Verification Coverage for Combining Test and Proof

Viet Hoang Le[1], Loïc Correnson[2], Julien Signoles[2], and Virginie Wiels[3]

[1] CEA MiPy,
viethoang.le@cea.fr
[2] CEA LIST, Software Reliability and Security Laboratory
loic.correnson@cea.fr,julien.signoles@cea.fr
[3] ONERA,
Virginie.Wiels@onera.fr

Abstract. The V&V practices of safety-critical industries (e.g. avionics) are currently based on either unit testing or unit proof to verify that a function satisfies its low-level requirements in order to be compliant with the highest certification levels [26] (e.g. DO-178C level A for avionic software). In this context, the verification engineer must assess sufficient coverage of both code (structural coverage) and specification (functional coverage). However, there is no shared method for test and proof to measure structural coverage. In practice, this prevents the verification engineer from combining test and automatic proof to verify low-level requirements of a common piece of code in order to mitigate the verification cost. This paper fills this gap between test and proof by introducing a new notion of verification coverage based on mutation coverage. It subsumes functional coverage and structural coverage for both unit testing and unit proof. Consequently, it allows the verification engineer to mix test tools and automatic provers in the verification process for the sake of reducing verification cost, in the sense that the more automation is used during the verification, the less resource is spent to verify the program.

Keywords: coverage criteria, combining test and proof

1 Introduction

In software development of critical systems, the code verification step is crucial since it prevents unexpected behaviors to arise during program execution. In particular, the verification engineers must ensure that the program satisfies its specifications. For this purpose, testing is the most commonly used technique to reach the expected level of confidence. It consists in running the tested program on some input data and comparing the expected results according to the given *oracles derived from the program specifications* [28,32]. Program proof is another suitable verification technique [9]. It consists in statically verifying the program with respect to its specifications for all possible executions by means of logical reasoning [17–19].

Whatever the underlying technique, one must ensure that enough verification has been performed. This part of the verification process is usually performed by measuring *coverage*, e.g. test coverage. For this purpose, the DO-178 standard for avionic software [29] introduces two processes: functional analysis and

structural analysis. The former guarantees that all the program specifications are verified, while the latter guarantees that every path and every piece of code in the program is reached and contributes to producing the expected results.

Functional analysis is independent from the verification technique. The verification of a function achieves full functional coverage as soon as one succeeds to *test* or *prove* all the specifications [12,13], thus all program specifications are verified. Structural analysis depends on the underlying verification technique. For testing, it relies on various structural coverage criteria (statement coverage, branch coverage, MC/DC coverage, *etc.*) [2] to ensure that executing a test suite covers each path and/or piece of code in the right way. For program proof, structural analysis may be performed by fulfilling different objectives. For instance, in DO-333 [30], one must ensure four objectives [10,26]: assumption coverage (each proof assumption is checked); completeness (the specifications specify outputs for every input condition and, conversely, input conditions for every output), data-flow (all the dependencies between inputs and outputs are found) and extraneous code (every piece of code depends on at least one specification). By guaranteeing their objectives, we ensure that neither path nor piece of code contributes to producing a result unexpected in any specification existed.

This current workflow has a major limitation: while test coverage on the one hand and proof coverage on the other hand are well-known concepts, it is not possible to use test coverage for proof and conversely. Test coverage can only be used when testing the entire program with the oracle derived from program specification, while proof coverage is only defined when all the program specifications are proved. Consequently, for a particular piece of code, it is not possible to test some specifications while proving the others, because there is no way to define the coverage of the combined verification.

Nowadays, during a proof campaign, the engineer relies on automated theorem provers in the hope to prove all program specifications and to ensure all objectives defined in DO-333 [9]. Usually most proof obligations are automatically discharged, but sometimes a few of them might not. In such a case, on account of the above-mentioned limitation, the engineer may either manually prove them and verify coverage through DO-333, or discard the proof campaign and rely on testing as defined in DO-178. In both cases, the verification process is much more expensive because it requires a lot of additional manual work.

This paper presents a new notion of verification coverage which aims at reducing the verification cost by keeping the existing proofs and adding only the necessary tests to complete the verification process. It subsumes functional coverage and structural coverage for both test and proof. It also relies on a *new notion of witness* that formalizes a verification activity and allows the verification engineer to sum up which verification technique has been used to validate that a particular piece of code contributes to enforcing some specifications. Furthermore, *we introduce a methodology and a companion algorithm that allow the verification engineer to check whether a verification campaign is complete with respect to this coverage.*

The remainder of our paper is organized as follows. First, Section 2 discusses related work. Then, Section 3 introduces a running example. Section 4 presents the general idea of our work, which is then formalized in Section 5. Section 6 explains how to automatize as much as possible a verification campaign with respect to our new verification coverage. Finally, Section 7 exemplifies our process on the running example.

2 Related Work and Discussion

As previously mentioned, coverage is a major obstacle for combining test and proof. Typically, it prevents us from complete a partial proof campaign by means of testing. Several existing works already study different kinds of combinations of testing and formal verification techniques [3, 8, 21, 33], but they do not deal with the coverage issue. According to Bishop et al. [8], these combinations can be divided into four levels described below. We aim at defining a notion of coverage for the last two ones:

- Level 1 (Separately): test and proof are applied separately to verify different parts of the system;
- Level 2 (Assistance): proof supports test or conversely;
- Level 3 (Friendship): proof contributes to the automated generation of tests and their results are combined;
- Level 4 (Unification): test and proof are fully combined.

Our notion of coverage is based on existing notions of *label coverage* and *mutation coverage*. Label coverage [5, 6] relies on *labels*. Labels are logical formulae attached at program points. They can encode most structural coverage criteria. Originally, they were used to automatically generate test suites that satisfy a given structural criterion. Then, in [4], their usage has been extended in order to detect unfeasible labels when combining program proof and abstract interpretation. However, in these works, formal methods were only used for supporting testing (level 2 above). In particular, structural coverage was still limited to testing and cannot be applied to program proof. More recently, labels have been extended to hyperlabels [23, 24] to enlarge the variety of criteria that can be represented. We believe that our technique can be extended to hyperlabels, but we leave this to future work.

