Time to Clean Your Test Objectives

Michaël Marcozzi*
Imperial College London
Department of Computing
London, United Kingdom
m.marcozzi@imperial.ac.uk

Sébastien Bardin†
CEA, List
Software Safety and Security Lab
Gif-sur-Yvette, France
sebastien.bardin@cea.fr

Nikolai Kosmatov
CEA, List
Software Safety and Security Lab
Gif-sur-Yvette, France
nikolai.kosmatov@cea.fr

Mike Papadakis
SnT, University of Luxembourg
Luxembourg
michail.papadakis@uni.lu

Virgile Prevosto‡
CEA, List
Software Safety and Security Lab
Gif-sur-Yvette, France
virgile.prevosto@cea.fr

Loïc Correnson
CEA, List
Software Safety and Security Lab
Gif-sur-Yvette, France
loic.correnson@cea.fr

ABSTRACT

Testing is the primary approach for detecting software defects. A major challenge faced by testers lies in crafting efficient test suites, able to detect a maximum number of bugs with manageable effort. To do so, they rely on coverage criteria, which define some precise test objectives to be covered. However, many common criteria specify a significant number of objectives that occur to be infeasible or redundant in practice, like covering dead code or semantically equal mutants. Such objectives are well-known to be harmful to the design of test suites, impacting both the efficiency and precision of the tester's effort. This work introduces a sound and scalable technique to prune out a significant part of the infeasible and redundant objectives produced by a panel of white-box criteria. In a nutshell, we reduce this task to proving the validity of logical assertions in the code under test. The technique is implemented in a tool that relies on weakest-precondition calculus and SMT solving for proving the assertions. The tool is built on top of the Frama-C verification platform, which we carefully tune for our specific scalability needs. The experiments reveal that the pruning capabilities of the tool can reduce the number of targeted test objectives in a program by up to 27% and scale to real programs of 200K lines, making it possible to automate a painstaking part of their current testing process.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Theory of computation** → *Program analysis*;

KEYWORDS

Coverage Criteria, Infeasible Objectives, Redundant Objectives

*A major part of this work has been performed as a CEA, List employee.
†This work has been partially funded by the French ANR (grant ANR-12-INSE-0002).
‡This work has been partially funded by the EU H2020 programme (grant 731453).

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-5638-1/18/05.
https://doi.org/10.1145/3180155.3180191

ACM Reference Format:
Michaël Marcozzi, Sébastien Bardin, Nikolai Kosmatov, Mike Papadakis, Virgile Prevosto, and Loïc Correnson. 2018. Time to Clean Your Test Objectives. In *Proceedings of ICSE '18: 40th International Conference on Software Engineering , Gothenburg, Sweden, May 27-June 3, 2018 (ICSE '18),* 12 pages.
https://doi.org/10.1145/3180155.3180191

1 INTRODUCTION

Context. Heretofore, software testing is the primary method for detecting software defects [2, 42, 44, 66]. It is performed by executing the programs under analysis with some inputs, and aims at finding some unintended (defective) behaviors. In practice, as the number of possible test inputs is typically enormous, testers do limit their tests to a manageable but carefully crafted set of inputs, called a *test suite*. To build such suites, they rely on so-called *coverage criteria*, also known as adequacy or test criteria, which define the objectives of testing [2, 66]. In particular, many *white-box* criteria have been proposed so far, where the test objectives are syntactic elements of the code that should be covered by running the test suite. For example, the *condition coverage* criterion imposes to cover all possible outcomes of the boolean conditions appearing in program decisions, while the *mutant coverage* criterion requires to differentiate the program from a set of its syntactic variants. Testers need then to design their suite of inputs to cover the corresponding test objectives, such as — for the two aforementioned cases — condition outcomes or mutants to kill.

Problem. White-box testing criteria are purely syntactic and thus totally blind to the semantics of the program under analysis. As a consequence, many of the test objectives that they define may turn out to be either

(a) *infeasible*: no input can satisfy them, such as dead code or equivalent mutants [2], or
(b) *duplicate* versions of another objective: satisfied by exactly the same inputs, such as semantically equal mutants [53], or
(c) *subsumed* by another objective: satisfied by every input covering the other objective [1, 37, 52], such as validity of a condition logically implied by another one in condition coverage.

We refer to these three situations as *polluting test objectives*, which are well-known to be harmful to the testing task [52, 53, 62, 63, 65] for two main reasons:

- While (early) software testing theory [66] requires all the criterion objectives to be covered, this seldom reflects the actual practice, which usually relies on test suites covering only a part of them [23]. This is due to the difficulty of generating the appropriate test inputs, but also to infeasible test objectives. Indeed, testers often cannot know whether they fail to cover them because their test suites are weak or because they are infeasible, possibly wasting a significant amount of their test budget trying to satisfy them.
- As full objective coverage is rarely reached in practice, testers rely on the ratio of covered objectives to measure the strength of their test suites. However, the working assumption of this practice is that all objectives are of equal value. Testing research demonstrated that this is not true [1, 13, 17, 52], as duplication and subsumption can make a large number of feasible test objectives redundant. Such coverable redundant objectives may artificially deflate or inflate the coverage ratio. This skews the measurement, which may misestimate test thoroughness and fail to evaluate correctly the remaining cost to full coverage.

Goal and Challenges. While detecting all polluting test objectives is undecidable [1, 52], our goal is to provide a technique capable to identify a significant part of them. This is a challenging task as it requires one to perform complex program analyses over large sets of objectives produced by various criteria. Moreover, duplication and subsumption should be checked for each pair of objectives, a priori putting a quadratic penalty over the necessary analyses.

Although many studies have demonstrated the harmful effects of polluting objectives, to date there is no scalable technique to discard them. Most related research works (see Tables 1, 2 and Section 8) focus on the equivalent mutant problem, i.e. the particular instance of infeasible test objectives for the mutant coverage criterion. These operate either in dynamic mode, i.e. mutant classification [57, 58], or in static mode, i.e. Trivial Compiler Equivalence (TCE) [53]. Unfortunately, the dynamic methods are unsound and produce many false positives [51, 58], while the static one does not deal with all forms of mutation and cannot detect subsumed mutants (whereas it handles duplicates in addition to infeasible ones). The LUncov technique [8] combines two static analyses to prune out infeasible objectives from a panel of white-box criteria in a generic way, but faces scalability issues.

	Sound	Scale	Kind of Pollution			Criterion Genericity
			Inf.	Dupl.	Subs.	
Mutant class. [58]	×	✓	✓	×	×	×
TCE [53]	✓	✓	✓	✓	×	×
LUncov [8]	✓	×	✓	×	×	✓
LClean (this work)	✓	✓	✓	✓	✓	✓

Table 1: Comparison with closest research techniques

	Analyses	Scope	Acuteness
TCE [53]	built-in compiler optimizations	interprocedural	+
LUncov [8]	value analysis and weakest-precondition	interprocedural	++
LClean (this work)	weakest-precondition	local function	+

Table 2: Static analyses available in closest techniques

Proposal. Our intent is to provide a *unified*, *sound* and *scalable* solution to prune out a significant part of *polluting objectives*, including *infeasible* but also *duplicate* and *subsumed* ones, while handling a large panel of white-box criteria in a *generic* manner. To achieve this, we propose reducing the problem of finding polluting objectives for a wide range of criteria to the problem of proving the validity of logical assertions inside the code under test. These assertions can then be verified using known verification techniques.

Our approach, called *LClean*, is the first one that scales to programs composed of 200K lines of C code, while handling all types of polluting test requirements. It is also generic, in the sense that it covers most of the common code-based test criteria (described in software testing textbooks [2]) and it is capable of using almost any state-of-the-art verification technique. In this study, we use weakest-precondition calculus [21] with SMT solving [18] and identify 25K polluting test objectives in fourteen C programs.

LClean introduces two acute code analyses that enable focusing the detection of duplicate and subsumed objectives over a limited amount of high-hit-rate pairs of objectives. This makes it possible to detect a significant number of redundant objectives while avoiding a quadratic penalty in computation time. The LClean tool is implemented on top of the Frama-C/LTest platform [34, 40], which features *strong* conceptual and technical foundations (Section 3). We specifically extend the Frama-C module dedicated to proving code assertions to make the proposed solution scalable and robust.

Contributions. To sum up, we make the following contributions:

- The *LClean* approach: a scalable, sound and unified formal technique (Sections 2 and 4) capable to detect the three kinds of polluting test objectives (i.e. infeasible, duplicate and subsumed) for a wide panel of white-box criteria, ranging from condition coverage to variants of MCDC and weak mutation.
- An *open-source prototype tool LClean* (Section 5) enacting the proposed approach. It relies on an industrial-proof formal verification platform, which we tune for the specific scalability needs of LClean, yielding a robust multi-core assertion-proving kernel.
- A thorough *evaluation* (Sections 6 and 7) assessing

(a) the scalability and detection power of LClean for three types of polluting objectives and four test criteria – pruning out up to 27% of the objectives in C files up to 200K lines,
(b) the impact of using a multi-core kernel and tailored verification libraries on the required computation time (yielding a speedup of approximately 45×), and
(c) that, compared to the existing methods, LClean prunes out four times more objectives than LUncov [8] in about half as much time, it can be one order of magnitude faster than (unsound) dynamic identification of (likely) polluting objectives, and it detects half more duplicate objectives than TCE, while being complementary to it.

