IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received November 2, 2018, accepted November 22, 2018, date of publication November 29, 2018,
date of current version January 11, 2019.

Digital Object Identifier 10.1109/ACCESS.2018.2883776

Domain-RIP Analysis: A Technique for Analyzing Mutation Stubbornness

HUAN LIN, YAWEN WANG, YUNZHAN GONG, AND DAHAI JIN
State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing 100876, China

Corresponding author: Yawen Wang (wangyawen@bupt.edu.cn)

This work was supported by the National Natural Science Foundation of China under Grant U1736110 and Grant 61702044.

ABSTRACT Existing mutation techniques generate vast numbers of equivalent and trivial mutants, which do not contribute on the improvement of test quality. One possible solution, as is proposed in this paper, is to measure the difficulty of killing a mutant and choose the stubborn (hard-to-kill) mutants in testing, which can be used to challenge the engineers to design new tests so to enhance the testing effectiveness. This paper introduces the domain-based reachability-infection-propagation analysis (D-RIP), a technique that can rank the mutants according to their difficulty of being revealed and present the reason for mutants being stubborn. More specifically, the D-RIP uses subdomain-based testing method and RIP analytical tool to measure the difficulty degree and decide whether a mutant is stubborn. Furthermore, it generates a report that displays the mutants that are hard-to-kill for a given testing method and presents the reasons to interpret why the test suites fail in detecting these mutants. Using this tool, the engineers are able to choose the useful mutants for improving test quality and obtain the guidance to design tests for detecting the stubborn mutants.

INDEX TERMS Mutation testing, subdomain-based testing, reachability-infection-propagation analysis, stubborn mutants, fault detection.

I. INTRODUCTION

Mutation testing [1]–[3] is a fault-based technique that uses a set of artificial faults, called mutants, to guide test process. Each mutant is created by injecting a syntactic change in code so to simulate the real fault in software [4]–[6]. If a test can distinguish the behaviors of mutant from original program, it is said that the test kills, or reveals the mutant. The underlying idea behind mutation analysis is that: test suites that kill more mutants tend to be of higher capability in fault detection than those that kill the fewers [7]. Since then, mutation testing has been widely studied and used for test data generation [8]–[11] as well as for assessing the test suite completeness [12]–[16].

Although powerful, the mutation testing has long suffered from being inefficient. According to the prior research [17]–[22], the most generated mutants are **not** useful for mutation testing: they are either equivalent or trivial mutants. Equivalent mutants are semantically identical with original program, which cannot be detected by any tests [23]–[25]. The trivial mutants, however, can be revealed by almost every test that reaches the mutated statement [21]. Any test suites that meet some structural coverage criteria [26] are guaranteed to detect the trivial mutants. Since mutation-based assessment is long used after the code coverage has been achieved by tests [12]–[16], both equivalent and trivial mutants do not contribute on the improvement of test suite quality.

A possible solution to improve the efficiency of mutation analysis is only to use the **stubborn mutants**. Informally, the stubborn mutants are those that remain unkilled (or survive) from a reasonably thorough test suite yet non-equivalent [27]. From a perspective of testing efforts, the term ‘‘stubbornness’’ is related with the difficulty of revealing a mutant. The more stubborn is a mutant, the more difficult it is to find the tests to kill the mutant in a given testing method. In this perspective, trivial mutants are the least stubborn that any tests generated from the criteria that are weaker than mutation-based criteria can detect them. Meanwhile, equivalent mutants are the most stubborn that it is impossible to be killed by any tests.

If we rank the mutants according to a spectrum from easy to hard that they are detected, the mutants in both ends refer to the least and most stubborn mutants, which are trivial and equivalent respectively. The others, however, refer to mutants that could survive from the test suites generated from a given testing method with high likelihood yet non-equivalent. Such stubborn mutants can provide the following benefits for test suite improvement and mutation testing.

- They could reveal the deficiency of the testing method, especially the coverage-based testing method, which is widely used in software practice [32]–[41].
- They can simulate the hard-to-kill faults in program and challenge the testers to design new tests so to enhance the test suite effectively [13].
- They are the reasons why mutation-based assessment is more powerful than traditional coverage criterions: the more mutants are stubborn, the more room we have for improving the existing test suite quality.

Note that, the stubbornness depends on the **testing method** considered in analysis, i.e. how the tests are selected from inputs space. Clearly, for different testing methods, difficulty for killing a mutant could be different. Although prior studies [27]–[31] proposed many techniques to measure the mutant stubbornness, none of them considered the relations between mutation stubbornness and testing methods used in analysis, making it difficult to identify the stubborn mutants precisely. Meanwhile, none of the prior studies investigates the reasons for the tests failing in detecting the stubborn mutants, which provides no guidance for the testers to design new tests so to enhance the test suite quality effectively.

To fill the research gaps, this paper introduces the domain based reachability-infection-propagation analysis, or **D-RIP**. The D-RIP is a technique that measure the difficulty of killing a mutant based on the given testing method, and presents the possible reasons for a mutant being stubborn.[1]

The technique uses **survival rate** to measure the difficulty of killing a mutant. Informally, survival rate is the likelihood that a mutant can survive from a random test suite generated by a testing method. The greater is the survival rate, the more difficult it is for the testing method to generate a test suite that can detect the mutant. Knowing the survival rate, an engineer can select the stubborn mutants to guide test improvement.

To measure the surival rate precisely, the D-RIP technique uses subdomain-based testing model [42]–[55], which can be used to describe the formal model of the testing method, and obtain the analytical solution to compute the survival rate. In terms of the test model, we also identify the selection-related reasons for the mutant being stubborn. More specifically, the selection-related reasons explain why a subdomain-based testing method fails in killing the mutants, and provide guidance for enhancing test quality by improving the random test selection strategy based on the given subdomains.

Meanwhile, D-RIP uses reachability-infection-propagation model [56]–[60], to investigate the relations between mutant stubbornness and the fault detection process. It can reveal at which stages in fault detection the tests are most likely to fail in revealing the mutant. In terms of RIP model, we investigate constraint-based reasons for stubborn mutant, which identify the key constraints for revealing a stubborn mutant and guide the testers to apply these constraints to generate tests that are efficient in killing the target mutants.

Finally, we implement the D-RIP technique in a prototype tool, JCMuta.[2] The JCMuta uses a dynamic-based algorithm to compute the theoretical solution of survival rate, in which a finite set of tests are required to simulate entire inputs space. The tool will produce a ranked list of mutants, according to their survival rates and generate a report that can display the stubborn mutants and their difficulty reasons automatically.

As far as we know, D-RIP and JCMuta is the first mutation technique that not only can measure the difficulty of killing a mutant but also provide the guidance to generate the tests for detecting the hard-to-kill mutant automatically. In short, this paper makes the following contributions:

- A theoretical model to obtain the analytical solution for computing the survival rate based on testing method.
- A classification framework and algorithm to identify the reasons for a mutant being hard-to-kill.
- A prototype tool, JCMuta, to identify stubborn mutants and report the reasons and guidance for killing them.
- An evaluation to validate the proposed technique.

The remainder of this paper is organized as follow. Section II provides the formal definition of survival rate, which is the basis to identify the stubborn mutants. Section III introduces the analytical solution of survival rate based on test methods and describes the selection-related reason for a mutant being stubborn. Section IV applies the RIP model to investigate the relations between stubbornness and fault detection processes which reveals constraint-based reason. Section V presents the dynamic-based implementation of D-RIP technique. Section VI reports the empirical evaluation results. The threats on the validity of the experiments, and the limitations of the D-RIP technique are discussed in section VII. Section VIII presents the related works on stubborn mutants, while section IX will conclude our works and contributions in this paper.

II. DEFINITION

In mutation testing, the stubbornness of mutant is related with the difficulty of revealing a mutant in the testing. Usually, the difficulty can be considered as the efforts needed to find a test suite that detects the mutant. Since the most of test generation techniques return the random test suite [32]–[41], we can use the survival rate of a mutant to measure this difficulty.

Formally, let p be the program under test, I be its inputs space. A test is an input that can drive the program to execute. If a test $x \in I$ produces different behaviors between mutant m and original program p, it is said that x kills m, denoted as $kill(x, m)$. A test suite is a subset of inputs $T \subseteq I$ which are selected by a testing method. Given the testing method ω, we introduce a probability mass function P_ω on 2^I such that the $P_\omega(T)$ is the probability that method ω returns test suite T as outputs. Then the likelihood that a mutant survives from

[1] In this paper, we use "the reason for the mutant being stubborn" and "reason for the test suite (generated by a testing method) failing in killing a mutant" interchangeably, since they all refer to the failure of detecting fault.

[2] https://github.com/dzt2/jcsa

(not killed by) a random test suite T generated from ω is defined as its survival rate, denoted as $S_{m,\omega}$. There we have:

$$S_{m,\omega} = \sum_{T \subseteq I} P_{\omega}(T) \cdot \{\bigwedge_{x \in T} \neg kill(x, m)\} \tag{1}$$

In the equation, the probability function P_{ω} is introduced for defining survival rate precisely, although it is uncomputable. The item $\bigwedge \neg kill(x, m) = 1$ iff. none of tests in T kills m, and its expected value on P_{ω} is the survival rate of mutant.

When $S_{m,\omega} \to 0$, it means the mutant m can be revealed by almost every test suite generated from a testing method ω. When $S_{m,\omega} \to 1$, it is very difficult for method ω to generate a test that can kill the mutant. In the extreme case $S_{m,\omega} = 1$, it is impossible for ω to generate any tests to kill the mutant. Clearly, when m is equivalent mutant, there is no input in I that kills m and $S_{m,\omega} = 1$ holds for any testing methods.

However, that $S_{m,\omega} = 1$ does not imply that the mutant m is equivalent. That's because a testing method may not select any tests that kill the mutant m. More formally, it means that $P_{\omega}(T) = 0$ holds for any test suite that contains the tests that can detect the mutant m.

To simplify the analysis, we introduce the notion of testing method completeness. Formally, a test method ω is complete iff. for any test $x \in I$, there exists a test suite $T \subseteq I$ so that $x \in T$ and $P_{\omega}(T) > 0$. In other words, for a complete testing method, any test in I might be generated in testing. In terms of complete testing method, we have the following theorem.

Theorem 1: Given a complete testing method ω, the mutant m is equivalent iff. its survival rate satisfies $S_{m,\omega} = 1$.

Proof: First, it is clear that $S_{m,\omega} = 1$ when m is equivalent mutant. We prove that, for any of complete testing method ω, that $S_{m,\omega} = 1$ implies m being an equivalent mutant. Let m be a non-equivalent mutant with $S_{m,\omega} = 1$. Then there is a test x in I such that x detects m. According to the definition, there is a test suite $T \subset I$ such that $x \in T$ and $P_{\omega}(T) > 0$. Therefore, we have $P_{\omega}(T) \cdot \bigwedge \neg kill(x, m) = 0$, which infers that $S_{m,\omega} < \sum P_{\omega}(T) = 1$. This conclusion contradicts with the premise of $S_{m,\omega} = 1$, and thus mutant m with $S_{m,\omega} = 1$ cannot be non-equivalent and must be equivalent.

The introduction of testing completeness and theorem-1 can simplify our analysis to distinguish the equivalent and the stubborn mutants. More specifically, the stubborn mutants are those with survival rate close to 1, of which $S_{m,\omega} \to 1$, while equivalent mutants are those with $S_{m,\omega} = 1$ for any complete testing method ω. Formally, let $\theta \in (0, 1)$ be a threshold used to decide whether detecting a mutant is difficult enough. The mutants with $S_{m,\omega} \geq \theta$ are considered as hard-to-kill, while those with $S_{m,\omega} < \theta$ are thought to be easily killed. Given a complete testing method ω, each mutant is dropped into one of the following categories.

- **Trivial mutant**: $S_{m,\omega} < \theta$;
- **Stubborn mutant**: $\theta \leq S_{m,\omega} < 1$;
- **Equivalent mutant**: $S_{m,\omega} = 1$.

Note that, since ω is complete testing method, that $S_{m,\omega} = 1$ implies that m is equivalent. Therefore, in this classification, the condition for a mutant being stubborn is $\theta \leq S_{m,\omega} < 1$. In the following studies, we use this condition to identify the stubborn mutants for any complete testing method ω.

III. SUBDOMAIN-BASED TESTING MODEL

A. SUBDOMAIN-BASED TESTING METHOD

The term subdomain-based testing, in its broadest sense, can describe a very general family of testing methods. The basic characteristic is that the program inputs space is divided into **subdomains** (or domain), with testers selecting one or more tests from each of them. The division of inputs space relies on a set of test requirements, in which each subdomain contains tests that can satisfy a particular requirement [51].

