Testing helps to find the reason. In *TAP*, volume 9762 of *LNCS*, pages 130–150. Springer, 2016.

[PKGJ14] Guillaume Petiot, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. How test generation helps software specification and deductive verification in Frama-C. In *TAP*, volume 8570 of *LNCS*, pages 53–60. Springer, 2014.

[PW10] Andreas Podelski and Thomas Wies. Counterexample-guided focus. In *POPL*, pages 249–260. ACM, 2010.

[Sig12] Julien Signoles. *E-ACSL: Executable ANSI/ISO C Specification Language*, 2012. `http://frama-c.com/download/e-acsl/e-acsl.pdf`.

[TFNM13] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. Program checking with less hassle. In *VSTTE*, volume 8164 of *LNCS*, pages 149–169. Springer, 2013.

[WMMR05] Nicki Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. PathCrawler: automatic generation of path tests by combining static and dynamic analysis. In *EDCC*, volume 3463 of *LNCS*, pages 281–292. Springer, 2005.