Our work also relies on mutants (in the sense of [16,25]) to check our coverage metrics. A mutant m of a program p is a program obtained by slightly modifying p. Mutation testing consists in verifying that the outputs for p and m differ. In that case, one says that the mutant m is killed. Mutation coverage is defined by the number of killed mutants. Our work extends the usage of mutants to program proof. Many different mutation schemes exist [1, 27] for various programming languages. Our work relies on statement deletion to create mutants inspired by Delamaro et al. [15]. But we also use other mutation operators like expression modification when it is more beneficial than statement deletion. Our proposed methodology is indeed independent from the underlying mutation schemes, but they may lead to different results: the choice of the best mutation scheme for a particular use case is let to the end-user. Mutation has also been explored

to propose a notion of coverage for model checking [11]. The model is mutated and the model part is considered covered if the mutant survives. Their notion of coverage does not apply to testing. However, it inspired our more general notion of verification coverage.

3 Running Example

This section introduces a running example that illustrates current issues when combining test and proof in order to verify a C function in the context of a DO-178 certification.

Fig. 1 presents a scheme of a function `transform` The complete C code is omitted for the sake of brevity This function is typical of reactive embedded software: it computes an output signal from an input by a linear regression depicted in Fig. 2. Here, all the values are bounded by an upper bound $s_{\max}$ and a lower bound $s_{\min}$. In C program, these bounds are global variables. The linear regression also depends on parameters x_1, x_2, y_1 and y_2 that must satisfy the following consistency constraints:

$$\mathrm{s_{min}} \leq \mathrm{x_1} < \mathrm{x_2} \leq \mathrm{s_{max}} \quad \text{and} \quad \mathrm{s_{min}} \leq \mathrm{y_1} \leq \mathrm{s_{max}} \quad \text{and} \quad \mathrm{s_{min}} \leq \mathrm{y_2} \leq \mathrm{s_{max}}.$$

Signals are implemented by a structure named `Signal`. Each `Signal` contains a floating-point value in interval [smin,smax], and an error flag indicating whether the constraints are satisfied. Parameters x_i and y_i $(i = 1, 2)$ are passed to the C function via another structure named `Block`. Furthermore, the function returns 0 when the constraints are satisfied and an error code otherwise.

```
1 int transform (Block *p, Signal *input, Signal *output){
2  // 1. verify block validity
3  // 2. modify the value of the output signal w.r.t. the input signal
4  // 3. set the error flag
5 }
```

Fig. 1: Scheme of function `transform`.

The verification objectives for this function are the informal requirements defined in the spirit of DO-178. They may be formalized and split in four groups of specification:

ERR 7 specifications defining the error code in case of invalid parameters.
OK 1 specification formalizing the result when the constraints are satisfied.
VALUE 3 specifications defining the expected value of the output signal.
VALID 1 specification controlling the error flag of the output signal.

The function has been implemented in C and formally specified in the ACSL specification language [7]. Then, we tried to verify this code with Frama-C [20]. This framework has already been (successfully employed) for experimenting combinations of test and proof of C programs [22]. Here, E-ACSL [31], the runtime verification plug-in of Frama-C, is first run to check that the C code satisfies its formal ACSL specifications. For this purpose, we need to manually define at least 10 test cases, 7 among them for testing all situations when the constraints

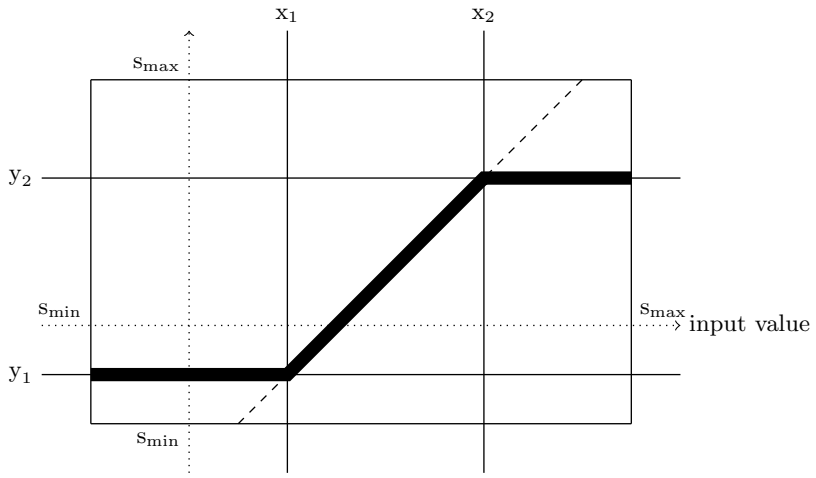

Fig. 2: Linear regression for transformation.

are invalid and 3 other used to test the expected value of output signal when the constraints are valid, to cover all the possible cases (or even more, depending on the chosen structural coverage criterion). Then, WP, the Frama-C plug-in for deductive verification, was used to (automatically) prove this function. Yet, one specification in group VALUE remained unproved because of floating-point computations in the code. Consequently, neither test nor proof alone allows us to complete the verification process with reasonable resource spent.

Careful code review allows us to argue that only a limited piece of code contributed to the unproved properties. Therefore, the intuitive goal of our approach is to complete the obtained proofs by adding only a few test cases that cover the remaining code fragments and the unproved specifications. We also aim at defining a notion of verification coverage for this use case.

4 Verification Campaign

This section provides additional details about the new kind of verification campaigns combining test and proof that we propose.

4.1 Labeled Mutant

Our technique is independent from a particular structural coverage criterion by relying on a notion of *labels* which can be used to encode most structural coverage criteria [6]. Each label is a property associated to a program point and divides the code source in two pieces that the below one is the corresponding code fragment of label. A label is satisfied when a verification activity demonstrates that there is an execution passing through this label and satisfying its associated property. A structural coverage criterion holds if and only if all the labels for this criterion are satisfied.