A test requirement can be any test objective. For example, the statement testing requires each statement being executed at least once. It seperates the inputs space into subdomains, each of which contains tests that cause a particular statement being executed. Branch testing also divides inputs space into subdomains, of which inputs cause a specific predicate being evaluated as true or false. Mutation testing is also subdomain based testing method, in which each domain defines tests that can kill a particular (non-equivalent) mutant [53].

In prior studies, the subdomains can be either overlapping (as in the branch testing), or disjoint (as in the path testing). If the subdomains are disjoint, we speak the testing method of the partition testing, which is a special form of subdomain-based testing [45]–[55]. In this paper, we do not assume that the subdomains are disjoint because the most test methods in practice are not partition, like branch testing [32]–[41].

In terms of the subdomain-based testing, a testing method is a collection, denoted as $\omega = (p, I, SD, \vec{n})$, in which:

- p is the program under testing;
- I is the inputs space of the program;
- $SD = \{D_1, D_2, \ldots, D_s\}$ defines the set of subdomains, in which the inputs space is divided into s subsets;
- $\vec{n} = (n_1, n_2, \ldots, n_s)$ specifies the test selection strategy, in which n_i is the number of tests selected from D_i.

In this model, a testing method ω will select n_i random tests from each subdomain D_i in SD, which is called a subdomain based testing method. The vector $\vec{n}$ specifies how many tests are randomly generated from each domain, which is called a selection vector. The vector $\vec{n}$ together with subdomains SD, precisely describes how a test suite T is generated from a test method ω by selecting random tests in each domain. We call the way which generates test suites by specifying the domains and the number of selected tests as a subdomain-based testing selection, of which details are discussed in next subsection.

To simplify the following analysis, we further make three assumptions for subdomain-based testing methods.

Assumption 2: The subdomains SD are complete. That is $\forall x \in I, \exists D_i \in SD$ such that $x \in D_i$.

Assumption 3: For each subdomain $D_i \in SD$, all of its inputs could be selected for testing.

Assumption 4: The inputs are independently selected from each subdomain with replacement.

The assumption-2 and assumption-3 guaranteed that, for any test selection in which $n_i > 0$ holds for each domain, the testing method ω must be complete. That's because any test in I might be selected from a subdomain of which $n_i > 0$. These two assumptions simplify the analysis to determine whether a testing method is complete in our study.

The assumption-4 implies that a test x can be repeatedly and independently selected from the same domain, such that the current selection does not rely on the prior ones, which is used to simplify the analysis to compute the survival rate.

B. SUBDOMAIN-BASED TEST SELECTION

Given a subdomain-based testing method ω with set SD and selection vector $\vec{n}$, we can precisely describe the subdomain-based selection strategy by presenting the set of all possible test suites generated by ω, denoted as $\widehat{T}_\omega$, in which we have:

$$\widehat{T}_\omega = \prod_{i=1}^{s} (D_i)^{n_i} \tag{2}$$

The item $(D_i)^{n_i}$ is the cartesian production $D_i \times D_i \times \ldots \times D_i$ which contains n_i subdomain. For $n_i = 0$, we define $D_i^0 = \emptyset$ and any set $D \times \emptyset = D$. The notation $\prod$ presents the cartesian production of all the sets of $(D_i)^{n_i}$. Note that, since domains are overlapping and tests can be repeatedly selected, the test suite $T \in \widehat{T}_\omega$ is a multiset that may contain duplicated tests.

Note that, not all of the possible s-dimensional vectors are **feasible** selection vector. The objective of subdomain-based testing is to construct a test suite that contains at least one test in each subdomain. Formally, it requires the test suite should satisify the following constraint.

- **Test objective**: $\forall D_i \in SD, \exists x \in T$ such that $x \in D_i$.

Since a subdomain refers to a test requirement, the objective implies that the test suite should satisfy all the requirements. We call a selection vector $\vec{n}$ as feasible when there exists $T \in \widehat{T}_\omega$ that satisfies this testing objective.

In terms of feasible selection, we consider three selection strategies that have been widely used in prior analysis.

- **All-n selection** α_n: The strategy selects n random tests from each subdomain. This selection strategy has been widely used as 'standard' selection for subdomain based testing in prior research [42]–[55]. The selection vector for all-n selection is $(n, n, \ldots, n)$ in which $n_i = n$.
- **Greedy selection** ϱ: This strategy selects a subdomain of which tests are not contained in test suite, and selects a random test from it. This process continues until test suite contain at least one test from each subdomain. The greedy selection simulates the random greedy algorithm used for tests minimization, which is widely used in test suite reduction [61]–[66]. In this selection, at most one is selected from each domain in which $n_i \leq 1$. Thereby, we also call the greedy selection as the **0-1 selection**, of which vector is like $\vec{n} = (1, 1, 0, \ldots, 1, \ldots, 0)$.
- **Random selection** γ: This strategy selects random tests from the entire input space I until the test suite contains tests from every subdomain. The random selection can simulate the random testing generation, which takes the subdomain coverage as its termination condition, which is used in coverage-based test generation [32]–[41]. The random selection selects tests based on the inputs space rather than the subdomains SD, whose vector contains one number (n), which is the size of generated tests.

There are two notable things about these subdomain-based test selection strategies. First, the requirement that $\exists x \in D_i$ for each domain does not imply that $n_i > 0$. Consider that two overlapping domains D_1, D_2 in which $D_1 \cap D_2 \neq \emptyset$, it is possible that a random test selected in D_1 is from $D_1 \cap D_2$ and thus covers the D_2 together. As a result, even in greedy selection, when $n_i = 0$ holds for some domains, the objective $\forall D_i, \exists x \in D_i$ can be met by the generated test suite.

Another notable thing is that: all-n selection is a complete testing method. Since $n_i = n > 0$ holds for any subdomain in SD, according to assumption-2 and 3, for any test x in inputs space, it is possible to be selected in test suite because x must belong to some subdomain D_i, of which $n_i > 0$.

C. ANALYTICAL SOLUTIONS OF SURVIVAL RATE

To obtain analytical solutions for survival rate in subdomain based test, we introduce the notion of **subdomain efficiency**. Informally, given a subdomain D_i in SD, the likelihood that a mutant m is killed by a random test in D_i is defined as the subdomain's efficiency, denoted as $e_{i,m}$. The greater is $e_{i,m}$, the more efficient it is to select tests of D_i in killing mutant.

Formally, let p_i be a probability mass function defined on D_i, such that $p_i(x)$ returns the probability, that x is selected from the subdomain D_i. Then the efficiency of domain D_i in killing the mutant m is defined as:

$$e_{i,m} = \sum_{x \in D_i} p_i(x) \cdot kill(x, m) \tag{3}$$

The probability function p_i specifies tests profile on domain. According to assumption-3, any tests in D_i may be selected and thus that $p_i(x) > 0$ holds for any $x \in D_i$.

Usually, the probability mass function p_i is uncomputable for the most testing methods. In previous research [42]–[55], it is assumed that p_i is a uniform distribution and the tests in subdomain are fairly selected. Let $|D_i|$ be the number of tests in subdomain D_i, and F_m be the set of tests that can kill the mutant m in inputs space. Then efficiency is computed as:

$$e_{i,m} = \frac{|D_i \cap F_m|}{|D_i|} \tag{4}$$

In other words, in the uniform distribution, the efficiency $e_{i,m}$ is the proportion of tests that can kill mutant m in domain D_i. In the following analysis, we assume that the tests in domain D_i are fairly selected and p_i is a uniform distribution.

Given the efficiency $e_{i,m}$ of each subdomain in SD, and the selection vector $\vec{n}$ for testing method, the survival rate of mutant m in the testing method ω is computed as:

$$S_{m,\omega} = \prod_{D_i} (1 - e_{i,m})^{n_i} \tag{5}$$

The equation (5) could be interpreted as: (1) the probability that m survives from one test selected from subdomain D_i is $1 - e_{i,m}$; (2) the probability that mutant m survives from n_i independent tests from D_i is $(1 - e_{i,m})^{n_i}$; (3) the probability that m survives from the test suite is the production of all the probabilities that it survives from tests in every domain.

In terms of the equation (5), we could obtain the analytical solutions of survival rates for the three selection strategies in section III-B. For all-n selection, of which $\vec{n} = (n, n, \ldots, n)$, the survival rate of mutant S_{m,α_n} is:

$$S_{m,\alpha_n} = \prod_{D_i} (1 - e_{i,m})^n \tag{6}$$

For greedy selection, let $SD_{\vec{n}}$ be the set of subdomains of which $n_i = 1$, the survival rate in this testing method is:

$$S_{m,\varrho} = \prod_{D_i \in SD_{\vec{n}}} (1 - e_{i,m}) \tag{7}$$

Clearly, the survival rate in greedy selection is the production of $1 - e_{i,m}$ for subdomain in which one test is selected.

Finally, random selection selects tests based on the entire inputs space. Let e_m be the efficiency of input space I and n be the number of selected tests in test suite. Then we have:

$$S_{m,\gamma} = (1 - e_m)^n \tag{8}$$

In the analysis above, the survival rate of mutant relies on both efficiency $e_{i,m}$ and the number of random tests selected from each subdomain. The $e_{i,m}$ depends on how inputs space is divided by SD. If the subdomains "well match" the tests that kill the mutant, in which $e_{i,m} \rightarrow 1$ holds for some D_i, the mutant can be easily killed when any test is selected from the domain, of which $n_i > 0$. Meanwhile, for any subdomain with $e_{i,m} > 0$, when n_i is large enough, $n_i \rightarrow \infty$, we have $(1-e_{i,m})^{n_i} \rightarrow 0$ and $S_{i,m} \approxeq 0$. That is: if the n_i is large enough for any subdomain with non-zero efficiency $e_{i,m}$, the mutant can be revealed with high likelihood.

Clearly, there are many reasons to explain, why a mutant is stubborn for the testing method. In following analysis, we introduce a classification framework that classifies a stubborn mutant into three categories, said selective, test-number and domain stubborn, each of which refers to a particular reason for tests failing in killing the mutant and provides guidance to generate tests for it by improving random test selection.

D. SELECTION-RELATED REASONS

Let's consider the three selection strategies based on domains in SD, called greedy selection ϱ, all-1 selection α_1, and all-n selection α_n. We further assume that n is a large number that is large enough for test generation in α_n. Then the follwing relations hold for any mutant's survival rate.

$$S_{m,\varrho} \geq S_{m,\alpha_1} \geq S_{m,\alpha_n} \tag{9}$$

That $S_{m,\varrho} \geq S_{m,\alpha_1}$ is clear. According to the equation (7), that $n_i = 1$ holds for a subset $SD_{\vec{n}}$ of SD in the greedy selection, but for all domains in all-1 selection α_1. We have:

$$S_{m,\varrho} = \prod_{SD_{\vec{n}}} (1 - e_{i,m}) \geq \prod_{SD} (1 - e_{i,m}) = S_{m,\alpha_1} \tag{10}$$

Meanwhile, that $S_{m,\alpha_1} \geq S_{m,\alpha_n}$ could be proved by the following inequation.

$$S_{i,\alpha_n} = (\prod_{SD} (1 - e_{i,m}))^n = (S_{m,\alpha_1})^n \leq S_{m,\alpha_1} \tag{11}$$

That is: S_{m,α_n} is n^{th} power of S_{m,α_1} and $S_{m,\alpha_1} \geq S_{m,\alpha_n}$. In terms of the mutant survival rates in these three selection strategies, we classify a mutant as one of following classes.

- **Not-A-Stubborn** (NaS): when $S_{m,\varrho} < \theta$, the mutant is not stubborn. Since greedy selection tends to generate a minimal test set covering the given domains [61]–[66], mutant that is not stubborn for such test suite will not be considered as hard-to-kill for any testing methods based on the subdomains SD.
- **Selective-Stubborn** (SeS): when $S_{m,\alpha_1} < \theta \leq S_{m,\varrho}$, the mutant is stubborn for greedy selection while not for all-1 selection.
- **Test-number-Stubborn** (TnS): if $S_{m,\alpha_n} < \theta \leq S_{m,\alpha_n}$, a mutant is test-number-stubborn, which is stubborn for all-1 selection but not for all-n selection.
- **Domain-Stubborn** (DoS): if $\theta \leq S_{m,\alpha_n} < 1$, a mutant is domain-stubborn which is stubborn for all-n selection.
- **Equivalent**: when $S_{m,\alpha_n} = 1$, the mutant is equivalent. That's because all-n selection α_n is a complete testing method and mutant with $S_{m,\alpha_n} = 1$ must be equivalent (see theorem-1 in section II).