Potential Impact. Infeasible test objectives have been recognized as a main cost factor of the testing process [62, 63, 65]. By pruning out a significant number of them with LClean, testers could reinvest the spared cost in targeting full coverage of the remaining objectives. This would make testing more efficient, as most faults are found within high levels of coverage [22]. Pruning out infeasible test objectives could also make the most meticulous testing criteria (e.g. mutation testing) less expensive and thus more acceptable in industry [53]. Furthermore, getting rid of redundant objectives should provide testers with more accurate quality evaluations of

```
1  // given three sides x,y,z of a valid triangle, computes
2  // its type as: 0 scalene, 1 isosceles, 2 equilateral
3  int type = 0;
4  // l1:  x == y && y == z; (DC) l2:  x != y || y != z; (DC)
5  // l3:  x == y; (CC)           l4:  x != y; (CC)
6  if( x == y && y == z ){
7      type = type + 1;
8  }
9  // l5: x==y || y==z || x==z; (DC)  l6: x!=y && y!=z && x!=z; (DC)
10 // l7: x == y; (CC)                l8: x != y; (CC)
11 // l9: x!=y && y==z && x==z; (MCC) l10: x==y && y!=z && x==z;(MCC)
12 if( x == y || y == z || x == z ){
13 // l11: type + 1 != type + 2; (WM)  l12: type + 1 != type; (WM)
14 // l13: type + 1 != -type + 1; (WM) l14: type + 1 != 1; (WM)
15     type = type + 1;
16 }
```

Figure 1: Example of a small C program with test objectives

their test suites and also result in sounder comparisons of test generation techniques [52].

2 MOTIVATING EXAMPLE

Figure 1 shows a small C program inspired by the classic triangle example [43]. Given three integers x, y, z supposed to be the sides of a valid triangle, it sets variable *type* according to the type of the triangle: equilateral ($type = 2$), isosceles ($type = 1$) or scalene ($type = 0$). Figure 1 also illustrates fourteen test objectives from common test criteria labelled from l_1 to l_{14}. l_1 and l_2 require the test suite to cover both possible decisions (or branches) of the conditional at line 6. For example, covering l_2 means to find test data such that, during test execution, the location of l_2 is reached and the condition `x != y || y != z` is true at this location, which ensures to execute the else branch. Similarly, l_5 and l_6 require the tests to cover both decisions at line 12. These four objectives are specified by the Decision Coverage (DC) criterion for this program. l_3 and l_4 (resp., l_7 and l_8) require the tests to cover both truth values of the first condition in the compound condition on line 6 (resp., line 12). They are imposed by Condition Coverage (CC) – the similar test objectives imposed by CC for the other conditions are not shown to improve readability. l_9 and l_{10} provide examples of objectives from Multiple Condition Coverage (MCC) for conditional at line 12. MCC requires the tests to cover all combinations of truth values of conditions. Finally, objectives l_{11} to l_{14} encode some Weak Mutants (WM) of the assignment on line 15 (see Bardin et al. [9, Theorem 2] for more detail).

We can easily notice that l_9 and l_{10} put unsatisfiable constraints over x, y and z. They are thus *infeasible* objectives and trying to cover them would be a waste of time. Other objectives are *duplicates*, denoted by $\Leftrightarrow$: they are always covered (i.e. reached and satisfied) simultaneously. We obviously have $l_3 \Leftrightarrow l_7$ and $l_4 \Leftrightarrow l_8$ since the values of x and y do not change in-between. Although syntactically different, l_{13} and l_{14} are also duplicates, as they are always reached together (we call them *co-reached* objectives) and satisfied if and only if $type \neq 0$. Finally, we refer to objectives like l_{11} and l_{12} as being *trivial* duplicates: they are co-reached, and always satisfied as soon as reached. While we do not have $l_1 \Leftrightarrow l_5$, covering l_1 necessarily implies covering l_5, that is, l_1 *subsumes* l_5, denoted $l_1 \Rightarrow l_5$. Other examples of subsumed objectives can be found, like $l_6 \Rightarrow l_2$. Duplicate and subsumed objectives are redundant objectives that can skew the measurement of test suite strength, as it should be provided by the test coverage ratio. For example, considering the objectives from the DC criterion, the test suite composed of the single test ($x = 1, y = 2, z = 1$) covers l_2 and l_5 but not l_1 and l_6, which implies a medium coverage ratio of 50%. The tester may be interested to know the achieved level of coverage without counting duplicate or subsumed objectives. Here, l_2 and l_5 are actually subsumed by l_1 and l_6. If the subsumed objectives are removed, the coverage ratio falls down to 0%. Discarding redundant objectives provides a better measurement of how far testers are from building an efficient test suite, only with the necessary inputs for covering the non-redundant objectives (l_1 and l_6 in this case).

The main purpose of this paper is to provide a lightweight yet powerful technique for pruning out infeasible, duplicate and subsumed test objectives. To do so, our approach first focuses on infeasible objectives. In Figure 1, one can notice, for instance, that the problem of proving l_9 to be infeasible can be reduced to the problem of proving that a code assertion `!(x!=y && y==z && x==z)` at line 11 will never be violated. Our approach then delegates this proof for each objective to a dedicated verification tool. While infeasibility should be checked once per objective, *duplication and subsumption require one to analyse all the possible pairs*. To avoid quadratic complexity, we focus on detecting duplicate and subsumed pairs only among the objectives that belong to the same sequential block of code, with no possible interruption of the control flow (with goto, break, ...) in-between. By construction, the objectives in these groups are always co-reached. In Figure 1, l_1–l_{10} and l_{11}–l_{14} are two examples of such groups. Examples of duplicate and subsumed objectives within these groups include $l_3 \Leftrightarrow l_7$, $l_4 \Leftrightarrow l_8$, $l_{11} \Leftrightarrow l_{12}$, $l_{13} \Leftrightarrow l_{14}$, $l_1 \Rightarrow l_5$ and $l_6 \Rightarrow l_2$. We call them *block-duplicate* and *block-subsumed* objectives. On the other hand, l_1 and l_{13} are duplicate (at line 14, `type` is nonzero if and only if `x`, `y`, and `z` are equal), but this will not be detected by our approach since those labels are not in the same block.

3 BACKGROUND

3.1 Test Objective Specification with Labels

Given a program P over a vector V of m input variables taking values in a domain $D \triangleq D_1 \times \cdots \times D_m$, a *test datum* t for P is a valuation of V, i.e. $t \in D$. A *test suite* $TS \subseteq D$ is a finite set of test data. A (finite) execution of P over some t, denoted $P(t)$, is a (finite) run $\sigma \triangleq \langle (loc_0, s_0), \ldots, (loc_n, s_n) \rangle$ where the loc_i denote successive (control-)locations of P ($\approx$ statements of the programming language in which P is written), loc_0 refers to the initial program state and the s_i denote the successive internal states of P ($\approx$ valuation of all global and local variables and of all memory-allocated structures) after the execution of each loc_i.

A test datum t *reaches* a location loc at step k with internal state s, denoted $t \leadsto^k_P (loc, s)$, if $P(t)$ has the form $\sigma \cdot (loc, s) \cdot \rho$ where σ is a partial run of length k. When focusing on reachability, we omit k and write $t \leadsto_P (loc, s)$.

Given a test objective $\mathbf{c}$, we write $t \leadsto_P \mathbf{c}$ if test datum t covers $\mathbf{c}$. We extend the notation for a test suite TS and a set of test objectives $\mathbf{C}$, writing $TS \leadsto_P \mathbf{C}$ when for any $\mathbf{c} \in \mathbf{C}$, there exists $t \in TS$ such that $t \leadsto_P \mathbf{c}$. A *(source-code based) coverage criterion* $\mathbb{C}$ is defined as a systematic way of deriving a set of test objectives $\mathbf{C} \triangleq \mathbb{C}(P)$ for any program under test P. A test suite TS satisfies (or achieves) a coverage criterion $\mathbb{C}$ if TS covers $\mathbb{C}(P)$.

Labels. *Labels* have been introduced in [9] as a code annotation language to encode concrete test objectives. Several common coverage criteria can be simulated by label coverage, in the sense that for a given program P and a criterion $\mathbb{C}$, every concrete test objective from $\mathbf{C} \triangleq \mathbb{C}(P)$ can always be encoded using a *corresponding* label.

Given a program P, a *label* $\ell \in \text{Labs}_P$ is a pair (loc, φ) where loc is a location of P and φ is a predicate over the internal state at loc. There can be several labels defined at a single location, which can possibly share the same predicate. More concretely, the notion of labels can be compared to labels in the C language, decorated with a pure (i.e. side-effect-free) boolean C expression.