In order to gain additional results during a verification campaign, each label is associated to a mutant that modifies the corresponding code fragment. Such a mutant is named *labeled mutant*. Any method of mutation is possible whenever it fulfills the following condition: *the mutant shall only modify the executions that pass the label*.

In this paper, two kinds of mutation operators are used to generate labeled mutants: the *replace* and *erase* operators. The former replaces a statement by another one, while the latter removes it. Their formal definitions (omitted here) inspired by [1] and fulfill the above-mentioned condition. Mainly, they consist of introducing a conditional statement guarded by the labeled property. Fig. 3 provides an example of such mutations for a small piece of code from our running example.

```
....
if(p->y2 < smin){
 // label Lreturn6:
 // label condition: true;
 // labeled mutant: replace(0) ;

 // initial statement: return (-6);

 // mutant created before simplify:
 // if (1)
 //  then {return 0;}
 //  else {return -6;}

 // mutant after simplify:
 return 1 ? 0 : -6;
}
....
```

(a) Replace

```
....
if(x->v >= p->x2){
 // label Lstmt2:
 // label condition: true;
 // labeled mutant: erase;

 // initial statement: y->v = p->y2;

 // mutant created before simplify:
 // if (1)
 //  then {}
 //  else {y->v = p->y2;}

 // mutant after simplify:
 if (! (1)) {y->v = p->y2;}
}
....
```

(b) Erase

Fig. 3: Example of labeled mutant.

Mutants are actually created for trying to kill them. Indeed, killing a mutant means that the corresponding statement in the initial program was meaningful for the checked criterion. Our method consists in finding out specifications that are validated in the initial program but not any more in a mutant (i.e. the mutant is killed by that specification) through various verification activities (inspired by the definition of mutation testing [16, 25]). It allows us to conclude that the mutated piece of code, denoted by a label, has a strong connection with these specifications. Full coverage is therefore achieved by establishing such a strong connection for each specification and piece of code corresponding to each label.

4.2 Verification Campaign

Our verification campaign assumes the existence of the source code, its specifications and labels encoding a particular structural coverage criterion. These labels may actually be automatically generated [5]. A verification campaign consists in a sequence of *verification activities* (either test or proof). Each of them provides two pieces of information: *verification information* indicating which

specifications S is validated and *coverage information* indicating which labels L are covered. These pieces of information are grouped together through the notion of witness, as informally explained below.

Proof Activities. Automated deductive verification may provide *verification information* about the validity or the invalidity of each specification S. In our context, invalidity of S means validity of its negation $\neg S$[4]. If S is validated in the initial code, it means that no execution passing any label contradicts the specification. This information is recorded through a *proof witness* between this specification and every label. However, if S is validated / invalidated in a labeled mutant (of label L), it means that S and L have no correlation at all (i.e. label L is associated to a piece of code that no matter how we modify this piece, specification S is still proved) / are strongly connected.

Consider our running example in which we successfully prove a specification of ERR named `error6`. Therefore, for each label L, one proof witness is recorded between `error6` and L. Furthermore, the specification is still proved in labeled mutant of label `Lstmt2` (Fig. 3b) but it is invalidated in labeled mutants of label `Lreturn6` (Fig. 3a). Thus, we conclude that there is no correlation between specification `error6` and label `Lstmt2`, while this specification and label `Lreturn6` are strongly connected.

Test Activities. Testing a specification S requires to manually define test cases. After defining them, testing the specification in source code and in mutants is an automatic process that provides us with *test witnesses* between labels and specifications. These witnesses are more precise than proof witnesses, since they assess that the corresponding label is reached.

In our running example, a particular test case activates the specification `error6` (its assumptions are fulfilled). It also covers several statements, one of them being the statement (a code fragment) associated to label `Lreturn6`. Yet, the statement associated to label `Lstmt2` is uncovered. Therefore, there is a test witness for the pair (`error6`, `Lreturn6`), but none for the pair (`error6`, `Lstmt2`).

Error Detection. If a specification S is invalidated in the original source code as a results of a verification activity, we get an error which is recorded as *error witness* between S and all the existing labels.

Coverage Analysis. A specification S and a label L are strongly connected if and only if there is one verification witness that validates the pair (S, L) in the original source code and one witness that invalidates this pair in the corresponding mutant. All the strong connections between specifications and labels are stored in a *coverage matrix*. Its columns represent specifications, while its rows represent labels. Fulfilling our verification coverage means positively filling this matrix. Indeed, it means that all the specifications S are verified (ensuring functional coverage), while all labels are covered (ensuring structural coverage).

The matrix may be quite large. Thus, analyzing it may be painful. Consequently, we provide a way to *consolidate* it by merging all the cells of the

[4] Usually, one only tries to prove S. Here, one tries to prove both S and $\neg S$ in order to get additional coverage information as explained later.

same column or row into a single one whenever possible. It helps the verification engineer in the analysis.

5 Formalization of Verification Witnesses

This section formalizes the underlying concepts of a verification activity introduced in the previous section, in particular *verification witnesses*.

5.1 Basic Concepts

Execution Given a program P with $\mathcal{L}$ list of program point and $\mathcal{M}$ list of possible memory state for P, $\overrightarrow{x}$ denotes an input vector for P. An execution $P(\overrightarrow{x})$ of a program P on some input datum $\overrightarrow{x} = x_1, ..., x_n$ is a (finite or infinite) sequence $(l_i, m_i)_{0 \leq i}$ of (program) states. Each state is a pair composed of a program point $l \in \mathcal{L}$ and a memory state $m \in \mathcal{M}$. A memory state m at a point l of an execution $P(\overrightarrow{x})$ denotes the association of a value to each variable when $P(\overrightarrow{x})$ reaches l. For a particular execution $P(\overrightarrow{x}) = (l_i, m_i)_{0 \leq i}$, a sub-sequence of states between two program points l_i and l_j $(i \leq j)$ is denoted by $(l_i, m_i) \hookrightarrow_{P(\overrightarrow{x})} (l_j, m_j)$.

