Efficiently Finding Higher-Order Mutants

Chu-Pan Wong*
Carnegie Mellon University
USA

Jens Meinicke*
Carnegie Mellon University
USA

Leo Chen*
Carnegie Mellon University
USA

João P. Diniz
Federal University of Minas Gerais
Brazil

Christian Kästner
Carnegie Mellon University
USA

Eduardo Figueiredo
Federal University of Minas Gerais
Brazil

ABSTRACT

Higher-order mutation has the potential for improving major drawbacks of traditional first-order mutation, such as by simulating more realistic faults or improving test-optimization techniques. Despite interest in studying promising higher-order mutants, such mutants are difficult to find due to the exponential search space of mutation combinations. State-of-the-art approaches rely on genetic search, which is often incomplete and expensive due to its stochastic nature. First, we propose a novel way of finding a complete set of higher-order mutants by using *variational execution*, a technique that can, in many cases, explore large search spaces completely and often efficiently. Second, we use the identified complete set of higher-order mutants to study their characteristics. Finally, we use the identified characteristics to design and evaluate a new search strategy, independent of variational execution, that is highly effective at finding higher-order mutants even in large codebases.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; *Search-based software engineering*.

KEYWORDS

mutation analysis, higher-order mutant, variational execution

ACM Reference Format:
Chu-Pan Wong, Jens Meinicke, Leo Chen, João P. Diniz, Christian Kästner, and Eduardo Figueiredo. 2020. Efficiently Finding Higher-Order Mutants. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3368089.3409713

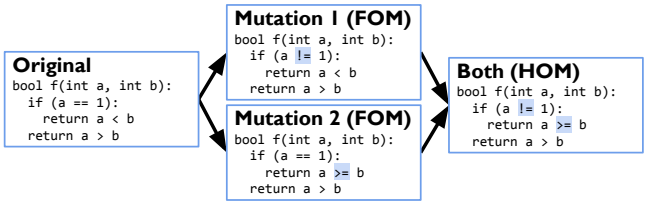
1 INTRODUCTION

Mutation analysis has been studied for decades in research [60] and is increasingly adopted in industry [63, 64]. Mutation analysis has many applications, including assessing and improving test suite quality, generating or minimizing a test suite, or as a proxy for evaluating other research techniques such as fault localization [28, 60].

*First three authors contributed equally to the paper

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11.
https://doi.org/10.1145/3368089.3409713



Test	Orig.	Mut. 1	Mut. 2	Both	Failure Cond.
T_1: `assert f(1, 2)`	✓	✗	✗	✗	$m_1 \vee m_2$
T_2: `assert !f(0, 3)`	✓	✗	✓	✓	$m_1 \wedge \neg m_2$
T_3: `assert !f(1, 1)`	✓	✓	✗	✓	$\neg m_1 \wedge m_2$

Figure 1: Example of mutations with their test outcomes.

Traditionally, mutation analysis injects syntactic mutations into an existing program and runs the existing tests to assess whether the tests are sensitive enough to detect the mutations.

Higher-order mutation is the idea of combining multiple mutations to represent more subtle changes, more complex changes, or changes that better mirror human mistakes [23]. To that end, Jia and Harman [23] distinguish *first-order mutants*, consisting of a single change, from *higher-order mutants* that combine multiple changes (cf. Fig. 1). While most research on mutation analysis has focused on first-order mutants, recent studies claim that higher-order mutants are less likely to be equivalent mutants [36, 49, 52, 61] and that higher-order mutants can reduce test effort [21, 23, 65]. In Section 2, we will discuss a specific use case with a motivating example.

A key challenge in adopting higher-order mutation is *identifying* beneficial higher-order mutants. Most higher-order mutants are as easy to kill as their constituent first-order mutants, due to coupling. Jia and Harman [23] argue that only a subset of all possible combinations better simulate real faults and increase the subtlety of the seeded faults. Specifically, Jia and Harman [21, 23] look for what they name a *strongly subsuming higher-order mutant (SSHOM)*, a particular kind of higher-order mutant that is harder to detect than its constituent first-order mutants, as we will explain in Section 2. However, SSHOMs are tricky to find among the vast quantity of possible combinations of first-order mutants. Current approaches use genetic-search techniques, guided by a simple fitness function [17, 21, 23, 43]. Since SSHOMs are difficult to find, little is known about them and their characteristics.