We say that a test datum t *covers a label* $\ell \triangleq (loc, \varphi)$ in P, denoted $t \overset{\text{L}}{\rightsquigarrow}_P \ell$, if there is a state s such that t reaches (loc, s) (i.e. $t \rightsquigarrow_P (loc, s)$) and s satisfies φ. An *annotated program* is a pair $\langle P, L \rangle$ where P is a program and $L \subseteq \text{Labs}_P$ is a set of labels for P. Given an annotated program $\langle P, L \rangle$, we say that a test suite TS satisfies the *label coverage criterion* (**LC**) for $\langle P, L \rangle$, denoted $TS \overset{\text{L}}{\rightsquigarrow}_{\langle P,L \rangle}$ **LC**, if TS covers every label of L (i.e. $\forall \ell \in L : \exists t \in TS : t \overset{\text{L}}{\rightsquigarrow}_P \ell$).

Criterion Encoding. Label coverage *simulates a coverage criterion* $\mathbb{C}$ if any program P can be *automatically* annotated with a set of *corresponding* labels L in such a way that any test suite TS satisfies **LC** for $\langle P, L \rangle$ if and only if TS covers all the concrete test objectives instantiated from $\mathbb{C}$ for P. The main benefit of labels is to *unify* the treatment of test requirements belonging to different classes of coverage criteria in a transparent way, thanks to the *automatic insertion* of labels in the program under test. Indeed, it is shown in [9] that label coverage can notably simulate basic-block coverage (**BBC**), branch coverage (**BC**), decision coverage (**DC**), function coverage (**FC**), condition coverage (**CC**), decision condition coverage (**DCC**), multiple condition coverage (**MCC**) as well as the side-effect-free fragment of weak mutations (**WM'**). The encoding of **GACC** comes from [50]. Some examples are given in Figure 1.

Co-reached Labels. We say that location loc is *always preceded by* location loc' if for any test datum t, whenever the execution $P(t) \triangleq \langle (loc_0, s_0), \ldots, (loc_n, s_n) \rangle$ passes through location loc at step k (i.e. $loc = loc_k$) then $P(t)$ also passes through loc' at some earlier step $k' \leq k$ (i.e. $loc' = loc_{k'}$) without passing through loc or loc' in-between (i.e. at some intermediate step i with $k' < i < k$). Similarly, loc' is said to be *always followed by* location loc if for any t, whenever the execution $P(t)$ passes through loc' at step k' then $P(t)$ also passes through loc at some later step $k \geq k'$ without passing through loc or loc' in-between. Two locations are *co-reached* if one of them is always preceded by the other, while the second one is always followed by the first one. Note that we exclude the case when one of locations is traversed several times (e.g. due to a loop) before being finally followed by the other one. In a sequential block of code, with no possible interruption of the control flow in-between (no goto, break, ...), all locations are co-reached. We finally say that two labels are *co-reached* if their locations are co-reached.

3.2 Polluting Labels

In the remainder of the paper, test objectives will often be expressed in terms of labels. This work addresses three kinds of polluting labels: infeasible, duplicate and subsumed. A label ℓ in P is called *infeasible* if there is no test datum t such that $t \overset{\text{L}}{\rightsquigarrow}_P \ell$. In other words, it is not possible to reach its location and satisfy its predicate.

We say that a label ℓ *subsumes* another label ℓ' (or ℓ' *is subsumed by* ℓ) in P, denoted $\ell \Rightarrow \ell'$, if for any test datum t, if $t \overset{\text{L}}{\rightsquigarrow}_P \ell$ then $t \overset{\text{L}}{\rightsquigarrow}_P \ell'$ as well. Finally, two labels ℓ and ℓ' are called *duplicate*[1], denoted $\ell \Leftrightarrow \ell'$, if each of them subsumes the other one. For the specific case where both labels ℓ and ℓ' belong to the same group of co-reached labels in a block, we call a duplicate (resp., subsumed) label *block-duplicate* (resp., *block-subsumed*).

Notice that if a label ℓ is infeasible, it subsumes by definition any other label ℓ'. We call this phenomenon *degenerate subsumption*. If ℓ' is feasible, it should be kept and covered. In this case, the truly polluting objective is ℓ rather than ℓ'. That is the reason why it is necessary to eliminate as many infeasible labels as possible before pruning out subsumed labels.

3.3 The Frama-C/LTest Platform

Frama-C [34] is an open-source industrial-strength framework dedicated to the formal analysis of C programs. It has been successfully used in several safety and security critical contexts. The tool is written in OCaml, and represents a very significant development (around 150K lines for the kernel and the main plug-ins alone).

Frama-C is based on a small kernel that takes care of providing an abstract representation of the program under analysis and maintaining the set of properties that are known about the program state at each possible execution step. These properties are expressed as ACSL [11] annotations. On top of the kernel, many plug-ins can perform various kinds of analysis, and can interact with the kernel either by indicating that a property ϕ holds, or by asking whether some other property ψ is true (in the hope that another plug-in will be able to validate ϕ later on).

In the context of this paper, we are mainly interested in the four following (open-source) plug-ins. *LAnnotate*, *LUncov* and *LReplay* are part of Frama-C/LTest [7, 40]. LAnnotate annotates the program with labels according to the selected criterion. LUncov combines weakest-precondition and value analysis to detect infeasible test objectives. LReplay executes a test suite and computes its coverage ratio. *WP* is a plug-in implementing weakest-precondition calculus [10, 28] in order to prove that an ACSL assertion holds.

4 THE LCLEAN APPROACH

The LClean technique involves three main steps (cf. Figure 2) preceded by a preprocessing phase. The first step aims at detecting infeasible label-encoded objectives. The second step targets trivial block-duplicate labels, while the third step focuses more generally on block-subsumed and block-duplicate labels.

Given a program P and a coverage criterion $\mathbb{C}$ that can be simulated by labels, the preprocessing consists in generating the corresponding labels L. For C programs, this is done by the LAnnotate plug-in of Frama-C. The LClean approach itself operates on the annotated program $\langle P, L \rangle$ and marks polluting labels so that they can be pruned out.

4.1 Step 1: Infeasible Labels

LClean systematically explores $\langle P, L \rangle$ and replaces every label $\ell \triangleq (loc, \varphi)$ by an assertion `assert(!`φ`)`, whose predicate is the negation of the label condition. The resulting assertion-laden code is

[1] The term *equivalent label* is not used here to avoid any confusion with the notion of *equivalent mutant*, which in mutation testing means *infeasible objective*.

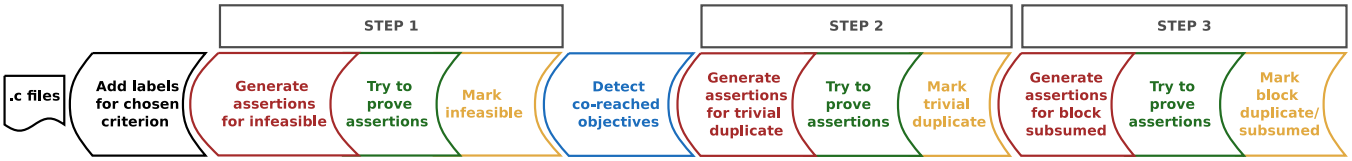


Figure 2: Process view of the LClean approach with main steps and substeps

sent to a deductive verification tool designed for proving that the received program is correct w.r.t. the defined assertions, i.e. that none of them can be violated during a possible run of the program. In practice, the verification tool returns the list of the assertions that it was able to prove correct. Since each assertion is by construction the negation of a label condition, the corresponding labels are formally proven to be infeasible, and are marked as so. These marks will be both used as a final result of the approach and as internal information transmitted to the next two steps of LClean. Regarding Figure 1, LClean indeed detects that l_9 and l_{10} are infeasible.

4.2 Detection of Co-reached Labels

```
1  void calledOnce () {
2    // l1: φ1
3    code1;
4  }
5  int main ( int i ) {
6  // l2: φ2
7  if (i>0) {
8    // l3: φ3
9    if (i==5) i++;
10   // l4: φ4
11   calledOnce ();
12   if (i==7) exit(0);
13   // l5: φ5
14   i++;
15   // l6: φ6
16 } else {
17   // l7: φ7
18   code2;
19 }
20 return i;
21 }
```

Figure 3: Co-reached locations

Prior to Steps 2 and 3, LClean performs the detection of blocks of co-reached locations. We illustrate it using the sample program of Figure 3. First, a basic syntactic analysis detects six blocks in the program: the global block of each of the two functions, the two branches of the outer conditional (line 7), and the then branches of the two nested conditionals. Second, a call-graph analysis discovers that the first function is only called once in the whole program, so that its outer block can be seen as executed as a part of the block containing the function call. The two blocks can then be merged. Finally, a conservative control-flow interruption analysis detects that the `exit(0);` statement at line 9 may interrupt the control-flow within the then branch of the outer conditional. The corresponding block is thus split into two blocks, gathering respectively the statements before and after the `exit(0);` statement. The identified blocks enabling us to conclude that there are four groups of mutually co-reached labels: $\{l_2\}$, $\{l_3, l_4, l_1\}$, $\{l_5, l_6\}$ and $\{l_7\}$.