Figure 1 describes the decision tree to classify the mutants based on its survival rates in three test selection strategies. In this decision tree, the mutant m is classified into one of five categories, each of which correspond to a particular reason to explain why testing method based on subdomains SD fails in detecting the target mutant. For NaS mutant, any minimal test suite that covers the domains in SD can kill the mutant with a high likelihood, of which $S_{m,\varrho} < \theta$. Hence, the mutant is not useful for enhancing test quality when coverage is met.

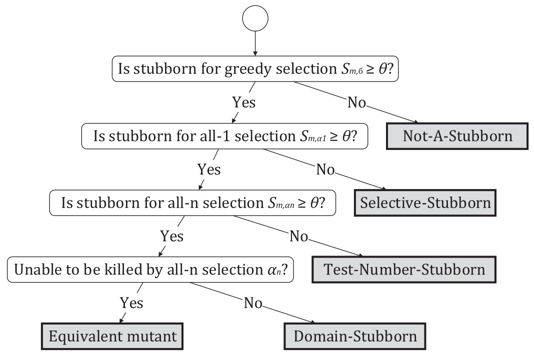
Is stubborn for greedy selection $S_{m,\varrho} \geq \theta$?
Yes
No
Not-A-Stubborn
Is stubborn for all-1 selection $S_{m,\alpha_1} \geq \theta$?
Yes
No
Selective-Stubborn
Is stubborn for all-n selection $S_{m,\alpha_n} \geq \theta$?
Yes
No
Test-Number-Stubborn
Unable to be killed by all-n selection α_n?
Yes
No
Equivalent mutant
Domain-Stubborn

FIGURE 1. Decision tree to classify mutants for selection-related reasons.

For SeS mutants, they can survive from the minimal tests that are generated by greedy selection with high likelihoods, while not for all-1 selection. The reason for SeS mutant being stubborn is that, the tests of the potential efficient subdomain are not selected by greedy selection. By selecting at least one test from each domain in *SD*, such mutant can be easily killed by the generated test suite, of which $S_{m,\alpha_1} < \theta$.

For TnS mutants, they are stubborn for all-1 selection, but not for all-n selection in which n is considered large enough. The reason for the tests of all-1 selection failing to reveal TnS mutants is that: the number of random tests selected in each domain is not large enough as for all-1 selection. According to the equation (5), if $n_i \rightarrow \infty$, we have $(1 - e_{i,m})^{n_i} \rightarrow 0$, for any subdomain with $e_{i,m} > 0$. In other words, once there exists subdomain with $e_{i,m}$ not that close to zero, TnS mutant can be killed by increasing the number of selected tests.

However, the n cannot be too large. From a perspective of practice, that $n \rightarrow \infty$ implies exhaustive testing in domains. And a upperbound for tests number n should be established in subdomain-based test selection.

Finally, for DoS mutants, even the test suites generated by all-n selection could hardly detect them in testing. The reason is that the inputs space is not ‘‘well divided’’ by *SD* such that the efficiency of each subdomain is extremely small, in which $e_{i,m} \approxeq 0$ holds for any $D_i \in SD$. It is impossible to improve the effectiveness of testing method by increasing the number of random tests. Usually, the tester has to refine the current *SD* by involving additional constraints [8], or turns to create another set SD' that is more efficient in killing the mutant.

Table 1 shows the reason and the guidance for each mutant class. It can be found that, except the DoS mutants, both the SeS and TnS mutants can be killed by enhancing the random test selection in existing subdomain in *SD*. Since the reasons for the mutants being stubborn are related with test selection strategy used, we call them **selection-related** reasons. These reasons provide the guidance to improve test quality by using the stronger selection strategy in the given subdomains, from the greedy selection to all-n selection.

TABLE 1. Selection-related reasons and their guidance for improving the subdomain-based random testing selection methods.

NaS: Not-A-Stubborn mutant class	
Property	It can be easily killed by any test suite that covers SD.
Guidance	The mutant is not needed for improving the test quality after the test coverage on subdomains is achieved.
SeS: Selective-Stubborn mutant class	
Reason	Efficient subdomains are not selected, of which $n_i = 0$.
Guidance	Select at least one test from each subdomain where $n_i > 0$.
TnS: Test-number-Stubborn mutant class	
Reason	The number of randomly selected tests is not large enough.
Guidance	Select more tests from each domain until the mutant is killed.
DoS: Domain-Stubborn mutant class	
Reason	The inputs space is not “well” divided by subdomains in SD.
Guidance	Additional constraints are required to generate the tests that are efficient in detecting the target mutant.

Meanwhile, the selection-related reasons do not answer the question of how to find a test for killing a DoS mutant. Since refining the subdomains requires understanding the program structure and constraints [8], the black-box based analysis in this section is **not** enough. In next section, we would consider how to identify the root reasons for the mutant being stubborn from a perspective of fault detection, and provide guidance to identify the key constraint for killing a DoS mutant.

IV. REACHABILITY-INFECTION-PROPAGATION

This section first reviews reachability-infection-propagation (RIP) model and defines key constraint. It then uses the RIP model to investigate the constraint-based reason for a mutant being stubborn and identify the key constraint for killing it.

A. FAULT DETECTION MODEL

The RIP model [56]–[60] divides the fault detection process into three successive stages, said reach, infect and propagate. Each stage refers to a constraint that the tests need to satisfy, shown as follows.

- C_R: The test x reaches the mutated statement l_m.
- C_I: The test causes data state errors in execution.
- C_P: The data state errors propagate to outputs.

From a perspective of constraint-based testing [8], the RIP model defines three constraints that a test needs to satisfy for revealing the mutant m. Specifically, the constraint C_R holds iff. the statement where m seeded, denoted as l_m, is executed by the test; the constraint C_I is satisfied by the test when the program states (mappings between variables and values) are different from original program after the mutated point l_m is executed; the constraint C_P holds when the errors in program state influence on and change the final outputs. Formally, for mutant being killed, the following condition needs to be met.

$$C_R \wedge C_I \wedge C_P \tag{12}$$

In [8], these constraints are used to generate the test data for detecting the mutant. Note that, in weak mutation testing [70] a mutant is determined as killed when it causes difference in program data state and propagation constraint is not needed. The condition for killing a mutant m in weak mutation testing context is shown as following:

$$C_R \wedge C_I \tag{13}$$

Since the constraints refer to three stages in fault detection, we also call them **RIP constraints** in following analysis.

From a perspective of test sets, each of the RIP constraints refers to a set of tests, including:

- **Reach-set** R_m: The set of tests that execute the mutated statement, which satisfy the constraint C_R.
- **Infect-set** I_m: The set of tests that cause data state errors in faulty point, which satisfy constraints $C_R \wedge C_I$.
- **Propagate-set** P_m: The set of tests in which data state errors propagate to outputs, satisfying $C_R \wedge C_I \wedge C_P$.

Since the test can cause data state errors only after the faulty statement is executed, there we have $I_m \subseteq R_m$. Meanwhile, the error propagation occurs, only after data state errors are caused, in which we have $P_m \subseteq I_m$.

$$P_m \subseteq I_m \subseteq R_m \subseteq I \tag{14}$$

In terms of tests sets R_m, I_m and P_m, the fault detection can be viewed as a screening process, in which less and less tests pass thorough the three successive stages, shown in figure 2. In figure 2, the tests in D_i are used to kill the mutant. Its tests are first filtered by the constraint C_R, which outputs $D_i \cap R_m$ as the tests that can execute the mutated statement in D_i. The tests are further filtered by condition C_I, which is satisfied by tests in $D_i \cap I_m$. Finally, the tests in $D_i \cap I_m$ are filtered by constraint C_P, which produces $D_i \cap P_m$ as outputs.

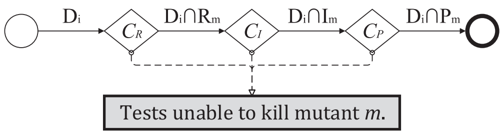

FIGURE 2. **RIP process model for screening tests.**

Note that, the propagation-set P_m is the set of tests that kill mutant m, which is the F_m in equation (4) in section III-C.

In this screening process model, we can identify the three **pass rates** for each stage in fault detection. They are:

- **Reachability-rate** $R_{i,m}$: The probability that a random test in D_i satisfies the constraint C_R, denoted as $R_{i,m}$.

$$R_{i,m} = \frac{|D_i \cap R_m|}{|D_i|} \tag{15}$$

- **Infection-rate** $I_{i,m}$: The probability that a random test that passes through the C_R can pass the constraint C_I, denoted as $I_{i,m}$, which is computed as:

$$I_{i,m} = \frac{|D_i \cap I_m|}{|D_i \cap R_m|} \tag{16}$$

If the denominator $D_i \cap R_m$ is empty, the infection-rate is defined as zero, where $I_{i,m} = 0$ for $D_i \cap R_m = \emptyset$.

- **Propagation-rate** $P_{i,m}$: The probability that a random test passing thorough C_R and C_I can meet the condition C_P, denoted as $P_{i,m}$. Then we have:

$$P_{i,m} = \frac{|D_i \cap P_m|}{|D_i \cap I_m|} \tag{17}$$

Similarly, we define $P_{i,m} = 0$ for $D_i \cap I_m = \emptyset$.

Each of the three probabilities refers to a RIP constraint and its corresponding fault detection stage. We call the probabilities as **RIP probabilities**. Since the outputs $D_i \cap P_m$ refers to the tests that can reveal the mutant in D_i, there we have:

$$R_{i,m} \cdot I_{i,m} \cdot P_{i,m} = \frac{|D_i \cap P_m|}{|D_i|} = e_{i,m} \tag{18}$$

The equation (18) reveals that, the subdomain efficiency is the production of the RIP probabilities. Meanwhile, in terms of process model in figure 2, it can be found that: if any pass rate in the three conditional branches is close with zero, then we have $e_{i,m} \approxeq 0$. This provides us a way to investigate the degree in how much a mutant is revealed by the tests in D_i, of which difficulty reasons will be discussed in following.

B. KEY CONSTRAINT IN SUBDOMAIN

In constraint-based test generation [8], test constraint is a set of conditions that need to be met by the tests. The solution of constraint C is a test that can satisfy the conditions in C. For example, in RIP model, the solution of constraint C_R is a test that can cause the mutated statement being executed.

Formally, given a subdomain D_i and constraint C. Unlike the random test generation [45], in which any tests in D_i can be fairly generated, the constraint-based test generation only selects tests in D_i that can satisfy the constraint C. We define the probability that tests generated based on C for killing the m as constraint-based domain efficiency, denoted as $e_{i,m,C}$.

$$e_{i,m,C} = \frac{|D_{i,C} \cap P_m|}{|D_{i,C}|} \tag{19}$$

in which $D_{i,C}$ is the set of tests that can satisfy C in domain D_i and P_m be the set of tests that kill the mutant m. If none of tests in D_i can satisfy C, in which $D_{i,C} = \emptyset$, we define its efficiency $e_{i,m,C} = 0$. Clearly, when $e_{i,m} = 0$, none of tests in D_i can kill the mutant, and thus $e_{i,m,C} = 0$ holds for any constraint C in this subdomain.

The subdomain efficiency $e_{i,m}$, as defined in section III-C, represents the efficiency of random testing in killing mutants, while $e_{i,m,C}$ is the efficiency of constraint-based testing. If $e_{i,m,C} > e_{i,m}$, we say that constraint C improves efficiency in killing mutant than random testing. If $e_{i,m,C} \gg e_{i,m}$, we say C dramatically increase the quality of generated tests and is the **key constraint** of D_i for killing mutant m, denoted as $K_{i,m}$. Formally, given D_i and m, a constraint C is defined as the key constraint when $e_{i,m}$ and $e_{i,m,C}$ satisfy:

$$0 < e_{i,m} < \alpha \text{ and } e_{i,m,C} > 1 - \beta \tag{20}$$

in which α and β are two small values, that are close with 0. With the key constraint used in testing, the probability that a generated test kills the mutant is increased from α (close to 0) to $1 - \beta$ (close with 1). Note that, for any subdomains with $0 < e_{i,m} < \alpha$, there **must** be the key constraints for it. That's because $C_R \wedge C_I \wedge C_P$ is a key constraint for any inefficient subdomains. When three RIP constraints are used in testing, the generated tests are guaranteed to kill mutant m, of which $e_{i,m,C} = 1$. $K_{i,m}$ may be **not unique** for given D_i and m.

C. KEY CONSTRAINTS IN RIP MODEL

Let's consider the constraint-based testing in RIP model [56]. In the three RIP constraints, said C_R, C_I, C_P, if any is used to guide the test generation in subdomain D_i, then only tests that can satisfy the constraint will be selected in testing. As a result, the test is guaranteed to pass thorough the stage which the constraint refers to. Let's consider eight compositions of RIP constraints and their efficiency, shown in table 2.