In this work, we develop a technique that can find a *complete* set of SSHOMs for *given first-order mutations and tests* on *small to medium-sized* programs, which enables us to study characteristics of SSHOMs. Based on the identified characteristics, we then develop a new heuristic search technique that is lightweight, scalable, and practical. Overall, we proceed in three steps:

(1) Variational Search: For the purpose of studying SSHOM in a controlled setting, we develop a new search strategy $\texttt{search}_{var}$ that allows us to find a *complete* set of higher-order mutants for a given test suite and given set of first-order mutants in small to medium-sized programs. Specifically, we use *variational execution* [3, 53, 54, 75], a dynamic-analysis technique that jointly explores many similar executions of a program. Conceptually, our approach searches for all possible higher-order mutants *at the same time*, identifying, with a propositional formula for each test case, which mutants and combinations of mutants cause a test to fail. From these formulas, we then encode search as a *Boolean satisfiability problem* to enumerate *all* SSHOMs. An exploration of *all* possible mutant combinations with variational execution is often feasible for *small to medium-sized* programs because variational execution shares commonalities among repetitive executions. Though it does not scale to all programs, analyzing a complete set of SSHOMs for smaller programs and their test suites allows us to study SSHOMs more systematically.

(2) Characteristics Analysis: We study the characteristics of the identified higher-order mutants from Step 1. Where previous approaches found only a few samples of higher-order mutants, we have a unique opportunity to study the characteristics of higher-order mutants on a much more complete set. We analyze characteristics, such as the typical number of mutants combined and their distance in the code. This helps us better understand higher-order mutants without the potential sampling bias from a search heuristic. For example, we found that most SSHOMs are composed of fewer than 4 first-order mutants and that constituent first-order mutants tend to locate within the same method or the same class.

(3) Prioritized Heuristic Search: Finally, we develop a second new search strategy $\texttt{search}_{pri}$ that prioritizes likely promising combinations of first-order mutants based on the characteristics identified in Step 2. The $\texttt{search}_{pri}$ is easy to implement and does not require the heavyweight variational analysis of $\texttt{search}_{var}$. Although it does not provide any completeness guarantees, it is highly efficient at finding higher-order mutants fast and scales to much larger systems with thousands of first-order mutants. We evaluate the new search strategy using a different set of larger systems to avoid overfitting. Our results indicate that the identified characteristics indeed effectively guide the search. For example, we found 390,533 SSHOMs among combinations of 103,663 first-order mutations, where existing search approaches can barely find any.

We make the following contributions in this work:

- We propose a novel way of using variational execution to find a *complete* set of SSHOMs for a given set of first-order mutations and tests, by formalizing the search as a Boolean satisfiability problem. An evaluation of small to medium-sized programs shows that we can achieve completeness and simultaneously increase efficiency (Section 3).
- Using the identified set of SSHOMs, we make the first step in studying the basic characteristics of SSHOMs to inform future research (Section 4).
- To show how useful the characteristics are, we use them to design a new lightweight prioritized search strategy, independent of variational execution. We evaluate the prioritized search strategy on a fresh set of larger benchmarks, showing that the new search is scalable and generalizable (Section 5).

2 HIGHER-ORDER MUTANTS

Mutation analysis introduces a set of syntactic changes to a software artifact and observes whether the previously passing test suite is sensitive enough to detect the changes (termed "to kill the mutant"). Traditionally, many simple small changes are explored in isolation, one at a time; several catalogs of mutation operators that perform small syntactic changes exist [35, 60].

In its simplest form, *higher-order mutants* are combinations of two or more first-order mutants [21, 23]. The set of possible second-order mutants grows quadratically with the size of the set of first-order mutants from which they are combined; if considering combining more than two first-order mutants, the set of possible higher-order mutants grows much faster.

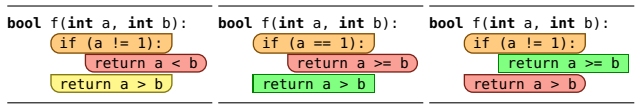
Many higher-order mutants are of little value in practice, because a test that would kill any constituent first-order mutant will likely also kill the higher-order mutant, discussed as the coupling effect hypothesis [55]. However, Jia and Harman [23] argue that there exist several classes of higher-order mutants that exhibit interesting behavior. They specifically highlight *strongly subsuming higher-order mutant (SSHOM)*, in which the constituent mutants interact in ways making the higher-order mutant hard to kill, as we will explain in detail in Section 2.2.

2.1 Usefulness of Higher-Order Mutants

A recent survey of over 39 papers on higher-order mutation testing [13] summarized a large number of different application scenarios for higher-order mutants claimed in prior research, including mutant reduction [12, 17, 19], coupling effect analysis [14, 23], equivalent mutant reduction [37, 49], test data evaluation [16], and test suite reduction [17, 52]. In the following, we illustrate a concrete example of how higher-order mutations can be useful to software-engineering researchers for creating synthetic, but challenging faults to evaluate various software engineering tools.

