Sound and Quasi-Complete Detection of Infeasible Test Requirements*

Sébastien Bardin*, Mickaël Delahaye*, Robin David*, Nikolai Kosmatov*, Mike Papadakis†, Yves Le Traon† and Jean-Yves Marion‡
*CEA, LIST, 91191, Gif-sur-Yvettes, France
†Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg
‡Université de Lorraine, CNRS and Inria, LORIA, France
*firstname.lastname@cea.fr, †michail.papadakis@uni.lu, †yves.letraon@uni.lu and ‡jean-yves.marion@loria.fr

***Abstract*—In software testing, coverage criteria specify the requirements to be covered by the test cases. However, in practice such criteria are limited due to the well-known infeasibility problem, which concerns elements/requirements that cannot be covered by any test case. To deal with this issue we revisit and improve state-of-the-art static analysis techniques, such as Value Analysis and Weakest Precondition calculus. We propose a lightweight greybox scheme for combining these two techniques in a complementary way. In particular we focus on detecting infeasible test requirements in an automatic and sound way for condition coverage, multiple condition coverage and weak mutation testing criteria. Experimental results show that our method is capable of detecting almost all the infeasible test requirements, 95% on average, in a reasonable amount of time, i.e., less than 40 seconds, making it practical for unit testing.**

***Keywords*—structural coverage criteria, infeasible test requirements, static analysis, weakest precondition, value analysis**

I. Introduction

For most safety critical unit components, the quality of the test cases is assessed through the use of some criteria known as *coverage* (or *testing*) *criteria*. Unit testing is mainly concerned with structural coverage criteria. These coverage criteria are normative test requirements that the tester must satisfy before delivering the software component under test.

In practice, the task of the tester is tedious, not only because he has to generate test data to reach the criterion expectations but mainly because he must justify why a certain test requirement cannot be covered. Indeed it is likely that some requirements cannot be covered due to the semantics of the program. We refer to these requirements as *infeasible*, and as *feasible* in the opposite case. The work that we present here, aims at making this justification automatic. We propose a generic and lightweight tooled technique that extends the LTest testing toolkit [6] with a component, called LUncov, dedicated to the detection of infeasible test requirements. The approach stands for any piece of software code (in particular C) that is submitted to strict test coverage expectations such as condition coverage, multiple condition coverage and weak mutation.

Coverage criteria thus define a set of requirements that should be fulfilled by the employed test cases. If a test case fulfills one or more of the test criterion requirements, we say that it *covers* them. Failing to cover some of the criterion requirements indicates a potential weakness of the test cases and hence, some additional test cases need to be constructed.

Infeasible test requirements have long been recognized as one of the main cost factors of software testing [40], [37], [42]. Weyuker [37] identified that such cost should be leveraged and reduced by automated detection techniques. This issue is due to following three reasons. First, resources are wasted in attempts to improve test cases with no hope of covering these requirements. Second, the decision to stop testing is made impossible if the knowledge of what could be covered remains uncertain. Third, since identifying them is an undecidable problem [17], they require time consuming manual analysis. In short, the effort that should be spent in testing is wasted in understanding why a given requirement cannot be covered.

By identifying the infeasible test requirements, such as equivalent mutants, testers can accurately measure the coverage of their test suites. Thus, they can decide with confidence when they should stop the testing process. Additionally, they can target full coverage. According to Frankl and Iakounenko [14] this is desirable since the majority of the faults are triggered when covering higher coverage levels, i.e., from 80% to 100% of decision coverage.

Despite the recent achievements with respect to the test generation problem [2], the infeasible requirements problem remains open. Indeed, very few approaches deal with this issue. Yet, none suggests any practical solution to it. In this paper we propose a heuristic method to deal with the infeasible requirements for several popular structural testing criteria. Our approach is based on the idea that the problem of detecting infeasible requirements can be transformed into the assertion validity problem. By using program verification techniques, it becomes possible to address and solve this problem.

We use *labels* [7], [6] to encode several structural testing criteria and implement a unified solution to this problem *based on existing verification tools*. In this study, we focus on *sound approaches*, i.e., identifying as infeasible only requirements that are indeed infeasible. Specifically, we consider two methods, the (forward) *Value Analysis* and the (backward) *Weakest Precondition* calculus. Value Analysis computes an over-approximation of all reachable program states while Weakest Precondition starts from the assertion to check and computes in a backward manner a proof obligation equivalent to the validity of the assertion.

We consider these approaches since they are representative

* Work partially funded by EU FP7 (project STANCE, grant 317753) and French ANR (project BINSEC, grant ANR-12-INSE-0002).

of current (sound) state-of-the-art verification technologies. Moreover, due to their nature, they are complementary and mutually advantageous to one another. We use *existing analyzers*, either in a pure *blackbox* manner or with light (*greybox*) combination schemes.

In summary our main contributions are:

- We revisit static analysis approaches with the aim of identifying infeasible test requirements. We classify these techniques as *State Approximation Computation*, such as Value Analysis, and *Goal-Oriented Checking*, such as Weakest Precondition.
- We propose a new method that combines two such analyzers in a greybox manner. The technique is based on easy-to-implement API functionalities on the State Approximation Computation and Goal-Oriented Checking tools. More importantly, it significantly outperforms each one of the combined approaches alone.
- We demonstrate that static analysis can detect almost all the infeasible requirements of the condition coverage, multiple condition coverage and weak mutation testing criteria. In particular, the combined approach identifies on average more than 95% of the infeasible requirements, while Value Analysis detects on average 63% and Weakest Precondition detects on average 82%. Computation time is very low when detection is performed after test generation (in order to evaluate coverage precisely), and we show how to keep it very reasonable (less than 40 seconds) if performed beforehand (in order to help the test generation process).
- We show that by identifying infeasible requirements before the test generation process we can speed-up the automated test generation tools. Results from an automated test generation technique, DSE* [7], show that it can be more than $\sim$55$\times$ faster in the best case and approximately $\sim$3.8$\times$ faster on the average case (including infeasibility detection time).

The rest of the paper is organized as follows: Sections II and III respectively present some background material and how static analysis techniques can be used to detect infeasible requirements. Section IV details our combined approach and its implementation. While section V describes the empirical study, Section VII discusses its implications. Finally, related work and conclusions are given in Sections VI and VIII.

II. Background

This section presents some definitions, the notation related to test requirements and the employed tools.

A. Test Requirements as 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., $t \in D$. The execution of P over t, denoted $P(t)$, is a run $\sigma \triangleq \langle (loc_1, s_1), \ldots, (loc_1, s_1) \rangle$ where the loc_i denote control-locations (or simply locations) of P and the s_i denote the successive internal states of P ($\approx$ valuation of all global and local variables as well as memory-allocated structures) before the execution of the corresponding loc_i. A test datum t reaches a location loc with internal state s, if $P(t)$ is of the form $\sigma \cdot \langle loc, s \rangle \cdot \rho$. A test suite $TS \subseteq D$ is a finite set of test data.

Recent work [7] proposed the notion of labels as an expressive and convenient formalism to specify test requirements. Given a program P, a *label* l is a pair $\langle loc, \varphi \rangle$ where loc is a location in P and φ is a predicate over the internal state at loc. We say that a test datum t *covers a label* $l = \langle loc, \varphi \rangle$ if there is a state s such that t reaches $\langle loc, s \rangle$ and s satisfies φ. An *annotated program* is a pair $\langle P, L \rangle$ where P is a program and L is a set of labels in P.

It has been shown that labels can encode test requirements for most standard coverage criteria [7], such as decision coverage (DC), condition coverage (CC), multiple condition coverage (MCC), and function coverage as well as side-effect-free weak mutations (WM) and GACC [29] (a weakened form of MCDC). Moreover, these encoding can be fully automated, as the corresponding labels can be inserted automatically into the program under test. Some more complex criteria such as MCDC or strong mutations cannot be encoded by labels. Fig. 1 illustrates possible encodings for selected criteria.

