Do Mutations of Strongly Subsuming Second-Order Mutants Really Mask Each Other?

João P. Diniz
Federal University of Minas Gerais
Brazil
jpaulo@dcc.ufmg.br

Fischer Ferreira
Federal University of Ceará
Brazil
fischer.ferreira@sobral.ufc.br

Fabiano Ferrari
Federal University of São Carlos
Brazil
fcferrari@ufscar.br

Eduardo Figueiredo
Federal University of Minas Gerais
Brazil
figueiredo@dcc.ufmg.br

***Abstract*—Mutation testing is a fault-based testing criterion that is used to measure the quality of the test suites of software systems. Due to its inherent high computational cost, many studies were published in the last decades aiming at reducing computational cost and human-effort for the mutation analysis. One of the most promising areas is searching for Strongly Subsuming Higher-Order Mutants (SSHOMs), which are rare and harder to kill than their constituent first-order mutants (FOMs). Therefore, they are valuable especially because they can replace their FOMs without loss of effectiveness in the mutation testing process. One of the explanations for the SSHOMs to be harder to kill than their constituent FOMs is that the single faults (mutations) can partially mask one another, so that the combination of them is harder to detect than any of the individual faults. However, we did not find in the literature an investigation of the masking phenomenon. Therefore, the goal of this paper is to start filling this gap. More specifically, for a mutation to mask the other one, it is necessary firstly that the execution of a test case reaches all involved mutations. Therefore, we designed two complementary studies to accomplish our goal. Study #1 focuses on searching for Strongly-Subsuming Second-Order Mutants (SS2OMs) and then analyzes reaching characteristics of their constituent FOMs. We found that almost half of the SS2OMs constituent FOMs are not even reaching the other FOM. In Study #2, we designed a search strategy that considers a second-order mutant killed only if both of their mutations are reached by a failing test case execution. This strategy found much more SS2OMs than in the first study.**

***Index Terms*—mutation testing, higher-order mutants, mutants reduction, mutants reachability, empirical study**

I. Introduction

Mutation testing [1]–[3] is a testing criterion that requires tests to reveal specified faults introduced into the program under test. It is a way to measure the quality of the test suite for a software system [4]. On the one hand, mutation testing is more reliable and fills more gaps than other (nonmutation-based) forms of coverage [5]. On the other hand, it demands high computational cost and has low popularity in industry [5]. Fortunately, this scenario has been changing in recent years [5]–[8]. In mutation testing literature, tools and novel analyses have been proposed, launched, and performed over the last decades, aiming at reducing the cost of the most expensive steps of the mutation testing process [9]–[15].

Performing mutation testing consists in systematically generating slightly modified versions of a program p, called *mutants*. *Mutation operators* embed the modification rules. Ideally, each mutant is a faulty program and should behave differently from p for at least one test case; when that happens, the mutant is said to be *killed*. Mutants that remain alive after the test suite execution either require additional tests to be killed, or are equivalent to the program under test and hence cannot be killed.

A First-Order Mutant (FOM) is created by inserting a single change into the program under test [16]. To create a Higher-Order Mutant (HOM), which simulates more complex faults, it is necessary to introduce two or more changes into the program. In other words, a HOM is created by the combination of two or more FOMs. HOMs have become an object of interest in the last years mainly after the Harman, Jia and colleagues' reports [16]–[19]. Gopinath et al. [20] showed that, for their setup study, a typical real fault involves about three to four tokens (changes).

Prior research [21], [22] has shown that HOMs are less likely to be equivalent to the original program. Besides that, while it has been shown that the large majority of the HOMs are killed by any test suite that kills all FOMs [23], [24], some of them are rare, increase faults subtlety, and are more difficult to kill than their constituent FOMs. Therefore, such HOMs are valuable and it is worth to analyze them. Particularly, a Strongly Subsuming Higher-Order Mutant (SSHOM) is a HOM that can only be killed by a subset of test cases that kill all its constituent FOMs [17]. SSHOMs have their importance once they can reduce test effort, either by replacing their constituent FOMs in a mutation testing process [13], [17], [25], [26], or by reducing the number of test cases in a given test suite [16], [25].

Due to the prohibitive search space for valuable HOMs, some studies focused on small (or even toy) systems or lower degrees of HOMs like Second-Order Mutants (2OMs) [27]–

[31]. In this study, we used nine Java systems, varying from toys to open source, industry accepted projects, and also focused on Strongly Subsuming Second-Order Mutants (SS2OMs). We understand that it is worth investigating valuable 2OMs since (i) due to the smaller search space when compared to mutants of higher orders, it is possible to perform exhaustive searches in specific contexts; and (ii) some studies showed that *valuable* HOMs of higher order are formed by valuable HOMs of lower order. For instance, Omar et al. [24] observed this phenomenon in Subtle HOMs while Wong et al. [26] observed it in SSHOMs. Therefore, the SS2OMs we analysed have potential to form SSHOMs of order higher than two, which can achieve higher reduction in the number of mutants.

Initially, we had a specific aim of investigating what makes the SS2OMs harder to kill than their constituent FOMs. One argument for this phenomenon is that, when the whole test suite exercises the FOMs, one interferes with the other (or, as commonly referred to in the literature, *one mutation partially masks the other one* [17], [18], [25]). As a result, it contributes to an output (or a program state) more difficult to be distinguished by the current test cases. For a code change to interfere with another code change, it implies both changes to be executed in the same run. In our context, if an SS2OM is harder to kill than its constituent FOMs because its mutations partially mask each other, it is expected that both mutations are *reached* by the execution of at least one test case of the SS2OM killing tests; otherwise, only the effect of one mutation execution holds. Prior to understand if and how the SS2OMs' constituent FOMs mask each other, we believe that it is necessary to verify if both mutations are actually being reached by the test cases in the SS2OMs killing tests.

Based on that, the *main goal* of this paper is to investigate the *FOMs reaching*. That is, whether each SS2OM's constituent FOM is indeed reached by the execution of the test cases of the SS2OM's killing tests. For this purpose, we designed two complementary studies to support our *investigation*. **Study #1** was designed to investigate the proportion of SS2OMs so that their killing tests reach both of their mutations. We ran an exhaustive search for SS2OMs in our subject systems. The main finding for Study #1 is that, for almost half of the SS2OMs, only one of their constituent FOMs is in fact executed by the SS2OMs' killing tests, while the other one is not even reached. Consequently, such mutations cannot mask each other. This and other findings motivated us to perform **Study #2**, which comprises a novel search strategy for SS2OMs that considers a 2OM into account if, and only if, a test case execution reaches both of their mutations. In other words, if a 2OM is killed by a test case that reaches both of their mutations, the test case is included in the 2OMs' killing tests set. The search performed in the second study found not only the SS2OMs on which one FOM reaches the other in the previous study, but many more distinct SS2OMs. Given the two sets of SS2OMs found by the distinct strategies in both studies, we finally compared the mutant reduction that both sets of SS2OMs can achieve. The reduction was measured in terms of the percentage of FOMs the SS2OMs can replace. Our results show that the SS2OMs found in the second study have the potential to achieve a higher reduction.

In summary, the contributions of this work are:

- we found that the constituent FOMs of many SS2OMs are not masking each other. Even so, such SS2OMs can still replace their FOMs to reduce the number of mutants in a mutation testing process (or mutation analysis);
- a novel search strategy for SS2OMs that can be extended for orders higher than two;

Hence, we believe that finding as many properties and characteristics of SS2OMs as possible can be useful to identify strongly subsuming mutants of third-, fourth-, and even higher-orders in general, which may potentially produce higher cost reductions for mutation testing.