The effectiveness of many approaches in software-engineering research needs to be evaluated on faults in software systems. For example, fault localization tools need to evaluate how accurately they can localize the faults, test suite generation tools need to evaluate how effective the generated tests are at finding bugs, and program repair tools need to evaluate how many faults they can repair. When evaluating their tools, researchers often have the choice of running evaluations on a curated, often small, set of real bugs or running on large numbers of synthetically seeded bugs. Both approaches have known benefits and drawbacks:

- Seeded faults are convenient: Easy to create and providing a perfect ground truth, they allow researchers to run experiments with very large numbers of faults on almost any system. For example, fault localization techniques were often evaluated on artificially seeded single-edit faults, such as those in the *Siemens* test suite [18] (e.g., [1, 25, 47, 62, 67]). Researchers have been critical of this style of evaluation, arguing that seeded single-edit faults are not representative of most real faults (which often require fixes in multiple locations) [28, 76] and that fault localization techniques may not generalize as they are over-optimized in finding such simple single-edit faults [62].
- In contrast, if curated well, datasets of real faults can be much more representative of realistic usage scenarios. Research on



(a) Mutation 1 (b) Mutation 2 (c) HOM

Figure 2: Suspicious lines based on spectrum-based fault localization [25]. The ranking is shown as the intensity of danger, suspicious, caution and safe.

automated program repair is almost exclusively evaluated on a few hundred real faults [46]. For example, the widely used *Defects4J* dataset [27] curated 438 faults from 5 libraries. Creating high-quality datasets of realistic and representative faults is challenging and typically requires significant human and engineering effort [27, 48, 71]. Therefore, while it is easy to seed millions of faults in almost any program, only a few datasets of curated real faults are available, often only with moderate numbers of faults in a small number of libraries or programs. Some researchers warn that overly focusing on a few shared datasets of faults leads to approaches that overfit the available faults [11, 71].

In this tension between simple seeded faults and expensive to curate real faults, higher-order mutation may provide a compromise. Certain kinds of higher-order mutants, in particular SSHOMs that we study in this work, are *more subtle* and *harder to kill* (shown both theoretically [14] and empirically [15, 21, 23, 43, 59]). They are more promising to simulate real faults than traditional first-order mutants: For example, Zhong and Su [76] and Just et al. [28] found that more than 50–70 % of real faults are caused by faults in more than two locations. Just et al. [28] also found that 73% of real faults are coupled to mutants, while on average 2 mutants are coupled to a single real fault. That is, certain kinds of higher-order mutants may be more representative of real faults. Thus, assuming we can find them efficiently, which is the goal of this paper, we can still automate their creation and seed thousands of these more challenging faults in almost any software system.

Let us illustrate the potential of higher-order mutation for fault localization in Figure 2. Our example program from Figure 1 is mutated with two first-order mutants, which are later combined to form a higher-order mutant; note how this higher-order mutant fails for fewer test cases than the constituent first-order mutants. In this simple setting, the classic fault localization technique Tarantula [25] works quite well for the first-order mutants, highlighting the mutated lines as shown in Figure 2; but Tarantula fails to report the two mutated lines of the higher-order mutant, instead highlighting the unchanged line. This example shows how fault localization fails to locate the faulty lines of interacting mutations, which, as discussed, may be expected for realistic faults [28, 76]. As a further consequence, a program repair technique based on spectrum-based fault localization may not even attempt to fix the first return statement [45].

To realize the full potential of higher-order mutants for these and other use cases, it is critical to have an efficient way of finding interesting higher-order mutants. In this work, we do not reevaluate the usefulness of HOMs for various use cases [13] or how well they represent real faults [23, 28, 76], which has been studied repeatedly and comprehensively in prior work [10]. Instead, we focus on a technical problem that made SSHOMs too costly and impractical: *How to efficiently find SSHOMs.*

2.2 Strongly Subsuming Higher-Order Mutants (SSHOMs)

Jia and Harman [23] classify higher-order mutants into several kinds, highlighting SSHOMs as useful. For this reason, our work targets SSHOMs, though we expect that it can be generalized to other classes of higher-order mutants. Specifically, Jia and Harman [23] define an SSHOM as a higher-order mutant that can only be killed by a subset of test cases that kill all its constituent first-order mutants. More formally, let h be a higher-order mutant composed of first-order mutants $f_1, f_2, \ldots, f_n$, T_h the set of test cases that kill the higher-order mutant h, and T_i the set of test cases that kill the first-order mutant f_i, then h is an SSHOM if and only if:

$$T_h \neq \emptyset \quad \wedge \quad T_h \subseteq \bigcap_{i \in 1 \ldots n} T_i \tag{1}$$

If we further restrict T_h to be a strict subset, we get an even stronger type of SSHOM, which we call *strict* strongly subsuming higher-order mutant, denoted as *strict-SSHOM*.[1] In other words, there must be at least one test case that kills one of the first-order mutants, but not the higher-order mutant. Thus, in a strict-SSHOM, multiple first-order mutants interact such that they mask each other at least for some test cases, making the strict-SSHOM harder to kill than all the constituent first-order mutants together.

Our (manually constructed) SSHOM in Figure 1 illustrates this relation: Intuitively, the first first-order mutant (replacing '==' by '!=') forces the execution to go into an unexpected branch, and the second (replacing '<' by '>=') inverts the return values. The two changes in control and data flow are easy to detect separately (i.e., killed by two test cases each), but the combination of them is more subtle and only detected by one test case.