Specification & Functional Coverage A program requirement R is formalized by a collection of specifications, denoted by $R \triangleq \{S_1, \ldots, S_n\}$. Functional coverage is achieved once all specifications in R are verified. A specification S in our framework is an implication $H \Rightarrow C$ which consists in a hypothesis H and a conclusion C. The hypothesis H is a pair (l, h) where l is a program point and $h \in \mathcal{P}(\mathcal{M})$ denotes a property over memory states. Similarly, the conclusion C is a triplet (l, l', r) where l and l' denote two program points and r is a relation between two memory states, i.e. a subset of $\mathcal{P}(\mathcal{M} \times \mathcal{M})$. For a specification $S \triangleq H \Rightarrow C$, the first program point l of C must be the same as the one of the hypothesis H.

Given an execution $P(\overrightarrow{x})$ of program P and a specification $S = H \Rightarrow C$, $P(\overrightarrow{x}) \rightsquigarrow (l, h)$ (resp. $P(\overrightarrow{x}) \rightsquigarrow (l, l', r)$) denotes that P *passes through* the hypothesis $H = (l, h)$ (resp. conclusion $C = (l, l', r)$). However, there is a different when $P(\overrightarrow{x})$ passes through the hypothesis H and $P(\overrightarrow{x})$ passes through the conclusion C. In the case of $P(\overrightarrow{x}) \rightsquigarrow (l, h)$, it means that this execution reaches l and the corresponding memory state satisfies h. However, $P(\overrightarrow{x}) \rightsquigarrow (l, l', r)$ means that each time l' is reached from l (i.e. each sequence $(l, m) \hookrightarrow_{P(\overrightarrow{x})} (l', m')$), then its corresponding memory state satisfies r (i.e. $(m, m') \in r$). More formally, passing through an hypothesis (resp. a conclusion) is defined as follows:

$$P(\overrightarrow{x}) \rightsquigarrow (l, h) \triangleq \exists (l_i, m_i) \in P(\overrightarrow{x}), l_i \equiv l \wedge m_i \in h;$$
$$P(\overrightarrow{x}) \rightsquigarrow (l, l', r) \triangleq \forall (m, m') \in \mathcal{M}^2 \text{ s.t. } (l, m) \hookrightarrow_{P(\overrightarrow{x})} (l', m'), (m, m') \in r.$$

Label & Structural Coverage A label exactly matches the notion of hypothesis introduced above: it is a pair of a program point l and a condition h that memory states must satisfy at l. Therefore, the notion of passing through an hypothesis is extended to a label.

Extending a label $\{l, h\}$ to a mutant M at label l (that is, the original program mutated at program point l) defines a new label $\{l, h, M\}$, named *label with mutant. From this point, all* labels *in the following are considered* labeled mutants. Formally, given a program P, a label $\{l, h, M\}$, and an input datum $\vec{x}$, $P(\vec{x})$ and $M(\vec{x})$ shall contain the same series of program states if and only if $P(\vec{x})$ does not pass through $\{l, h, M\}$, since the mutant shall modify the execution trace of the original program.

Verification Activity In our context, a verification activity is either a unit proof or a unit test. Both of them tries to provide evidence that each program execution satisfies each program specification. However, both processes are not performed in the same way in practice. It results in difference when measuring coverage.

Consider a program P, an hypothesis $H = (l, h)$, a conclusion $C = (l, l', r)$ and a specification $S = H \Rightarrow C$. Unit test checks that the specification is satisfied in the program by observing some program executions. Each observation is a test case. A successful test case with input datum $\vec{x}$ validates both the hypothesis H and the conclusion C. It provides us the evidence $P(\vec{x}) \rightsquigarrow H \wedge C$. A test is successful whenever all its test cases are themselves successful.

For unit proof, verifying a specification ensures that no execution violates the specification, which means either the verification passes through both H and C, or through the negation of H. Therefore, the evidence for a successful unit proof is $\forall \vec{x},$ either $(P(\vec{x}) \rightsquigarrow H \wedge C)$ or $(P(\vec{x}) \rightsquigarrow \neg H)$. Thus, even if such an evidence ensures that every possible execution satisfies the specification, it does not ensure that the goal C is actually satisfied since the hypothesis H could be invalidated. (contrary to evidence provided by unit testing).

5.2 Verification Witness about the Initial Program

A witness results from a verification activity. It consists of two pieces of information: a *verification technique* (either proof or test) and a *verdict* indicating which specification is satisfied and by which means. We introduce one kind of witness by verification technique:

- A test witness (denoted by $\mathtt{T}$) represents the existence of some test evidence, passing through the label L:

$$\mathtt{T}(S, L, \vec{x}) \quad \triangleq \quad P(\vec{x}) \rightsquigarrow H \wedge C \wedge L.$$

- A proof witness (denoted by $\mathtt{P}$) indicates the existence of a proof for some specification S. It ensures that S is satisfied for every execution of program P:

$$\mathtt{P}(S) \quad \triangleq \quad \forall \vec{x}, \text{ either } (P(\vec{x}) \rightsquigarrow H \wedge C) \text{ or } (P(\vec{x}) \rightsquigarrow \neg H).$$

Another witness (less considered here than the other ones) is the error witness. It comes when the verification technique finds an error in the code.

- Error Witness (denoted by $\mathtt{ER}$) indicates the existence of an error during the verification of some specification $S = H \Rightarrow C$:

$$\mathtt{ER}(S) \quad \triangleq \quad \exists \vec{x}, (P(\vec{x}) \rightsquigarrow H \wedge \neg C).$$

Hence, witnesses provide us with a formal evidence of all activities performed during our verification campaign.

5.3 Verification Witness about a Mutant

As already explained, our methodology requires a verification of mutants: combining the result of verification in source code and in labeled mutant allows us to deduce relationships between specifications and pieces of code (denoted by labels). In order to separate witnesses provided by verification of a labeled mutant M from the ones coming from the verification of the original code, we introduced additional types of witnesses. Even if the verification of a specification $S = H \Rightarrow C$ for a mutant may produce many different results, only the following two cases are useful in our contexts:

- The specification $H \Rightarrow C$ is satisfied in the mutant,
- The *opposite specification* of S, $H \Rightarrow \neg C$, is satisfied by the mutant.