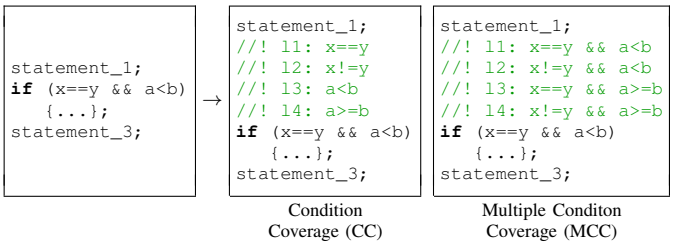

Fig. 1. A label encoding of test requirements for standard coverage criteria

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. While the question of automatic test generation was explored in the original paper on labels [7], we focus in this paper on the problem of infeasibility detection.

B. FRAMA-C *Framework and* LTEST *Toolset for Labels*

This work relies on FRAMA-C [22] for our experimental studies. This is an open-source framework for analysis of C code. FRAMA-C provides an extensible plugin-oriented architecture for analysis collaboration, and comes with various analyzers that can share analysis results and communicate through a common abstract syntax tree (AST) and a common specification language ACSL [22] to express annotations. In our context, FRAMA-C offers the following advantages: it implements two different (sound) verification techniques in the same environment (Abstract Interpretation and Weakest Precondition, cf. Section III-C), it is open-source[1], robust and already used in several industrial contexts [22, Sec. 11].

We also rely on and further extend LTEST [6], an all-in-one testing platform for C programs annotated with labels, developed as a FRAMA-C plugin[2]. LTEST provides the following services: (1) annotation of a given C program with labels according to chosen coverage criteria; (2) replay of a given test suite and coverage reporting; (3) detection of infeasible labels

[1]Available at `http://frama-c.com/`.
[2]Available at `http://micdel.fr/ltest.html`.

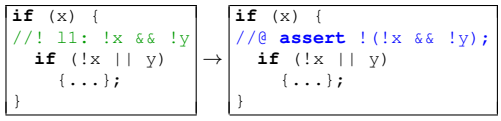

Fig. 2. Transforming an (infeasible) test requirement into a (valid) assertion

based on sound analysis of the source program (Forward Value Analysis); and finally (4) automatic test generation through the DSE* procedure [7], an extension of Dynamic Symbolic Execution [16], [35], [38] which handles labels in a native and very optimized way. Interestingly, all services can cooperate together by sharing a database of label statuses. For example, test generation takes advantage of the results of infeasibility detection in order to prune its own search space [6]. The infeasibility detection service (LUNCOV) is extended by the present work.

III. PROOF OF ASSERTION VALIDITY

In the sequel, we assume that test requirements are expressed in terms of labels. They can be either automatically inserted for common classes of structural test requirements (cf. Sec. II-B, tool LTEST), or manually added for some very specific test purposes. This section gives a characterization of infeasible test requirements in terms of valid assertions, provides a brief presentation and classification of existing static analysis techniques for their detection, and describes the specific techniques and tools we use throughout the paper.

A. Infeasible Test Requirements as Valid Assertions

Let $\langle P, L\rangle$ be an annotated program. A label $l \in L$ (and the corresponding test requirement) is called *feasible* if there exists a test datum covering l. Otherwise l is called *infeasible*. To avoid any confusion between the original label and its proven counterpart, along with labels we will use the dual notion of assertions. Syntactically, an *assertion* a is also a pair of the form $\langle loc, \psi\rangle$. The assertion a is *valid*[3] if it cannot be false, i.e. if there does not exist any test datum t such that t covers $\langle loc, \neg\psi\rangle$. Given a label $l = \langle loc, \varphi\rangle \in L$, let a_l denote the assertion $\langle loc, \neg\varphi\rangle$, that is, the negation of the predicate φ of l asserted at the same location loc. For a set of labels L over P, we define the set of assertions $A_L = \{\, a_l \,|\, l \in L \,\}$. The following lemma easily follows from the definitions.

Lemma 1. *A label l is infeasible if and only if the assertion a_l is valid. The problem of detecting infeasible labels in L is equivalent to the problem of detecting valid assertions in A_L.*

Fig. 2 illustrates the transformation of a test requirement, expressed as a label in a C code, into an assertion expressed in the ACSL specification language. In this example, the label is infeasible and the assertion is valid.

B. Classification of Sound Static Analyses

The assertion validity problem can often be solved by static analysis techniques. To do so, given a program P with a set of assertions A, such a technique needs to offer a validity checking procedure

$$\mathcal{V}_P : A \to \{\mathbf{1}, \mathbf{0}, ?\}$$

[3]A different definition is sometimes used in the literature, where validity also includes reachability.

mapping an assertion to one of three possible verdicts: valid ($\mathbf{1}$), invalid ($\mathbf{0}$) or unknown ($?$). We consider here only sound analysis techniques. In this context, $\mathcal{V}_P$ is *sound* if whenever the procedure outputs a verdict $\mathcal{V}_P(a) = \mathbf{1}$ (resp., $\mathcal{V}_P(a) = \mathbf{0}$), the assertion a is indeed valid (resp., invalid). Since the assertion validity problem is undecidable (cf. Sec. I), such procedures are in general *incomplete* and may return the unknown verdict.

We review and classify such techniques into two categories: State Approximation Computation and Goal-Oriented Checking. These two categories are built on orthogonal approaches (eager exploration vs property-driven analysis).

State Approximation Computation. The objective of a State Approximation Computation (SAC) technique is to compute an over-approximation of the set of reachable states of the given program at each program location. Let $\mathbb{S}$ denote the set of over-approximations S that can be computed by a particular SAC technique. In order to be applicable for assertion validity checking, SAC should provide the following procedures:

$$\begin{array}{rrl} \text{Analysis} & \mathcal{A}^{SAC} : & \mathbb{P} \to \mathbb{S} \\ \text{Implication check} & \mathcal{I}^{SAC} : & \mathbb{S} \times A \to \{\mathbf{1}, \mathbf{0}, ?\} \end{array}$$

The analysis procedure $\mathcal{A}^{SAC}$ computes a state over-approximation $\mathcal{A}^{SAC}(P)$ for a given program P. The implication (or state inclusion) checking procedure $\mathcal{I}^{SAC}$ determines if a given assertion a is implied by a given reachable state approximation S. It returns a verdict $\mathcal{I}^{SAC}(S, a)$ stating if the procedure was able to deduce from the state approximation S that the assertion a is valid ($\mathbf{1}$), invalid ($\mathbf{0}$) or if the result was inconclusive ($?$). The implication check depends on the specific form of the state approximation S. Based on these procedures, the assertion validity check is easily defined as follows:

$$\begin{array}{rrl} \text{SAC validity check} & \mathcal{V}_P^{SAC} : & A \to \{\mathbf{1}, \mathbf{0}, ?\} \\ & & a \mapsto \mathcal{I}^{SAC}(\mathcal{A}^{SAC}(P), a) \end{array}$$

An important characteristic of this class of techniques is that the analysis step needs to be executed only once for a given program P even if there are several assertions in A. Indeed, only the implication check $\mathcal{I}^{SAC}(\mathcal{A}^{SAC}(P), a)$ should be executed separately for each assertion $a \in A$, based on the same state approximation $\mathcal{A}^{SAC}(P)$. Typical State Approximation Computation implementations include (forward) abstract interpretation based tools, such as implemented in ASTRÉE [11] and Clousot [13], and software model checking tools, such as BLAST [9] and SLAM [4].

Goal-Oriented Checking. The second category includes Goal-Oriented Checking (GOC) techniques that perform a specific analysis for each assertion to be proved valid. Unlike for State Approximation Computation, such an analysis can be simpler since it is driven by the assertion to check, yet it should be repeated for each assertion. We can thus represent a GOC technique directly by the procedure

$$\begin{array}{rrl} \text{GOC validity check} & \mathcal{V}_P^{GOC} : & A \to \{\mathbf{1}, \mathbf{0}, ?\} \end{array}$$

that returns the validity status $\mathcal{V}_P^{GOC}(a)$ for an assertion $a \in A$. Typical Goal-Oriented Checking tools include weakest precondition checkers, backward abstract state exploration, backward bounded model checking, and also CEGAR software model checkers (which belong to both GOC and SAC).

C. Choice for SAC and GOC Tools

Choice for State Approximation Computation. We select the value analysis plugin VALUE of FRAMA-C [22]. VALUE is a forward data-flow analysis based on the principles of abstract interpretation [10], which performs a whole-program analysis based on non-relational numerical domains (such as intervals and congruence information) together with a byte-level region-based memory model. VALUE computes a sound approximation S of values of all program variables at each program location, that can be used to deduce the validity of an assertion $a = \langle loc, \psi \rangle$.