The remainder of this paper is structured as follows. Section II presents essential background that includes a formalization of SS2OMs and a motivating example. Section III shows the subject systems, mutation operators, and mutation generation strategy used in both studies. Sections IV and V present Studies #1 and Study #2, respectively. Section VI discusses some findings and their implications. Section VII lists some limitations and threats to validity. Section VIII compares our studies to related work, and, finally, Section IX concludes this paper and points out to possible future work.

II. Background

In this section, we describe the fundamental concepts necessary to understand the two studies reported in this paper (Sections II-A and II-B, together with a motivating example (Section II-C).

A. Strongly Subsuming Second-Order Mutants

A First-Order Mutant (FOM) is created by inserting a single syntactic change into the program under testing [16], whereas a Second-Order Mutant (2OM) is created with the introduction of two changes in the same mutant (mutated program). The set of 2OMs grows quadratically with the size of the set of FOMs from which they are combined. However, the problem becomes more complicated when more than two FOMs are combined to compose the set of HOMs. In this case, the problem becomes exponential, requiring an innovative technique to deal with the combinatorial explosion. Therefore, we focused this study on an exhaustive combination of FOMs to generate all possible 2OMs.

Given their underlying nature, HOMs are more likely to be killed than FOMs; this is a property discussed as the coupling effect hypothesis [23]. However, a small fraction of HOMs represents subtler faults, making them harder to be killed than their constituent FOMs. Jia and Harman [17] named such HOMs as *strongly subsuming higher-order mutants* (SSHOMs) and argue that they can replace their constituent FOMs without loss of test effectiveness (the mutants remain failing on the test suite) but with increased test efficiency

(fewer mutants to be executed). Therefore, SSHOMs keep effectiveness and improve efficiency.

Based on a formal definition of an SSHOM as a HOM that can only be killed by a subset of the intersection of the test cases that kill all its constituent FOMs [17], we simplified the formalism from SSHOMs to SS2OMs as follows. Let s be a 2OM composed of FOMs f_1 and f_2, T_s be the set of test cases that kills s, and T_i be the set of test cases that kills the FOM f_i. Then, s is an SS2OM if, and only if, $T_s \neq \emptyset$ and $T_s \subseteq T_1 \cap T_2$. It is possible to find examples of SS2OMs in the research reported by Wong et al. [32] (figure 1), Diniz et al. [31] (listing 1), and Jia and Harman [17] (table 4 and program 5).

B. Mutants Reduction via SS2OMs

From the literature, it is known that SSHOMs are valuable so they can provide a reduction in the number of mutants to be executed in a mutation testing process. This reduction is achieved by replacing the FOMs they subsume by the SSHOMs themselves. There are some examples in the literature on how to calculate such reduction, as presented in previous studies [25], [31]. Due to the similarity of the studies, i.e., SSHOMs being limited to the second order, and believing in more realistic results, we follow similar steps of previous work [31], as follows:

1) Split the FOMs into *subsumed* (the ones subsumed by, at least, one SS2OM) and *non-subsumed*;
2) Reduce the number of SS2OMs. This step is necessary because the number of SS2OMs are (much) larger than the number of FOMs they subsume. Choosing only the smallest fraction of SS2OMs, which are sufficient to replace all the subsumed FOMs, is modeled as an instance of the Minimum Set Cover Problem. The solution of this optimization problem is the set named *SS2OMsRed*.
3) Obtain the overall reduced set of mutants as *non-subsumed* $\cup$ *SS2OMsRed*;
4) Compute the *reduction* as the percentage of the size of the final resulting set of mutants in relation to the original number of FOMs.

C. Motivating Example

Listing 1 shows (i) the original source code of the `classify` method of the Triangle program; (ii) three out of many possible mutations represented as comments at lines 6, 16 and 18, named FOM1, FOM2 and FOM3, respectively; and (iii) two of the original test cases and, as comments, the FOMs they kill. We show only these test cases because, in the whole Triangle test suite, `testIsosceles1` is the unique test case that kills FOM1 and FOM3, while `testIsosceles1` and `testIsosceles2` are the only test cases that kill FOM2. Therefore, the killing tests (KTs) of these three FOMs are:

- $KT_{FOM1} = \{testIsosceles1\}$
- $KT_{FOM2} = \{testIsosceles1, testIsosceles2\}$
- $KT_{FOM3} = \{testIsosceles1\}$

In this example, we focused on two 2OMs: {FOM1,FOM2} and {FOM1,FOM3}. By running the test suite against

Listing 1. Triangle

```
public Type classify(int a, int b, int c) {                    1
  if (a <= 0 || b <= 0 || c <= 0)                               2
    return INVALID;                                             3
  int trian = 0;                                                4
  if (a == b)                                                   5
    trian = trian + 1;  //FOM1: trian - 1                       6
  if (a == c)                                                   7
    trian = trian + 2;                                          8
  if (b == c)                                                   9
    trian = trian + 3;                                          10
  if (trian == 0)                                               11
    if (a + b < c || a + c < b || b + c < a)                    12
      return INVALID;                                           13
    else                                                        14
      return SCALENE;                                           15
  if (trian > 3)  //FOM2: trian != 3                            16
    return EQUILATERAL;                                         17
  if (trian == 1 && a + b > c)  //FOM3: a % b                   18
    return ISOSCELES;                                           19
  else if (trian == 2 && a + c > b)                             20
    return ISOSCELES;                                           21
  else if (trian == 3 && b + c > a)                             22
    return ISOSCELES;                                           23
  return INVALID;                                               24
}                                                               25
                                                                26
// test cases                                                   27
void testIsosceles1() { //kills FOM1,FOM2,FOM3                  28
  assertEquals(ISOSCELES, classify(2, 2, 3));                   29
}                                                               30
void testIsosceles2() { //kills FOM2                            31
  assertEquals(ISOSCELES, classify(2, 3, 2));                   32
}                                                               33
```

{FOM1,FOM2}, this 2OM fails on `testIsosceles1` and `testIsosceles2`. While `testIsosceles1` executes the mutations of both FOMs, `testIsosceles2` executes only the mutation of *FOM2*, both returning `EQUILATERAL` instead of `ISOSCELES`. By running the Triangle test suite against {FOM1,FOM3}, the 2OM fails only on `testIsosceles1`. By analyzing its trace, after the mutation of *FOM1* has been executed at line 6, the `trian` variable gets `-1`, and *the expression mutated by FOM3 is not executed*; at the end, the `classify` method execution finishes at line 24, returning `INVALID` instead of `ISOSCELES`. Therefore, the killing tests and classification of the two subject 2OMs are:

- $KT_{FOM1,FOM2} = \{testIsosceles1, testIsosceles2\} \Rightarrow$ not an SS2OM
- $KT_{FOM1,FOM3} = \{testIsosceles1\} \Rightarrow$ SS2OM

Regardless of how many mutations of a HOM are executed, if a test case fails against it, the mutation testing process or any search strategy recognizes such HOM as killed. Recall that, for a HOM to be classified as an SSHOM, it depends on its killing tests.