2.3 Finding SSHOMs

SSHOM is defined in terms of test results of *first-order mutants*. All existing search strategies aim to find SSHOMs in terms of a given test suite and a given set of first-order mutants. The search space can be large due to the exponential combinatorial explosion of possible mutant combinations. Since only very few of the combinations are interesting and those are hard to find in a vast search space, higher-order mutation testing has long been considered too expensive.

Jia and Harman [23] explored several search techniques to find SSHOMs, finding that genetic search performs best. We will use their genetic-search strategy, together with a brute-force strategy, as baselines for our evaluations. Although the genetic search has been shown to successfully find SSHOMs, it requires considerable resources to evaluate many candidates, involves significant randomness, and cannot enumerate all SSHOMs in the search space.

It is conceptually possible to define SSHOMs in terms of an idealized test suite that represents all (possibly infinite) possible

[1] SSHOMs have been defined inconsistently in the literature as subset [17] and strict subset [21, 23]. We inherit the definition of SSHOMs from Harman et al. [17], as it is the most recent work. As we will see in the evaluation, the difference between subset and strict subset is significant, so we make the distinction explicit, introducing strict-SSHOM as a distinct subclass and reporting results for both.

```
def findSSHOMs(program P, mutants M, testsuite T):
  failing_conditions = Map[Test, FailingCondition]()
  # Merge all first-order mutants into one meta-program
  mutated_P = encode_all_mutants(P, M)
  for (test ← T):
    # Variational execution returns under what combinations of mutants the
    # given test fails, compactly represented as a propositional formula.
    f_t = variational_execution(mutated_P, # mutated program
                                test,      # entry point of execution
                                M)         # symbols representing mutants
    failing_conditions.add(test → f_t)
  # search of SSHOMs as a satisfiability problem, using Equation 2-4
  constraint = encodeSAT(failing_conditions, T, M)
  # allSAT returns all solutions to the constraint, each represented
  # as a set of activated variables (first-order mutants)
  found_SSHOMs = allSAT(constraint)
  return foundSSHOMs
```

Figure 3: Using variational execution to find SSHOMs.

behaviors in the program. In practice though, all search strategies have to work with existing test suites and whether a combination of first-order mutants is considered an SSHOM is evaluated in terms of a given test suite. Different test suites and different first-order mutants may result in different SSHOMs; SSHOMs found with a specific test suite could be interpreted as *approximations* of SSHOMs potentially found with an idealized test suite. A specific test suite and set of first-order mutants form a large but finite search space and it is possible to define and find a *complete* set of SSHOMs *with regard to those given mutants and tests*, as we will discuss in Section 3.

In this paper, in line with prior work on finding SSHOMs, we focus on finding SSHOMs with regard to a fixed test suite and fixed set first-order mutants, as would be useful in the fault localization and program repair scenarios discussed above. Although orthogonal to the goal of this work, which is improving existing search strategies, in Appendix A, we discuss the notion of SSHOMs in terms of a theoretical idealized test suite and the influence of test suite size on identifiable SSHOMs.

3 STEP 1: COMPLETE SEARCH WITH VARIATIONAL EXECUTION ($\texttt{search}_{var}$)

In this step, we develop $\texttt{search}_{var}$ to compute a *complete* set of SSHOMs with respect to given tests and first-order mutants, so that we can study their properties later. Figure 3 shows the pseudo-code of applying variational execution to find SSHOMs.

First, given a program under analysis P, we mutate it into P' by applying our mutation operators *exhaustively* at every applicable location to generate all first-order mutants upfront. We represent each first-order mutant as a Boolean option and use a ternary conditional operator to encode the change. Mutations to the same expression are expressed as nested ternary conditional expressions. For example, we show below how we encode the two first-order mutants from Figure 1.

```
bool f(int a, int b):
    if ( m1 ? a != 1 : a == 1 ):
        return m2 ? a >= b : a < b
    return a > b
```

After encoding first-order mutants, we use variational execution as a black-box technique to explore, for each test case, under what combinations of first-order mutants the test would fail. In a nutshell, variational execution runs the program by dynamically tracking the differences in program state that are caused by mutations (similar to executing the program symbolically with symbolic values for all mutations) [3, 53, 54, 75]. Conceptually, a single run of variational execution is equivalent to running all combinations of first-order mutations in a brute-force fashion, but it is usually much faster due to the sharing of similar executions at runtime [53, 54, 75]. For each test execution, variational execution will return a failing condition, which is a propositional formula that represents exactly the combinations of mutations for which the test fails. We show examples of failing conditions in our running example in Figure 1 (last table column).

Finally, we collect all propositional failing conditions for all test cases and use them to search for SSHOMs by encoding the search as a Boolean satisfiability problem. Using BDDs or SAT solvers, we can then enumerate all solutions, which correspond directly to all SSHOMs. Although the formulas can be large if we have many first-order mutants and test cases and finding satisfiable assignments is NP-hard, modern SAT solving techniques are scalable enough. Our implementation and data are available on GitHub: https://github.com/poosomooso/SSHOM-Search.