Each result can be obtained by any verification activity (either test or proof). Hence, the verification of a specification S for a mutant M of a label L can lead to one of the four following witnesses:

- *Witness* `SP` *of proof for the labeled mutant:* when the specification S is *proved* on the mutant M of label L:

$$\texttt{SP}(S, L) \quad \triangleq \quad \forall \vec{x}, \text{ either } (M(\vec{x}) \rightsquigarrow H \wedge C) \text{ or } (M(\vec{x}) \rightsquigarrow \neg H).$$

- *Witness* `ST` *of test for the labeled mutant:* when the specification S is *tested* on the mutant M of label L with input datum $\vec{x}$:

$$\texttt{ST}(S, L, \vec{x}) \quad \triangleq \quad M(\vec{x}) \rightsquigarrow H \wedge C \wedge L.$$

- *Witness* `OP` *of proof for the opposite specification on the labeled mutant:* when the *opposite* specification $H \Rightarrow \neg C$ is *proved* on the mutant M of label L:

$$\texttt{OP}(S, L) \quad \triangleq \quad \forall \vec{x}, \text{ either } (M(\vec{x}) \rightsquigarrow H \wedge \neg C) \text{ or } (M(\vec{x}) \rightsquigarrow \neg H).$$

- *Witness* `OT` *of test for for the opposite specification on the labeled mutant:* when the *opposite* specification $H \Rightarrow \neg C$ is *tested* by an execution $M(\vec{x})$ that passes the label L:

$$\texttt{OT}(S, L, \vec{x}) \quad \triangleq \quad M(\vec{x}) \rightsquigarrow H \wedge \neg C \wedge L.$$

5.4 Witness Precedence

Since formal proof assesses properties for all input data, while testing only checks a sample of data, there is a natural precedence of proof witnesses (`P`, `SP`, `OP`) over test ones (`T`, `ST`, `OT`). Other combinations of witnesses are also comparable.

In particular, it is possible to have two contradicting witnesses: one witness shows that a specification S is satisfied by the mutant M, while the other one shows that S is violated by M. It could arrive in two different cases:

– Both witnesses are proof witnesses $\texttt{SP}(S, L)$ and $\texttt{OP}(S, L)$. From those witnesses, we know that the specification S and its opposite were proved. This situation only occurs when no execution satisfies the hypothesis of S (i.e. the specification is completely useless in the mutant). In this case, $\texttt{OP}(S, L)$ (which is required to claim that S and L are strongly connected) brings harm to the coverage measure. Therefore, we only keep witness SP and reject OP.
– Both witnesses are test witnesses $\texttt{ST}(S, L, \vec{x})$ and $\texttt{OT}(S, L, \vec{y})$. This situation can only occur when $\vec{x} \neq \vec{y}$. In this case, we only keep witness OT because it leads to better coverage.

Hence, we can define a partial ordering over witnesses, illustrated by the diagram below. It shows that SP is greater than any witness over labeled mutants.

$$\begin{array}{ccc} \texttt{P} & \texttt{SP} & \rightarrow \texttt{OP} \\ \downarrow & \downarrow & \downarrow \\ \texttt{T} & \texttt{ST} & \leftarrow \texttt{OT} \end{array}$$

6 Formalization of Verification Campaigns

6.1 Coverage Matrix

A coverage matrix allows an engineer to check the advancement of a verification campaign. Each column of such a matrix represents a specification, while each row represents a label. The matrix records the results of a (possibly still ongoing) verification campaign. Table 1 depicts the possible marks stored in the matrix cells with respect to verification witnesses of specification S that have been computed for the original program P and the mutant M associated to the label L.

Spec. on P	Spec on M	Witness	Verification information	Coverage information
no witness	no witness		(empty)	
error	any witness	$\texttt{ER}(S)$	✗	
no witness	proved & opp. proved	$\texttt{SP}(S, L)$	?	
		$\texttt{OP}(S, L)$		
proved	no witness & tested	$\texttt{P}(S)$	P	?
		$\texttt{P}(S) \wedge (\exists \vec{x}.\texttt{ST}(S, L, \vec{x}))$		
tested	tested	$\exists \vec{x}.\ (\ \texttt{T}(S, L, \vec{x}) \wedge \texttt{ST}(S, L, \vec{x})\ )$	T	
proved	proved	$\texttt{P}(S) \wedge \texttt{SP}(S, L)$	P	–
tested	proved	$(\exists \vec{x}.\texttt{T}(S, L, \vec{x})) \wedge \texttt{SP}(S, L)$	T	
proved	opp. proved & opp. tested	$\texttt{P}(S) \wedge \texttt{OP}(S, L)$	P	✓
		$\texttt{P}(S) \wedge (\exists \vec{x}.\texttt{OT}(S, L, \vec{x}))$		
tested	opp. proved & opp. tested	$(\exists \vec{x}.\texttt{T}(S, L, \vec{x})) \wedge \texttt{OP}(S, L)$	T	
		$\exists \vec{x}.(\ \texttt{T}(S, L, \vec{x}) \wedge \texttt{OT}(S, L, \vec{x})\ )$		

Table 1: Marks recording verification and coverage information.

Marks in cells encode verification and coverage information for a pair of a specification S and a label L. Fig. 4 provides a synthetic view of the possible

connections between labels and specifications. An empty cell means that no verification activity occured. Otherwise, each line contains either one or two marks. When one mark is used, mark ✗ means that S is invalidated in the original program (thus no coverage information is required), while mark ? means that we only have coverage information. When two marks are used, the first mark (either T or P) denotes verification information (either test or proof). The second mark denotes coverage information. Here, mark ? means that S is validated while there is no information about the connection between S and L. Mark $-$ means no correlation between S and L, while S is validated. Mark ✓ means validation of S in source code and invalidation of S in the mutant of L, so S and L are strongly connected.