One of the main well-established understandings for the SSHOMs being more difficult to kill than their constituent FOMs is that such FOMs mask each other. Nonetheless, as far as we are concerned, there is no study in the literature

investigating the proportion of SSHOMs whose constituent FOMs are actually masking each other. Moreover, there is no search strategy for SSHOMs that proposes to take into account a HOM only when at least one FOM masks any other FOM. For such kind of strategy, killing tests clearly change and, for the same 2OMs above, we get distinct classifications. Therefore, the killing tests and classification of the two subject 2OMs are:

- $KT_{FOM1,FOM2} = \{testIsosceles1\} \Rightarrow$ SS2OM
- $KT_{FOM1,FOM3} = \emptyset \Rightarrow$ not an SS2OM

The difference between the SS2OMs killing tests computed from both strategies is that the ones generated by the second strategy are smaller by one test case. On the one hand, without `testIsosceles2` in the {FOM1,FOM3} killing tests, it becomes an SS2OM. On the other hand, without `testIsosceles1`, {FOM1,FOM3} is no longer an SS2OM.

III. Setup for both studies

This section describes the common setup for both studies. Table I presents basic information about the selected Java systems.[1] They are from different domains and Table I shows their sizes, lines of code, number of test cases, test coverage, and FOMs we generated. We chose them aiming at a generalized sample to reflect a piece of the universe of types of systems.

The first two, Vending Machine and Triangle, are small toy programs explored in prior mutation testing [17], [26], [27], [31] and configurable systems testing [33] research. Monopoly, ECal, and Chess are larger systems than the first ones; they are basically maintained for research purposes and, consequently, are also used in previous testing-related studies [30], [34]. Finally, four systems are well-established in the industry and two of them are also present in previous works [26], [31]: Commons CSV, Commons CLI, Commons Validator (from Apache ©), and Gson (from Google ©). In both studies, we used only the original test suite available with each system.

TABLE I
Subject systems.

System	*Version*	*LoC*	*# Classes*	*# Test cases*	*% Test coverage*	*# FOMs generated*
Vending M	Exceptions	155	6	35	55.5	57
Triangle	n/a	32	2	12	96.4	138
Monopoly	n/a	1,181	40	123	97.4	866
C CSV	1.8	1,880	11	328	92.1	925
C CLI	1.4	2,702	24	318	96.0	1,082
ECal	2003.10	3,626	74	224	78.8	1,207
C Validator	1.6	7,409	70	536	86.2	3,197
Gson	2.9.0	11,036	70	1,089	68.2	3,712
Chess	n/a	4,924	38	930	58.4	5,287

We rely on the four mutation operators shown in Table II. The first three replace binary Java operators like arithmetic, relational, and logical. The last one removes Java statements. With respect to the SBR operator, given the `if` statement on its respective example, the operator is able to generate two mutations: one by removing the expression statement "`a = -b;`", and another by removing the whole `if` statement (which was represented by an empty set).

[1]We choose systems implemented in Java since it is more common in the mutation testing literature [5] and due to the ease of use of the JavaParser open source AST generator library necessary to our mutants instrumentation.

Listing 2. Metamutant code snippet.

```
trian = (READ_MUTANT(4) ? (trian % 1) :
         (READ_MUTANT(3) ? (trian \ 1) :
         (READ_MUTANT(2) ? (trian * 1) :
         (READ_MUTANT(1) ? (trian - 1) :
                           (trian + 1)))));
```

TABLE II
Mutation operators implemented.

Mutation operator		*Examples in Java* *Original*	*Mutations*
AOR	Arithmetic Operator Replacement	`a - b`	`a + b` `a * b` `a / b` `a % b`
ROR	Relational Operator Replacement	`a <= b`	`a >= b` `a > b` `a == b` `a < b` `a != b`
LCR	Logical Connector Replacement	`a || b`	`a && b`
SBR	Statement Block Removal	`if(a > b){ a = -b; }`	`if(a > b){}` `∅`

In addition, instead of generating one version of each first- or second-order mutant over a subject system, we generate a *metamutant*, i.e., a single variant of the system that encodes all possible syntactic changes (mutants). This approach is inspired by Mutant Schemata [35]. For example, when generating all mutations for the original expression `trian + 1` in line 6 of Listing 1, the resulting source code looks like that shown in Listing 2.[2] The `READ_MUTANT` method returns whether the target mutant is enabled or not. As can be seen, the original expression is also present and is executed only if all four mutants are disabled.

This technique saves compilation costs because the *metamutant* needs to be compiled just once. Moreover, mutations can be enabled/disabled when necessary at runtime. To ensure schemata to work correctly, we also track information useful to disregard mutations that cannot be enabled simultaneously. For instance, in the original expression "`c = a + b`", despite of being possible to create four FOMs while replacing the "+" arithmetic operator with the four other arithmetic operators, it is impossible to combine any of those four FOMs into a HOM.

IV. Study #1

While the great majority of HOMs are more likely to be killed than their constituent FOMs, SSHOMs are rare and valuable HOMs that are more difficult to be killed [16]–[19].

[2]The indentation was intentional, just for better code comprehension.

The most convincing explanation for this phenomenon is that the SSHOMs' constituent FOMs are acting in such a way to partially mask each other. However, previous work [31] observed that the killing tests of more than 84% of the SS2OMs found are exactly the same killing tests of their constituent FOMs. Therefore, we wonder if the SS2OMs' mutations are actually masking each other or if there may be situations in which only one FOM is reached by the killing tests.

To the best of our knowledge, there is no study investigating the following three situations, from the most specific to the most general, about SSHOMs: (i) how the mutations mask each other, (ii) whether the mutations are actually masking each other, and (iii) whether all the mutations are even being reached by a given test case that kills an SSHOM. Before being able to investigate (ii) and to understand (i), it is necessary to investigate (iii).

After manually inspecting the execution trace of a small sample of SS2OMs, we found occurrences that only one of their mutations was indeed executed by some killing tests, like in the motivating example (Section II-C). To go further in this analysis and due to search space limitations, we designed this first study in order to investigate how many test cases that kill an SS2OM (i.e., the SS2OM's killing tests) are actually reaching both mutated parts of its code.

A. Definitions

First of all, we formally define the killing tests of an SS2OM in terms of its constituent FOMs and the test cases that kill it:

- FOM_i: FOM that is part of (or is subsumed by), at least, one SS2OM;
- $SS2OM_{ij}$: SS2OM formed by (that subsumes) FOM_i and FOM_j, $i \neq j$;
- $t_k \in KT_{ij}$: test case k that kills FOM_i, FOM_j, and $SS2OM_{ij}$;
- KT_{ij}: killing tests of $SS2OM_{ij}$.

Given the definitions above, we are able to define the *reach function* r, which indicates whether the test case t_k reaches the first-order mutant FOM_i:

$$r(t_k, FOM_i) = \begin{cases} true & \text{if } t_k \text{ reaches } FOM_i \\ false & \text{otherwise} \end{cases} \quad (1)$$

For KT_{ij}, it is known that all of its test cases kill FOM_i, FOM_j, and $SS2OM_{ij}$. On the one hand, if all of those test cases only reach a single FOM, it is impossible for one FOM to mask the other. To track such situations, we define $Reach_{min}$:[3]

$$Reach_{min_{ij}} = \forall t_k \in KT_{ij} : r(t_k, FOM_i) \oplus r(t_k, FOM_j) \quad (2)$$

[3] $\oplus$ is the XOR operator; for $Reach_{min}$, either a test case only reaches one FOM or it only reaches the other FOM, never both.