3.1 Mutant Generation

We generate first-order mutants *exhaustively* and encode them all at once into a metaprogram, which is later used for finding SSHOMs. This compact encoding of mutations defines a finite search space, which is critical for variational execution to be efficient [74]. Similar encodings have been explored in different contexts, such as speeding up mutation testing [29, 50, 72]. Using this encoding, we also ensure a fair comparison with baseline approaches by excluding compilation time and using the same metaprograms.

For our experiments, we implemented 3 mutation operators: (1) *Arithmetic Operator Replacement* (AOR, mutating `+`, `-`, `*`, `/`, `%`) (2) *Relational Operator Replacement* (ROR, mutating `==`, `!=`, `<`, `>`, `<=`, `>=`) and (3) *Logical Connector Replacement* (LCR, mutating `||` and `&&`). These comprise 3 of 5 most well-studied mutation operators [56, 57], excluding two further based on recent insights: (4) *Absolute Value Insertion* (ABS) has been shown to be less useful in practice [63], so we excluded it to avoid a meaninglessly large search space. (5) *Unary Operator Insertion* (UOI) would add many more mutants, most of which are likely equivalent to the ones generated from other mutation operators (e.g., mutating `a+b` to `a+-b` using UOI is equivalent to `a-b` using AOR) [35, 50, 63].

3.2 Variational Execution

We use variational execution to determine which combinations of mutants fail a test case. The novelty of using variational execution lies in the efficient and complete exploration of all mutants, as opposed to one mutant at a time in traditional search-based approaches. For this work, we use variational execution as a black-box technique and so the technical details of how it works are not relevant. We only provide intuition and refer the interested readers to the existing literature for a more in-depth discussion [3, 53, 54, 75].

Conceptually, variational execution is similar to symbolic execution, in that it executes a program with symbolic Boolean values representing mutants and concrete values for test inputs. Specifically, variational execution performs computations with *conditional*

values [75], which may represent multiple alternative concrete values. For example, a conditional value $\langle \alpha, 1, -1 \rangle$ indicates that it has the value `1` under α, and `-1` otherwise. Conditional values can represent a *finite* number of alternative *concrete* values distinguished by propositional conditions over symbolic values (representing mutations). Variational execution computes with conditional values and propagates them along data and control flow. At control-flow decisions, both branches are explored under corresponding symbolic path conditions; afterward, state is merged again into conditional values to exploit sharing in subsequent statements. In a nutshell, variational execution can be considered as an extreme design choice among various forms of symbolic program evaluation [5, 6, 8, 34, 69] for finite domains, in which computations are maximally performed on concrete values, but Boolean symbolic values may distinguish between multiple concrete values in program state [3, 53, 75].

For our purposes, we consider all Boolean options representing first-order mutants as symbolic options. This way, all state changes caused by mutants can be compactly tracked, which enables us to explore all combinations of mutants at the same time. As output, we determine under which combinations of mutants a test case fails (propositional formula over first-order mutants as illustrated in Fig. 1), by simply observing under which condition any asserted expression evaluates to *false*.

In theory, mutant interactions can cause a combinatorial explosion in conditional values where exponentially many alternative values for different combinations of mutants need to be tracked for a single variable. However, in practice, not all mutants affect each test and not all mutants interact, enabling an often reasonably efficient exploration of all feasible combinations. We defer the discussion of this scalability issue to Section 3.4.

Multiple implementations of variational execution exist for a number of programming languages [3, 4, 31, 53, 54, 68, 75]. For details on variational execution and how it differs from symbolic execution (e.g., finite symbolic domain, efficient state sharing), previous work describes implementations [31, 54] and provided formal semantics for a core calculus [3]. We use *VarexC*, a state-of-the-art implementation of variational execution for Java [75].

3.3 SSHOM Search as a SAT Problem

We use the output of variational execution—propositional formulas indicating under which combinations of mutations each test fails—to construct a single formula that is satisfiable exactly for those assignments that represent SSHOMs, based on our definition of SSHOM in Section 2.2. This way, the search for SSHOMs is transformed into a Boolean satisfiability problem, which we can solve with BDDs or SAT solvers. To derive the formula, we outline the criteria for identifying SSHOMs as defined by Jia and Harman [23] (see Sec. 2.2) and construct a logical expression for each criterion.