4.3 Step 2: Trivial Block-Duplicate Labels

As in Step 1, LClean systematically explores $\langle P, L\rangle$ and replaces labels by assertions. Except for the labels marked as infeasible in Step 1, which are simply dropped out, each label $\ell \triangleq (loc, \varphi)$ is replaced by an assertion `assert(`φ`)`. This time, the predicate is directly the label condition. The resulting assertion-laden code is sent to the verification tool. The proven assertions correspond to labels that will be always satisfied as soon as their location is reached. Afterwards, LClean identifies among these always-satisfied-when-reached the groups of co-reached labels (cf. Section 4.2). The labels within each of the groups are trivial block-duplicates, and they are marked as being clones of a single label chosen among them. Again, these marks will be both final results and internal information transmitted to the next step. For the example of Figure 1, LClean will identify that l_{11} and l_{12} are trivial block-duplicate labels. Similarly, if we assume that all predicates φ_i are always satisfied for the code of Figure 3, Step 2 detects that l_3, l_4 and l_1 are trivial duplicates, and l_5 and l_6 are as well. As a subtle optimization, LClean can detect that label l_2 is always executed simultaneously with the outer conditional, so that l_2 will be covered if and only if at least one of the labels l_3 and l_6 is covered. l_2 can thus be seen as duplicate with the pair (l_3,l_6) and is marked as so.

4.4 Step 3: Block-Subsumed Labels

Within each group of co-reached labels, the labels previously detected as infeasible by Step 1 are removed and those detected as trivial block-duplicates by Step 2 are merged into a single label. Afterwards, every label $\ell_i = (loc_i, \varphi_i)$ remaining in the group is replaced by a new statement `int vl`$_i$ `=` φ_i`;`, which assigns the value of the label condition to a fresh variable `vl`$_i$. Then, for each pair $(\ell_i, \ell_j)_{i\neq j}$ of co-reached labels in the group, the assertion `assert(vl`$_i \implies$ `vl`$_j$`)`; is inserted at the end of the corresponding block of co-reached locations. If this assertion is proven by the verification tool, then label ℓ_i subsumes label ℓ_j. Indeed, their locations are co-reached, and the proven assertion shows that every input satisfying φ_i will also satisfy φ_j. As a consequence, every input that covers ℓ_i also covers ℓ_j.

The graph of subsumption relations detected in a group of co-reached labels is then searched for cycles. All labels in a cycle are actually duplicates and can be marked as mergeable into a single label. Among the labels that survive such a merging phase, those that are pointed to by at least one subsumption relation are marked as subsumed labels. For the example of Figure 1, LClean will identify, for instance, $l_1 \Rightarrow l_5$, $l_6 \Rightarrow l_2$, $l_3 \Leftrightarrow l_7$ and $l_{13} \Leftrightarrow l_{14}$.

4.5 Discussion of LClean Design

Once the third and final step finished, LClean returns a list of polluting labels composed of the infeasible ones returned by Step 1 and of the duplicate and subsumed ones returned by Steps 2 and 3. It should be noted that the approach is incremental and that each of the three main steps can even be run independently of the others. However, removing infeasible objectives before Steps 2 and 3 is important, as it reduces the risk of returning degenerate subsumption relations. Similarly, Step 2 detects duplicate labels that would be identified by Step 3 anyway, but Step 2 finds them at much lower cost. Indeed, the number of proofs required by Step 2 is linear in the number of labels as it does not have to consider pairs of labels. The incremental nature of the approach, coupled with the fact that assertion proving has become reasonably fast (c.f. Section 6) and that it can be parallelised, as well as performed independently over stand-alone code units (e.g. C functions), makes a continuous computation of polluting objectives conceivable during software

Criterion	Labels	Block Pairs	Function Pairs	Program Pairs
CC	27,638	94,042	3,013,940	428,075,244
MCC	30,162	314,274	3,961,004	503,856,852
GACC	27,638	94,042	3,013,940	428,075,244
WM	136,927	2,910 908	80,162,503	8,995,885,473
TOTAL	**222,365** (×1/15)	**3,413,266** (×1)	**90,151,387** (×26)	**10,355,892,813** (×3034)

Figure 4: Number of pairs of labels in 14 C programs

development. This could be used for continuous integration to enforce test suites of specific coverage levels.

The LClean approach might be extended to detect duplicate or subsumed labels that are not in the same basic block, by generating more complex assertions that would be flow-sensitive. However, limiting the analysis to block-duplicate and block-subsumed labels turns out to be a sweet spot between detection power and computation time. Indeed, Figure 4 details the total number of pairs of labels for four common criteria in the 14 C programs used in the evaluation in Section 6 (cf. Figure 6). Figure 4 also presents the total number of pairs of labels taken inside the same block, inside the same function or over the whole program. We can see that focusing the analysis on block pairs enables reducing the number of necessary proofs by one to four orders of magnitude. At the same time, it seems reasonable to think that a significant part of the duplicate or subsumed labels reside within the same basic block, as those labels are always executed together and typically describe test objectives related to closely interconnected syntactic elements of the program.

5 IMPLEMENTATION

The three steps of the LClean approach are implemented in *three independent open-source Frama-C plug-ins*[2] (≈5,000 locs in OCaml). These plug-ins share a common architecture depicted in Figure 5. It relies on the Frama-C kernel (in black) and features four modules (in color) performing the different substeps of an LClean step. It receives as input an annotated program $\langle P, L \rangle$, in which labels have already been generated with plug-in LAnnotate [7] in order to simulate the coverage criterion of interest. As a starting point, the program is parsed by the Frama-C kernel, which makes its abstract syntax tree (AST) available for all the components of the architecture. We now present the four modules performing the analysis.

Assertion Generator. The Assertion Generator replaces the labels in the code by assertions according to the corresponding step (cf. Section 4). Frama-C primitives are used to explore the AST, locate the nodes corresponding to labels and replace them by the required assertions, written in ACSL.

[2] Available from http://icse18.marcozzi.net.

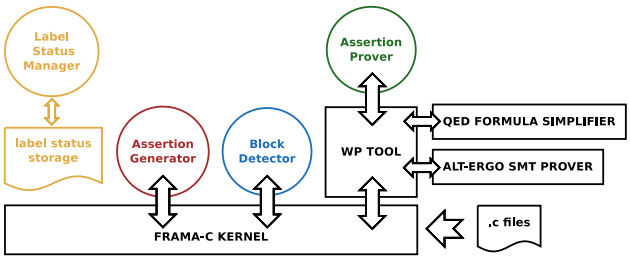


Figure 5: Frama-C plug-in implementing one LClean step

Robust Multicore Assertion Prover. The Assertion Prover deals with proving the assertions introduced in the AST by the Assertion Generator and relies on the WP plug-in. It is not a simple wrapper for WP: the Assertion Prover introduces *crucial optimizations ensuring its scalability and robustness*:

- First, it embeds a version of WP that we carefully optimized for our specific needs, making it capable to prove several different assertions independently in a single run of the tool. This version factors out a common part of the analysis (related to the program semantics) that would have to be repeated uselessly if WP was called once per assertion.
- Second, its multi-core implementation ensures a significant speedup. The assertions to be proved are shared among several parallel WP instances running on different cores.
- Third, the Assertion Prover also guarantees robustness and adaptability of the process. Indeed, the WP tool can consume a high amount of memory and computation time when analyzing a large and complex C function. The Assertion Prover can smoothly interrupt a WP session when a threshold w.r.t. the used memory or elapsed time has been reached.

All these improvements to Frama-C/WP have been proven crucial for large-scale experiments (cf. Section 6). A technical description of how they were actually implemented, comparing the optimised and non-optimised source code of the tool, can be found on the companion website[2] of this paper.

Label Status Manager. The Label Status Manager maintains and gives access to a set of files storing a status for each label. Each label is identified by a unique integer ID used both in the AST and in the status files. The status of a label can be a) infeasible, b) duplicate to another ID (or a pair of IDs), c) subsumed by other IDs, or d) unknown. The status files are updated by the plug-ins when they detect that some labels can be marked as polluting. The plug-ins for Steps 2 and 3 also check the files in order to drop out the labels marked as polluting during the previous steps.

Block Detector. The detector of blocks of co-reached labels is only used before Steps 2 and 3. It relies on the Frama-C primitives to explore the AST and perform the analyses detailed in Section 4.2. For each block found, it returns the label IDs of co-reached labels belonging to the block.

6 EXPERIMENTAL EVALUATION

To evaluate experimentally LClean, we consider the following three research questions:

Research Question 1 (RQ1): Is the approach effective and useful? Especially, (a) Does it identify a significant number of objectives from common criteria, all being real polluting objectives? b) Can it scale to real-world applications, involving many lines of code and complex language constructs?