	Specification
Label	(empty) ✗ ? $P?$ $T?$ $P-$ $T-$ $P✓$ $T✓$

Fig. 4: Possible marks in coverage matrix cells.

6.2 Consolidated Coverage Matrix

A coverage matrix is usually quite large. For instance, our running example have 11 specifications and 12 labels, with made a total 132 cells in the coverage matrix. Consequently, it is not easy for a verification engineer to handle it in a useful way. For solving this issue, we provide a way to consolidate it by gathering columns and rows. This consolidation results in adding one column named *specification consolidation* and one row named *label consolidation* as shown in Fig. 5.

	Specification	label consolidation
Label	(empty) ✗ ? $P?$ $T?$ $P-$ $T-$ $P✓$ $T✓$	✓ $-$? ✗
specification consolidation	P T ? $-$ ✗	

Fig. 5: Consolidated coverage matrix.

Mark meanings are slightly modified in the new cells as depicted in Table 2. For a specification S, mark P (*resp.* T) denotes that S is tested (*resp.* proved) and strongly connected to at least one label. Mark ✗ means that S is invalidated.

Mark − indicates no correlation of S with any label, which means that either one piece of code is missing (in other words, one expected functionality is probably not implemented), or the specification is absurd (e.g. it has unsatisfiable hypotheses). This kind of information may be particularly useful for debugging code and/or specification. Mark ? means that the verification is currently inconclusive: an additional verification activity is required.For a label L, mark ✓ means that L is strongly connected to at least one specification. Mark − means no correlation between L and any specification. Mark ? is used for all the other cases which are inconclusive with respect to the coverage criterion.

If specification S has $\cdots$	Then the symbol for the consolidation cell of S is $\cdots$	It means $\cdots$
At least one cell P✓	P	S are proved
At least one cell T✓	T	S is tested
All cells are either $P-$ or $T-$	−	either S is absurd, or one label is missing for S
At least one cell ✗	✗	S is invalidated
Other case	?	Verification of S is inconclusive

If label L has $\cdots$	Then the symbol for the consolidation cell of L is $\cdots$	It means $\cdots$
At least one cell P✓ At least one cell T✓	✓	L is strongly connected to at least one specification
All cells are either $P-$ or $T-$	−	no correlation between L and any specification
Other case	?	not enough coverage information for label L

Table 2: Consolidation rules for specifications and labels.

Full coverage (as per DO-178) is reached if, after consolidating the coverage matrix, the resulting consolidated specification table only contains marks P or T, meaning that every specification is verified, while the resulting consolidated label table only contains mark ✓, meaning that every label is covered.

6.3 Verification Campaign Automatization

This section proposes an algorithm, shown in Fig. 6 and currently being implemented as a new Frama-C plugin. It consists of two consecutive steps that automatize as much as possible a verification campaign, in order to quickly reach full coverage.

1. **Proving.**
   (a) *Manually* provide all the necessary data (initial code, formal specification, labels and mutants) to an automatic proof technique (*e.g.* plug-in WP of Frama-C with an associated SMT solver).
   (b) *Automatically*, from the results, fill the (consolidated) coverage matrix.
   (c) *Manually* choose the next action according to the coverage matrix:
   − if full coverage is reached, the verification campaign is complete;

- if *an error is found*, correct it and restart the verification campaign;
- if *no error is found* but coverage is yet incomplete, continue the verification campaign by using another technique (e.g. another prover) or testing. If one goes for testing, goto step 2.

2. **Testing.**
   (a) *Manually* define test cases in order to test the uncovered specifications (not containing any mark ✓ in the coverage matrix). In order to quickly reach full coverage, try as much as possible to choose test cases which can pass through the remaining labels (not containing any mark ✓ in the coverage matrix).
   (b) *Manually* provide all the necessary data (specification, code, label, mutant) and test case to a testing tool (e.g. plug-in E-ACSL of Frama-C).
   (c) *Automatically*, from the results, fill the (consolidated) coverage matrix.
   (d) *Manually* choose the next action according to the coverage matrix:
   - if full coverage is reached, the verification campaign is complete;
   - if *an error is found*, correct it and restart the verification campaign;
   - if *no error is found* but coverage is yet incomplete, continue the verification campaign by defining (at least) one other test case.

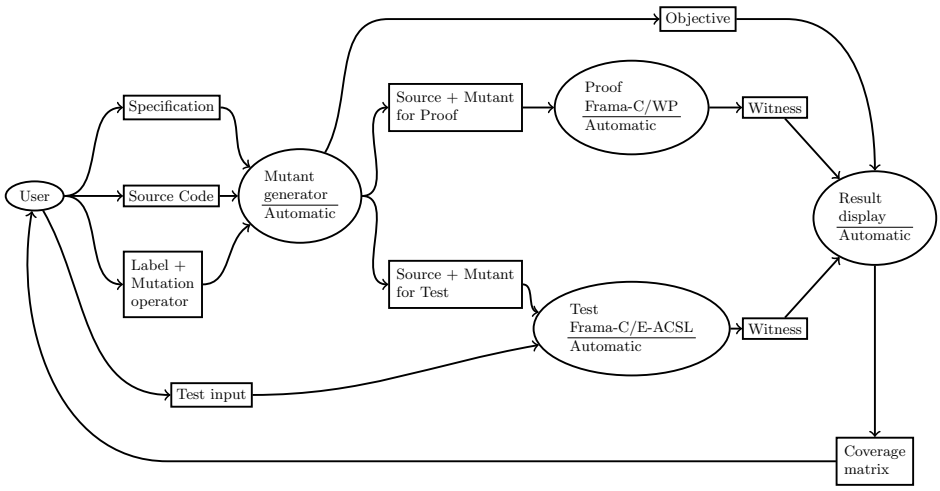

Fig. 6: Algorithm for automatizing a verification campaign.

7 Experiment

This section applies the previous algorithm to our running example of Section 3. To demonstrate that our verification coverage is able to detect a missing specification, we intentionally remove the VALID specification from our example.