Table 2 presents the constraint C used for test generation, its effect on pass-rates of each stage in fault detection and the efficiency of generated test in killing mutant based on C.

In the first row, the constraint 'true' means no constraint is used in test generation and a random test selection is used to generate tests, of which $e_{i,m,C} = e_{i,m}$.

TABLE 2. Constraint-based testing and its effects on subdomain efficiency.

Constraint C	Effect on pass-rates	Efficiency $e_{i,m,C}$
true	None	$R_{i,m} \cdot I_{i,m} \cdot P_{i,m}$
C_R	$R_{i,m} = 1$	$I_{i,m} \cdot P_{i,m}$
C_I	$I_{i,m} = 1$	$R_{i,m} \cdot P_{i,m}$
C_P	$P_{i,m} = 1$	$R_{i,m} \cdot I_{i,m}$
$C_R \wedge C_I$	$R_{i,m} = I_{i,m} = 1$	$P_{i,m}$
$C_R \wedge C_P$	$R_{i,m} = P_{i,m} = 1$	$I_{i,m}$
$C_I \wedge C_P$	$I_{i,m} = P_{i,m} = 1$	$R_{i,m}$
$C_R \wedge C_I \wedge C_P$	$R_{i,m} = I_{i,m} = P_{i,m} = 1$	1

For $C = C_R$, C_I and C_P, only one of the RIP constraint is used to generate tests, which are guaranteed to pass thorough the corresponding stage in fault detection, of which pass rate satisfies $R_{i,m} = 1$, $I_{i,m} = 1$ and $P_{i,m} = 1$, respectively. The constraint-based efficiency $e_{i,m,C}$ is computed by substituting the corresponding pass-rate for 1 in equation (18).

Similarly, when two of RIP constraints are used in testing, the generated tests are guaranteed to pass thorough the stages in fault detection that refer to the constraints. For example, in $C = C_R \wedge C_I$, the generated tests are guaranteed to weakly kill the mutant, and C_P is the only condition these tests need to meet for killing the mutant. The efficiency for $C_R \wedge C_I$ is $P_{i,m}$, which is the pass-rate in error propagation stage.

Finally, when all of the RIP constraints are used to generate tests, in which $C = C_R \wedge C_I \wedge C_P$, the generated tests are guaranteed to pass thorough all stages in fault detection, and kill the mutant m. The efficiency for $C_R \wedge C_I \wedge C_P$ is 100%.

Table 2 provides us a way to identify the key constraint for domains based on the computed $e_{i,m,C}$ in the third column. For example, if $R_{i,m} > 1 - \beta$, it implies that $C_I \wedge C_P$ is the key constraint because its efficiency $e_{i,m,C} = R_{i,m}$.

The algorithm-1 is used to determine the key constraint for subdomain based on RIP probabilities. The first if-condition validates, whether $e_{i,m} = 0$; if so, none of tests in D_i can kill the mutant and no key constraint exists for the subdomain. In second condition, $e_{i,m} \geq \alpha$, it investigates whether the D_i is inefficient enough. As is defined in section IV-B, $e_{i,m} < \alpha$ is premise of using key constraint, since $K_{i,m}$ can dramatically increase $e_{i,m,C}$, only when the $e_{i,m}$ is inefficient enough. The algorithm will compute the $e_{i,m,C}$ for each of the eight constraints in table 2 and return the corresponding $K_{i,m}$. The condition $e_{i,m,C} > 1 - \beta$ is used to decide that $e_{i,m,C} \approx 1$, where α, β are close to 0, as is defined in section IV-B.

The key constraint provided by this algorithm can be used to guide test generation for killing a domain-stubborn mutant. More specifically, given a domain-stubborn mutant, it selects the subdomain with maximal efficiency in killing the mutant, of which $0 < e_{i,m} < \alpha$ where $\alpha = 1 - \sqrt[n]{\theta}$. That's because, based on equation (5), $S_{m,\alpha_n} > \theta$ implies $(1 - e_{i,m})^n > \theta$, which infers $e_{i,m} < 1 - \sqrt[n]{\theta}$ holds for any subdomain in SD. Since n is a large number (see definition in section III-D), we have $\sqrt[n]{\theta} \to 1$ when $n \to \infty$. As a result, let $\alpha = 1 - \sqrt[n]{\theta}$, we have $\alpha \to 0$. Given the inefficient domain D_i with $e_{i,m} < \alpha$, the algorithm is performed to identify the key constraints for the domain, which is further used to guide test generation for killing the domain-stubborn mutant.

Algorithm 1 Simplest Key Constraint Identification

```
 1: function getKeyConstraint($R_{i,m}$, $I_{i,m}$, $P_{i,m}$, $\alpha$, $\beta$)
 2:     if $R_{i,m} \cdot I_{i,m} \cdot P_{i,m} = 0$ then
 3:         return false;            ▷ No key constraint exists
 4:     else if $R_{i,m} \cdot I_{i,m} \cdot P_{i,m} \geq \alpha$ then
 5:         return true;             ▷ Constraint is not needed
 6:     else if $R_{i,m} \cdot I_{i,m} > 1 - \beta$ then
 7:         return $C_P$;
 8:     else if $R_{i,m} \cdot P_{i,m} > 1 - \beta$ then
 9:         return $C_I$;
10:     else if $I_{i,m} \cdot P_{i,m} > 1 - \beta$ then
11:         return $C_P$;
12:     else if $R_{i,m} > 1 - \beta$ then
13:         return $C_I \wedge C_P$;
14:     else if $I_{i,m} > 1 - \beta$ then
15:         return $C_R \wedge C_P$;
16:     else if $P_{i,m} > 1 - \beta$ then
17:         return $C_R \wedge C_I$;
18:     else
19:         return $C_R \wedge C_I \wedge C_P$;
20:     end if
21: end function
```

D. CONSTRAINT-BASED REASONS

The key constraint introduced in section IV-B and IV-C can be used to explain, at which stage in fault detection, the tests are most likely to fail in satisfying the constraint, and killing a domain-stubborn mutant. For example, if $K_{i,m} = C_I$, it is implied that $R_{i,m}P_{i,m} \approxeq 1$ (see algorithm-1). In other words, both $R_{i,m}$ and $P_{i,m}$ are close to 1. Meanwhile, the premise of $R_{i,m} \cdot I_{i,m} \cdot P_{i,m} < \alpha$, where $\alpha \to 0$, implies that $I_{i,m}$ is close to 0. Therefore, for domain with $K_{i,m} = C_I$, the tests of D_i are very likely to fail in the stage of infection while can easily pass thorough the reaching and propagation stages.

The following lists seven reasons to interpret, why the tests of a subdomain fail to detect a domain-stubborn mutant.

Hard-to-Reach (HR). When $K_{i,m} = C_R$, that mutant m is hard-to-reach, in which constraint C_R is difficult to be met by tests in D_i. That's because, $K_{i,m} = C_R$ implies that $I_{i,m}$ and $P_{i,m}$ are close to 1 while $R_{i,m} \approx 0$.

Hard-to-Infect (HI). If $K_{i,m} = C_I$, we say that mutant is hard-to-infect, since it is difficult for tests in D_i to pass the stage of infection, in which $I_{i,m} \approx 0$ and $R_{i,m} \cdot P_{i,m} \approx 1$.

Hard-to-Propagate (HP). When $K_{i,m} = C_P$, the mutant is hard-to-propagate, in which tests of D_i can pass thorough the reachability and infection stages ($R_{i,m} \cdot I_{i,m} \approx 1$), while it is difficult for state errors to propagate ($P_{i,m} \approx 0$).

Hard-to-Reach-Infect (HRI). When $K_{i,m} = C_R \wedge C_I$, we say a mutant is hard to reach and infect. According to the algorithm-1, it implies that $P_{i,m} \approxeq 1$, while $R_{i,m} \cdot I_{i,m} \approx 0$. In other words, it is difficult for tests in D_i to reach the faulty statement and cause data state errors (satisfying $C_R \wedge C_I$).

Hard-to-Reach-Propagate (HRP). In this case, we have $K_{i,m} = C_R \wedge C_P$, which implies that tests of D_i can easily satisfy the constraint C_I, of which $I_{i,m} \approxeq 1$, while difficult to meet the C_R and C_P, where $R_{i,m} \cdot P_{i,m} \approx 0$.

Hard-to-Infect-Propagate (HIP). If $K_{i,m} = C_I \wedge C_P$, a mutant is said hard to infect and propagate. The cause for test failing in detecting such mutant is that the tests of D_i are very likely to reach the faulty statement ($R_{i,m} \approxeq 1$) but fail to pass the infection and propagation stage ($I_{i,m} \cdot P_{i,m} \approx 0$).

Hard-to-Reach-Infect-Propagate (HRIP). In this case, $K_{i,m} = C_R \wedge C_I \wedge C_P$. In other words, the RIP probabilities are all close to 0, in which $R_{i,m}, I_{i,m}, P_{i,m} < 1 - \beta$ (see the algorithm-1). It reveals that it is difficult for the tests in D_i to pass any of the three stages in fault detection. For a tester to design a test that can kill this mutant, the three constraints need to be conquered.

Clearly, each of these reasons correspond to a unique key constraint in table 2, and can be used to explain why the tests of a subdomain fail to detect a domain-stubborn mutant, with high likelihoods, of which $0 < e_{i,m} < \alpha$. Meanwhile, as is defined in section IV-B, the key constraint $K_{i,m}$ can be used to guide the test generation and improve the test suite quality. For the domain-stubborn mutant, the constraint-based reason can reveal at which stage in fault detection, said reach, infect and propagate, the tests in subdomains are most likely to fail in passing thorough; and key constraints provide guidance for test design to kill the domain-stubborn mutant.

Finally, for the subdomain of which $e_{i,m} = 0$, the mutant is unkillable and no key constraints exist. In this case, there are three reasons for the mutant being unkillable.

- **Unable-to-reach** (UR): None of tests in D_i can execute the mutated statement, where $R_{i,m} = 0$.
- **Unable-to-infect** (UI): None of tests in D_i causes the data state error in program, where $I_{i,m} = 0$.
- **Unable-to-propagate** (UP): None of tests in D_i causes the error propagation in testing, and $P_{i,m} = 0$.

For an unkillable mutant, these reasons reveal at which stage in fault detection the tests must fail in detecting the mutant.

V. DYNAMIC-BASED IMPLEMENTATION

The analysis to estimate survival rate and identify the reason for mutant being stubborn requires an implementation of our theoretical framework in section III. This section introduces a dynamic-based technique to implement the D-RIP model, which need to conquer the following undecidable problems.

- **Subdomain division**: How to divide the program inputs into subdomains based on given test coverage criteria?
- **Tests classification**: How to classify tests to reachability, infection and propagation set defined in section IV?

We implement the dynamic-based technique in a prototype tool *JCMuta*, which can automatically identify subdomains for three coverage criterions: statement, branch and MC/DC coverage criteria, estimate the survival rate of the mutant for random, all-n and greedy selection, and report the reasons for mutants being stubborn. The idea of the dynamic technique is to use a finite and comprehensive test suite to simulate the entire inputs space and transform the analytical solution of $e_{i,m}$ and $S_{m,\omega}$ to the case that I is the finite set of tests.

The following of this section will present the details of the technique and answer how the problems above are solved by the dynamic-based implementation.

A. SUBDOMAIN DIVISION

Informally, the purpose of subdomain division is to divide the inputs space into subdomains based on the test requirements provided by the given test adequacy criteria. Formally, let I be program inputs space and R be the set of test requirements defined by test coverage criteria C. The test requirement can be any test objective, such as a statement being executed. The tests that can satisfy a requirement are defined as its domain. The output of subdomain division is a multiset of the inputs space, denoted as $SD = \{D_1, \ldots, D_s\}$, such that for each r_i in R, there is a unique subdomain $D_k \in SD$, so that D_k is the domain of the requirement r_i.

Usually, it is impossible to create a complete subdomain, because the inputs in domain can be infinite, i.e. $|D_i| \rightarrow \infty$. In static analysis, the subdomain is usually represented by the constraints that the tests need to satisfy to meet requirement, such as the path constraint for causing a statement or branch being executed [32]–[41]. However, it is impractical to obtain test constraint for each requirement, especially for programs with complicated logics.

In this paper, we develop a dynamic-based implementation to construct the subdomains approximately. It uses a finite set of tests T to simulate the (infinite) inputs space I. In this way, a subdomain can be created by clustering the inputs of T that can satisfy a specific requirement r_i.