Research Question 2 (RQ2): Do the optimizations (Section 5) improve the time performance in a significant way, impacting LClean acceptability in practice?

Research Question 3 (RQ3): How does our approach compare with the closest approaches like LUncov, mutant classification and TCE, especially in terms of pruning power and time performance?

The experimental artefacts used to answer these questions and the fully detailed results that we obtained are available on the

companion website[2] of the paper. The tool and artefacts have also been installed in a Linux virtual machine provided on the website and enabling an easy reproduction of the experiments described in the next subsections. All these experiments were performed on a Debian Linux 8 workstation equipped with two Intel Xeon E5-2660v3 processors, for a total of 20 cores running at 2.6Ghz and taking advantage of 25MB cache per processor and 264GB RAM.

6.1 RQ1: Effectiveness and Scalability

We consider fourteen C programs of various types and sizes (min: 153 locs, mean: 16,166 locs, max: 196,888 locs) extracted from five projects: the seven Siemens programs from [29], four libraries taken from the cryptographic OpenSSL toolkit [49], the full GNU Zip compression program [27], the complete Sjeng chess playing IA application [59] and the entire SQLite relational database management system [60]. Every program is annotated successively with the labels encoding the test objectives of four common coverage criteria: Condition Coverage (CC), Multiple-Condition Coverage (MCC), General Active Clause Coverage (GACC) and Weak Mutations (WM, with sufficient mutation operators [46]). The LClean tool is then run to detect polluting objectives for each (program,criterion) pair.

For each step of the LClean process, the number of marked objectives and the computation time are reported in Figure 6. 11% of the 222,365 labels were marked as polluting in total (min: 4% for CC/MCC with SQLite, max: 27% for WM in Siemens/printokens.c). The global ratio of marked polluting objectives is 5% for CC, 5% for MCC, 6% for GACC and 15% for WM. In total, 13% of the detected polluting objectives were infeasible, 46% were duplicate (about one half were marked during Step 2 and the other during Step 3) and 41% were subsumed. The computation time ranges from 10s for MCC in Siemens/schedule.c (410 locs and 58 objectives) to ~69h for WM in SQLite (197K locs and 90K objectives). Globally, computation time is split into 10% for Step 1, 8% for Step 2 and 82% for Step 3. While the computation time is acceptable for a very large majority of the experiments, Step 3 becomes particularly costly when applied on the largest programs with the most meticulous criteria. This is of course due to the fact that this step is quadratic in the number of labels. While we limit our analysis to block pairs, the number of resulting proof attempts still gets large for bigger applications, reaching 1.8M proofs for SQLite and WM (which remains tractable). Yet, limiting LClean to Steps 1 & 2 still marked many labels and is much more tractable: on SQLite, it detects 4566 polluting objectives in only 9h (13692 objectives in 69h for full LClean). Moreover, this should be compared to the fact that running the SQLite TH3 test suite[3] and computing the mutation score takes many days and that identifying polluting objectives is a time-consuming manual task (authors of [58] report 15 minutes per instance). As the SQLlite developers report[3] that they work hard to obtain test suites with a 100% coverage score for different criteria, they should immediately benefit from our tool.

Conclusion: These results indicate that LClean is a useful approach able to detect that a significant proportion of the test objectives from various common criteria are polluting ones, even for large and complex real-word applications. In practice, for very large programs and demanding criteria, LClean can be limited to Steps 1 & 2, keeping a significant detection power at a much lower expense.

[3] https://www.sqlite.org/testing.html

6.2 RQ2: Impact of Optimizations

We repeat the experiments performed in **RQ1** for the WM criterion over the seven Siemens programs, but we deactivate the optimizations that we implemented in the Assertion Prover of our tool, namely tailored WP tool and multi-core implementation (Section 5). Figure 7 details the obtained computation times (*in logarithmic scale*) for the three steps of the LClean process, considering three levels of optimizations. At level 0 (oblique-lined blue), the Assertion Prover uses a single instance of the classical Frama-C/WP running on a single core. At level 1 (horizontal-lined red), the Assertion Prover uses 20 instances of the classical version WP running on 20 cores. Level 2 (plain beige) corresponds to the actual version of the tool used in **RQ1**, when all the optimizations are activated: the Assertion Prover uses 20 instances of our tailored version WP running on 20 cores.

We observe that the total computation time is reduced by a *factor of 2.4* when switching from level 1 to level 2, and that it is reduced by a *factor of 45* when switching from level 0 to level 2. These factors are very similar for all the steps of the LClean process. The analysis results remained unchanged across the optimization levels.

Conclusion: These results show that our optimizations have a very significant impact over the time performance of our tool, making the experiments on large programs intractable without them. The measured speedup of 45x has a sensible influence over the perceived speed of the tool, improving its acceptability in practice.

6.3 RQ3: LClean vs. Closest Related Works

6.3.1 LUncov. We apply both LUncov [8] and LClean on the same benchmarks [8]. The measured computation time and detection power for LUncov and LClean are compared in Figure 8. As LUncov is limited to infeasibility, we also provide results for Step 1 of LClean. It appears that LClean detects 4.2× more polluting labels than LUncov in 1.8× less time. When LClean is limited to Step 1, it detects 1.6× less polluting labels than LUncov, but in 10× less time.

Conclusion: LClean provides a more extensive detection of polluting objectives than LUncov (especially as it goes beyond infeasibility) at cheaper cost, thanks to modularity and optimized implementation.

6.3.2 Mutant Classification. The core principle of mutant classification [57, 58] is to rely on dynamic coverage data to identify (in an approximated way) polluting mutants. As a comparison between LClean and such a dynamic pruning principle, Figure 9 reveals that the time necessary to run a high-coverage test suite (Siemens test suite), save coverage data and find likely-polluting objectives can be one order of magnitude higher than running LClean over the same test objectives. In the same time, it appeared that many of the objectives detected in this way were false positives, leading to a 89% rate of labels to be considered as likely polluting (mainly because of duplication and subsumption). Actually, while the Siemens test suite achieves high coverage of standard metrics, it is not built to reveal different coverage behaviours between feasible test objectives. Crafting new test cases to do so would reduce the number of false positives but even more penalize the computation time.

Conclusion: By relying on lightweight static analyses, LClean provides a sound and quick detection of a significant number of both infeasible and redundant test objectives, while dynamic detection is expensive and unsound, yielding many false positives even based on high-quality test suites.

Benchmark	Labels	STEP 1		STEP 2		STEP 3			TOTAL			Criterion
		marked as infeasible	*time*	*marked as duplicate*	*time*	*marked as duplicate*	*marked as subsumed*	*time*	*marked as polluting* ratio	*%*	*time*	
siemens	654	0	35s	0	38s	2	41	83s	43/654	7%	156s	CC
(agg. 7 programs)	666	20	36s	0	40s	0	16	78s	36/666	5%	154s	MCC
3210 locs	654	1	37s	0	39s	18	17	77s	36/654	6%	153s	GACC
	3543	37	114s	123	126s	134	336	723s	630/3543	18%	963s	WM
openssl	1022	28	67s	3	67s	4	57	391s	92/1022	9%	525s	CC
(agg. 4 programs)	1166	134	77s	0	83s	2	24	294s	160/1166	14%	454s	MCC
4596 locs	1022	29	70s	0	81s	30	24	324s	83/1022	8%	475s	GACC
	4978	252	356s	270	372s	200	326	4214s	1048/4978	21%	5122s	WM
gzip	1670	23	149s	5	152s	19	54	578s	101/1670	6%	879s	CC
7569 locs	1726	44	170s	5	171s	11	34	628s	94/1726	5%	969s	MCC
	1670	31	154s	5	156s	43	34	555s	113/1670	7%	865s	GACC
	12270	267	1038s	942	1210s	542	895	10029s	2646/12270	22%	12277s	WM
sjeng	4090	34	351s	15	354s	82	215	798s	346/4090	8%	1503s	CC
14070 locs	4746	358	417s	9	436s	34	26	1912s	427/4746	9%	2765s	MCC
	4090	35	349s	15	353s	82	210	751s	342/4090	8%	1453s	GACC
	25722	353	5950s	483	4791s	640	706	19586s	2182/25722	8%	31478s	WM
sqlite	20202	120	1907s	3	1416s	130	456	4646s	709/20202	4%	7969	CC
196888 locs	21852	394	2295s	0	1902s	178	255	11958s	827/21852	4%	16155	MCC
	20202	129	2065s	0	1613s	803	223	4773s	1155/20202	6%	8451	GACC
	90240	878	18104s	3688	13571s	2962	6164	216140s	13692/90240	15%	247815s	WM
TOTAL	27638	205	2509s	26	2027s	237	823	6496s	**1291/27638**	**5%**	**3h3m52**	**CC**
226333 locs	30156	950	2995s	14	2632s	225	355	14870s	**1544/30156**	**5%**	**5h41m37**	**MCC**
	27638	225	2675s	20	2242s	976	508	6480s	**1729/27638**	**6%**	**3h9m57**	**GACC**
	136753	1787	25562s	5506	20070s	4478	8427	250692s	**20198/136753**	**15%**	**82h18m44**	**WM**
	222185	**3167**	**9h22m21**	**5566**	**7h29m31**	**5916**	**10113**	**77h22m18**	**24762/222185**	**11%**	**94h14m10**	**TOTAL**