On the other hand, if all the killing tests of an SS2OM reach both of its mutations, it is possible to make an assumption that the FOMs are partially masking each other. We define $Reach_{max}$ to investigate these cases:

$$Reach_{max_{ij}} = \forall t_k \in KT_{ij} : r(t_k, FOM_i) \wedge r(t_k, FOM_j) \quad (3)$$

Finally, we can observe an *intermediate* situation. At least one test case reaches both FOMs as well as at least one test case reaches only one FOM. This way, we define $Reach_{int}$ to track such SS2OMs:

$$Reach_{int_{ij}} = \exists t_k, t_l \in KT_{ij} : r(t_k, FOM_i) \wedge r(t_k, FOM_j) \wedge (r(t_l, FOM_i) \oplus r(t_l, FOM_j)) \quad (4)$$

In our motivating example (see Section II-C), {FOM1,FOM2} is $Reach_{int}$ and {FOM1,FOM3} is $Reach_{min}$.

B. Research Method

In this study, we intend to answer the following research questions.

RQ1: What is the proportion the SS2OMs so that their constituent FOMs can mask each other?

RQ2: What is the reduction achieved by SS2OMs that their constituent FOMs can mask or do not mask each other?

This study consists of four phases for each subject system's respective *metamutant*. The first three regards an *exhaustive search* for SS2OMs. The last one performs the *reaching analysis*. We implemented some optimizations based on well-known and established mutation testing tools like PIT [10] and PIT-HOM [11]. As seen next, we run only the test cases that reach the target mutants.

First, we executed the test suite with all FOMs disabled against the *metamutant* system. As output, we have all test cases that reached each FOM. Second, we enabled each FOM and executed the test cases recorded in the previous step against it. As output, we have all FOMs' killing tests. Third, for each pair of FOMs with a non-empty set of killing tests, we check if both form a valid 2OM and if their killing tests overlap, i.e., have an intersection. If the two checks hold, we executed the union of the test cases that reached each FOM against the 2OM. If the killing tests of the current 2OM is equals to or is a subset of the intersection of the killing tests of their constituent FOMs, it is a SS2OM.

Fourth, we also retrieve information about which of the SS2OMs' constituent FOMs each test case has reached. Listing 3 shows the necessary code instrumentation for the `READ_MUTANT` method. Before the execution of each test case, we reset `mutantsReached`. After the execution of a test case, if it kills the current SS2OM, we check in `mutantsReached` which mutations of the SS2OM have been executed. As output, we have all SS2OMs classified as $Reach_{max}$, $Reach_{int}$, or $Reach_{min}$.

Listing 3. READ_MUTANT snippet.

```
boolean READ_MUTANT(int mut) {
    if (isEnabled(mut)) {
        mutantsReached.add(mut);
        return true;
    }
    return false;
}
```

C. Results

The second column of Table III shows, for each subject system, the number of SS2OMs found by the strategy described in the previous subsection. The next three columns show the division of the SS2OMs according to the three reach classifications defined in this section, both in absolute numbers and percentages. Moreover, for each subject system, the number of SS2OMs representing the classification with the majority of the SS2OMs is highlighted in bold. The last three columns compare the reduction in percentage that three sets of SS2OMs achieve by replacing the subsumed FOMs: the complete set of SS2OMs, $Reach_{max}$ and $Reach_{min}$ SS2OMs, respectively. We also highlighted the highest values of the last two columns, for each system. The last row shows the overall numbers for all systems.

Overall, only 3.01% of the SS2OMs found are classified as $Reach_{int}$. The analysis of each system separately reveals variations that range from 0.51 to 8.00%. Thus, the minority of the SS2OMs have part of their killing tests reaching their two mutations and part reaching only one of their mutations. Then, we decided to focus the rest of this section on the remaining 96.99% SS2OMs, particularly analyzing the reduction they achieve.

The majority of the SS2OMs are $Reach_{max}$ (more than 53%) but, surprisingly, the number of $Reach_{min}$ SS2OMs is only 10% less. Individually, four systems have more $Reach_{max}$ and five have more $Reach_{min}$ SS2OMs. In both toy systems, up to 80.00% and 91.35% of the SS2OMs are $Reach_{min}$. For the larger systems, Commons CSV and Monopoly have much more $Reach_{max}$ SS2OMs, with 80.18% and 64.62% respectively, while Commons CLI has much more $Reach_{min}$ SS2OMs. Commons Validator, the system with more SS2OMs found, has 55.27% $Reach_{max}$. Gson and Chess, the largest systems, have few more than 50% of $Reach_{min}$ SS2OMs. Finally, ECal is the system with the most well-balanced ratio of $Reach_{max}$ SS2OMs and $Reach_{min}$, a difference of only 13 in absolute numbers in favour of the first.

Answering RQ1: Overall, in 53.52% of the SS2OMs, their constituent FOMs can mask each other because the execution of all test cases of these SS2OMs' killing tests reaches their two FOMs. In 43.47% of the SS2OMs, their FOMs do not mask each other because only one FOM is in fact executed by the killing tests. In 3.01%, the FOMs can mask each other in part of the killing tests.

Despite the number of $Reach_{max}$ and $Reach_{min}$ SS2OMs being balanced, the reductions they achieve are not, as shown in the last two columns of Table III. This result was also surprising for us. Only Commons Validator have their $Reach_{max}$ SS2OMs providing higher reduction than their $Reach_{min}$. For the other eight systems and overall, $Reach_{min}$ SS2OMs achieve higher reduction. However, when both reduction percentages are compared to the one that all SS2OMs can achieve (sixth column of Table III), it is clear that only one kind of SS2OMs is not enough. Therefore, it is necessary to combine both $Reach_{max}$ and $Reach_{min}$ SS2OMs to achieve the highest possible reduction.

Answering RQ2: The reduction $Reach_{min}$ SS2OMs achieve is higher than the reduction $Reach_{max}$ SS2OMs do overall. Respectively, 17.95% and 14.48%. However, SS2OMs from both sets are necessary to achieve the highest possible reduction.

The *most important finding of this study* is that a considerable amount of the SS2OMs' mutations are not masking the other, given that the execution of the SS2OMs' killing tests is not even reaching both mutations. In addition, *all killing tests of all $Reach_{min}$ SS2OMs found in seven subject systems execute the same mutation, while the other is never executed*. The exceptions are ECal and Commons Validator. In the former, only one $Reach_{min}$ SS2OM has part of their killing tests executing one mutation and part executing the other. In the latter system, it happens with only six $Reach_{min}$ SS2OMs. Such SS2OMs simply subsume their constituent FOMs, i.e., they can replace their constituent FOMs without loss of effectiveness and, consequently, they are still useful. Interestingly, this phenomenon may also be present in search techniques that use HOMs generated and compiled individually, techniques that generate mutations at Java *bytecode* level, or even techniques that employ mutant schemata.

Given that the constituent FOMs of many SS2OMs are not both reached by the SS2OMs' killing tests, we propose another search technique that only looks for SS2OMs such that both constituent FOMs are actually reached. In such SS2OMs, their FOMs have the potential to mask one another. The next section describes a study with the proposed search technique.

V. Study #2

We list below the facts that motivated this second study.