Algorithm-2 presents how inputs space T are divided into subdomains based on test requirements R provided by a test coverage criteria C. The algorithm first constructs domain for each $r_i \in R$, and then eliminates the duplicated subdomains from SD, in which $D_i = D_j$. Finally, SD without duplicated subdomains are returned as a division of the inputs space.

Algorithm 2 Subdomain Division Algorithm

```
Input: $R$ – set of test requirements
Input: $T$ – finite set of inputs space
Output: $SD$ – subdomains based on $R$
 1: function DivideDomains($R$, $T$)
 2:     $SD = \{\}$;
 3:     for $r_i$ in $R$ do
 4:         $D_i = \{\}$; $SD = SD + \{D_i\}$;
 5:         for $t$ in $T$ do              ▷ Cluster tests based on $r_i$
 6:             if $t$ satisfies $r_i$ then
 7:                 $D_i = D_i + \{t\}$;
 8:             end if
 9:         end for
10:     end for
11:         ▷ Eliminate duplicated domains $D_i = D_j$ from $SD$
12:     $SD$ = eliminateDuplicatedDomains($SD$);
13:     return $SD$;
14: end function
```

```
int max(int x, y) {
    if(x > y)
        return x;
    else return y;
}
```
(1) Original program

```
int max(int x, y) {
    if(trap())
        return x;
    else return y;
}
```
(2) Coverage mutation

```
int max(int x, y) {
    if(assert_equals(
    x > y, x!=y))
        return x;
    else return y;
}
```
(3) Weak mutation

```
int max(int x, y) {
    /* (>, !=) */
    if(x != y)
        return x;
    else return y;
}
```
(4) Strong mutation

FIGURE 3. Example of three versions of syntactic mutations. The function `assert_equals(a,b)` **returns *a* as output when *a* = *b*, or throws exception to** `stderr` **when $a \neq b$. The function** `trap()` **will terminate the program immediately once it is called in the execution.**

Note that, the tests set used to simulate inputs space should be reasonably thorough such that for each feasible $r_i \in R$, at least one input in its domain should be selected. Meanwhile, for two non-equivalent domains $D_i \neq D_j$, at least one test in $D_i - D_j$ and $D_j - D_i$ are needed. If $D_i - D_j \neq \emptyset$ and none of tests in $D_i - D_j$ are contained in T, then algorithm-2 will consider $D_i - D_j$ as empty and obtain finite subdomains D'_i, D'_j and $D'_i \subseteq D'_j$, which may distort the results of the subdomain division.

B. MUTATION-DIRECTED TEST CLASSIFICATION

According to the RIP analysis in section IV, to estimate the domain efficiency $e_{i,m}$, reachability-rate, infection-rate and propagation-rate, we need to identify the reach-set R_m, the infect-set I_m and propagate-set P_m in finite inputs space T.

To identify the three test sets, we introduce three mutation models for a mutant, said coverage-mutation, weak-mutation and strong-mutation [70]. It means that, a mutant m could be translated into three versions of mutated code. The coverage mutation is killed once the mutated statement is executed by the test. The weak mutation is killed once the test causes the data state errors in execution. Strong mutation corresponds to the original version of mutated program seeded by m.

Figure 3 presents the three versions of syntactic mutations for the same mutant "`x!=y`" in predicate "`x>y`". In strong mutation, the predicate is directly mutated as `x!=y`. In weak mutation, an assertion is introduced to determine whether the value of $x \neq y$ equals with $x > y$. If not, the program throws an exception and terminates the execution immediately. For coverage mutation, the function call "`trap()`" immediately terminates once it is executed in testing. Obviously, the tests that detect the coverage, weak and strong mutation of a mutant refer to the R_m, I_m, P_m, respectively.

With the aids of coverage and weak mutation, the JCMuta is able to optimize the mutation testing. The idea is to use the relation $P_m \subseteq I_m \subseteq R_m$ in RIP model: the tests that cannot kill the coverage mutation of a mutant ($x \notin R_m$) will not kill its weak mutation ($x \notin I_m$), which do not need to be used to be executed against the weak mutation. Meanwhile, the tests that do not detect the weak mutation cannot detect the strong mutation, which will not be used in strong mutation testing.

Figure 4 presents the procedure: the JCMuta first executes all the tests against coverage mutation of m, and obtain R_m; it then executes only the tests in R_m against weak mutation to obtain the I_m; finally, the tests in I_m are executed against the strong mutation to obtain the set P_m. As a result, the times of execution is smaller than that of exhaustive testing, of which time complexity is $O(|M| \times |T|)$.

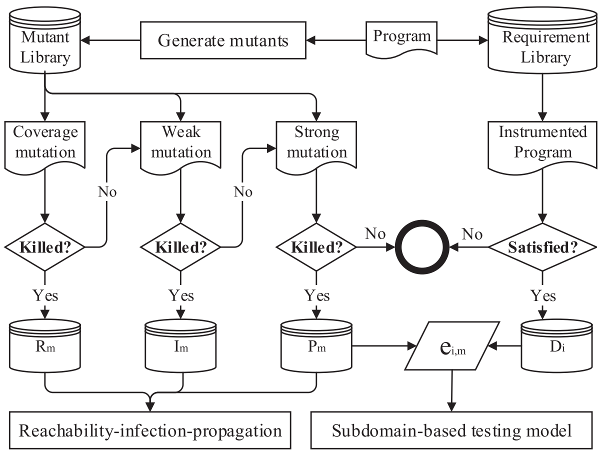

FIGURE 4. Flow graph for *JCMuta* in mutation testing and analysis.

C. PROTOTYPE TOOL

We implemented the dynamic-based technique for the D-RIP analysis in a prototype tool, named *JCMuta*. The JCMuta is a Java-implemented mutation system for testing the ANSI-C programs. It integrates all the 78 C mutation operator classes in [68] to generate various mutants and their coverage, weak and strong mutation programs automatically. It also supports the subdomain division based on finite tests T for statement, branch and MC/DC coverage criteria [26].

The tool only requires users to provide the source code of program for generating mutants and a finite set T to simulate the inputs space in analysis. Figure 4 presents a flow graph to describe how JCMuta is performed in mutation testing.

In figure 4, the mutants set M and requirements R are first generated. For each mutant m, its coverage, weak and strong mutation are successively executed against the tests to obtain sets R_m, I_m, P_m. For each requirement r_i, its instrumented program is executed against the T to verify of which tests can satisfy the requirement and obtain subdomain D_i. Finally, the D_i, R_m, I_m, P_m are used to estimate $e_{i,m}$ and survival rate by the D-RIP analysis. The outputs of JCMuta contain:

- A ranked list of the mutants from trivial to stubborn and finally to the unkilled mutants.
- The efficiency $e_{i,m}$ for each mutant and subdomain, and its survival rates in greedy, all-n and random selection.

TABLE 3. Subject programs, complexity, subdomains, mutants, tests and description.

Name	Complexity of program code				#Subdomains			Data source description			
	Lines	Block	Branch	Condition	stmt	brch	mcdc	Mutant	KillRate	#Test	Functionality
bubble_sort	39	33	14	7	5	6	9	1027	82.86%	9675	Bubble-sorting algorithm.
insert_sort	41	36	16	8	5	7	10	1042	82.53%	9597	Insert-sorting algorithm.
quick_sort	44	43	16	10	6	6	8	1017	81.91%	9601	Quick-sorting algorithm.
minmax	45	36	16	8	4	6	8	956	73.85%	9817	Minimum/maximum in numbers.
medium	53	20	12	6	11	12	17	548	85.77%	4096	Medimum in three integers.
prime	52	24	10	5	7	9	10	475	82.74%	26154	Determine a number as prime.
profit	30	18	12	6	12	13	18	1414	96.96%	30018	Salesperson's commission.
triangle	33	42	18	15	10	11	19	1048	80.73%	4096	Determine the triangle class.
days	38	46	32	18	19	21	27	1567	89.92%	4601	Day order for given date.
hero_triangle	50	33	14	8	12	13	17	1211	85.30%	4266	Heronian triangle.
max_path	49	33	12	6	6	8	10	1363	82.17%	8301	Maximal path in directed graph.
prime_factor	39	32	14	8	8	9	13	881	78.55%	7564	Prime factorization.
print_tokens	456	289	170	83	51	62	77	5370	81.56%	4072	Lexical analyzer.
print_tokens2	489	283	140	81	65	72	104	5359	84.70%	4072	Lexical analyzer.
replace	564	299	144	87	93	134	216	9010	78.18%	11082	Pattern matching.
schedule	412	148	62	36	41	50	72	2846	82.15%	2634	Priority scheduler.
schedule2	204	170	82	47	38	52	79	3035	75.91%	2679	Priority scheduler.
tcas	103	99	14	11	14	15	49	2995	81.97%	1593	Aircrafct collision avoidance.
tot_info	236	160	68	36	33	43	67	4986	82.35%	1096	Statistical analyzer.
calendar	253	105	48	27	25	28	44	4437	83.34%	6000	Calendar for given year.
md4	271	170	34	17	6	8	12	30424	82.82%	5501	MD4 message-digest algorithm.
md5	198	76	36	18	8	10	12	15542	89.16%	5501	MD5 message-digest algorithm.
Total.	3699	2195	1528	548	479	600	896	96553	83.38%	172016	Summary of all data items.

- The set of stubborn mutants and report on the cause for each of them being hard-to-kill in testing.

Note that, in dynamic-based technique, the survival rates of unkilled mutants satisfy $S_{m,\omega} \equiv 1$, which does not imply that the mutant is equivalent. However, when the test set T is mutation-adequate, for all-n selection that $S_{m,\alpha_n} = 1$ in dynamic-based technique implies that m is equivalent.

Unfortunately, it is undecidable to determine whether the unkilled mutant is equivalent until a careful proof and tedious analysis are performed. The JCMuta will assume the unkilled mutants are equivalent, until the users complement the tests that can reveal the unkilled mutants into test set T.

VI. EXPERIMENTAL EVALUATION

This section uses the techniques introduced in section III and IV to compute the survival rates of 96,000 mutants with a set of widely used programs and identify the reason for stubborn mutants being hard-to-kill in testing. We have three goals in the empirical study. They are:

- Examine the distribution of $S_{m,\omega}$ in different coverage criteria (statement, branch and MC/DC coverage), and different test selection strategies (greedy, all-1, random).
- Estimate prevalence of stubborn mutants for thresholds $\theta = 0.5, 0.75, 0.9$, respectively.
- Highlight the possible reasons that test methods studied, said statement, branch and MC/DC testing, fail in killing the identified stubborn mutants in our experiment.

The following of the section presents the subject programs used in experiment, and the results corresponding to each of the research questions above.

A. SUBJECTS

Table 3 presents the details about the subject programs used in the evaluation, including the program name, line of code, the number of blocks, branches and conditions, the number of subdomains identified by algorithm-2 in statement (stmt), branch (brch) and MC/DC (mcdc) test criteria, the number of generated mutants, the proportion of the mutants killed by tests and the number of tests in T, which are used to simulate the inputs space and compute the survival rates based on the dynamic-based technique in Section V.

The first twelve subjects from bubble_sort to prime_factor are trivial programs that have been studied in prior research on mutation stubbornness [27]. Among them, the max_path, hero_triangle and prime_factor come from the Rosetta Code[3] with one prime factorization program and two algorithms for searching problems. These trivial subjects provide the simple functionalities and input data structure that are easy-to-read. In the experiment, we perform the approximately exhaustive testing for each of these trivial programs which involves large number of test inputs. The highly exhaustive testing achieves the mutation-adequate objectives, and guarantees that all the non-equivalent mutants are killed in trivial programs.

The following seven subjects from print_tokens to tot_info are part of the Siemens Suite (available at SIR,[4]) which are widely used as a standard benchmark in research of software testing. They come with existing test suites. In the study, we used all of the test inputs provided by the benchmark in test suite. Since the original test suites are not branch-coverage, we further apply the random test generation to enhance tests until the multiple condition coverage is met.

[3]http://rosettacode.org/

[4]https://sir.unl.edu/content/sir.php

In total, more than 80% of mutants in Siemens Suite subjects are killed.

Finally, the subjects calendar, md4 and md5 are three real world utilities which provide practical applications. Calendar is used to print calendar, while md4 and md5 are the famous algorithms for message-digest in network communications. The code of md4, md5 come from IETF tools[5] and github.[6] We applied the random testing to generate thousands of tests which kill about 89% of mutants in these subjects. However, we are not sure whether the unkilled mutants are equivalent or extremely stubborn in testing. In the following analysis, we only consider stubborn mutants within the mutants killed by the tests in the study.

The programs used in the study are not very complicated: the line of code is less than 1000. The reason we consider the trivial and medium programs in empirical evaluation is that, with the increase of program complexity, the number of inputs and subdomains will explosively grow up, making it expensive to estimate the survival rates precisely, threatening the validity of our computation results.