Figure 6: Pruning power and computation time of LClean over 14 various "real-world" C programs

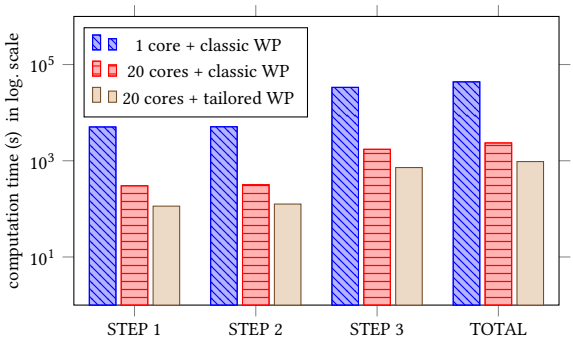


Figure 7: Tool optimization impact (Siemens, WM)

Criterion	LUncov		LClean (step 1)		LClean (all steps)	
	marked	*time*	*marked*	*time*	*marked*	*time*
CC	4/162	97s	4/162	12s	51/162	46s
MCC	30/203	125s	30/203	15s	51/203	53s
WM	84/905	801s	41/905	75s	385/905	463s
TOTAL	**9%** (×1)	**17m3s** (×1)	**6%** (÷1.6)	**1m42s** (÷10)	**38%** (×4.2)	**9m22s** (÷1.8)

Figure 8: LUncov [8] vs LClean (benchmarks from [8])

6.3.3 Trivial Compiler Equivalence (TCE). A direct comparison with TCE [53] is not possible, as TCE aims at identifying strong mutant equivalences, which are fundamentally different from the structural ones we handle. Killing strong mutants requires indeed the propagation of the mutated program states to the program outputs, which is more complex to formalize [20]. Thus, the only way to compare the two approaches is to assume that weakly polluting mutants are also strongly polluting ones. This assumption is true for the case of equivalent mutants, but not entirely true for the case of the duplicated mutants. Weakly duplicates mutants might not be strongly duplicates due to failed mutated state propagation. However, this is usually quite rare, as most weakly killed mutants propagate to the program outputs [47]. Nevertheless, we report these results for demonstrating the capabilities of the approaches and not for suggesting a way to detect redundant strong mutants.

To perform the comparison, we generated some strong mutants as well as our corresponding weak ones for the replace program. We selected only the replace program as our purpose here is to demonstrate the relative differences of the approaches: replace is one of the largest program from the Siemens suite, for which TCE performs best with respect to equivalent mutant detection [31]. Our results show that among the 1,579 mutants involved, our approach detected 103 (7%) as infeasible, while TCE detected 96 (6%). Among these, 91 are shared, which means that 12 of the infeasible mutants were only found by our approach and 5 only by TCE. Regarding duplicated mutants, our approach detected 555 (35%) duplicates, and TCE detected 352 (22%). 214 were shared, which means that both techniques together identify 693 (44%) duplicated mutants.

Conclusion: Overall, the results show that our approach outperforms TCE in terms of detection power and form a relatively good complement of it. Moreover, LClean is able to detect subsumption. Yet, TCE is much more efficient, relying on compiler optimizations.

7 DISCUSSION

7.1 Threats to Validity

Common to all studies relying on empirical data, this one may be of limited generalizability. To diminish this threat we used, in addition to the Siemens benchmark programs, four large real-world applications composed of more than 200 kloc (in total), like SQLite, and showed that our approach is capable of dealing with many types of polluting objectives, which no other approach can handle.

Our results might also have been affected by the choice of the chosen test criteria and in particular the specific mutation operators we employ. To reduce this threat, we used popular test criteria (CC, MCC, GACC and WM) included in software testing standards [55, 56], and employed commonly used mutation operators included in recent work [1, 16].

Criterion	Dynamic Detection					LClean				
	possibly infeasible	*possibly duplicate*	*possibly subsumed*	***total ratio for possibly polluting***	***time***	*marked as infeasible*	*marked as duplicate*	*marked as subsumed*	***total ratio for marked as polluting***	***time***
CC	37/654	243/654	230/654	80% (510/654)	3132s	0/654	2/654	41/654	7% (43/654)	156s
MCC	76/666	221/666	215/666	77% (512/666)	3142s	20/666	0/666	16/666	5% (36/666)	154s
GACC	46/654	249/654	212/654	78% (507/654)	3134s	1/654	18/654	17/654	6% (36/654)	153s
WM	386/3543	2327/3543	641/3543	95% (3354/3543)	8399s	37/3543	257/3543	336/3543	18% (630/3543)	963s
TOTAL	545/5517	3040/5517	1298/5517	**89%** (4883/5517)	**4h56m47** (×12)	58/5517	277/5517	410/5517	**14%** (745/5517)	**23m46s** (×1)

Figure 9: Dynamic detection of (likely) polluting objectives vs. LClean (Siemens)

The validity of our experimental results have been crosschecked in several ways. First, we compared our results on the Siemens benchmark with those of other tools, namely LUncov and TCE. We knew by design that infeasible objectives detected by LClean should be detected by LUncov as well, and we checked manually the status of each duplicate objective reported by LClean and not by TCE. No issue was reported. Second, we used the existing tests suites for the Siemens programs as a redundant sanity check, by verifying that every objective reported as infeasible (resp. duplicated, subsumed) by LClean was indeed seen as infeasible (resp. duplicated, subsumed) when running the test suite. These test suites are extremely thorough [30, 51] and are thus likely to detect errors in LClean. Third, for larger programs, we picked a random selection of a hundred test objectives reported as infeasible, duplicated or subsumed by LClean and manually checked them – this was often straightforward due to the local reasoning of LClean. All these sanity checks succeeded.

Another class of threats may arise because of the tools that we used, as it is likely that Frama-C or our implementation are defective. However, Frama-C is a mature tool with industrial applications in highly demanding fields (e.g., aeronautics) and thus, it is unlikely to cause important problems. Moreover, our sanity checks would have likely spotted such issues.

Finally, other threats may be due to the polluting nature of the objectives that we target. However, infeasible objectives are a well-known issue, usually acknowledged in the literature as one of the most time consuming tasks of the software testing process [2, 38, 53, 58], and redundant objectives have been stated as a major problem in both past and recent literature [37, 38, 52].

7.2 Limitations

Labels cannot address all white-box criteria. For example, dataflow criteria or full MCDC require additional expressive power [41]. Currently, parts of the infeasibility results from LClean could be lifted to these classes of objectives. On the other hand, it is unclear how it could be done for redundancy. Extending the present work to these criteria is an interesting future work direction.

From a more technical point of view, the detection of subsumption is limited more or less to basic blocks. While it already enables catching many cases, it might be possible to slightly extend the search while retaining scalability. In the same vein, the proofs are performed in LClean on a *per* function basis. This is a problem as it is often the case that a given function is always called within the same context, reducing its possible behaviors. Allowing a limited degree of contextual analysis (e.g., inlining function callers and/or callees) should allow to detect more polluting objectives while retaining scalability.

Finally, as we are facing an undecidable problem, our approach is sound, but not complete: SMT solvers might answer *unknown*. In that case, we may miss polluting objectives.

8 RELATED WORK

8.1 Infeasible Structural Objectives

Early research studies set the basis for identifying infeasible test objectives using constraint-based techniques [24, 48]. Offutt and Pan [48] suggested transforming the programs under test as a set of constraints that encode the test objectives. Then, by solving these constraints, it is possible to identify infeasible objectives (constraints with no solution) and test inputs. Other attempts use model checking [14, 15] to prove that specific structural test objectives (given as properties) are infeasible. Unfortunately, constraint-based techniques, as they require a complete program analysis, have the usual problems of the large (possibly infinite) numbers of involved paths, imprecise handling of program aliases [35] and the handling of non-linear constraints [3]. Model-checking faces precision problems because of the system modelling and scalability issues due to the large state space involved. On the other hand, we rely on a modular, hence not too expensive, form of weakest precondition calculus to ensure scalability.

Perhaps the closest works to ours are the ones by Beckman *et al.* [12], Baluda *et al.* [4–6] and Bardin *et al.* [8] that rely on weakest precondition. Beckman *et al.* proves infeasible program statements, Baluda *et al.* infeasible program branches and Bardin *et al.* infeasible structural test objectives. Apart from the side differences (Beckman *et al.* targets formal verification, Baluda *et al.* applies model refinement in combination to weakest precondition and Bardin *et al.* combines weakest precondition with abstract interpretation) with these works, our main objective here is to identify all types of polluting test objectives (not only infeasible ones) for real-world programs in a generic way, i.e. for most of the test criteria, including advanced ones such as multiple condition coverage and weak mutation. Another concern regards the scalability of the previous methods, which remains unknown under the combinatorial explosion of test objectives that mutation criteria introduce.