- The results of Study #1 show that the mutations of almost 47% of the SS2OMs found are not both reached by the SS2OMs' killing tests.
- The literature about searching for SSHOMs [16], [18], [26], [31] or performing higher-order mutation testing [11] is rich of many distinct strategies designed for these purposes. The way to select the HOMs is totally dependent of the strategy, for instance, via a genetic algorithm that chooses two HOMs to cross. These pieces of research have in common that, if a test case fails for a given HOM (despite the search strategy that suggested

TABLE III
DISTINCT SS2OMS, THEIR CLASSIFICATION AND THE REDUCTION THEY PROVIDE.

System	*SS2OMs*	$Reach_{max}$ # (%)	$Reach_{int}$ # (%)	$Reach_{min}$ # (%)	SS2OMs reduction (%)	$Reach_{max}$ reduction (%)	$Reach_{min}$ reduction (%)
Vending M	25	3 (12.00)	2 (8.00)	**20** (80.00)	14.04	1.75	**12.28**
Triangle	393	32 (8.14)	2 (0.51)	**359** (91.35)	36.23	13.77	**34.78**
Monopoly	3,324	**2,148** (64.62)	38 (1.14)	1,138 (34.24)	24.36	15.59	**19.40**
C CSV	4,430	**3,552** (80.18)	46 (1.04)	832 (18.78)	20.32	15.03	**15.24**
C CLI	1,852	419 (22.62)	76 (4.10)	**1,357** (73.27)	31.05	13.77	**28.10**
ECal	1,421	**678** (47.71)	74 (5.21)	669 (47.08)	22.54	15.16	**15.74**
C Validator	17,546	**9,697** (55.27)	736 (4.19)	7,113 (40.54)	24.52	**22.71**	20.21
Gson	6,970	3,243 (46.53)	217 (3.11)	**3,510** (50.36)	20.42	12.45	**15.76**
Chess	8,959	4,268 (47.64)	161 (1.80)	**4,530** (50.56)	20.38	10.81	**16.41**
Overall	**44,920**	**24,040** (53.52)	**1,352** (3.01)	**19,528** (43.47)	**22.37**	**14.48**	**17.95**

it), *it is treated as killed, regardless of how many of its mutated parts have indeed been executed.*

- The most accepted explanation that SSHOMs are mutants harder to kill than their constituent FOMs is because their mutations can partially mask each other [17], [18].
- The second part of the Motivating Example (Section II-C), in which we obtain distinct SS2OMs if the execution of their killing tests reaches both of their mutations.

In this study, we propose a novel search strategy for SS2OMs that considers an arbitrary 2OM killed if, and only if, at least one test case reaches both its mutations and causes the mutant to fail. It is possible to achieve this goal (i) by a dynamic search with a backtracking approach that executes each test case repeatedly until all possible combinations of mutants are exercised,[4] (ii) via static analysis to identify whether a piece of (mutated) code can reach another piece of (mutated) code, or (iii) by improving the *metamutant* instrumentation from the previous study to track whether each test case executed the desired mutated points. We implemented (iii) and named this strategy *forced reach search* (FRS). For comparison purposes in the remainder of this section, we named the search strategy of Study #1 as *exhaustive search* (ES).

A. Research Method

In this study, we formulate the following research questions.

RQ3: Are the SS2OMs found by FRS similar to the ones found by ES, specially the $Reach_{max}$ SS2OMs?

Since it is expected to have a different set of SS2OMs found by this strategy in comparison to the SS2OMs found in Study #1, it is necessary to compare them with a metric. Therefore, we also investigate the mutants reduction, as in the previous section.

RQ4: What is the *potential*[5] reduction achieved by the SS2OMs found by the FRS strategy?

To perform this study, we reused the results of the second phase described in Section IV-B and the `READ_MUTANT` method of Listing 3. Moreover, we adapted the third and fourth phases described in Section IV-B as follows. After the execution of each test case, if a given 2OM fails on it, we check in `mutantsReached` if the execution reached both mutations of the 2OM. If both checks hold, we add the current test case into the given 2OM killing tests set. FRS produces the same result presented in the second part of our motivating example in Section II-C.

B. Results

Table IV shows the results for this study. In the second column, we present the number of SS2OMs the FRS strategy was able to find. In the next three columns, we present three set operations to compare the similarities and differences between the SS2OMs found by the ES and FRS strategies. The last column presents the reduction rate for the potential SS2OMs found in this study.

TABLE IV
SS2OMS FOUND BY THE FRS STRATEGY

Systems	*FRS (# SS2OMs)*	*ES \ FRS*	*ES ∩ FRS*	*FRS \ ES*	*Reduction (%)*
Vending M	**28**	20	5	23	12.28
Triangle	83	359	34	49	15.94
Monopoly	**8,285**	1,199	2,125	6,160	34.41
C CSV	**9,946**	1,061	3,369	6,577	29.30
C CLI	**15,662**	1,358	494	15,168	36.23
ECal	**4,275**	668	753	3,522	25.27
C Validator	**27,540**	7,111	10,435	17,105	26.62
Gson	**79,189**	3,510	3,460	75,729	30.55
Chess	**20,981**	4,661	4,298	16,683	18.24
Overall	**165,989**	**19,947**	**24,973**	**141,016**	**25.75**

As shown in the second column, Triangle is the only system for which FRS found less SS2OMs than ES. The number of SS2OMs found for the other systems are considerable higher than the ones in the Study #1.

To show how distinct the SS2OMs found by ES and FRS are, we used the *set difference operation,*[6] two times for each system. First, we present in the third column of Table IV the difference from ES to FRS, i.e., the number of SS2OMs in ES that are not in FRS. To calculate the fifth column, we

[4]Such a backtracking approach is the main idea of SPLat, which is a technique implemented to test all possible feature interactions in configurable systems.

[5]We explain why using potential reduction instead of just reduction as a threat in Section VII.

[6]represented by the \ operator

did the opposite. Since ES is much smaller than FRS, except for Triangle, the same occurs for their set difference. In the other systems, FRS found many more SS2OMs than ES did. The most significant discrepancy occurs for Commons CLI and Gson, where FRS found about nine and 21 times more SS2OMs than ES, respectively.

To show the similarities of ES and FRS, in the fourth column of Table IV, we show the intersection of both sets. We can interpret that most of the $Reach_{max}$ SS2OMs found by ES can be in $ES \cup FRS$ because both have SS2OMs that one of their FOMs reaches the other. The slight difference among these sets can be understood since both strategies works differently when considering an SS2OM killed. As could be seen in our motivating example, a test case can be part of a SS2OM's killing tests set via ES and not via FRS.

Answering RQ3: The overall number of the SS2OMs found by the FRS is considerably higher than the ones found by ES. Their intersection are similar to the $Reach_{max}$ SS2OMs.

After acknowledging how different the SS2OMs found by ES and FRS are, we still need to compare such SS2OMs in terms of their usefulness. As described above, we calculate the *potential* reduction the SS2OMs found by *FRS* can achieve and show the values for each subject system in the last column of Table IV. SS2OMs from FRS of the first two toy systems have potential reductions smaller than the SS2OMs from ES. However, for the larger systems, SS2OMs from FRS have higher *potential* reduction. Therefore, it is worth to perform a more solid analysis over the SS2OMs from FRS to understand if the *potential* reduction they can provide is in fact a relevant reduction.

Answering RQ4: Not only the number of SS2OMs found by FRS is higher than those by ES, but also the *potential* reduction they can achieve is higher for all non-toy larger subject systems.