Choice for Goal-Oriented Checking. We select WP [22], a FRAMA-C plugin for weakest precondition calculus. It takes as input a C program annotated in ACSL. From each ACSL annotation, WP generates a formula (also called *proof obligation*) that, if proven, guarantees that the annotation is valid. It may then be proved automatically via an SMT solver (Alt-Ergo, Z3, CVC4). WP can take advantage of handwritten ACSL assertions, yet since we are interested in fully automated approaches, we do not add anything manually. Notice that WP considers each function independently, and, therefore, is more scalable than non modular analysis like VALUE.

IV. GREYBOX COMBINATION OF STATE APPROXIMATION COMPUTATION AND GOAL-ORIENTED CHECKING

This section presents a greybox combination of State Approximation Computation and Goal-Oriented Checking techniques that we propose for detecting valid assertions. We first present our combination in a generic way, for any SAC and GOC procedures (SAC $\oplus$ GOC). Ultimately, we use it to combine Value Analysis and Weakest Precondition (VA$\oplus$WP), and more especially the FRAMA-C plugins VALUE and WP.

A. The Combined Method

Intuition. The main idea of this new combination is to strengthen Goal-Oriented Checking with additional properties computed by State Approximation Computation, but taking only properties relevant to the assertion whose validity is to be checked. These properties must be put into a form that GOC can use. We will call them *hypotheses*, or *assumes*. Syntactically, hypotheses are pairs $\langle loc, \psi \rangle$ like assertions.

Yet, a State Approximation Computation produces lots of information about variable values at different program points, that can often be irrelevant for a particular assertion. Providing a lot of irrelevant or redundant information to Goal-Oriented Checking would make it less efficient. Our first point is thus to determine which variables are relevant to a given assertion, and at which program points they are. Then, relevant variables are used to produce a set of hypotheses, that will be assumed and used during the GOC step.

Extended API. Let P be a program, and A a set of assertions in P whose validity we want to check. Let us denote by $\mathbb{L}$ the set of locations in P, and by $\mathbb{V}$ the set of variables in P. We represent relevant variables (and locations) for an assertion a by a subset $R_a = \{(loc_1, v_1), \ldots, (loc_n, v_n)\}$ of $\mathbf{Set}(\mathbb{L} \times \mathbb{V})$. That is, R_a is a set of couples (loc_i, v_i) such that the value of the variable v_i at location loc_i is considered relevant to the assertion a.

In addition to the general API for SAC and GOC techniques (introduced in Sec. III-B), the greybox combination relies on some extensions. First, we require an enhanced GOC validity check taking into account hypotheses:

$$\text{GOC validity check} \qquad \mathcal{V}_P^{GOC} : \quad \mathbb{H} \times A \to \{\mathbf{1}, \mathbf{0}, ?\},$$

where $\mathbb{H}$ denotes possible sets of hypotheses. Second, we require two additional procedures. The first one computes the set of relevant variables R_a (defined above) for the considered GOC technique for each assertion a:

$$\text{Relevant variables} \qquad \mathcal{R}_P^{GOC} : \quad A \to \mathbf{Set}(\mathbb{L} \times \mathbb{V}) \qquad a \mapsto R_a$$

Given a computed state approximation S and a set of relevant variables, the second procedure

$$\text{Hypotheses creation} \qquad \mathcal{H}_P^{SAC} : \quad \mathbb{S} \times \mathbf{Set}(\mathbb{L} \times \mathbb{V}) \to \mathbb{H} \qquad (S, R) \mapsto H$$

deduces a set of verified properties H for these variables and locations that will be used as hypotheses by the GOC analysis step. For instance, a pair $(loc_i, v_i) \in R$ may lead to a hypothesis $(loc_i, m \leq v_i \leq M)$ if the state approximation S guarantees this interval of values for the variable v_i at location loc_i. Here again, we consider only sound hypotheses creation procedures.

The method steps. The complete method SAC $\oplus$ GOC is depicted in Fig. 3, where the boxes denote the main steps, while their output, assumed to be when necessary also part of the next steps' input, is indicated on the arrow. First, a SAC analysis step ($\mathcal{A}^{SAC}$) computes a state approximation S. This step is performed only once, while the other ones are executed once for each assertion $a \in A$. Next, if the implication check ($\mathcal{I}^{SAC}$) returns $\mathbf{1}$ showing that a is valid, the method terminates. Otherwise, the greybox part starts by extracting the set R_a of variables and locations relevant to a. Next, the set R_a and the previously computed approximation S are used to deduce properties of relevant variables that will be submitted as hypotheses to the last step. Finally, a GOC analysis step ($\mathcal{V}_P^{GOC}$) checks if the assertion a can be proven valid using the additional hypotheses in H.

Advantages. The proposed combined approach takes benefit both from the global precision of the approximation computed by SAC for the whole program and the local precision of analysis for a given assertion ensured by GOC. Therefore, this technique can be expected to provide a better precision than the two methods used separately. Careful selection of information transferred from SAC to GOC tries to minimize information exchange: the amount of useless (irrelevant or redundant) data is reduced thanks to the greybox combination. On the other hand, even if not being blackbox, the greybox part remains lightweight and non-invasive: only basic knowledge of the GOC technique is required to implement the $\mathcal{R}_P^{GOC}$ step, and only basic knowledge of datastructures and contents of approximations computed by SAC is necessary to query them and to produce hypotheses at the $\mathcal{H}_P^{SAC}$ step. Neither SAC nor GOC requires any modification of the underlying algorithms. Moreover, the approximation S is computed only once, and then used for all assertions. These elements constitute the cornerstone of the proposed combination of the two methods.

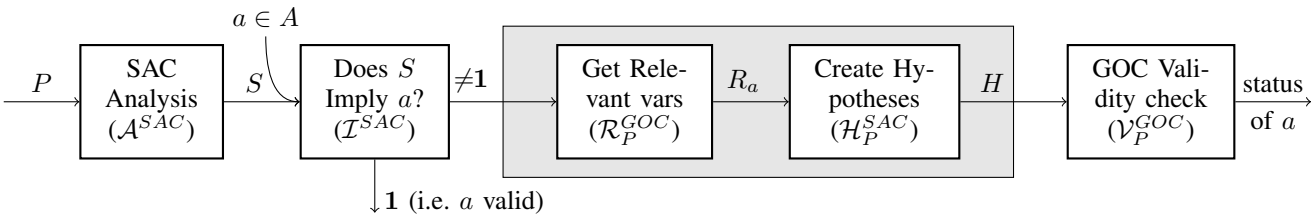

Fig. 3. Computing assertion validity status by the combined approach SAC $\oplus$ GOC for a given program P and all given assertions $a \in A$

Regarding soundness of the approach, notice that soundness of the $\mathcal{R}_P^{GOC}$ step is not required: various heuristics can be used to select relevant variables and locations that only impact the number and location of the future hypotheses and not their correctness. Yet, as mentioned above, the hypotheses construction $\mathcal{H}_P^{SAC}$ as well as the procedures defined in Sec. III-B are required to be sound. Soundness of the combined approach easily follows from soundness of both techniques:

Theorem 2 (Soundness)**.** *Assume that the combined techniques SAC and GOC are both sound. Then* SAC $\oplus$ GOC *is a sound Goal-Oriented Checking technique.*

B. Implementation in FRAMA-C: VA⊕WP

The proposed SAC$\oplus$GOC method is implemented on top of the FRAMA-C framework, and extends the label infeasibility detection service LUNCOV of LTEST (cf. Sec. II-B). We use the VALUE and WP plugins as implementations of SAC and GOC, respectively, and we denote by VA⊕WP their combination. These plugins have been presented in Sec. III-C. They readily offer the requested API for SAC and GOC analyzers.

Let us provide more implementation details on the greybox combination, which is now implemented in the LUNCOV plugin following the formal presentation of Sec. IV-A. The possibility for several FRAMA-C analyzers to work on the same program in different projects and to communicate through annotations (e.g. assertions, assumes) in the common specification language ACSL [22] significantly facilitates the implementation. Basically, hypotheses H are implemented as ACSL annotations assumed to be true. Here are a few hints about the implementation of the additional requirements on SAC and GOC described in Sec. IV-A.