Labels and their associated mutant are defined according to statement coverage criterion. Therefore, one label is attached to each statement. Two kinds of mutant operators, *replace* and *erase*, are used to define the labeled mutant of each label, as explained in Section 4.1.

	ERR					OK	VALUE			consol. label
	error 1	error 2	···	error 6	error 7	ok	low signal	medium signal	high signal	
`Lreturn1`	P✓	$P-$	···	$P-$	$P-$	$P-$	$P-$		$P-$	✓
`Lreturn2`	$P-$	P✓	···	$P-$	$P-$	$P-$	$P-$		$P-$	✓
					···					···
`Lreturn6`	$P-$	$P-$	···	P✓	$P-$	$P-$	$P-$		$P-$	✓
`Lreturn7`	$P-$	$P-$	···	$P-$	P✓	$P-$	$P-$		$P-$	✓
`Lstmt1`	$P-$	$P-$	···	$P-$	$P-$	$P-$	P?		$P-$	?
`Lstmt2`	$P-$	$P-$	···	$P-$	$P-$	$P-$	$P-$		P?	?
`Lstmt3`	$P-$	$P-$	···	$P-$	$P-$	$P-$	$P-$		$P-$	?
`Lstmt4`	$P-$	$P-$	···	$P-$	$P-$	$P-$	$P-$		$P-$	?
`Lreturn0`	$P-$	$P-$	···	$P-$	$P-$	P✓	$P-$		$P-$	✓
consol. spec.	P	P	···	P	P	P	?	?	?	

Fig. 7: Coverage matrix after running plug-in WP.

We now apply the algorithm of Section 6.3. First, we try to prove that the function satisfies its specifications by using plug-in WP of Frama-C. The results are stored in the coverage matrix (partially) shown in Fig. 7.

Except for the four rows from label `Lstmt1` to label `Lstmt4`, all rows of the matrix contain at least one mark P✓ which means a mark ✓ in the consolidated label table. Similarly, each column in categories ERR and OK contains at least one mark P✓ which means a mark P in the consolidated specification table. It means that the verification campaign already succeeds for the corresponding labels and specifications: plug-in WP was able to validate them alone.

The four rows from label `Lstmt1` to label `Lstmt4` do not contain mark P✓. They lead to four marks ? in the consolidated label table. Also, no column in the category VALUE contains mark P✓, hence three marks ? in the corresponding cells of the consolidated specification table. Consequently, these 4 labels and the 3 specifications in category VALUE were not covered by plug-in WP.

We now choose the plug-in E-ACSL to test them. For that purpose, we define three test cases which pass through the remaining labels `Lstmt1` to `Lstmt4`. Fig. 8 shows the resulting updated cells of the coverage matrix. From the consolidated label table, we conclude that label `Lstmt4` is the only label not yet covered. It prevents us to complete the verification campaign. Proofreading the code allows us to establish that the related piece of code has no correlation with any specification.

	low signal	medium signal	high signal	consol label
`Lstmt1`	P✓		$P-$	✓
`Lstmt2`	$P-$		P✓	✓
`Lstmt3`	$P-$	T✓	$P-$	✓
`Lstmt4`	$P-$	T?	$P-$	?
consol. spec	T	T	T	

Fig. 8: Interesting subset of the coverage matrix after running plug-in E-ACSL.

We now add an additional specification corresponding to that piece of code. It matches the previously removed VALID specification. Running the very same test cases again leads to an updated coverage matrix. Fig. 9 shows the only interesting cells. It allows us to conclude that our verification campaign is complete: we reach full functional and structural coverage by combining proof with plug-in WP and test with plug-in E-ACSL run on only three test cases.

	VALUE	VALID	consol
	medium signal	valid flag	label
`Lstmt4`	T?	T✓	✓
consol spec	T	T	

Fig. 9: Coverage cells of the added specification after running E-ACSL again.

8 Conclusion and Future Work

By combining and enhancing labels [5, 6] and mutations [16, 25], we introduce a new notion of verification coverage that allows us to combine test and proof for verifying a group of specifications related to the same piece of code. It subsumes both the usual notions of functional coverage and structural coverage.

Our verification coverage establish connections between pieces of code represented by labels and functional specifications. It allows us to measure verification and coverage rates through the number of specifications and labels strongly connected. Thus it provides a way to decrease the influence of a particular verification method when measuring coverage.

Based on this verification coverage, we also introduce an algorithm that automatizes most parts of a verification campaign combining test and proof in order to complete the verification process as quickly as possible. This algorithm is currently being implemented as a new Frama-C plug-in.

We also formalize our work thanks to new notions of verification witnesses and coverage matrices. Coverage matrices are consolidated *per* specification and *per* label in order to synthesize the verification and the coverage results. Such consolidations help the verification engineer to decide the next verification activity to be performed.

Future work includes studying the impact of mutation and coverage criteria on verification coverage. We also aim at extending existing toolchains in order to automatize label generation, choice of mutation operators and test case generation with respect to different coverage criteria. It would reduce the parts of our algorithm that are not yet automated.

Acknowledgment

The authors would like to thank the anonymous reviewers for their helpful comments and feedbacks. This work was partly supported by project VESSEDIA, which has received funding from the European Union's Horizon 2020 Research and Innovation Program under grant agreement No 731453. It was also partly supported by European Union's Involvement in Midi-Pyrénées through its Regional Development Fund.