VI. Discussion and Implications

In Section IV, we discuss about the $Reach_{min}$ SS2OMs constituent FOMs'. They may have their similarity, since their killing tests are the same and only one mutation is executed when running all test cases of the SS2OMs killing tests. However, we did not investigate whether they produce the same incorrect outputs or distinct incorrect outputs, which could help identify how similar they are.

Some of these FOMs may be redundant [36]. Redundant mutants are killed whenever other ones are killed and, therefore, do not contribute to the testing process [5]. It is worth investigating, preferably statically, whether such FOMs are similar or redundant and, consequently, not eligible to be part of a HOM in a search strategy for SS2OMs (or even SSHOMs). The related literature includes many studies that help to avoid, identify and eliminate redundant mutants [21], [37]–[41] as well as useless mutants [42], [43]. Complementary, some FOMs may subsume other FOMs [44] and, before executing a search strategy for SSHOMs, it is worth eliminating the subsumed FOMs.

In our motivating example (Section II-C), {FOM1,FOM3} is recognized as an SS2OM not only by our implemented ES in Study #1 (Section IV) but also, as far as we know, by all other search strategies for SSHOMs proposed to date. These strategies have a step that selects the next HOM to be executed by the test cases. When the execution finishes, all test cases that caused the target HOM to fail compose that HOM's killing tests set, regardless of which of its mutated code was indeed executed. However, {FOM1,FOM3} is not recognized by FRS in Study #2 (Section V). With respect to {FOM1,FOM2}, it is recognized as an SS2OM only by FRS and not by all other known strategies. In that context, the following question arises: *Are the SS2OMs found by FRS also valuable and useful?* To answer this question, it is necessary deeper investigations about these SS2OMs, such as (i) if they are indeed harder to kill and (ii) if they are able to improve either the *mutation score* or the quality of the test suite in comparison to "traditional" SS2OMs (i.e., SS2OMs characterized in research prior to ours).

The primary reduction effect that SS2OMs achieve concerns the number of mutants to be executed. One usefulness of SSHOMs is that they can replace their constituent FOMs without loss of effectiveness in the mutation testing process (Section II-A). Reducing the test suite, which we did not analyse in this work, is a direct consequence of reducing the number of mutants, given that only the smallest number of test cases necessary to kill the considered mutants should be kept. The reduction proposed by Amman et al. [43] consists of minimal FOMs sets, also based on the subsumption concept. We understand that both research initiatives are complementary. In the future, we expect the research about SSHOMs could generate enough knowledge to make a mutation testing tool able to generate the minimal mutant sets (containing FOMs and HOMs) without executing any test case.

We claim FRS as an alternative, but not an improved search strategy for valuable 2OMs. It only considers situations where all test cases that kill two FOMs and the 2OM they constitute also reach both mutations in the respective 2OM. Study #2 analyzed the potential reduction the valuable 2OMs found by FRS can provide in comparison to the SS2OMs found by ES. Consequently, we shaded light on a not yet explored research topic, which can, in the future, validate or refute the *potential* reduction we found. Moreover, we believe our studies generate useful knowledge for novel and faster search strategies. However, analyzing the performance of ES and FRS is not our concern given the current stage of the research.

One may wonder (i) how often a QA team needs to run the mutation testing process to improve quality on the test suite of a given system, and (ii) how useful research like ours and the one of Amman et al. [43] are. On the one hand, due to the high computational cost, running a mutation testing process after each small change performed on a software system is impracticable. On the other hand, the mutation testing process is known as the most effective technique to improve test suites,

which is likely to decrease the number of bugs in production.

Comparing our research with that of Amman et al. [43], both process costly computational tasks by running test suites against all target mutants. Such tasks generate useful information that reveals the existence of redundant, useless or subsumed mutants that can be either removed or replaced, which results in minimum mutant sets (with either FOMs or HOMs). Therefore, both research initiatives only identify room to advance on finding as much knowledge about FOMs and SSHOMs as possible. In the future, one could generate the smallest mutant set without executing any test case.

With that, even a periodically performed mutation testing process could benefit since the SS2OMs found in Study #1 achieve 23.7% of reduction in the number of mutants. Consequently, strongly subsuming mutants of orders higher than two could achieve an even more effective reduction.

We hope the results we found in both studies can inspire the community of mutation testing researchers to investigate mutants with orders higher than two with the same analysis and novel analysis based on ours. For instance, depending on the $Reach_{min}$ or $Reach_{max}$ SS2OMs characteristics, a search strategy may favor more occurrences of one instead of the other. However, recall that both types of SS2OMs are necessary to provide the highest possible reduction.

VII. Limitations and Threats to validity

In this section, we discuss some limitations about mutant generation, test case executions, and the proposed strategies, as well as some threats to the validity of the studies and actions to mitigate them.

There may be implementation errors in our tool and in both search strategies. To minimize this threat, we randomly choose some FOMs and 2OMs to replicate their mutations manually into the original source code of the systems. We then ran all test suites against each of the modified programs and compared the test cases that failed with the killing tests reported by the strategies.

The results do not generalize, since our setup consists of four mutation operators and nine Java systems. The systems may not be sufficiently large and representative, although other studies about mutation testing also present such limitations. To mitigate this threat, we implemented mutation operators that can be applied in other programming languages and chose systems of distinct domains. Chess and Monopoly were developed and are maintained by software engineering groups for research purposes. Gson, from Google, and the other three from Apache (Commons CSV, CLI and Validator) are well-known open source projects and widely used in industry.

One may point the small set of four mutation operators. First, they are were used in recent studies [26], [31]. Second, three of them, the binary ones, are a subset of a set of five operators reported by Offutt et al. [42] as the most representative, and recent works also recognized the importance of deletion operators [45], which are represented by SBR in this paper. Third, we configured our mutation tool to generate all possible mutants in each subject system.

We opted for not modifying the source code and the original test suites available in the selected versions of the subject systems. Therefore, such assets may not present a high quality. In this case, we ran code coverage tool and ensured all test cases selected for both studies passed when no mutants are enabled. However, mutant executions can result in side-effects, such as changing the value of a static attribute. As a consequence, side effects can lead a mutant to fail due to an invalid state caused by the previous execution of another mutant. Therefore, we adapted our tool to help us identifying all side effects situations and it was necessary to disregard a few test cases and mutants in some of our subject systems. For Monopoly, it was necessary to create a *reset function* to restore the dice values. It simulates the behavior of running each test case in a different JVM instance.

Finally, regarding the FRS search strategy, note that it finds only SS2OMs that one of their constituent FOMs reaches the other. In Section V, we refer to the reduction in mutation testing process they can achieve by replacing the subsumed FOMs as *potential* reduction. This is because, differently from all other strategies, a test case somehow forces the 2OMs existence. Therefore, it deserves further analysis, as described in Section VI, to conclude such reduction is real. Otherwise, the SS2OMs can be as valuable as any random set of 2OMs.

VIII. Related work

We list below works about HOMs which are somehow similar to the context and the setup of the two studies in this paper. To the best of our knowledge, there is no work specifically about the SSHOMs' mutations masking (or even reaching) each other. However, the work of Gopinath et al. [46] presents a theory of composite faults on which complex faults of a program have their simple faults masking each other.