- Support of hypotheses (in $\mathcal{V}_P^{GOC}$): the ability to take into account assumed hypotheses is an essential feature of WP. The implementation of hypotheses insertion relies on the existing services offered by the FRAMA-C kernel.
- Relevant locations and variables ($\mathcal{R}_P^{GOC}$): WP does not provide such API function, so we add our own simple function on top of it. Since the plugin works in a modular way, our implementation just returns all pairs of locations and variables found in the function containing the assertion to check.
- Hypotheses creation ($\mathcal{H}_P$): we rely on the primitives offered by the VALUE plugin to explore the computed approximation of reachable states, and we export them as ACSL annotations. The actual implementation returns annotations (on the form of interval and congruence constraints) only for expressions evaluating to integer values (including primitive variables, structure fields, array values or values referenced by a pointer). Note that missing other kind of information affects the completeness of our method, but not its soundness.

Extension of LUNCOV. The earlier version [6] of LUNCOV, the label infeasibility detection service of the LTEST toolset, used only VALUE in a blackbox manner. The present work extends it by two new modes. In the second mode, LUNCOV constructs for each label l a program with its negation a_l and runs WP in blackbox to check if a_l is valid (that is, by Lemma 1, if l is infeasible). The last mode implements the greybox combination VA⊕WP described in Sec. IV-A and IV-B.

C. Use case

Fig. 4 illustrates how the combined technique VA⊕WP can prove properties that could not be proven by each technique separately. `Frama_C_interval` is a built-in FRAMA-C function returning a non-deterministic value in the given range.

```
int main() {
  int a = Frama_C_interval(0,20);
  int x = Frama_C_interval(-1000,1000);
  return g(x,a);
}
int g(int x, int a) {
  int res;
  if(x+a >= x)
    res = 1;
  else
    res = 0;
  //@assert res == 1;
}
```

Fig. 4. A valid assertion that is proven neither by VALUE nor by WP alone

The assertion in function `g` can be proven neither by VALUE nor by WP alone on a platform like FRAMA-C. VALUE is unable to prove that `x+a >= x` is always true because it lacks relational domains and does not recognize that both occurrences of `x` refer to the same variable and have the same value. Working on a per function level, WP ignores possible values of `x` and `a` in the function `g` and cannot prove the validity of the assertion because of possible overflows.

In VA⊕WP, we first run VALUE on the whole program, then use its results to insert relevant variable information into `g` as shown in Fig. 5, that allows WP to prove the assertion.

V. EXPERIMENTAL EVALUATION

A. Research Questions

In this study we investigate whether static analyzers are capable of detecting infeasible test requirements. Therefore,

```
int g(int x, int a) {
  //@assume a >= 0 && a <= 20;
  //@assume x >= -1000 && x <= 1000;
  int res;
  if(x+a >= x)
    res = 1;
  else
    res = 0;
  //@assert res == 1;
}
```

Fig. 5. Function g of Fig. 4 enriched with hypotheses for WP

a natural question to ask is about their relative effectiveness and efficiency. By showing that these techniques can provide a practical solution to the infeasibility problem, testers and practitioners can adequately measure the true coverage of their test suites. Another benefit is that test generation tools can focus on covering feasible test requirements and hence, improve their performance. In view of this, we seek to answer the following three Research Questions (RQs):

RQ1: How effective are the static analyzers in detecting infeasible test requirements?

RQ2: How efficient are the static analyzers in detecting infeasible test requirements?

RQ3: To what extent can we speed-up the test generation process by detecting infeasible test requirements?

B. Tools, subjects and test requirements

In our experiments we use the FRAMA-C and LTEST tools as they were explicitly defined in Sections II-B, III-C and IV-B. For **RQ3**, we consider the automatic test generation procedure of LTEST, based on DSE* (cf. Section II-B). We consider 12 benchmark programs[2] taken from related works [7], [6], mainly coming from the Siemens test suite (`tcas` and `replace`), the Verisec benchmark (`get_tag` and `full_bad` from Apache source code), and MediaBench (`gd` from libgd). We also consider three coverage criteria: CC, MCC and WM [1]. Each of these coverage criteria were encoded with labels as explained in Section II-A. In the case of WM, the labels mimic mutations introduced by MuJava [23] for operators AOIU, AOR, COR and ROR [1], which are considered very powerful in practice [28], [39]. Each label is considered as a single test requirement. Overall, our benchmark consists of 26 pairs program–test requirements. Among the 1,270 test requirements of this benchmark, 121 were shown to be infeasible in a prior manual examination. Experiments are performed under Linux on an Intel Core2 Duo 2.50GHz, 4GB of RAM. In the following only extracts of our experimental results are given. Further details are available online in an extended version of this paper[2].

*C. Detection power (**RQ1**)*

Protocol. To answer **RQ1** we compare the studied methods in terms of detected infeasible test requirements. Thus, we measure the number and the percentage of the infeasible requirements detected, per program and method. In total we investigate 26 cases, i.e., pairs of program and criterion, with 3 methods. Therefore, in total we perform 78 (26 X 3) runs. The methods we consider are: (1) the value analysis technique, through abstract interpretation, denoted as VA; (2) the computation of weakest preconditions, denoted as WP; and (3) the proposed combination of the VA and WP, denoted VA⊕WP. It is noted that for the WP and VA⊕WP, a timeout is set on the solver calls of 1 second (thanks to WP API).

Results. Table I records the results for **RQ1**. For each pair of program and criterion, the table provides the total number of infeasible labels (from a preliminary manual analysis [7]), the number of detected infeasible requirements and the percentage that they represent per studied method. Since the studied methods are sound, false positives are impossible.

From these results it becomes evident that all the studied methods detect numerous infeasible requirements. Out of the three methods, our combined method VA⊕WP performs best as it detects 98% of all the infeasible requirements. The VA and WP methods detect 69% and 60% respectively. Interestingly, VA and WP do not always detect the same infeasible labels. For instance, WP identifies all the 11 requirements in `fourballs`–WM while VA finds none. Regarding the `utf8-3`–WM, VA identifies all the 29 labels while WP finds only two. This is an indication that a possible combination of these techniques, such as the VA⊕WP method, is fruitful. Thus, VA⊕WP finds at least as much as VA and WP methods on all the cases, while in some, i.e., `replace`-WM and `full_bad`–WM, it performs even better.

TABLE I. INFEASIBLE LABEL DETECTION POWER

Program	LOC	Crit.	#Lab	#Inf	VA		WP		VA ⊕ WP	
					#D	%D	#D	%D	#D	%D
trityp	50	CC	24	0	0	–	0	–	0	–
		MCC	28	0	0	–	0	–	0	–
		WM	129	4	4	100%	4	100%	4	100%
fourballs	35	WM	67	11	0	0%	11	**100%**	11	100%
utf8-3	108	WM	84	29	29	100%	2	7%	29	100%
utf8-5	108	WM	84	2	2	100%	2	100%	2	100%
utf8-7	108	WM	84	2	2	100%	2	100%	2	100%
tcas	124	CC	10	0	0	–	0	–	0	–
		MCC	12	1	0	0%	**1**	**100%**	1	100%
		WM	111	10	6	60%	6	60%	**10**	**100%**
replace	100	WM	80	10	5	50%	3	30%	10	100%
full_bad	219	CC	16	4	2	50%	4	100%	4	100%
		MCC	39	15	9	60%	15	100%	15	100%
		WM	46	12	7	58%	9	75%	11	**92%**
get_tag-5	240	CC	20	0	0	–	0	–	0	–
		MCC	26	0	0	–	0	–	0	–
		WM	47	3	2	67%	0	0%	2	**67%**
get_tag-6	240	CC	20	0	0	–	0	–	0	–
		MCC	26	0	0	–	0	–	0	–
		WM	47	3	2	67%	0	0%	2	**67%**
gd-5	319	CC	36	0	0	–	0	–	0	–
		MCC	36	7	7	100%	7	100%	7	100%
		WM	63	1	0	0%	0	0%	**1**	**100%**
gd-6	319	CC	36	0	0	–	0	–	0	–
		MCC	36	7	7	100%	7	100%	7	100%
		WM	63	0	0	–	0	–	0	–
Total			1,270	121	84	69%	73	60%	118	**98%**
Min				0	0	0%	0	0%	2	**67%**
Max				29	29	100%	15	100%	29	100%
Mean				4.7	3.2	63%	2.8	**82%**	4.5	**95%**

#D: number of detected infeasible labels %D: ratio of detected infeasible labels
–: no ratio of detected infeasible labels due to the absence of infeasible labels

*D. Detection speed (**RQ2**)*

In this section we address **RQ2**, that is about the required time to detect infeasible requirements per studied method. To this end, we investigate three scenarios; a) *a priori* which consists of running the detection process before the test generation,

b) *mixed* which starts with a first round of test generation, then applies the detection method and ends with a second round of test generation and c) *a posteriori* which consists of running the detection approach after the test generation process.