Other techniques attempt to combine infeasible test objectives detection techniques as a means to speed-up test generation and refine the coverage metric. Su *et al.* [61] combines symbolic execution with model checking to generate data flow test inputs. Baluda *et al.* [6] combines backward (using weakest precondition) and forward symbolic analysis to support branch testing and Bardin *et al.* [8, 9] combines weakest precondition with dynamic symbolic execution to support the coverage of structural test objectives. Although integrating such approaches with ours may result in additional benefits, our main objective here is to demonstrate that lightweight symbolic analysis techniques, such as weakest precondition, can be used to effectively tackle the general problem of polluting objectives for almost all structural test criteria in real-world settings.

Another line of research attempts diminishing the undesirable effects of infeasible paths in order to speed-up test generation.

Woodward *et al.* [64] suggested using some static rules called allegations to identify infeasible paths. Papadakis and Malevris [54] and Lapierre *et al.* [39] used a heuristic based on the k-shortest paths in order to select likely feasible paths. Ngo and Tan [45] proposed some execution trace patterns that witness likely infeasible paths. Delahaye *et al.* [19] showed that infeasibility is caused by the same reason for many paths and thus, devised a technique that given an infeasible path can identify other, potentially unexplored paths. All these methods indirectly support test generation and contrary to ours do not detect polluting test objectives.

8.2 Equivalent Mutants

Automatically determining mutant equivalence is an instance of the infeasibility problem and is undecidable [48]. There are numerous propositions on how to handle this problem, however most of them have only been evaluated on example programs and thus, their applicability and effectiveness remains unexplored [31]. Due to space constraints we discuss the most recent and relevant approaches. Details regarding the older studies can be found in the recent paper by Kintis *et al.* [31], which extensively covers the topic.

One of the most recent methods is the Trivial Compiler Optimization (TCE) [31, 53]. The method assumes that equivalent mutant instances can be identified by comparing the object code of the mutants. The approach works well (it can identify 30% of the equivalent mutants) as the compiler optimisations turn mutant equivalencies into the same object code. In contrast our approach uses state-of-the-art verification technologies (instead of compilers) and targets all types of polluting objectives.

Alternative to static heuristics are the dynamic ones. Grun *et al.* [26] and Schuler *et al.* [57] suggested measuring the impact of mutants on the program execution and program invariants in order to identify likely killable mutants. Schuler and Zeller [58] investigate a large number of candidate impact measures and found that coverage was the most appropriate. Along the same lines Kintis *et al.* [33] found that higher order mutants provide more accurate predictions than coverage. Overall, these approaches are unsound (they provide many false positives) and they depend on the underlying test suites. In contrast our approach is sound and static.

8.3 Duplicate and Subsumed Test Objectives

The problems caused by subsumed objectives have been identified a long time ago. Chusho introduced essential branches [17], or non-dominated branches [13], as a way to prevent the inflation of the branch coverage score caused by redundant branches. He also introduced a technique devising graph dominator analysis in order to identify the essential branches. Bertolino and Marré [13] also used graph dominator analysis to reduce the number of test cases needed to cover test objectives and to help estimate the remaining testing cost. Although these approaches identify the harmful effects of redundant objectives, they rely on graph analysis, which results in a large number of false positives. Additionally, they cannot deal with infeasible objectives.

In the context of mutation testing, Kintis *et al.* [32] identified the problem and showed that mutant cost reduction techniques perform well when using all mutants but not when using non-redundant ones. Amman *et al.* [1] introduced minimal mutants and dynamic mutant subsumption and showed that mutation testing tools generate a large number of subsumed mutants.

Although mutant redundancies were known from the early days of mutation testing [37], their harmful effects were only recently realised. Papadakis *et al.* [52] performed a large-scale study and demonstrated that subsumed mutants inflate the mutation score measurement. Overall, Papadakis *et al.* [52] showed that arbitrary experiments can result in different conclusions when they account for the cofounding effects of subsumed mutants. Similarly, Kurtz *et al.* [37, 38] compared selective mutation testing strategies and found that they perform poorly when the mutation score is free of redundant mutants.

Overall, most of the studies identify the problem but fail to deal with it. One attempt to reduce mutant redundancies uses TCE [31, 53] to remove duplicate mutants. Other attempts are due to Kurtz *et al.* [36] who devised differential symbolic execution to identify subsumed mutants. Gong *et al.* [25] used dominator analysis (in the context of weak mutation) in order to reduce the number of mutants. Unfortunately, both studies have limited scope as they have been evaluated only on example programs and their applicability and scalability remain unknown. Conversely, TCE is applicable and scalable, but it only targets specific kinds of subsumed mutants (duplicated ones) and cannot be applied on structural test objectives.

9 CONCLUSION

Software testing is the primary method for detecting software defects. In that context, polluting test objectives are well-known to be harmful to the testing process, potentially wasting the tester's efforts and misleading them on the quality of their test suites. We have presented LClean, the only approach to date that handles in a unified way the detection of (the three kinds of) polluting objectives for a large set of common criteria, together with a dedicated (open-source) tool able to prune out such polluting objectives. LClean reduces the problem of detecting polluting objectives to the problem of proving assertions in the tested code. The tool relies on weakest-precondition calculus and SMT solving to prove these assertions. It is built on top of the industry-proof Frama-C verification platform, specifically tuned to our scalability needs. Experiments show that LClean provides a useful, sound, scalable and adaptable means for helping testers to target high levels of coverage (where most faults are detected) and to evaluate more accurately the strength of their test suites (as well as of the tools possibly used to generate them). It could immediately benefit to all application developers that aim at specific test suite coverage levels in their current testing process, like for example in the well-known SQLite database management system. A promising direction for future work is the extension of LClean to the few remaining unsupported classes of test objectives, like data-flow criteria.

ACKNOWLEDGMENTS

We thank the anonymous reviewers of ICSE'2018 for their valuable comments and remarks.

REFERENCES

[1] Paul Ammann, Márcio Eduardo Delamaro, and Jeff Offutt. 2014. Establishing Theoretical Minimal Sets of Mutants. In *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*. 21–30.

[2] Paul Ammann and Jeff Offutt. 2008. *Introduction to Software Testing* (1 ed.). Cambridge University Press.

[3] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John A. Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. 2013. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software* 86, 8 (2013), 1978–2001.

[4] Mauro Baluda, Pietro Braione, Giovanni Denaro, and Mauro Pezzè. 2010. Structural coverage of feasible code. In *The 5th Workshop on Automation of Software Test, AST 2010, May 3-4, 2010, Cape Town, South Africa*. 59–66.

[5] Mauro Baluda, Pietro Braione, Giovanni Denaro, and Mauro Pezzè. 2011. Enhancing structural software coverage by incrementally computing branch executability. *Software Quality Journal* 19, 4 (2011), 725–751.

[6] Mauro Baluda, Giovanni Denaro, and Mauro Pezzè. 2016. Bidirectional Symbolic Analysis for Effective Branch Testing. *IEEE Trans. Software Eng.* 42, 5 (2016), 403–426.

[7] Sébastien Bardin, Omar Chebaro, Mickaël Delahaye, and Nikolai Kosmatov. 2014. An All-in-One Toolkit for Automated White-Box Testing. In *TAP*. Springer.

[8] Sébastien Bardin, Mickaël Delahaye, Robin David, Nikolai Kosmatov, Mike Papadakis, Yves Le Traon, and Jean-Yves Marion. 2015. Sound and Quasi-Complete Detection of Infeasible Test Requirements. In *ICST*.

[9] Sébastien Bardin, Nikolai Kosmatov, and François Cheynier. 2014. Efficient Leveraging of Symbolic Execution to Advanced Coverage Criteria. In *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on (pp. 173-182). IEEE.*

[10] Mike Barnett and Rustan Leino. 2005. Weakest-Precondition of Unstructured Programs. In *ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE)*. 82–87.

[11] Patrick Baudin, Pascal Cuoq, Jean C. Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. [n. d.]. *ACSL: ANSI/ISO C Specification Language*. http://frama-c.com/acsl.html

[12] Nels E. Beckman, Aditya V. Nori, Sriram K. Rajamani, Robert J. Simmons, SaiDeep Tetali, and Aditya V. Thakur. 2010. Proofs from Tests. *IEEE Trans. Software Eng.* 36, 4 (2010), 495–508.

[13] Antonia Bertolino and Martina Marré. 1994. Automatic Generation of Path Covers Based on the Control Flow Analysis of Computer Programs. *IEEE Trans. Software Eng.* 20, 12 (1994), 885–899.

[14] Dirk Beyer, Adam Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. 2004. Generating Tests from Counterexamples. In *26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom*. 326–335.

[15] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. 2007. The software model checker Blast. *STTT* 9, 5-6 (2007), 505–525.

[16] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. 2017. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. 597–608.

[17] Takeshi Chusho. 1987. Test Data Selection and Quality Estimation Based on the Concept of Esssential Branches for Path Testing. *IEEE Trans. Software Eng.* 13, 5 (1987), 509–517.