The literature is plenty of studies investigating 2OMs under distinct purposes. Polo et al. [27] empirically analysed mutation equivalence and mutation testing adequacy of 2OMs in comparison to FOMs. Papadakis and Malevris [28] compared the application benefits in terms of the number of produced mutants, the number of equivalent ones, and the number of test cases needed to satisfy each mutation variant criterion. Mateo et al. [30] extended Polo et al's work [27] to investigate the adequacy of second-order mutation at system level in larger systems. Kintis et al. [21] proposed four second-order mutation testing strategies: *Relaxed Dominator* and *Strict Dominator* using only 2OMs, *Mixed 20%*, and *Mixed 50%* using both FOMs and 2OMs. They empirically evaluate collateral coverage and the same aspects of Papadakis and Malevris' work [28]. In our both studies, we are dealing with valuable and hard to kill 2OMs (the strongly subsuming) by analysing the reduction they can provide in the mutation testing process.

In some of the works mentioned above, the authors proposed different forms for combining FOMs to generate 2OMs. For instance, *LastToFirst*, *DifferentOperators*, and *RandomMix* [27]; *FirstToLast*, *SameNode*, *SameUnit*, *SameUnit FirstToLast*, and *SameUnit DifferentOperators* [28]; *LastToFirst*, *BetweenOperators*, *Random*, and *EachChoice* Mateo

et al. [30]. In our both studies, our target are SS2OMs, from all possible combinations of FOMs.

Due to the exponential search space for SSHOMs, many studies in literature proposed distinct search-based software engineering (SBSE) techniques to find such valuable mutants. Most of them based on evolutionary algorithms. For instance, greedy [17], hill climbing [17], genetic algorithms [17], [25], multi-objective optimization algorithms [47], [48], and multi-objective hyper-heuristic approach to search for 2OMs, including SS2OMs [49]. In our study, we implemented two search strategies, one exhaustive and one inspired on configurable systems testing and to find all possible SS2OMs, given the setup and tooling described in Section III.

Wong et al. [26] adapted a tool to tackle the feature interactions problem in configurable systems testing to the context of mutation testing. They treated mutations as variability and adapted VarexC [50] to design $Search_{var}$, a search strategy for SSHOMs in Java systems. Based on information gathered from the discovered SSHOMs, they proposed another search strategy: $Search_{pri}$. In the second study of this paper (Section V), we implemented a search strategy for SS2OMs inspired by SPLat [51], a tool designed to find all possible feature interactions in configurable systems. One particular difference is the different type of SS2OMs we are looking for: one of their constituent FOMs must reach the other.

IX. Conclusion and future work

This paper reported on two complementary studies that were motivated by what makes the Strongly Subsuming Second-Order Mutants (SSHOMs) harder to kill than their constituent First-Order Mutants (FOMs). One of the most plausible explanations in the literature is that the SSHOMs' constituent FOMs (partially) mask each other [17], [18]. Before analyzing the masking phenomenon in general SSHOMs, we investigated whether the SS2OMs' killing tests reached both of their mutations.

The first study encompassed an exhaustive search for SS2OMs in nine subject systems. Surprisingly, in a large proportion (almost 44%) of the SS2OMs found, their constituent FOMs were not both reached by the execution of the test cases of the SS2OM's killing tests. *In other words, the FOMs are not masking each other*.

In the second study, we employed a novel search strategy for SS2OMs, the so-called *forced reach search* (FRS). FRS relies on an extra code instrumentation for the *metamutant* used in Study #1. FRS takes a 2OM into account if, and only if, the 2OM fails on a test case and its two mutations are reached by such a test case. The strategy was applied to nine programs that were used in the first study and has shown (i) a potential to find more SS2OMs than the exhaustive traditional approach, and (ii) a potential to achieve higher cost reduction for mutation testing regarding the number of mutants to be executed.

As *future work*, we intend to perform static analysis for a preliminary diagnostic if the mutations of a given 2OM can be reached by an arbitrary execution of a system under test. Such information can supply novel search strategies to avoid executing useless HOMs. Moreover, inspired on the RIPR model [52], FRS can be improved to track not only reachability but also whether the infection of a mutation propagates to the other mutation of an SS2OM candidate. It is also worth investigating strongly subsuming mutants with orders larger than two. Moreover, we intend to enlarge the set of operators used in our metamutant-based mutation implementation, as well as the set of systems that compose our dataset. We also plan to extend the study of Guimarães et al. [53], who investigated dynamic mutant subsumption relations for FOMs, to the context of HOMs, as well as to explore higher-order mutation testing tools like PIT-HOM [11].

We also afforded an online repository for our experimental artifacts, which is publicly available at https://jpaulodiniz.github.io/SSHOMs.

Acknowledgment

This research was partially supported by CNPq, CAPES, and FAPEMIG.

References

[1] R. G. Hamlet, "Testing Programs with the Aid of a Compiler," *IEEE Trans. Softw. Eng.*, vol. 3, no. 4, pp. 279–290, Jul. 1977.

[2] 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, 1978.

[3] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Mutation Analysis," School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, USA, Technical Report GIT-ICS-79/08, 1979.

[4] J. Offutt and R. H. Untch, *Mutation 2000: Uniting the Orthogonal*. Springer US, 2001, pp. 34–44.

[5] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman, "Mutation testing advances: an analysis and survey," in *Advances in Computers*. Elsevier, 2019, pp. 275–378.

[6] G. Petrovic and M. Ivankovic, "State of mutation testing at google," in *Proceedings of the 40th International Conference on Software Engineering—Software Engineering in Practice (ICSE-SEIP)*, 2018, pp. 163–171.

[7] G. Petrović, M. Ivanković, G. Fraser, and R. Just, "Does mutation testing improve testing practices?" in *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 910–921.

[8] ——, "Practical mutation testing at scale: A view from google," *IEEE Trans. Softw. Eng.*, vol. 48, no. 10, pp. 3900–3912, 2021.

[9] R. Gopinath, M. A. Alipour, I. Ahmed, C. Jensen, and A. Groce, "On the limits of mutation reduction strategies," in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 511–522.

[10] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: a practical mutation testing tool for Java," in *Proceedings of the 25th Intl. Symposium on Software Testing and Analysis (ISSTA)*, 2016, pp. 449–452.

[11] T. Laurent and A. Ventresque, "Pit-hom: An extension of pitest for higher order mutation analysis," in *Proceedings of the 14th International Workshop on Mutation Analysis (Mutation)*. IEEE, 2019, pp. 83–89.

[12] A. V. Pizzoleto, F. C. Ferrari, J. Offutt, L. Fernandes, and M. Ribeiro, "A systematic literature review of techniques and metrics to reduce the cost of mutation testing," *Journal of Systems and Software*, vol. 157, p. 110388, 2019.

[13] J. A. do Prado Lima and S. R. Vergilio, "A systematic mapping study on higher order mutation testing," *Journal of Systems and Software*, vol. 154, pp. 92–109, 2019.

[14] D. Basile, M. H. t. Beek, S. Lazreg, M. Cordy, and A. Legay, "Static detection of equivalent mutants in real-time model-based mutation testing: An empirical evaluation," *Empirical Software Engineering*, vol. 27, no. 7, p. 160, 2022.

[15] A. B. Sánchez, P. Delgado-Pérez, I. Medina-Bulo, and S. Segura, "Mutation testing in the wild: findings from github," *Empirical Software Engineering*, vol. 27, no. 6, p. 132, 2022.

[16] Y. Jia and M. Harman, "Constructing subtle faults using higher order mutation testing," in *Proceedings of the 8th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2008, pp. 249–258.