In the study, we use the JCMuta tool to generate mutants of each subject program, and execute them to obtain the survival rate and stubbornness reasons. JCMuta implements the 78 C mutation operator classes in [68]. In total, more than 96,000 mutants are generated and 80,503 of them (83.4%) are killed by tests, which are used to investigate mutant stubbornness.

B. DISTRIBUTION OF SURVIVAL RATE

Figure 5 presents the overall distributions of survival rates in all-1 selection under statement, branch, MC/DC coverage. From this figure, it is observed that more than 70,000 mutants are extremely trivial, of which $S_{m,\omega} = 0$ for three coverage criteria. On the other hand, there are 16,050 unkilled mutants in the study, accounting for 16.63% of all generated mutants. As a result, the remaining 10,000 mutants are neither unkilled nor extremely trivial, with $0 < S_{m,\alpha_1} < 1$, within which we identify the stubborn mutants in our study.

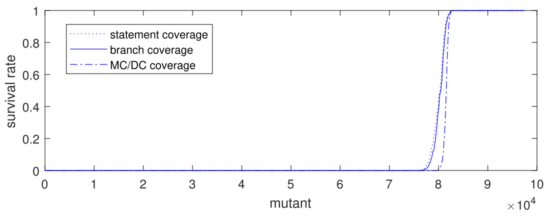

FIGURE 5. Overall distribution of survival rate in three coverage criteria (all-1 selection).

Figure 6 shows the detailed distribution of the survival rate for greedy, all-1 and random selection in the three coverage criteria. The survival rates of 15,000 mutants are presented. In this figure, the red lines refer to the survival rate computed based on subdomains of statement testing; the blue and black lines correspond to the branch and MC/DC testing. For each coverage criteria, the dashed line represents survival rates for greedy selection, while solid line and dash-dotted line refer to the survival rates computed in all-1 and random selection.

From the figure 6, it can be observed that, for almost every mutant in study, given the subdomains of a coverage criteria, its survival rates in all-1 selection (α_1), greedy selection (ϱ), and random selection (γ) satisfy the following inequalities.

$$S_{m,\varrho} \geq S_{m,\alpha_1} \geq S_{m,\gamma} \tag{21}$$

The relation $S_{m,\varrho} \geq S_{m,\alpha_1}$ has been proved in section III-D. Let's consider $S_{m,\alpha_1} \geq S_{m,\gamma}$, which is explained as follows: all-1 selection directly selects tests from each subdomain and achieve full-coverage on SD; while random selection selects tests from entire inputs space until full-coverage is met. As a result, random selection tends to select more than one tests in each subdomain and is more powerful than all-1 selection.

Another interesting finding in figure 6 is that: given a test selection strategy, survival rates in branch testing are slightly smaller than statement testing, while those in MC/DC testing is much smaller than branch testing. In other words, it reveals that the probability that tests kill a mutant in branch coverage is close with that in statement coverage for the subjects in our study. We are not sure whether this finding can hold for other programs, since the prior research [50] concludes that branch testing is much more effective than statement testing in fault detection. The further research may be needed to investigate the generalization of the findings in figure 6.

C. PREVALENCE OF STUBBORN MUTANTS

Figure 7 shows the proportion of stubborn mutants identified based on $S_{m,\omega} \geq \theta$, in which $\theta = 0.5, 0.75, 0.9$. The three thresholds identify mutants with a high likelihood to survive from each testing method in the study.

From the figure 7, it can be found that, the prevalence of stubborn mutants relies on the testing methods. For example, the weakest test strategy, greedy selection based on statement coverage, identifies 5.4% to 2.5% of generated mutants as hard-to-kill. Meanwhile, for strongest test method in study, the random selection under MC/DC coverage, only 0.1% to 0.6% of mutants are identified as stubborn. In all-1 selection, the prevalence of stubborn mutants is 2.3%, 2.1% and 1.3% in statement, branch and MC/DC coverage, where $\theta = 0.75$. In branch testing, it ranges from 2.3% to 0.4% for $\theta = 0.90$.

In total, the proportion of stubborn mutants ranges from 0.1% to 5.4%, accounting for very small number of mutants. In other words, in mutation testing, the prevalence of mutants that are useful for improving test quality in mutation testing is small, and even smaller than those of trivial mutants (78%) and the equivalent (or unkilled) mutants (16%).

D. REASONS FOR STUBBORN MUTANTS

In the experiment, we used all-n selection, of which $n = 32$, to identify the domain-stubborn mutants, whose $S_{m,\alpha_n} \geq \theta$. Meanwhile, we assigned $\alpha = \beta = 10^{-6}$ to the arguments in

[5]https://tools.ietf.org/

[6]https://github.com/pod32g/MD5

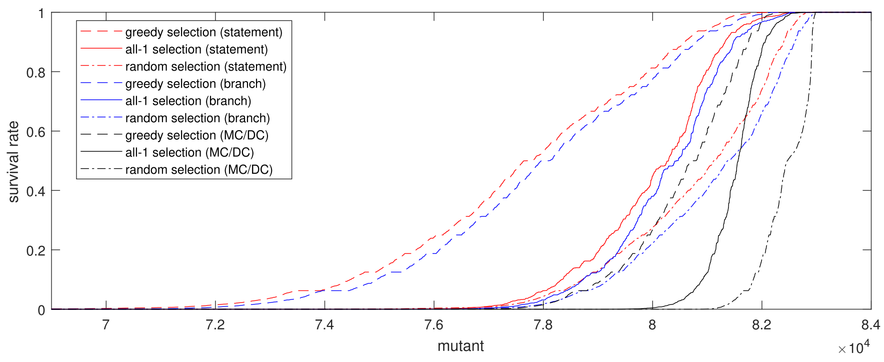

FIGURE 6. **Detailed distribution of survival rates in all-1, greedy and random selection ($69000 \leq x \leq 84000$).**

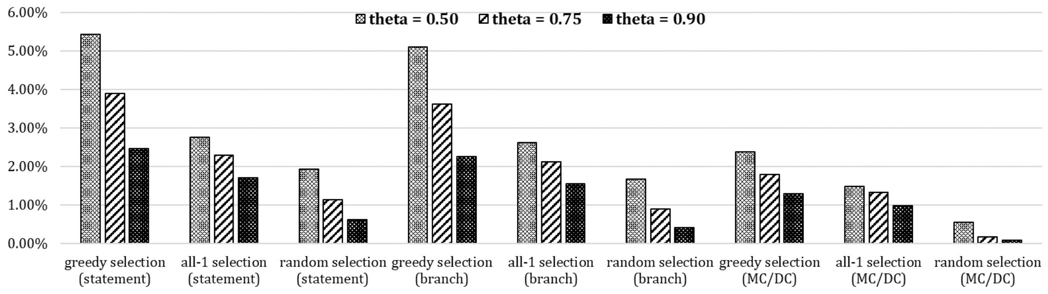

FIGURE 7. **The proportion of stubborn mutants in greedy, all-1 and random selection ($\theta = 0.5, 0.75, 0.9$).**

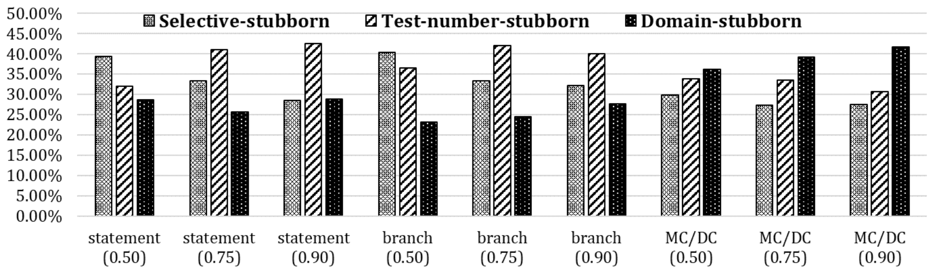

FIGURE 8. **The proportion of stubborn mutants in each selection-related reason.**

the algorithm-1, which is used to identify the key constraints, and the constraint-based reasons to explain why the test suite generated in statement, branch and MC/DC testing can fail to detect the domain-stubborn mutants in our study.

Figure 8 shows the proportion of mutants referring to each of selection-related reasons in stubborn mutants identified by greedy selection. The value in parenthesis is the threshold θ. From this figure, we can find that, the contributions made by each reason vary across different testing method. In statement and branch test, selective-stubborn and test-number stubborn are the main reasons for test suite failing in killing the mutant. In MC/DC testing, domain-stubborn is the main reason for stubborn mutants ($36\% \sim 42\%$). The reason may be that the test suite in MC/DC testing can reveal the more selective and test number stubborn mutants in statement and branch testing but less domain-stubborn mutants. The reason might be that: more tests are created for predicate with multiple conditions in MC/DC coverage; as a result, the test-number and selective stubborn mutants in branch testing can be killed with higher likelihood by test suite that meets the MC/DC coverage.

Meanwhile, the contribution made by each reason can vary across the threshold θ. For the statement and branch testing, with the increase of θ, the proportions of selective-stubborn mutants decreases, while increased is test-number-stubborn. In MC/DC testing, the proportions of the domain-stubborn ones increase with the increase of threshold. It implies that, for extremely hard-to-kill mutant ($\theta = 0.9$), its main reason is that number of tests is not large enough in branch testing, while in MC/DC testing, such mutant is extremely stubborn because the subdomains are not well divided.

From figure 8, it is concluded that: in statement and branch testing, many mutants are not killed because the tests number is not large enough; while in MC/DC testing, the number of tests for each predicate is increased by covering its conditions and the most mutants that survive from MC/DC tests refer to the reason that the subdomains are inefficient in killing them.

Table 4 further shows the proportions of domain-stubborn and unkilled mutants in each constraint-based reason.

TABLE 4. Constraint-based reason for domain-stubborn ($\theta = 0.75$) and unkilled mutants.

	hard-reach	hard-infect	hard-propagate
Statement	67.46%	5.70%	26.84%
Branch	68.22%	3.72%	28.07%
MC/DC	69.45%	3.67%	26.88%
	unable-reach	unable-infect	unable-propagate
unkilled	27.93%	19.19%	52.88%

From this figure, it is observed that, only three of the seven constraint-based reasons in section IV contribute on domain stubborn mutants. In three reasons, hard-to-reach is the main reason for subdomain being inefficient, which refers to 68% of domain-stubborn mutant. Hard-to-propagate is the second major reason, referring to 27% of stubborn mutants. Finally, only 3% $\sim$ 6% of stubborn mutants are hard-to-infect.

The observation above provides the guidance for testers to improve test quality: for killing a domain-stubborn mutant, the testers need to be focused on the constraint of either C_R or C_P, since the reasons for most domain-stubborn mutants either refers to hard-to-reach or hard-to-propagate, of which key constraint is reaching statement or error propagation.

Meanwhile, table 4 shows the proportions of unkilled mutants in each constraint-based reason. Unable-to-propagate is the major reason for a mutant being unkilled in our study, which contributes on 53% of unkilled mutants. The unable-to-reach is the second main reason which refers to 28% of the unkilled mutants, while unable-to-infect contributes on 19% of unkilled mutants in the study.

Note the difference between the unkilled and the stubborn mutants. The key constraint for mutant being unkilled is the error propagation C_P while it is the C_R for mutant being hard-to-kill in our study. For both the unkilled and stubborn mutant, the least number of them are hard (unable) to infect, which survive from tests that fail in infecting program state.

VII. DISCUSSION

This section will discuss the threats on validity of our empirical study in section VI and the limitation of D-RIP technique.

A. CONSTRUCT VALIDITY

These threats regard the used measure, i.e. survival rate and threshold θ. The survival rate provides the quantiative way to measure the difficulty of killing mutant. However, threshold θ can divide this difficulty degree in two levels (hard-to-kill or trivial). For two mutants with $S_{m,\omega}$ around θ, their difficulty of being killed is close, but one classified as stubborn, while another as the trivial. Therefore, θ-based classification might be sensitive to θ in identifying stubborn mutants.

In empirical study, we consider $\theta = 0.5, 0.75, 0.9$. From figure 7, it is found that the prevalence of stubborn mutants is sensitive to θ. For example, in greedy selection of branch testing, the proportions of stubborn mutants range from 5.2% to 2.3%. Almost the half stubborn mutants in $\theta = 0.5$ are not stubborn for $\theta = 0.9$. The sensitivity threats on identification of stubborn mutants. The θ-based classification needs to be improved in future works. In the study, we are interested in the proportion of stubborn mutant since our research purpose is to verify whether the stubborn mutants can reduce the costs of the mutation testing. In the study, no matter which θ is, the proportion of the stubborn mutants is trivial (0.1% $\sim$ 5.4%), which confirms our expectation.