Let T be the set of all tests, M be the set of all first-order mutants, and f_t be the propositional formula over literals from M describing the mutant configurations in which test $t \in T$ fails (f is generated with variational execution, see above). As a shorthand, let $\Gamma(m, t)$ be the result of evaluating f_t with first-order mutant m assigned to *true* and all other mutants assigned to *false*; in other words, whether test t fails for first-order mutant m. To identify SSHOMs, we encode three criteria:

First, we ensure that a mutant combination is killed by at least one test, encoding $T_h \neq \emptyset$ in Formula 1 (Sec. 2.2):

(1) The SSHOM must fail at least one test (i.e., must not be an equivalent mutant):

$$\bigvee_{t \in T} f_t \tag{2}$$

Second, if a given mutant combination (i.e., higher-order mutant) is killed by a test t, the same test must kill each constituent first-order mutant. That is, for all tests and first-order mutants, the first-order mutant must either be killed by the test ($\Gamma(m, t)$) or not be part of the higher-order mutant ($\neg m$). This is the encoding of $T_h \subseteq \bigcap_{i \in 1 \ldots n} T_i$ in Equation 1 (Sec. 2.2):

(2) Every test that fails the SSHOM must fail each constituent first order mutant:

$$\bigwedge_{t \in T} (f_t \Rightarrow \bigwedge_{m \in M} (\neg m \vee \Gamma(m, t))) \tag{3}$$

In addition, we can optimize for SSHOMs that are harder to kill than the constituent first order mutants, excluding those that are equally difficult to kill [23]. As discussed in Section 2.2, we call these *strict-SSHOM* and require a strict subset relation in Equation 1 (i.e., $T_h \subset \bigcap_{i \in 1 \ldots n} T_i$ rather than $T_h \subseteq \bigcap_{i \in 1 \ldots n} T_i$), which requires the additional encoded condition:

(3) There exists a test that can kill all constituent first-order mutants but cannot kill the strict-SSHOM.

$$\bigvee_{t \in T} \left(\neg f_t \wedge \bigwedge_{m \in M} (\neg m \vee \Gamma(m, t))\right) \tag{4}$$

To find SSHOMs and strict-SSHOMs, we take the conjunction of Equations 2–3 and 2–4, respectively, and use BDD or SAT solver to iterate over all possible solutions. For example, if our approach returns a satisfiable assignment in which m_1 and m_3 are selected and all other mutants are deselected, then the combination of m_1 and m_3 is a valid (strict-)SSHOM.

We use BDDs to enumerate all satisfiable solutions by default. While constructing BDDs can be expensive, getting a solution from a BDD is fast ($O(n)$, where n represents the number of Boolean variables [7]). In some rare cases where we cannot construct a BDD due to insufficient memory, we fall back to using a SAT solver. With a SAT solver, we ask for one possible solution, then add the negation of that solution as an additional constraint before asking for the next solution, repeating the process until all solutions are enumerated. We can usually efficiently enumerate *all* possible SSHOMs for the given set of first-order mutants and the variational-execution result of a given test suite.

3.4 Limitations

While variational execution and the SAT encoding provide a new strategy to find SSHOMs, this approach comes also with severe restrictions, mostly regarding scalability and engineering limitations inherited from the tools we use, which limits broad applicability in practice (which we address with an alternative strategy in Sec. 5).

Combinatorial Explosion. Recent studies show that combinatorial explosion is uncommon for the types of highly-configurable programs analyzed with variational execution in the past [53, 66], mainly because programs are usually written by human developers to have manageable interactions among options. When applied to

higher-order mutation testing, we did observe some combinatorial explosion caused by random combinations of first-order mutants. For example, we observed cases where interactions of first-order mutants create more than 15,000 alternative concrete values in one single local variable. We argue that this is the essential complexity of the mutated program, and it would be equally difficult for other approaches to exhaustively explore a complex search space like this. However, it is possible to find efficient search strategies when giving up the completeness goal, as we will show in Sec. 5.

In the evaluation of $\texttt{search}_{var}$, we manually removed some problematic first-order mutants and test cases that caused an excessive number of interactions (See Table 1). For fairness, we remove these mutants and test cases across all compared approaches.

Environment Barrier. Similar to symbolic execution, variational execution needs to handle the environment barrier carefully when interacting with an external runtime environment that is not aware of conditional values or path conditions. This barrier often manifests as I/O or native method calls. There are several common strategies to mitigate this issue, such as creating models for these operations [8, 69, 75]. In our study, only a few tests and mutants triggered problematic environment interactions. While solvable with engineering effort, we consider them noncritical for our goal and removed the problematic tests or mutants after manual inspection.

3.5 Evaluation

In addition to using $\texttt{search}_{var}$ to get a complete set of SSHOMs with regard to given tests and first-order mutants, we compare the efficiency and effectiveness of $\texttt{search}_{var}$ against the existing state-of-the-art *genetic search* ($\texttt{search}_{gen}$) and a baseline *brute-force* strategy ($\texttt{search}_{bf}$), based on subject systems previously used in evaluating the genetic search strategy [17].