We investigate these scenarios since WP, as a Goal-Oriented Checking, is strongly dependent on the number of considered requirements. Thus, the goal of the a) scenario is to measure the required time before performing any test generation and hence, check all the considered requirements. The b) scenario aims at measuring the time needed when having a fairly mixed set of feasible and infeasible requirements. The goal of the c) scenario is to measure the required time when almost all of the considered requirement are infeasible.

Protocol. We consider the time required to run each detection method per program and scenario, i.e., *a priori*, *mixed* and *a posteriori*. In the *a priori* approach, the detection consider all labels as inputs. In the *mixed* approach, we chose to use a fast but unguided test generation on the first round: random testing with a budget of 1 sec. This time frame was used for both generation and test execution (needed to report coverage). In our system, 984 to 1124 tests are generated per each program in the specified time. To increase the variability of the selected tests, we chose tests between 20 random generations with the median number of covered requirements. The uncovered requirements after this random generation step are the input of the infeasible detection process. In the *a posteriori* approach, we use DSE* as our test generation method. The labels not covered after DSE* are the inputs of the detection.

Overall, by combining the 26 pairs of programs and requirements with the 3 detection methods and the 3 scenarios, a total number of 234 runs are performed.

Results. A summary of our results is given in Table II. The table records for each detection method and studied scenario the number of the considered requirements, and the total required time to detect infeasible labels. It also records the minimum, maximum, and arithmetic mean of the needed time to run the detection on all programs. The average times are also represented as a bar plot on Fig. 6.

From these results we can see that the detection time is reasonable. Indeed, even in the worst case (max) 130.1 sec. are required. Within this time, 2 out of 84 labels are detected. These results also confirm that the required time of WP and VA⊕WP depend on the number of considered requirements. We observe a considerable decrease with the number of labels when using WP or VA⊕WP. The results also shows that in the *mixed* scenario less than half of the time of the *a priori* scenario is required on average.

TABLE II. DETECTION SPEED SUMMARY (IN SECONDS)

	a priori				mixed approach				a posteriori			
	#Lab	VA	WP	VA ⊕WP	#Lab	VA	WP	VA ⊕WP	#Lab	VA	WP	VA ⊕WP
Total	1,270	21.5	994	1,272	480	20.8	416	548	121	13.4	90.5	29.4
Min	10	0.5	5.2	5.5	0	0.5	0.9	1.2	0	0.5	0.4	0.7
Max	129	1.9	127	130	68	1.9	62.5	64.6	29	1.9	50.7	3.9
Mean	48.8	0.8	38.2	48.9	18.5	0.8	16.7	21.9	4.7	0.8	5.7	1.8

#Lab: number of considered labels: in the *a priori* approach, all labels are considered, in the *a posteriori* approach, only labels not covered by DSE*, and in the mixed approach, only labels not covered by the random testing

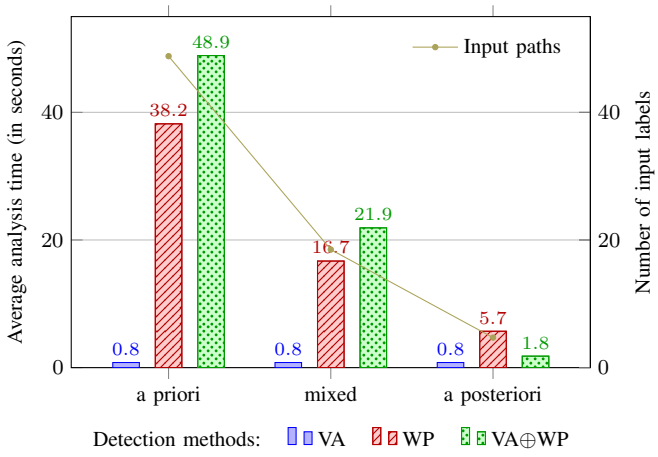

Fig. 6. Average detection time of the studied methods per considered scenario

E. Impact on test generation (*RQ3*)

This section focuses on **RQ3**, that is, on measuring the impact of the knowledge of infeasible requirements on the automated test generation process.

Protocol. In this experiments, we consider only two approaches: (1) LUNCOV+DSE*: first use one of the detection method of LUNCOV, then the DSE* test generation; (2) RT+LUNCOV+DSE*: first exploit random testing to find easy coverable labels, then run LUNCOV, finally run the DSE* test generation to complete the test suite. Recall that LUNCOV forms the implementation of the VA⊕WP approach. Each experiment includes both test generation and infeasible test requirement detection. Various data are recorded, in particular, the reported coverage ratio, as well as, the time needed by the test generation and by the infeasible test requirement detection. Note that the reported coverage ratio remove from consideration the detected infeasible labels.

Results. Table III shows a summary of the coverage ratio reported by DSE* for both approaches (they report the same coverage ratio). As a reference, we provide also the coverage ratio for DSE* without detection and given a manual and perfect detection of infeasible labels. It shows the three methods improve the coverage ratio. In particular the minimum coverage ratio gains goes from 90.5% to more than 95%. Our hybrid method by detecting more infeasible considerably impacts the reported coverage. It allows in our benchmark to report automatically a nearly complete coverage with a 99.2% average coverage ratio.

TABLE III. SUMMARY OF REPORTED COVERAGE RATIOS

	Coverage ratio reported by DSE*				
Detection method	**None**	**VA**	**WP**	**VA ⊕WP**	**Perfect***
Total	90.5%	96.9%	95.9%	99.2%	100.0%
Min	61.54%	80.0%	67.1%	91.7%	100.0%
Max	100.00%	100.0%	100.0%	100.0%	100.0%
Mean	91.10%	96.6%	97.1%	**99.2%**	100.0%

* preliminary, manual detection of infeasible labels

Table VI summarizes the speed-up on the total test generation and infeasible label detection. We observe that the infeasible label detection cost is not always counterbalanced

by a speed-up in the test generation. In fact, for approach (1), LUncov+DSE*, for both WP-based detection, a slow-down occurs. Approach (2), RT+LUncov+DSE*, obtains better results with a mean speed-up of 3.8x. However, we observe in some cases very good speed-ups with multiple two-digit speed-ups as well as a tree-digit speed-up of 107x. Overall the speed-up on the whole benchmark is systematically good.

Fig. 7 shows as a bar plot the average time test generation plus detection. The average time of DSE* without detection is marked by a red line. It shows that the average time generation plus detection in both approaches and for all detection method is well under the DSE* line. We also observe the clear difference between the two approaches, RT+LUncov+DSE* being the more efficient.

TABLE IV. Detection and Test Generation Speed-up Summary

		LUncov = VA	LUncov = WP	LUncov = VA⊕WP
		Speedup	**Speedup**	**Speedup**
LUncov +DSE*	**Total**	**1.3x**	**1.1x**	**1.1x**
	Min	0.7x	0.03x	0.05x
	Max	**10.3x**	2.4x	2.3x
	Mean	**1.4x**	0.5x	0.4x
RT(1s) +LUncov +DSE*	**Total**	**2.4x**	**2.2x**	**2.2x**
	Min	0.5x	0.1x	0.1x
	Max	**107.0x**	**74.1x**	**55.4x**
	Mean	**7.5x**	**5.1x**	**3.8x**

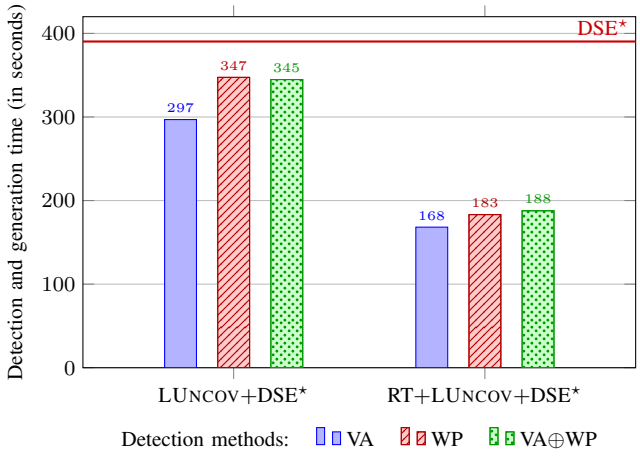

Fig. 7. Average detection and test generation times

F. Evaluation Conclusions

RQ1. Our evaluation shows that sound static analyzers can be used to detect *most infeasible test requirements*. In particular, our implementation of the greybox analysis achieves a nearly perfect detection of infeasible test requirements.