B. INTERNAL VALIDITY

These threats refer to the extend to which the findings in our study is warrantied. In evaluation, we use the dynamic-based technique to compute survival rate, which depends on a finite set of tests to simulate the entire inputs space. As a result, we are not sure whether the unkilled mutants are equivalent, or stubborn mutants. The ignorance of these potential stubborn mutants may bring errors in estimating the prevalence and the contribution of each reason in section VI-C and VI-D. On the other hands, the dynamic algorithm in section V-A identifies the subdomains based on finite set T. If T is not adequate to cover all the requirements, then SD are not correctly divided.

In order to reduce the errors, we manually check unkilled mutants for all the trivial mutants and the schedule(2), tcas in Siemens Suite. We further added large numbers of random tests to Siemens Suite subjects and other subjects to achieve multiple condition coverage (stronger than MC/DC).

C. EXTERNAL VALIDITY

The programs used in our study are basically the trivial and medium, of which code line is less than 1000. Therefore, the results in section VI could only provide guidance to improve the test quality in these subjects. More real-world programs are needed to confirm the generalizability of the results.

D. LIMITATION OF D-RIP TECHNIQUE

The domain-based RIP method provides a theoretical model to explain the survival rate and the reason for a mutant being hard-to-kill. However, the theoretical solutions in section III are usually unknown. The dynamic-based implementation in section V can estimate the solutions by using large number of tests. However, this method is not practical for engineers who want to know the stubbornness of mutants before tests have been developed and provided.

Meanwhile, to estimate the survival rate theoretically is an undecidable problem, since it requires determining on which tests in a subdomain can kill the mutant. In this paper, we only use the dynamic-based technique to validate whether D-RIP is effective in measuring difficulty of killing mutant, and can be used to optimize mutation testing. Further research is needed to make the technique more practical.

VIII. RELATED WORKS

The stubbornness problems have been studied in a variety of research contexts. Weyuker [51] and Hamlet [54] observed that the fault detection probability can be used to measure the effectiveness of testing methods. Their formal analysis based on subdomain testing is then widely used to compare the test effectiveness between the partition testing and random testing [42]–[50]. The early study is focused on using fault detection probability to measure test quality, rather than considering it as a metric to measure the utility of mutants.

Until 2015, Namin *et al.* [29] first consider the probability for mutants being revealed could be used to rank the mutants and identify the useful mutants for analysis. In their study, they use the *similarity* between program representations to predict the exposability of mutants. However, the study does not consider that a mutant's survival rate can vary across the testing methods. As a result, the survival rates are estimated based on the entire inputs space, without considering how the tests are selected from each subdomain in testing practice.

Yao *et al.* [27] further propose the notion of the stubborn mutants. It is realized that the stubborn mutants are different from equivalent mutants, which can be potentially useful for guiding test process. In their study, the stubborn mutants are identified as those remain unkilled from a branch-coverage test suite while are non-equivalent. The mutation operators that are efficient in producing the stubborn mutants are further identified. However, the stubborn mutants defined in this way strongly depend on the test suite used in study, and no reasons for tests failing in killing these mutants are reported, making it difficult to guide the testers to design tests that can reveal these hard-to-kill faults effectively.

Just *et al.* [28] further developed a syntactic-pattern based technique to predict the stubbornness of mutants. In the study, the survival rate is measured based on the proportion of test suites that can detect the target mutants in random sampling. However, the proportion does not reveal any factors that can influence on stubbornness, making it difficult to understand the reasons for mutant being stubborn and provide guidance to design tests that detect the stubborn mutants.

In conclusion, the lack of such theoretical foundation and model for understanding mutation stubbornness, and relating it with testing methods and fault detection process, makes the prior research tend to identify the mutants stubborn for the specific research contexts, fails in interpreting the reasons for the difficulty of killing the mutants, and providing guidance to improve the test suite quality effectively.

IX. CONCLUSIONS

The mutation stubbornness problem has focused attention in recent studies [27]–[31], which is used to select hard-to-kill yet non-equivalent mutants (called stubborn), and guide the test improvement after structural coverage has been achieved by test suite. However, fews of these works provide a solid foundation for estimating the stubbornness, neither do they investigate the possible reasons to explain, why tests fail in killing a stubborn mutant, making it difficult to give engineer useful suggestion about how to design tests for detecting the hard-to-kill faults and improve test quality effectively.

This paper proposes domain-based RIP technique, which measures the survival rate of each mutant based on the given testing method and presents the possible reasons to interpret why the tests fail in killing the mutants, which can be used to guide the test generation and enhance testing effectiveness.

The D-RIP uses the subdomain-based testing to obtain the analytical solutions of survival rates, in which we investigate the survival rates of mutants in three classical domain-based test selection strategies, including the all-n selection, greedy selection and random selection. Based on the analytical solution of survival rate, we identify the selective-stubborn, test-number-stubborn, domain-stubborn as three mutants classes, each of which refers to the reason for a mutant being stubborn and provides guidance for testers to design tests that detect them by improving the random test selection strategies.

The D-RIP also uses RIP analytical tool to investigate the relations between difficulty of revealing mutants and the fault detection process. It reveals at which stage the test is likely to fail in revealing faults and the key constraints for killing a mutant. In the analysis, we discuss constraint-based reasons for domain being inefficient in killing mutant, which provide guidance for testers to identify the key constraints to design tests for killing the domain-stubborn mutants.

We further develop a dynamic-based implementation for D-RIP technique and integrate it to prototype tool, JCMuta. The tool can automatically compute survival rate of a mutant based on different testing methods, and generate a report that displays the reasons to explain why killing the mutant is hard, which provide guidance for improving the test suites quality.

Although dynamic-based implementation is not useful for a practical engineer, who want to know the stubborn mutants before the tests are developed, it is useful for researchers who want to investigate the reasons for mutant being hard-to-kill. It is also useful for the testers who need the key constraints to generate tests for killing the stubborn mutants and improving the test suite quality effectively.

With the aids of D-RIP and JCMuta, an empirical study is performed to evaluate our proposed method. In the study, we measure the survival rates of 96,000 mutants from 22 subject programs. The results show that only 0.1% – 5.4% of mutants are stubborn. It is also found that the contributions made by each reason for mutant being stubborn are uneven: the selective and test number stubborn are the major reasons for mutants being stubborn in statement and branch testing, while it is domain-stubborn for MC/DC tests in our study.

Meanwhile, the empirical results also reveal that, the hard-to-reach and hard-to-propagate are main reasons for a mutant being domain-stubborn, while unable-to-propagate is the major reason for a mutant being unkilled in the study. The results imply that the reachability and error propagation are the key constraints for detecting stubborn mutants and determining on whether a mutant is equivalent for programs in our study.

To our knowledge, D-RIP is the first mutation technique to relate stubbornness with the testing methods and present the possible reasons for mutant being hard-to-kill. However, it is undecidable to determine the theoretical survival rate of each mutant and the dynamic-based implementation requires vast numbers of tests to estimate the survival rate precisely, which is not useful for practical engineers who want to identify the stubborn mutants before tests are developed.

The future work might involve predicting the survival rate and identifying stubborn mutants without using test data. It is possible to be achieved by using the syntactic pattern of a mutant and machine learning technique to predict whether a mutant is stubborn, as in [28]. Another future work refers to investigating whether the stubbornness reasons can improve the efficiency of tests design in killing the stubborn mutants. Further experiments are required to address these issues.

REFERENCES

[1] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, Sep./Oct. 2011.

[2] T. A. Budd, "Mutation analysis of program test data," Ph.D. dissertation, Dept. Comput. Sci., Yale Univ., New Haven, CT, USA, 1981.

[3] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, Apr. 1978, doi: 10.1109/C-M.1978.218136.

[4] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, 2014, pp. 654–665, doi: 10.1145/2635868.2635929.

[5] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proc. 27th Int. Conf. Softw. Eng. (ICSE)*, Saint Louis, MO, USA, 2005, pp. 402–411, doi: 10.1109/ICSE.2005.1553583.

[6] M. Daran and P. Thévenod-Fosse, "Software error analysis: A real case study involving real faults and mutations," in *Proc. ISSTA*, 1996, pp. 158–171.

[7] M. Papadakis, D. Shin, S. Yoo, and D.-H. Bae, "Are mutation scores correlated with real fault detection? A large scale empirical study on the relationship between mutants and real faults," in *Proc. 40th Int. Conf. Softw. Eng. (ICSE)*, 2018, pp. 537–548, doi: 10.1145/3180155.3180183.

[8] R. A. DeMilli and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Trans. Softw. Eng.*, vol. 17, no. 9, pp. 900–910, Sep. 1991, doi: 10.1109/32.92910.

[9] M. Harman, Y. Jia, and W. B. Langdon, "Strong higher order mutation-based test data generation," in *Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Found. Softw. Eng. (ESEC/FSE)*, 2011, pp. 212–222, doi: 10.1145/2025113.2025144.

[10] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. de Halleux, and H. Mei, "Test generation via dynamic symbolic execution for mutation testing," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Timisoara, Romania, Sep. 2010, pp. 1–10, doi: 10.1109/ICSM.2010.5609672.

[11] M. Papadakis and N. Malevris, "Automatic mutation test case generation via dynamic symbolic execution," in *Proc. IEEE 21st Int. Symp. Softw. Rel. Eng.*, San Jose, CA, USA, Nov. 2010, pp. 121–130, doi: 10.1109/ISSRE.2010.38.

[12] R. Baker and I. Habli, "An empirical evaluation of mutation testing for improving the test quality of safety-critical software," *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 787–805, Jun. 2013, doi: 10.1109/TSE.2012.56.

[13] B. H. Smith and L. Williams, "On guiding the augmentation of an automated test suite via mutation analysis," *Empirical Softw. Eng.*, vol. 14, no. 3, pp. 341–369, Jun. 2009, doi: 10.1007/s10664-008-9083-7.

[14] A. J. Offutt, J. Pan, K. Tewary, and T. Zhang, "An experimental evaluation of data flow and mutation testing," *Softw., Pract. Exper.*, vol. 26, no. 2, pp. 165–176, Feb. 1996.

[15] P. G. Frankl, S. N. Weiss, and C. Hu, "All-uses vs mutation testing: An experimental comparison of effectiveness," *J. Syst. Softw.*, vol. 38, no. 3, pp. 235–253, 1997.

[16] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Trans. Softw. Eng.*, vol. 32, no. 8, pp. 608–624, Aug. 2006, doi: 10.1109/TSE.2006.83.

[17] G. Kaminski, P. Ammann, and J. Offutt, "Better predicate testing," in *Proc. 6th Int. Workshop Automat. Softw. Test (AST)*, 2011, pp. 57–63, doi: 10.1145/1982595.1982608.

[18] P. Ammann, M. E. Delamaro, and J. Offutt, "Establishing theoretical minimal sets of mutants," in *Proc. IEEE 7th Int. Conf. Softw. Test., Verification Validation*, Cleveland, OH, USA, Mar. 2014, pp. 21–30, doi: 10.1109/ICST.2014.13.

[19] B. Kurtz, P. Ammann, M. E. Delamaro, J. Offutt, and L. Deng, "Mutant subsumption graphs," in *Proc. IEEE 7th Int. Conf. Softw. Test., Verification Validation Workshops*, Cleveland, OH, USA, Mar. 2014, pp. 176–185, doi: 10.1109/ICSTW.2014.20.

[20] B. Kurtz, P. Ammann, and J. Offutt, "Static analysis of mutant subsumption," in *Proc. IEEE 8th Int. Conf. Softw. Test., Verification Validation Workshops (ICSTW)*, Graz, Austria, 2015, pp. 1–10, doi: 10.1109/ICSTW.2015.7107454.

[21] B. Kurtz, P. Ammann, J. Offutt, and M. Kurtz, "Are we there yet? How redundant and equivalent mutants affect determination of test completeness," in *Proc. IEEE 9th Int. Conf. Softw. Test., Verification Validation Workshops (ICSTW)*, Chicago, IL, USA, Apr. 2016, pp. 142–151, doi: 10.1109/ICSTW.2016.41.

[22] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. Le Traon, "Threats to the validity of mutation-based test assessment," in *Proc. 25th Int. Symp. Softw. Test. Anal. (ISSTA)*, 2016, pp. 354–365, doi: 10.1145/2931037.2931040.

[23] T. A. Budd and D. Angluin, "Two notions of correctness and their relation to testing," *Acta Inform.*, vol. 18, pp. 31–45, Mar. 1982.

[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, pp. 23–42, Jan. 2014, doi: 10.1109/TSE.2013.44.