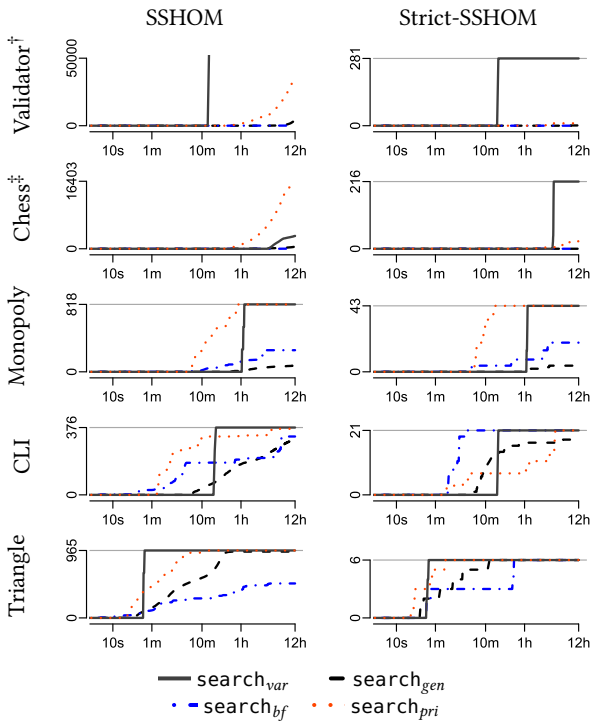
Subject Systems. We replicate the setup of the largest previous study on higher-order mutation testing [17]. While we cannot perform an exact replication since we could not obtain the original tools from the authors, not all relevant details and parameters have been published, and some engineering limitations discussed earlier, we still select the same subject systems and reimplement search strategies in our own infrastructure. That is, our results cannot be compared directly against the numbers reported in prior work [17], but we report comparable numbers within a consistent setup.

We use the same four small to medium-sized Java programs, *Monopoly*, *Cli*, *Chess*, and *Validator*, all of which come with good quality test suites that are deemed complete by developers [17]. In addition, we use the *triangle* program commonly used in mutation testing [23]. We report the statistics of our subject systems in Table 1 (top), which are comparable to those reported in prior work [17], with slight differences likely caused by different mutation operators used and excluded tests (as discussed in Section 3.4).

Baseline Search Strategies. We compare our approach against the state-of-the-art genetic algorithm [17, 21, 23] and a naive brute-force search. The brute-force search iterates over all valid higher-order mutants, starting from all pairs, then all triples, and so on until a time limit is reached. The brute-force search serves as a reliable baseline as there is no randomness involved.

We reimplemented the genetic algorithm approach based on the description in Jia et al.'s work [20, 21, 23]. As the exact setup was not available or documented, we leave undocumented parameters at default values. The core of the genetic algorithm is a fitness function for candidate higher-order mutants. Following existing work [20, 21, 23] and using the notations in Equation 1, we calculate the fitness as $|T_h| / |\bigcap_{i \in 1 \ldots n} T_i|$.[2] The intuition is that an SSHOM should fail only for a subset of test cases that kill all its constituent first-order mutants. Thus, we use it as a piece-wise function: a fitness of $(0, 1]$ indicates an SSHOM and $(0, 1)$ a strict-SSHOM, with lower fitness more preferable; a fitness of 0 and larger than 1 indicate potential equivalent mutants and non-SSHOMs, respectively, which are discarded between generations of the genetic algorithm.

Measurements. All experiments were performed on AWS EC2 instances, each of which has an Intel 4-core Xeon CPU with 16GB of RAM. We ran benchmarks to confirm that the performance is stable enough for our measurements across different instances; given that we often demonstrate order-of-magnitude differences in outcomes, differences are unlikely explained by measurement noise. For each search strategy (i.e., $\texttt{search}_{gen}$, $\texttt{search}_{bf}$, $\texttt{search}_{var}$), we measure each subject system three times and report the average, like the three restarts in the work of Harman et al. [17]. We ran each trial of genetic algorithm and brute force for 12 hours.



† We cap the plot for Validator since there are 13.4 billion SSHOMs; ‡ we could not enumerate all nonstrict-SSHOMs for Chess due to the difficulty of the SAT problem and report only those found within the time limit

Figure 4: (Strict-)SSHOMs found over time in each subject system, averaged over 3 executions. Note that time is plotted in log scale as most SSHOMs are found within the first hour.

[2] The fitness function has been defined either using intersect of T_i [20] or union [21, 23]. We use the former in our reimplementation as it more precisely captures our intuition of SSHOMs.

Table 1: Subjects and Found (strict-)SSHOMs; the last three subjects and the *Pri* column are discussed in Section 5.