RQ2. Detecting infeasible requirements requires a reasonable amount of time. Our experiment reveals the link between the number of test requirements and the speed of the detection process. Thus, we propose a simple approach that reduces significantly the time required by the analyzers through a preliminary step of (cheap) random testing.

RQ3. Detecting infeasible test requirements influences test generation in two ways. First, it allows us to report coverage that are better (higher ratio) and closer to the truth. Second, it speed-up test generation. In particular, our approach that combines random testing, infeasible requirement detection and DSE* is on average 3.8 times faster than DSE* alone.

VI. Related Work

This section discusses techniques dealing with infeasible requirements for both structural testing, VI-A, and mutation testing, VI-B, as our approach applies in both contexts.

A. Infeasible test requirements for structural testing

Most of the techniques found in literature aim at reducing the effects of infeasible paths and thus, help the test generation process. Ngo and Tan [25] suggested using trace patterns to identify unexplored paths that are likely to be infeasible. In a similar manner Delahaye *et al.* [12] showed many paths are infeasible due to the same reason. Thus, they suggested inferring what causes the infeasibility and generalize it to identify unexplored paths. They also show that when this approach is combined with dynamic symbolic execution considerable savings can be gained. Fraser and Arcuri [15] suggested aiming at all the test requirements and not separately at each one. This way, the wasted effort, i.e., the effort expended on generating test cases for infeasible requirements, is reduced. All these techniques aim at improving the efficiency of the test generation method and not detecting infeasible requirements. Thus, they can be adopted and used instead of our DSE*.

Goldberg *et al.* [17] suggested that when all the paths leading to a test requirement are infeasible then, this requirement is infeasible. Thus, they used symbolic execution and theorem provers to identify infeasible paths and some infeasible test requirements. In a similar way, Offutt and Pan [27] used constraint based testing to encode all the constraints under which a test requirement can be covered. If these constraints cannot be solved then, the requirements are infeasible. However, these methods are not applicable even on small programs due to the infinite number of the involved paths [40]. Additionally, the imprecise handling of program aliases [33] and non-linear constraints [2] further reduce the applicability of the methods.

Detecting infeasible requirements has been attempted using model checkers. Beyer *et al.* [9] integrate symbolic execution and abstraction to generate test cases and prove the infeasibility of some requirements. Beckman *et al.* [8] adopt the computation of the weakest precondition to prove that some statements are not reachable. Their aim was to formally verify some properties on the tested system and not to support the testing process. This was done by Baluda *et al.* [5]. Baluda *et al.* used model abstraction refinement based on the weakest precondition and integrate it with dynamic symbolic execution to support structural testing. Our approach differs from this one, by using a hybrid combination of value analysis with weakest precondition independently of the test generation process. Additionally, our approach is the first one that employs static analysis approaches to automatically detect infeasible requirements for a wide range of testing criteria such as the multiple condition coverage and weak mutation.

B. Equivalent Mutants

Detecting equivalent mutants is a known undecidable problem [3]. This problem is an instance of the infeasibility

problem [27] in the sense that equivalent mutants are the infeasible requirements of the mutation criterion. Similar to the structural infeasible requirements, very few approaches exist for equivalent mutants. We briefly discuss them here.

Baldwin and Sayward [3] observed that some mutants form optimized or de-optimized versions of the original program and they suggested using compiler optimization techniques to detect them. This idea was empirically investigated by Offutt and Craft [26] and found that on average 45% of all the existing equivalent mutants can be detected. Offutt and Pan [27] model the conditions under which a mutant can be killed as a constraint satisfaction problem. When this problem has no solution the mutants are equivalent. Empirical results suggest that this method can detect on average 47% of all the equivalent mutants. Note that like in our case, these approaches aim at identifying weak equivalent mutants and not strong ones. However, they have the inherent problems of the constraint-based methods such as the imprecise handling of program aliases [33] and non-linear constraints [2]. Papadakis *et al.* [31] demonstrated that 30% of the strongly equivalent mutants can be detected by using compilers. Our approach differs from this one in two essential ways. First, we handle weak mutants while they target strong ones. Second, we use state-of-the-art verification technologies while they use standard compiler optimizations. Note that the two approaches are complementary for strong mutation: our method identifies mutants, 95%, that can be neither reached nor infected, while the compiler technique identifies mutants, 45%, that cannot propagate.

Voas and McGraw [36] suggested using program slicing to assist the detection of equivalent mutants. This idea was developed by Hierons *et al.* [20] who formally showed that their slicing techniques can be employed to assist the identification of equivalent mutants and in some cases to detect some of them. Hierons *et al.* also demonstrated that slicing subsumes the constraint based technique of Offutt and Pan [27]. Harman *et al.* [18] showed that dependence analysis can be used to detect and assist the identification of equivalent mutants. These techniques were not thoroughly evaluated since only synthetic data were used. Additionally, they suffers from the inherent limitations of the slicing and dependence analysis technology.

Other approaches tackle this problem based on mutant classification, i.e., classify likely equivalent and non-equivalent mutants based on run-time properties of the mutants. Schuler and Zeller [34] suggested measuring the impact of mutants on the program execution. They found that among several impact measures, coverage was the most effective one. This idea was extended by Kintis *et al.* [21] using higher order mutants. Their results indicate that higher order mutants can provide more accurate results than those provided by Schuler and Zeller. Papadakis, *et al.* [30] defined the mutation process when using mutant classification. They demonstrated that using mutant classification is profitable only when low quality test suites are employed and up to a certain limit. Contrary to our approach, these approaches are not sound, i.e., they have many false positives. They can also be applied in a complementary way to our approach by identifying likely equivalent mutants from those not found by our approach [34]. Further details about the equivalent mutants on other mutation domains and can be found at a relevant survey about equivalent mutants [24].

VII. Discussion

Our findings suggest that it is possible to identify almost all infeasible test requirements. This implies that the accuracy of the measured coverage scores is improved. Testers can use our technique to decide with confidence when to stop the testing process. Additionally, since most of the infeasible requirements can be removed, it becomes easier to target full coverage. According to Frankl and Iakounenko [14] this is desirable since the majority of the faults are triggered when covering higher coverage levels, i.e., from 80% to 100% of decision coverage.

Although our approach handles weak mutation, it can be directly applied to detect strong equivalent mutants. All weakly equivalent mutants are also strongly equivalent mutants [41] and thus, our approach provides the following two benefits. First, it reduces the involved manual effort of identifying equivalent mutants. According to Yao *et al.* [41], equivalent mutant detection techniques focusing on weak mutation have the potential to detect approximately 60% of all the strong equivalent mutants. Therefore, since our approach detects more than 80% of the weak mutants, we can argue that the proposed approach is powerful enough to detect approximately half of all the involved mutants. Second, it reduces the required time to generate the test cases as our results show. The current state-of-the-art in strong mutation-based test generation aims at weakly killing the mutants first and then at strongly killing them [19], [32]. Therefore, along these lines we can target strong mutants after applying our approach.

Finally, it is noted that our method can be applied to MCDC criterion by weakening its requirements into GACC requirements. GACC requirements can be encoded as labels [29].

A. Threats to Validity and Limitations

As it is usual in software testing studies, a major concern is about the representativeness, i.e., *external validity*, of the chosen subjects. To reduce this threat we employed a recent benchmark set composed of 12 programs [7]. These vary both with respect to application domain and size. We were restricted to this benchmark since we needed to measure the extent of the detected infeasible requirements.