[25] D. Schuler and A. Zeller, "Covering and uncovering equivalent mutants," *Softw. Test. Verification Rel.*, vol. 23, pp. 353–374, Aug. 2013, doi: 10.1002/stvr.1473.

[26] H. Zhu, P. A. V. Hall, and H. R. John, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, no. 4, pp. 366–427, Dec. 1997, doi: 10.1145/267580.267590.

[27] X. Yao, M. Harman, and Y. Jia, "A study of equivalent and stubborn mutation operators using human analysis of equivalence," in *Proc. 36th Int. Conf. Softw. Eng. (ICSE)*, 2014, pp. 919–930, doi: 10.1145/2568225.2568265.

[28] R. Just, B. Kurtz, and P. Ammann, "Inferring mutant utility from program context," in *Proc. 26th ACM SIGSOFT Int. Symp. Softw. Test. Anal. (ISSTA)*, 2017, pp. 284–294, doi: 10.1145/3092703.3092732.

[29] A. S. Namin, X. Xue, O. Rosas, and P. Sharma, "MuRanker: A mutant ranking tool," *Softw. Test. Verification Rel.*, vol. 25, pp. 572–604, Aug. 2015, doi: 10.1002/stvr.1542.

[30] W. Visser, "What makes killing a mutant hard," in *Proc. 31st IEEE/ACM Int. Conf. Automat. Softw. Eng. (ASE)*, 2016, pp. 39–44, doi: 10.1145/2970276.2970345.

[31] B. Lindstrom, J. Offutt, L. Gonzalez-Hernandez, and S. F. Andler, "Identifying useful mutants to test time properties," in *Proc. IEEE Int. Conf. Softw. Test., Verification Validation Workshops (ICSTW)*, Västerås, Sweden, Apr. 2018, pp. 69–76, doi: 10.1109/ICSTW.2018.00030.

[32] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, 2005, pp. 213–223, doi: 10.1145/1065010.1065036.

[33] C. Cadar *et al.*, "Symbolic execution for software testing in practice: Preliminary assessment," in *Proc. 33rd Int. Conf. Softw. Eng. (ICSE)*, 2011, pp. 1066–1071, doi: 10.1145/1985793.1985995.

[34] N. Alshahwan and M. Harman, "State aware test case regeneration for improving Web application test suite coverage and fault detection," in *Proc. Int. Symp. Softw. Test. Anal. (ISSTA)*, 2012, pp. 45–55, doi: 10.1145/2338965.2336759.

[35] Q. Yang, J. J. Li, and D. M. Weiss, "A survey of coverage-based testing tools," in *Proc. Int. Workshop Automat. Softw. Test (AST)*, 2006, pp. 99–103, doi: 10.1145/1138929.1138949.

[36] K. Lakhotia, M. Harman, and H. Gross, "AUSTIN: An open source tool for search based software testing of C programs," *Inf. Softw. Technol.*, vol. 55, no. 1, pp. 112–125, Jan. 2013, doi: 10.1016/j.infsof.2012.03.009.

[37] J. Malburg and G. Fraser, "Combining search-based and constraint-based testing," in *Proc. 26th IEEE/ACM Int. Conf. Automat. Softw. Eng. (ASE)*, Nov. 2011, pp. 436–439, doi: 10.1109/ASE.2011.6100092.

[38] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, May 2018, Art. no. 50, doi: 10.1145/3182657.

[39] R. Majumdar and K. Sen, "Hybrid concolic testing," in *Proc. 29th Int. Conf. Softw. Eng. (ICSE)*, May 2007, pp. 416–426, doi: 10.1109/ICSE.2007.41.

[40] S. J. Galler and B. K. Aichernig, "Survey on test data generation tools," *Int. J. Softw. Tools Technol. Transf.*, vol. 16, no. 6, pp. 727–751, Nov. 2014, doi: 10.1007/s10009-013-0272-3.

[41] P. McMinn, "Search-based software test data generation: A survey," *Softw. Test. Verification Rel.*, vol. 14, no. 2, pp. 105–156, 2004, doi: 10.1002/stvr.294.

[42] A. Ganjali. (2009). A requirements-based partition testing framework using particle swarm optimization technique. UWSpace. [Online]. Available: http://hdl.handle.net/10012/4242

[43] J. Mayer and C. Schneckenburger, "An empirical analysis and comparison of random testing techniques," in *Proc. ACM/IEEE Int. Symp. Empirical Softw. Eng. (ISESE)*, New York, NY, USA, 2006, pp. 105–114, doi: 10.1145/1159733.1159751.

[44] S. Morasca and S. Serra-Capizzano, "On the analytical comparison of testing techniques," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 4, pp. 154–164, Jul. 2004, doi: 10.1145/1013886.1007533.

[45] W. J. Gutjahr, "Partition testing vs. random testing: The influence of uncertainty," *IEEE Trans. Softw. Eng.*, vol. 25, no. 5, pp. 661–674, Sep. 1999, doi: 10.1109/32.815325.

[46] N. Simeon, "On random and partition testing," in *Proc. ACM SIGSOFT Int. Symp. Softw. Test. Anal. (ISSTA)*, W. Tracz, Ed. New York, NY, USA: ACM, 1998, pp. 42–48, doi: 10.1145/271771.271785.

[47] F. T. Chan, T. Y. Chen, I. K. Mak, and Y. T. Yu, "Proportional sampling strategy: Guidelines for software testing practitioners," *Inf. Softw. Technol.*, vol. 38, no. 12, pp. 775–782, 1996.

[48] T. Y. Chen and Y. T. Yu, "On the expected number of failures detected by subdomain testing and random testing," *IEEE Trans. Softw. Eng.*, vol. 22, no. 2, pp. 109–119, Feb. 1996, doi: 10.1109/32.485221.

[49] T. Y. Chen and Y. T. Yu, "On the relationship between partition and random testing," *IEEE Trans. Softw. Eng.*, vol. 20, no. 12, pp. 977–980, Dec. 1994, doi: 10.1109/32.368132.

[50] P. G. Frankl and E. J. Weyuker, "Provable improvements on branch testing," *IEEE Trans. Softw. Eng.*, vol. 19, no. 10, pp. 962–975, Oct. 1993, doi: 10.1109/32.245738.

[51] E. J. Weyuker, "Can we measure software testing effectiveness?" in *Proc. 1st Int. Softw. Metrics Symp.*, Baltimore, MD, USA, 1993, pp. 100–107, doi: 10.1109/METRIC.1993.263796.

[52] P. G. Frankl and E. J. Weyuker, "A formal analysis of the fault-detecting ability of testing methods," *IEEE Trans. Softw. Eng.*, vol. 19, no. 3, pp. 202–213, Mar. 1993, doi: 10.1109/32.221133.

[53] E. J. Weyuker and B. Jeng, "Analyzing partition testing strategies," *IEEE Trans. Softw. Eng.*, vol. 17, no. 7, pp. 703–711, Jul. 1991, doi: 10.1109/32.83906.

[54] R. Hamlet, "Theoretical comparison of testing methods," in *Proc. ACM SIGSOFT 3rd Symp. Softw. Test., Anal., Verification (TAV3)*, R. A. Kemmerer, Ed. New York, NY, USA: ACM, 1989, pp. 28–37, doi: 10.1145/75308.75313.

[55] J. W. Duran and S. C. Ntafos, "An evaluation of random testing," *IEEE Trans. Softw. Eng.*, vol. SE-10, no. 4, pp. 438–444, Jul. 1984, doi: 10.1109/TSE.1984.5010257.

[56] P. Ammann and J. Offutt, *Introduction to Software Testing*, 1st ed. New York, NY, USA: Cambridge Univ. Press, 2008.

[57] A. J. Offutt and J. H. Hayes, "A semantic model of program faults," in *Proc. ACM SIGSOFT Int. Symp. Softw. Test. Anal. (ISSTA)*, S. J. Zeil and W. Tracz, Eds. New York, NY, USA: ACM, 1996, pp. 195–200, doi: 10.1145/229000.226317.

[58] L. J. Morell, "A theory of fault-based testing," *IEEE Trans. Softw. Eng.*, vol. 16, no. 8, pp. 844–857, Aug. 1990, doi: 10.1109/32.57623.

[59] J. M. Voas, "PIE: A dynamic failure-based technique," *IEEE Trans. Softw. Eng.*, vol. 18, no. 8, pp. 717–727, Aug. 1992.

[60] B. Korel, "PELAS-program error-locating assistant system," *IEEE Trans. Softw. Eng.*, vol. SE-14, no. 9, pp. 1253–1260, Sep. 1988.

[61] J. A. Jones and M. J. Harrold, "Test-suite reduction and prioritization for modified condition/decision coverage," *IEEE Trans. Softw. Eng.*, vol. 29, no. 3, pp. 195–209, Mar. 2003, doi: 10.1109/TSE.2003.1183927.

[62] Y. Yu, J. A. Jones, and M. J. Harrold, "An empirical study of the effects of test-suite reduction on fault localization," in *Proc. 30th Int. Conf. Softw. Eng. (ICSE)*, 2008, pp. 201–210, doi: 10.1145/1368088.1368116.

[63] D. Jeffrey and N. Gupta, "Test suite reduction with selective redundancy," in *Proc. 21st IEEE Int. Conf. Softw. Maintenance (ICSM)*, Sep. 2005, pp. 549–558, doi: 10.1109/ICSM.2005.88.

[64] M. J. Harrold, R. Gupta, and M. L. Soffa, "A methodology for controlling the size of a test suite," *ACM Trans. Softw. Eng. Method*, vol. 2, no. 3, pp. 270–285, 1993, doi: 10.1145/152388.152391.

[65] G. Fraser and F. Wotawa, "Redundancy based test-suite reduction," in *Proc. 10th Int. Conf. Fundam. Approaches Softw. Eng. (FASE)*, M. B. Dwyer and A. Lopes, Eds. Berlin, Germany: Springer-Verlag, 2007, pp. 291–305.

[66] X.-Y. Ma, Z.-F. He, B.-K. Sheng, and C.-Q. Ye, "A genetic algorithm for test-suite reduction," in *Proc. IEEE Int. Conf. Syst., Man Cybern.*, vol. 1, Oct. 2005, pp. 133–139, doi: 10.1109/ICSMC.2005.1571134.

[67] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Softw. Eng.*, vol. 10, no. 4, pp. 405–435, Oct. 2005.

[68] H. Agrawal *et al.*, "Design of mutant operators for the C programming language," Dept. Comput. Sci., Purdue Univ., West Lafayette, IN, USA, Tech. Rep., Mar. 1989.

[69] M. E. Delamaro and J. C. Maldonado, "Proteum/IM 2.0: An integrated mutation testing environment," in *Mutation Testing for the New Century* (International Series on Advances in Database Systems), vol. 24, W. E. Wong, Ed. Norwell, MA, USA: Kluwer, 2001, pp. 91–101.

[70] W. E. Howden, "Weak mutation testing and completeness of test sets," *IEEE Trans. Softw. Eng.*, vol. SE-8, no. 4, pp. 371–379, Jul. 1982, doi: 10.1109/TSE.1982.235571.

HUAN LIN was born in Xinjiang, China, in 1989. He received the B.S. degree in electrical engineering from Center South University, Changsha, Hunan, China, in 2012, and the M.S. degree in software engineering from the School of Computer Science and Engineering, Beijing University of Aeronautics and Astronautics, Beijing, China, in 2015. He is currently pursuing the Ph.D. degree with the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications. His research interests include program analysis and mutation testing.

YAWEN WANG was born in Shaanxi, China, in 1983. She received the Ph.D. degree in communication and information system from the Beijing University of Posts and Telecommunications, Beijing, in 2010. She is currently an Associate Professor with the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications. Her research interests include static analysis and automatic software testing. She is a member of CCF.

YUNZHAN GONG was born in Shandong, China, in 1962. He received the master's degree in computer from the National University of Defense Technology in 1986, and the Ph.D. degree in computer from the Institute of Computing Technology, Chinese Academy of Sciences, in 1991. From 1995 to 2006, he was a Professor with the Armored Engineering Institute of the PLA. He has been a Professor with the Research Institute of Networking Technology, Beijing University of Posts and Telecommunications, since 2006. His current research interests include software testing and software reliability. He received the China Engineering Science Realistic Award, the Chinese Academy of Natural Sciences Award, and the Military Progress Prize in Science and Technology.

DAHAI JIN was born in Liaoning, China, in 1974. He received the master's and Ph.D. degrees in information security from the Armored Engineering Institute of the PLA in 2002 and 2006, respectively. In 2008, he was a Post-Doctoral Fellow with the Beijing University of Posts and Telecommunications, where he has been an Associate Professor since 2010. His current research interests include software testing and static analysis.