References

1. Agrawal, H., Demillo, R.A., Hathaway, B., Hsu, W., Hsu, W., Krauser, E.W., Martin, R.J., Mathur, A., Spafford, E.: Design Of Mutant Operators For The C Programming Language. Tech. Rep. SERC-TR-41-P, Purdue University (1999)
2. Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press (2008)
3. Artho, C., Biere, A.: Combined Static and Dynamic Analysis. Electronic Notes in Theoretical Computer Science (ENTCS) **131**, 3–14 (2005)
4. Bardin, S., Delahaye, M., David, R., Kosmatov, N., Papadakis, M., Traon, Y.L., Marion, J.Y.: Sound and quasi-Complete Detection of Infeasible Test Requirements. In: IEEE 8th International Conference on Software Testing, Verification and Validation, ICST 2015. pp. 1–10 (2015)
5. Bardin, S., Kosmatov, N., Chebaro, O., Delahaye, M.: An All-in-One Toolkit for automated White-box Testing. In: Tests and Proofs (TAP 2014). pp. 53–60 (2014)
6. Bardin, S., Kosmatov, N., Cheynier, F.: Efficient Leveraging of Symbolic Execution to Advanced Coverage Criteria. In: IEEE 7th International Conference on Software Testing, Verification and Validation, ICST 2014. pp. 173–182 (2014)
7. Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, `http://frama-c.com/acsl.html`
8. Bishop, P., Bloomfield, R., L.Cyra: Combining Testing and Proof to Gain High Assurance in Software: A case study. In: IEEE International Symposium on Software Reliability Engineering (ISSRE). pp. 248–257 (2013)
9. Brahmi, A., Delmas, D., Essoussi, M.H., Randimbivololona, F., Atki, A., Marie, T.: Formalise to automate: deployment of a safe and cost-efficient process for avionics software. In: European Congress on Embedded Real Time Software and Systems (ERTSS'18) (2018)
10. Brown, D., Delseny, H., Hayhurst, K., Wiels, V.: Guidance for Using Formal Methods in a Certification Context. In: European Congress on Embedded Real Time Software and Systems (ERTS'10) (2010)
11. Chockler, H., Kupferman, O., Vardi, M.: Coverage Metrics for Formal Verification. In: Correct Hardware Design and Verification Methods. pp. 111–125 (2003)
12. Dadeau, F., Giorgetti, A., Bouquet, F., Enderlin, I.: Contract-based testing for PHP with Praspel. Journal of Systems and Software **136**, 209–222 (2018)
13. Dadeau, F., Ledru, Y., du Bousquet, L.: Measuring a Java Test Suite Coverage Using JML Specifications (2007)
14. Delahaye, M., Kosmatov, N., Signoles, J.: Common specification language for static and dynamic analysis of C programs. In: Symposium on Applied Computing (SAC'13). pp. 1230–1235. ACM (2013)
15. Delamaro, M.E., Deng, L., Durelli, V.H.S., Li, N., Offutt, J.: Experimental Evaluation of SDL and One-Op Mutation for C. In: IEEE 7th International Conference on Software Testing, Verification and Validation, ICST 2014. pp. 203–212 (2014)
16. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on Test Data Selection: Help for the Practicing Programmer. IEEE Computer **11**, 34–41 (1978)
17. Filliâtre, J.C.: Deductive software verification. In: International Journal on Software Tools for Technology Transfer. vol. 13, p. 397 (2011)
18. Floyd, R.W.: Assigning Meanings to Programs, pp. 65–81. Springer Netherlands, Dordrecht (1993)
19. Hoare, C.A.R.: An Axiomatic Basis for Computer Programming, pp. 89–100. Springer New York, New York, NY (1978)

20. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. In: Formal Aspects of Computing. pp. 573–609 (2015)
21. Kiss, B., Kosmatov, N., Pariente, D., Puccetti, A.: Combining Static and Dynamic Analyses for Vulnerability Detection: Illustration on Heartbleed. In: Hardware and Software: Verification and Testing: 11th International Haifa Verification Conference (HVC 2015). vol. 9434, pp. 39–50 (2015)
22. Kosmatov, N., Signoles, J.: Runtime assertion checking and its combinations with static and dynamic analyses - tutorial synopsis. In: International Conference on Tests and Proofs (TAP 2014). pp. 165–168 (2014)
23. Marcozzi, M., Bardin, S., Delahaye, M., Kosmatov, N., Prevosto, V.: Taming Coverage Criteria Heterogeneity with LTest. In: IEEE 10th International Conference on Software Testing, Verification and Validation, ICST 2017. pp. 500–507 (2017)
24. Marcozzi, M., Delahaye, M., Bardin, S., Kosmatov, N., Prevosto, V.: Generic and Effective Specification of Structural Test Objectives. In: IEEE 10th International Conference on Software Testing, Verification and Validation, ICST 2017. pp. 436–441 (2017)
25. Morell, L.J.: A Theory of Fault-Based Testing. IEEE Transactions on software engineering **16**, 844–857 (1990)
26. Moy, Y., Ledinot, E., Delseny, H., Wiels, V., Monate, B.: Testing or Formal Verification: DO-178C Alternatives and Industrial Experience. In: IEEE Software. pp. 50–57 (2013)
27. Offutt, A.J., J.Voas, J.Payne: Mutation Operators for Ada. Technical Report ISSE-TR-96-09 (1996)
28. Peters, D.K., Parnas, D.L.: Using Test Oracles Generated from Program Documentation. In: IEEE Transactions on software engineering. vol. 24, pp. 161–173 (1998)
29. RTCA (Firm). and EUROCAE (Agency).: DO-178C, Software Considerations in Airborne Systems and Equipment Certification. Document (RTCA (Firm))) (2011)
30. RTCA (Firm). SC-205 and EUROCAE (Agency). Working Group 71: Formal Methods Supplement to DO-178C and DO-278A. Document (RTCA (Firm))), RTCA, Incorporated (2011), `https://books.google.fr/books?id=nYhyMwEACAAJ`
31. Signoles, J., Kosmatov, N., Vorobyov, K.: E-ACSL, a Runtime Verification Tool for Safety and Security of C Programs. Tool Paper. In: International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES'17). pp. 164–173 (2017)
32. Software and Engineering Standards Committee: IEEE Standard for Software and System Test Documentation. In: IEEE Std 829-2008. pp. 1–118 (2008)
33. Williams, N., Marre, B., Mouy, P., Roger, M.: PathCrawler: Automatic Generation of Path Tests by Combining Static and Dynamic Analysis. In: Dependable Computing - EDCC 5. pp. 281–292 (2005)