Another issue is the *scalability* of our approach since we did not demonstrate its applicability on large programs. While, this is an open issue that we plan to address in the near future, it can be argued that our approach is as applicable and scalable as the techniques that we apply. We rely on Value Analysis and Weakest Precondition methods as implemented within the Frama-C framework. These particular implementations are currently used by industry [22, Sec. 11] to analyze safety-critical embedded software (Airbus, Dassault, EdF) or security-critical programs (PolarSSL, QuickLZ). Moreover, our implementation handles all C language constructs except of multi-thread mechanisms and recursive functions. Thus, we believe that our propositions are indeed applicable to real-world software. Moreover, note that Weakest Precondition methods are inherently scalable since they work in a modular way. Hence, we can strongly expect that the (good) experimental results reported in Sec. V for WP still hold on much larger programs. Though, the primary contribution of this article is to demonstrate that static analysis techniques can be used to detect infeasible test requirements such as equivalent mutants. Future research will focus on scalability issues.

Other threats are due to possible defects in our tools, i.e., *internal validity*. To reduce this threat we carefully test our implementation. Additionally, the employed benchmark, which has known infeasible test requirements, served as a sanity check for our implementation. It is noted that the employed tools have also passed successfully the NIST *SATE V Ockham Sound Analysis Criteria*[4] thus, providing confidence on the reported results. Furthermore, to reduce the above-mentioned threats we made our tool and all the experimental subjects publicly available[2].

Finally, additional threats can be attributed to the used measurements, i.e., *construct validity*. However, infeasible requirements form a well known issue which is usually acknowledged by the literature as one of the most important and time consuming tasks of the software testing process. Similarly, the studied criteria might not be the most appropriate ones. To reduce this threat we used a wide range of testing criteria, most of which are included in software testing standards and are among the most popular ones in the software testing literature.

VIII. Conclusion

In this paper we used static analysis techniques to detect infeasible test requirements for several structural testing criteria, i.e., condition coverage, multiple condition coverage and weak mutation. We leverage two state-of-the-art techniques, namely Value Analysis and Weakest Precondition, and determined their ability to detect infeasible requirements in an automatic and sound way. Going a step further, we proposed a lightweight greybox scheme that combines these techniques. Our empirical results demonstrate that our method can detect a high ratio of infeasible test requirements, on average 95%, in a few seconds. Therefore, our approach improves the testing process by allowing a precise coverage measurement and by speeding-up automatic test generation tools.

Acknowledgment

The authors would like to thank the Frama-C team members for providing the tool, their support and advice.

References

[1] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge University Press, 2008.

[2] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, 2013.

[3] D. Baldwin and F. G. Sayward, "Heuristics for determining equivalence of program mutations," Yale University, Research Report 276, 1979.

[4] T. Ball and S. K. Rajamani, "The SLAM project: Debugging system software via static analysis," *SIGPLAN Notices*, vol. 37, no. 1, 2002.

[5] M. Baluda, P. Braione, G. Denaro, and M. Pezzè, "Enhancing structural software coverage by incrementally computing branch executability," *Software Quality Journal*, vol. 19, no. 4, 2011.

[6] S. Bardin, O. Chebaro, M. Delahaye, and N. Kosmatov, "An all-in-one toolkit for automated white-box testing," in *TAP*. Springer, 2014.

[7] S. Bardin, N. Kosmatov, and F. Cheynier, "Efficient leveraging of symbolic execution to advanced coverage criteria," in *ICST*. IEEE, 2014.

[8] N. E. Beckman, A. V. Nori, S. K. Rajamani, R. J. Simmons, S. D. Tetali, and A. V. Thakur, "Proofs from Tests," *IEEE Trans. Softw. Eng.*, vol. 36, no. 4, Jul. 2010.

[9] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker Blast," *STTT*, vol. 9, no. 5-6, 2007.

[10] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL*, 1977.

[11] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "The ASTRÉE analyzer," in *ESOP*. Springer, 2005.

[12] M. Delahaye, B. Botella, and A. Gotlieb, "Infeasible path generalization in dynamic symbolic execution," *Inf. and Softw. Technology*, 2014.

[13] M. Fähndrich and F. Logozzo, "Static contract checking with abstract interpretation," in *FoVeOOS*, 2010.

[14] P. G. Frankl and O. Iakounenko, "Further empirical studies of test effectiveness," *ACM SIGSOFT Softw. Eng. Notes*, vol. 23, no. 6, 1998.

[15] G. Fraser and A. Arcuri, "Whole Test Suite Generation," *IEEE Trans. Softw. Eng.*, vol. 39, no. 2, 2013.

[16] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *PLDI*. ACM, 2005.

[17] A. Goldberg, T. C. Wang, and D. Zimmerman, "Applications of feasible path analysis to program testing," in *ISSTA*. ACM, 1994.

[18] M. Harman, R. M. Hierons, and S. Danicic, "The relationship between program dependence and mutation analysis," in *MUTATION*, 2001.

[19] M. Harman, Y. Jia, and W. B. Langdon, "Strong higher order mutation-based test data generation," in *ESEC/FSE*. ACM, 2011.

[20] R. M. Hierons, M. Harman, and S. Danicic, "Using program slicing to assist in the detection of equivalent mutants," *STVR*, vol. 9, no. 4, 1999.

[21] M. Kintis, M. Papadakis, and N. Malevris, "Employing second-order mutation for isolating first-order equivalent mutants," *STVR*, 2014.

[22] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-C: A Program Analysis Perspective," *Formal Aspects of Computing Journal*, 2015.

[23] Y. Ma, J. Offutt, and Y. R. Kwon, "MuJava: a mutation system for Java," in *ICSE*. ACM, 2006.

[24] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Jozala, "Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation," *IEEE Trans. Softw. Eng.*, vol. 40, no. 1, 2014.

[25] M. N. Ngo and H. B. K. Tan, "Heuristics-based infeasible path detection for dynamic test data generation," *Inf. and Softw. Technology*, vol. 50, no. 7-8, 2008.

[26] A. J. Offutt and W. M. Craft, "Using compiler optimization techniques to detect equivalent Mutants," *STVR*, vol. 4, no. 3, 1994.

[27] A. J. Offutt and J. Pan, "Automatically Detecting Equivalent Mutants and Infeasible Paths," *Software Testing, Verification and Reliability*, vol. 7, no. 3, 1997.

[28] A. J. Offutt, G. Rothermel, and C. Zapf, "An experimental evaluation of selective mutation," in *ICSE*. IEEE/ACM, 1993.

[29] R. Pandita, T. Xie, N. Tillmann, and J. de Halleux, "Guided test generation for coverage criteria," in *ICSM*. IEEE CS, 2010.

[30] M. Papadakis, M. Delamaro, and Y. L. Traon, "Mitigating the effects of equivalent mutants with mutant classification strategies," *Science of Computer Programming*, 2014.

[31] M. Papadakis, Y. Jia, M. Harman, and Y. LeTraon, "Trivial compiler equivalence: A large scale empirical study of a simple fast and effective equivalent mutant detection technique," in *37th International Conference on Software Engineering (ICSE)*, 2015.

[32] M. Papadakis and N. Malevris, "Automatic mutation test case generation via dynamic symbolic execution," in *ISSRE*. IEEE, 2010.

[33] ——, "Mutation based test case generation via a path selection strategy," *Inf. and Softw. Technology*, vol. 54, no. 9, 2012.

[34] D. Schuler and A. Zeller, "Covering and uncovering equivalent mutants," *STVR*, vol. 23, no. 5, 2013.

[35] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *ESEC/FSE*. ACM, 2005.

[36] J. Voas and G. McGraw, *Software Fault Injection: Inoculating Programs Against Errors*. John Wiley & Sons, 1997.

[37] E. Weyuker, "More experience with data flow testing," *IEEE Trans. Softw. Eng.*, vol. 19, no. 9, 1993.

[38] N. Williams, B. Marre, and P. Mouy, "On-the-fly generation of k-paths tests for C functions : towards the automation of grey-box testing," in *ASE*. IEEE CS, 2004.

[39] W. E. Wong and A. P. Mathur, "Reducing the cost of mutation testing: An empirical study," *JSS*, vol. 31, no. 3, 1995.