[18] Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability Modulo Theories: Introduction and Applications. *Commun. ACM* 54, 9 (Sept. 2011), 69–77.

[19] Mickaël Delahaye, Bernard Botella, and Arnaud Gotlieb. 2015. Infeasible path generalization in dynamic symbolic execution. *Information & Software Technology* 58 (2015), 403–418.

[20] Richard A. DeMillo and A. Jefferson Offutt. 1991. Constraint-Based Automatic Test Data Generation. *IEEE Trans. Software Eng.* 17, 9 (1991), 900–910.

[21] E. W. Dijkstra. 1976. *A Discipline of Programming*. Prentice Hall.

[22] Phyllis G. Frankl and Oleg Iakounenko. 1998. Further Empirical Studies of Test Effectiveness. In *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '98/FSE-6)*. ACM, New York, NY, USA, 153–162.

[23] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. 2015. Guidelines for Coverage-Based Comparisons of Non-Adequate Test Suites. *ACM Trans. Softw. Eng. Methodol.* 24, 4 (2015), 22:1–22:33.

[24] Allen Goldberg, Tie-Cheng Wang, and David Zimmerman. 1994. Applications of Feasible Path Analysis to Program Testing. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis, ISSTA 1994, Seattle, WA, USA, August 17-19, 1994*. 80–94.

[25] Dunwei Gong, Gongjie Zhang, Xiangjuan Yao, and Fanlin Meng. 2017. Mutant reduction based on dominance relation for weak mutation testing. *Information & Software Technology* 81 (2017), 82–96.

[26] Bernhard J. M. Grün, David Schuler, and Andreas Zeller. 2009. The Impact of Equivalent Mutants. In *Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, Colorado, USA, April 1-4, 2009, Workshops Proceedings*. 192–199.

[27] GZip (SPEC) 2018. https://www.spec.org/cpu2000/CINT2000/164.gzip/docs/164.gzip.html. (2018).

[28] C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (October 1969), 576–580 and 583.

[29] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. 1994. Experiments of the Effectiveness of Dataflow- and Controlflow-based Test Adequacy Criteria. In *Proceedings of the 16th International Conference on Software Engineering (ICSE '94)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 191–200.

[30] Monica Hutchins, Herbert Foster, Tarak Goradia, and Thomas J. Ostrand. 1994. Experiments of the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria. In *Proceedings of the 16th International Conference on Software Engineering*. 191–200.

[31] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. Le Traon, and M. Harman. 2017. Detecting Trivial Mutant Equivalences via Compiler Optimisations. *IEEE Transactions on Software Engineering* PP, 99 (2017), 1–1.

[32] Marinos Kintis, Mike Papadakis, and Nicos Malevris. 2010. Evaluating Mutation Testing Alternatives: A Collateral Experiment. In *17th Asia Pacific Software Engineering Conference, APSEC 2010, Sydney, Australia, November 30 - December 3, 2010*. 300–309.

[33] Marinos Kintis, Mike Papadakis, and Nicos Malevris. 2015. Employing second-order mutation for isolating first-order equivalent mutants. *Softw. Test., Verif. Reliab.* 25, 5-7 (2015), 508–535.

[34] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A Program Analysis Perspective. *Formal Aspects of Computing Journal* (2015).

[35] Nikolai Kosmatov. 2008. All-Paths Test Generation for Programs with Internal Aliases. In *19th International Symposium on Software Reliability Engineering (ISSRE 2008), 11-14 November 2008, Seattle/Redmond, WA, USA*. 147–156.

[36] Bob Kurtz, Paul Ammann, and Jeff Offutt. 2015. Static analysis of mutant subsumption. In *Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015*. 1–10.

[37] Bob Kurtz, Paul Ammann, Jeff Offutt, Márcio Eduardo Delamaro, Mariet Kurtz, and Nida Gökçe. 2016. Analyzing the validity of selective mutation with dominator mutants. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. 571–582.

[38] Bob Kurtz, Paul Ammann, Jeff Offutt, and Mariet Kurtz. 2016. Are We There Yet? How Redundant and Equivalent Mutants Affect Determination of Test Completeness. In *Ninth IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2016, Chicago, IL, USA, April 11-15, 2016*. 142–151.

[39] Sébastien Lapierre, Ettore Merlo, Gilles Savard, Giuliano Antoniol, Roberto Fiutem, and Paolo Tonella. 1999. Automatic Unit Test Data Generation Using Mixed-Integer Linear Programming and Execution Trees. In *1999 International Conference on Software Maintenance, ICSM 1999, Oxford, England, UK, August 30 - September 3, 1999*. 189–198.

[40] Michaël Marcozzi, Sébastien Bardin, Mickaël Delahaye, Nikolai Kosmatov, and Virgile Prevosto. 2017. Taming Coverage Criteria Heterogeneity with LTest. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*. 500–507.

[41] Michaël Marcozzi, Mickaël Delahaye, Sébastien Bardin, Nikolai Kosmatov, and Virgile Prevosto. 2017. Generic and Effective Specification of Structural Test Objectives. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*. 436–441.

[42] Aditya P. Mathur. 2008. *Foundations of Software Testing*. Addison-Wesley Prof.

[43] Glenford J. Myers and Corey Sandler. 2004. *The Art of Software Testing*. John Wiley & Sons.

[44] Glenford J. Myers, Corey Sandler, and Tom Badgett. 2011. *The Art of Software Testing* (3 ed.). Wiley.

[45] Minh Ngoc Ngo and Hee Beng Kuan Tan. 2008. Heuristics-based infeasible path detection for dynamic test data generation. *Information & Software Technology* 50, 7-8 (2008), 641–655.

[46] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. 1996. An Experimental Determination of Sufficient Mutant Operators. *ACM Trans. Softw. Eng. Methodol.* 5, 2 (April 1996), 99–118.

[47] A. Jefferson Offutt and Stephen D. Lee. 1994. An Empirical Evaluation of Weak Mutation. *IEEE Trans. Software Eng.* 20, 5 (1994).

[48] A. Jefferson Offutt and Jie Pan. 1997. Automatically Detecting Equivalent Mutants and Infeasible Paths. *Softw. Test., Verif. Reliab.* 7, 3 (1997), 165–192.

[49] OpenSSL 2018. https://www.openssl.org. (2018).

[50] Rahul Pandita, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. 2010. Guided Test Generation for Coverage Criteria. In *ICSM*.

[51] Mike Papadakis, Márcio Eduardo Delamaro, and Yves Le Traon. 2014. Mitigating the effects of equivalent mutants with mutant classification strategies. *Sci. Comput. Program.* 95 (2014), 298–319.

[52] Mike Papadakis, Christopher Henard, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Threats to the Validity of Mutation-based Test Assessment. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, New York, NY, USA, 354–365.

[53] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. 2015. Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 936–946.

[54] Mike Papadakis and Nicos Malevris. 2012. Mutation based test case generation via a path selection strategy. *Information & Software Technology* 54, 9 (2012), 915–932.

[55] Radio Technical Commission for Aeronautics. 1992. RTCA DO178-B Software Considerations in Airborne Systems and Equipment Certification. (1992).

[56] Stuart C. Reid. 1995. The Software Testing Standard — How you can use it. In *3^{rd} European Conference on Software Testing, Analysis and Review (EuroSTAR '95)*. London.

[57] David Schuler, Valentin Dallmeier, and Andreas Zeller. 2009. Efficient mutation testing by checking invariant violations. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009*. 69–80.

[58] David Schuler and Andreas Zeller. 2013. Covering and Uncovering Equivalent Mutants. *Softw. Test., Verif. Reliab.* 23, 5 (2013), 353–374.

[59] SJeng (SPEC) 2018. https://www.spec.org/cpu2006/Docs/458.sjeng.html. (2018).

[60] SQLite 2018. https://www.sqlite.org. (2018).

[61] Ting Su, Zhoulai Fu, Geguang Pu, Jifeng He, and Zhendong Su. 2015. Combining Symbolic Execution and Model Checking for Data Flow Testing. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. 654–665.

[62] E. J. Weyuker. 1993. More Experience with Data Flow Testing. *IEEE Trans. Softw. Eng.* 19, 9 (Sept. 1993), 912–919.

[63] M. R. Woodward, D. Hedley, and M. A. Hennell. 1980. Experience with Path Analysis and Testing of Programs. *IEEE Trans. Softw. Eng.* 6, 3 (May 1980), 278–286.

[64] Martin R. Woodward, David Hedley, and Michael A. Hennell. 1980. Experience with Path Analysis and Testing of Programs. *IEEE Trans. Software Eng.* 6, 3 (1980), 278–286.

[65] D. Yates and N. Malevris. 1989. Reducing the Effects of Infeasible Paths in Branch Testing. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3)*. ACM, New York, NY, USA, 48–54.

[66] Hong Zhu, Patrick A. V. Hall, and John H. R. May. 1997. Software Unit Test Coverage and Adequacy. *ACM Comput. Surv.* 29, 4 (1997).