[17] ——, "Higher order mutation testing," *Information and Software Technology*, vol. 51, no. 10, pp. 1379–1393, 2009.

[18] M. Harman, Y. Jia, and W. B. Langdon, "A manifesto for higher order mutation testing," in *Proceedings of the 5th Intl. Workshop on Mutation Analysis (Mutation)*. IEEE Computer Society, 2010, pp. 80–89.

[19] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[20] R. Gopinath, C. Jensen, and A. Groce, "Mutations: How close are they to real faults?" in *Proceedings of the 25th IEEE International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2014, pp. 189–200.

[21] M. Kintis, M. Papadakis, and N. Malevris, "Evaluating mutation testing alternatives: A collateral experiment," in *Proceedings of the 17th Asia Pacific Software Engineering Conference (APSEC)*, 2010, pp. 300–309.

[22] 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 Transactions on Software Engineering*, vol. 40, no. 1, pp. 23–42, 2013.

[23] A. J. Offutt, "Investigations of the software testing coupling effect," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 1, no. 1, pp. 5–20, 1992.

[24] E. Omar, S. Ghosh, and D. Whitley, "Subtle higher order mutants," *Information and Software Technology*, vol. 81, pp. 3–18, 2017.

[25] M. Harman, Y. Jia, P. R. Mateo, and M. Polo, "Angels and monsters: An empirical investigation of potential test effectiveness and efficiency improvement from strongly subsuming higher order mutation," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2014, pp. 397–408.

[26] C.-P. Wong, J. Meinicke, L. Chen, J. P. Diniz, C. Kästner, and E. Figueiredo, "Efficiently finding higher-order mutants," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020, pp. 1165–1177.

[27] M. Polo, M. Piattini, and I. García-Rodríguez, "Decreasing the cost of mutation testing with second-order mutants," *Software Testing, Verification and Reliability*, vol. 19, no. 2, pp. 111–131, 2009.

[28] M. Papadakis and N. Malevris, "An empirical evaluation of the first and second order mutation testing strategies," in *Proceedings of the 5th Intl. Workshop on Mutation Analysis (Mutation)*, 2010, pp. 90–99.

[29] M. Kintis, M. Papadakis, and N. Malevris, "Isolating first order equivalent mutants via second order mutation," in *Proceedings of the 5th IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2012, pp. 701–710.

[30] P. R. Mateo, M. P. Usaola, and J. L. F. Aleman, "Validating second-order mutation at system level," *IEEE Transactions on Software Engineering*, vol. 39, no. 4, pp. 570–587, 2013.

[31] J. P. Diniz, C.-P. Wong, C. Kästner, and E. Figueiredo, "Dissecting strongly subsuming second-order mutants," in *Proceedings of the 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2021, pp. 171–181.

[32] C.-P. Wong, J. Meinicke, and C. Kästner, "Beyond testing configurable systems: Applying variational execution to automatic program repair and higher order mutation testing," in *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2018, pp. 749–753.

[33] F. Ferreira, G. Vale, J. P. Diniz, and E. Figueiredo, "Evaluating t-wise testing strategies in a community-wide dataset of configurable software systems," *Journal of Systems and Software*, vol. 179, p. 110990, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016412122100087X

[34] P. R. Mateo, M. P. Usaola, and J. Offutt, "Mutation at the multi-class and system levels," *Science of Computer Programming*, vol. 78, no. 4, pp. 364–387, 2013.

[35] R. H. Untch, J. Offutt, and M. J. Harrold, "Mutation analysis using mutant schemata," in *Proceedings of the 2nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 1993, pp. 139–148.

[36] B. Kurtz, P. Ammann, J. Offutt, and M. Kurtz, "Are we there yet? how redundant and equivalent mutants affect determination of test completeness," in *Proceedings of the 11th International Workshop on Mutation Analysis (Mutation)*, 2016, pp. 142–151.

[37] K.-C. Tai, "Theory of fault-based predicate testing for computer programs," *IEEE Transactions on Software Engineering*, vol. 22, no. 8, pp. 552–562, 1996.

[38] R. Just, G. M. Kapfhammer, and F. Schweiggert, "Do redundant mutants affect the effectiveness and efficiency of mutation analysis?" in *Proceedings of the 5th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2012, pp. 720–725.

[39] G. Kaminski, P. Ammann, and J. Offutt, "Improving logic-based testing," *Journal of Systems and Software*, vol. 86, no. 8, pp. 2002–2012, 2013.

[40] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon, "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique," in *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE)*, vol. 1, 2015, pp. 936–946.

[41] L. Fernandes, M. Ribeiro, L. Carvalho, R. Gheyi, M. Mongiovi, A. Santos, A. Cavalcanti, F. Ferrari, and J. C. Maldonado, "Avoiding useless mutants," in *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, 2017, pp. 187–198.

[42] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 5, no. 2, pp. 99–118, 1996.

[43] P. Ammann, M. E. Delamaro, and J. Offutt, "Establishing theoretical minimal sets of mutants," in *Proceedings of the 7th International Conference on Software Testing, Verification and Validation (ICST)*, 2014, pp. 21–30.

[44] R. Gheyi, M. Ribeiro, B. Souza, M. Guimarães, L. Fernandes, M. d'Amorim, V. Alves, L. Teixeira, and B. Fonseca, "Identifying method-level mutation subsumption relations using z3," *Information and Software Technology*, vol. 132, p. 106496, 2021.

[45] M. E. Delamaro, J. Offutt, and P. Ammann, "Designing deletion mutation operators," in *Proceedings of the 7th International Conference on Software Testing, Verification and Validation (ICST)*, 2014, pp. 11–20.

[46] R. Gopinath, C. Jensen, and A. Groce, "The theory of composite faults," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2017, pp. 47–57.

[47] Q. V. Nguyen and L. Madeyski, "Searching for strongly subsuming higher order mutants by applying multi-objective optimization algorithm," in *Advanced Computational Methods for Knowledge Engineering*. Springer, 2015, pp. 391–402.

[48] ——, "Addressing mutation testing problems by applying multi-objective optimization algorithms and higher order mutation," *Journal of Intelligent & Fuzzy Systems*, vol. 32, no. 2, pp. 1173–1182, 2017.

[49] J. A. P. Lima and S. R. Vergilio, "A multi-objective optimization approach for selection of second order mutant generation strategies," in *Proceedings of the 2nd Brazilian Symposium on Systematic and Automated Software Testing*, 2017, pp. 1–10.

[50] C.-P. Wong, J. Meinicke, L. Lazarek, and C. Kästner, "Faster variational execution with transparent bytecode transformation," *Proceedings of the ACM on Programming Languages*, vol. 2, no. 117, pp. 1–30, 2018.

[51] C. H. P. Kim, D. Marinov, S. Khurshid, D. Batory, S. Souto, P. Barros, and M. d'Amorim, "Splat: Lightweight dynamic analysis for reducing combinatorics in testing configurable systems," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (FSE)*. New York, NY, USA: Association for Computing Machinery, 2013, pp. 257–267.

[52] N. Li and J. Offutt, "Test oracle strategies for model-based testing," *IEEE Transactions on Software Engineering*, vol. 43, no. 4, pp. 372–395, 2016.

[53] M. A. Guimarães, L. Fernandes, M. Ribeiro, M. d'Amorim, and R. Gheyi, "Optimizing mutation testing by discovering dynamic mutant subsumption relations," in *Proceedings of the 13th IEEE International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 198–208.