[40] M. Woodward, D. Hedley, and M. Hennell, "Experience with Path Analysis and Testing of Programs," *IEEE Trans. Softw. Eng.*, vol. SE-6, no. 3, 1980.

[41] X. Yao, M. Harman, and Y. Jia, "A study of equivalent and stubborn mutation operators using human analysis of equivalence," in *ICSE*. ACM, 2014.

[42] D. Yates and N. Malevris, "Reducing the effects of infeasible paths in branch testing," *ACM SIGSOFT Softw. Eng. Notes*, vol. 14, no. 8, 1989.

[4] See `http://samate.nist.gov/SATE5OckhamCriteria.html`.

APPENDIX

A. Automated Test Generation for Labels: DSE*

Let us recall here a few basic facts about Symbolic Execution (SE) and Dynamic Symbolic Execution (DSE) [16], [35], [38]. A simplified view of SE is depicted in Algorithm 1. We assume that the set of paths of P, denoted $Paths(P)$, is finite. In practice, testing tools enforce this assumption through a bound on path lengths. The algorithm iteratively builds a test suite TS by exploring all program paths. The key insight of SE is to compute a *path predicate* ψ_σ for a path $\sigma \in Paths(P)$ such that for any input valuation $t \in D$, we have: t satisfies ψ_σ iff $P(t)$ covers σ. As most SE tools, Algorithm 1 uses a procedure for path predicate computation (with predicates in some theory T), and relies on a solver taking a formula $\psi \in T$ and returning either *sat* with a solution t or *unsat*. DSE enhances SE by interleaving concrete and symbolic executions. The dynamically collected information can help the symbolic step, for example, by suggesting relevant approximations.

Algorithm 1: Symbolic Execution algorithm

Input: a program P with finite set of paths $Paths(P)$
Output: TS, a set of pairs (t, σ) such that $P(t)$ covers σ

```
1  TS := ∅;
2  S_paths := Paths(P);
3  while S_paths ≠ ∅ do
4  |  choose σ ∈ S_paths; S_paths := S_paths \ {σ};
5  |  compute path predicate ψ_σ for σ ;
6  |  switch solve(ψ_σ) do
7  |  |  case sat(t):  TS := TS ∪ {(t, σ)};
8  |  |  case unsat:  skip;
9  |  endsw
10 end
11 return TS;
```

Based on dynamic symbolic execution, the DSE* method, proposed in [7] and further improved in [6], offers an efficient test generation technique natively supporting labels. Compared to other DSE-based algorithms for coverage criteria, DSE* brings several benefits. First, it has the major advantage of being generic due to labels. Second, thanks to a tight instrumentation of the program and other specific optimizations for labels, the method's overhead in terms of explored paths over the classic DSE is linear in the number of labels. Previous methods, such as [29], reported an exponential increase of the search space.

B. Research question ***Q2***

C. Research question ***Q3***

TABLE V. DETAILED RESULTS FOR INFEASIBLE LABEL DETECTION SPEED (IN SECONDS)

Program	Crit.	A priori				Mixed approach				A posteriori			
		#Lab	VA	WP	VA ⊕ WP	#Lab	VA	WP	VA ⊕ WP	#Lab	VA	WP	VA ⊕ WP
trityp	CC	24	0.6	11.9	14.9	12	0.6	6.4	6.8	0			
	MCC	28	0.5	13.9	14.7	12	0.5	6.8	6.9	0			
	WM	129	0.7	74.4	81.0	68	0.7	39.7	42.3	4	0.7	2.2	0.8
fourballs	WM	67	0.5	27.2	28.2	42	0.5	16.2	17.3	11	0.5	2.8	2.7
utf8-3	WM	84	0.5	127.2	75.5	31	0.5	54.8	4.7	29	0.5	50.7	0.7
utf8-5	WM	84	0.6	127.2	130.1	35	0.6	62.1	64.4	2	0.6	0.9	0.8
utf8-7	WM	84	0.6	127.2	128.4	35	0.6	62.5	64.6	2	0.6	0.9	0.8
tcas	CC	10	0.7	5.2	5.5	0				0			
	MCC	12	0.5	6.4	6.9	1	0.5	1.0	1.2	1	0.5	0.4	1.1
tcas'	WM	111	0.7	55.4	54.8	21	0.7	8.2	8.1	10	0.7	3.6	3.0
replace	WM	80	0.8	39.1	39.6	15	0.8	8.9	4.2	10	0.8	4.8	1.6
full_bad	CC	16	0.5	10.3	17.7	7	0.5	4.1	6.6	4	0.5	1.2	0.9
	MCC	39	0.6	22.0	37.5	23	0.6	11.7	17.4	15	0.6	3.9	2.3
	WM	46	0.7	27.8	39.0	18	0.7	9.4	13.0	12	0.7	4.2	2.5
get_tag-5	CC	20	0.7	11.1	18.5	1	0.7	1.1	2.2	0			
	MCC	26	0.7	15.8	27.0	2	0.7	0.9	4.3	0			
	WM	47	0.7	31.3	58.9	13	0.7	9.9	21.3	3	0.7	1.9	1.3
get_tag-6	CC	20	0.6	11.1	18.5	2	0.6	1.7	4.8	0			
	MCC	26	0.6	15.7	27.0	2	0.6	1.9	4.3	0			
	WM	47	0.7	31.3	59.0	15	0.7	11.3	25.3	3	0.7	4.1	2.2
gd-5	CC	36	1.2	23.6	53.8	21	1.2	13.7	38.5	0			
	MCC	36	1.9	31.0	48.8	22	1.9	15.8	32.4	7	1.9	3.5	2.0
	WM	63	1.7	47.2	92.0	20	1.7	18.4	42.8	1	1.7	1.9	3.9
gd-6	CC	36	1.1	23.7	53.8	20	1.1	14.9	40.1	0			
	MCC	36	1.9	30.2	49.0	22	1.9	16.1	31.8	7	1.9	3.5	2.7
	WM	63	1.5	47.0	92.0	20	1.5	18.7	42.4	0			
Total		1,270	21.5	994	1,272	480	20.8	416	548	121	13.4	90.5	29.4
Min		10	0.5	5.2	5.5	0	0.5	0.9	1.2	0	0.5	0.4	0.7
Max		129	1.9	127.2	130.1	68	1.9	62.5	64.6	29	1.9	50.7	3.9
Mean		48.8	0.8	38.2	48.9	18.5	0.8	16.7	21.9	4.7	0.8	5.7	1.8

#Lab: number of considered labels: in the *a priori* approach, all labels are considered,
in the *a posteriori* approach, only labels not covered by DSE*,
and in the mixed approach XXX

TABLE VI. DETAILED SUMMARY OF DETECTION AND TEST GENERATION TIMES (IN SECONDS)

		LUNCOV = VA				LUNCOV = WP				LUNCOV = VA⊕WP			
	DSE*	LUNCOV	DSE*	Total	Speedup	LUNCOV	DSE*	Total	Speedup	LUNCOV	DSE*	Total	Speedup
LUNCOV+DSE*													
Total	10,147.3	21.48	7,696.7	7,718.2	**1.3**	994.0	8,037.2	9,031.2	**1.1**	1,272.0	7,693.2	8,965.2	**1.1**
Min	1.4	0.47	1.4	2.0	0.7	5.2	1.4	6.8	0.03	5.5	1.4	7.1	0.05
Max	3,434.9	1.91	2,033.3	2,034.8	**10.3**	127.2	2,338.1	2,369.4	2.4	130.1	2,033.3	2,125.3	2.3
Mean	390.3	0.8	296.0	296.9	**1.4**	38.2	309.1	347.4	0.5	48.9	295.9	344.8	0.4
Random testing (1s)+LUNCOV+DSE*													
Total	10,147.3	21	4,160	4,206	**2.4**	416	4,139	4,581	**2.2**	548	4,123	4,695	**2.2**
Min	1.4	0.5	1.6	3.1	0.5	0.9	1.5	3.5	0.1	1.2	1.5	4.9	0.1
Max	3,434.9	1.9	1,948.2	1,949.8	**107.0**	62.5	1,948.2	1,960.4	**74.1**	64.6	1,948.2	1,974.5	**55.4**
Mean	390.3	0.8	166.4	168.2	**7.5**	16.7	165.6	183.2	**5.1**	21.9	164.9	187.8	**3.8**