						Found SSHOM (and strict-SSHOM)			
Subject	LOC	Tests (%used)	LCov	FOMs (%used)	MutScore	Var	Gen	BF	Pri
Validator	7,563	302 (83%)	54%	1,941 (97%)	36% (68%)	$1.34*10^{10}$ (281)	4,041 (0)	273 (4)	36,995 (10)
Chess	4,754	847 (84%)	74%	956 (26%)	81% (86%)	3,268† (216)	484 (0)	19 (6)	16,403 (24)
Monopoly	4,173	99 (89%)	74%	366 (90%)	80% (83%)	818 (43)	81 (4)	349 (15)	817 (43)
Cli	1,585	149 (95%)	92%	249 (51%)	71% (81%)	376 (21)	309 (18)	326 (21)	369 (21)
Triangle	19	26 (100%)	100%	128 (100%)	92% (92%)	965 (6)	949 (6)	493 (6)	965 (6)
Ant	108,622	1354 (77%)	53%	18,280 (92%)	57% (94%)	- (-)	1 (0)	0 (0)	44,496 (61)
Math	104,506	5177 (79%)	90%	103,663 (100%)	66% (71%)	- (-)	0 (0)	0 (0)	390,533 (2,830)
JFreeChart	90,481	2169 (99%)	59%	36,307 (99%)	21% (45%)	- (-)	0 (0)	6 (0)	576,725 (513)

LOC represents lines of code, excluding test code, measured with *sloccount*. **Tests** and **FOMs** report the numbers of test cases and first-order mutants we used in experiments, with the percentages relative to the total numbers in parentheses. **LCov** reports line coverage of the tests used in our experiments. **MutScore** reports the mutation score of all used first-order mutants and the score in parenthesis considers only FOMs that are covered by the tests. **Var**, **Gen**, **BF**, **Pri** denote our approach (Step 1, $\texttt{search}_{var}$), the genetic algorithm ($\texttt{search}_{gen}$), brute force ($\texttt{search}_{bf}$), and our prioritized search (Step 3, $\texttt{search}_{pri}$) respectively.

† incomplete results, solutions found with SAT solving within the 12 hours budget.

Results. In Table 1, we report the number of (strict-)SSHOMs found with all three search strategies within the 12-hour time budget and, in Figure 4, we plot the numbers of (strict-)SSHOMs found over time. Note that by construction, if $\texttt{search}_{var}$ terminates (all cases except *Chess*, where solving the satisfiability problem takes considerable time), it enumerates *all* SSHOMs, thus provides an upper bound for other search strategies—without $\texttt{search}_{var}$ this upper bound would not be known.

These results show clear trends: $\texttt{search}_{var}$ requires a relatively long time to find the first SSHOM because variational execution must finish executing all tests for all combinations of first-order mutants. However, once variational execution finishes, it can enumerate *all* SSHOMs very quickly by solving the Boolean satisfiability problem. Variational execution takes longer with more and longer test cases and with more first-order mutants but still outperforms a brute-force execution by far, indicating significant sharing, as found in prior analyses of highly-configurable systems [53, 54, 75].

In contrast, $\texttt{search}_{gen}$ and $\texttt{search}_{bf}$ can test many candidate SSHOMs before variational execution terminates and finds some actual SSHOMs early, but both approaches take a long time to find a substantial number of SSHOMs and miss at least some SSHOMs in all subject system within the 12h time budget given. In some systems with moderate numbers of first-order mutants, $\texttt{search}_{bf}$ is fairly effective as it systematically prioritizes pairwise combinations which are more common among SSHOMs than combinations of more than two mutants, as we will discuss.

In summary, for systems where variational execution scales, $\texttt{search}_{var}$ can find *all* SSHOMs whereas other approaches find only an often much smaller subset within a 12h time window. Whereas prior approaches often find their first SSHOMs faster, $\texttt{search}_{var}$ needs more time upfront for variational execution but can then enumerate SSHOMs very quickly. To scale $\texttt{search}_{var}$ to more realistic programs, more engineering is needed to overcome the limitations discussed in Sec. 3.4. Nevertheless, $\texttt{search}_{var}$ is valuable to the research community as it provides a precise and efficient way of identifying all SSHOMs.

4 STEP 2: SSHOM CHARACTERISTICS

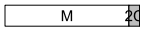
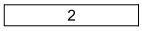
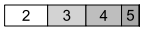
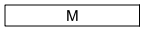
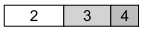
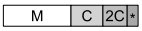
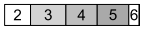
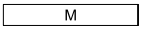
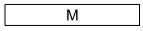
In this second step, we study the characteristics of (strict-)SSHOMs, with the goal to inform subsequent heuristic search strategies (Step 3) and future research in general. Using the *complete* set derived for the subject systems in the previous step, rather than a (potentially biased) sample of SSHOMs, we can study characteristics with higher confidence.

We explored the dataset in an iterative exploratory fashion, focusing primarily on characteristics that may guide future search strategies, such as specific composition patterns and proximity of constituent first-order mutants for the set of all higher-order mutants. Kurtz et al. [40] argue that mutation operators should be specialized for individual programs, so we focus on high-level characteristics that are largely independent of specific mutation operators to avoid overfitting. We started by randomly sampling a large number of identified SSHOMs (among the pool of all SSHOMs). We manually inspected the sampled SSHOMs to pose hypotheses about common characteristics. We then operationalized the hypothesized characteristics (i.e., develop measures to apply across all SSHOMs) to quantitatively validate them. We repeated the process until we could not identify additional hypotheses. Due to space constraints, we only report characteristics for which we could quantitatively identify strong support.

Mutation Order. SSHOMs and strict-SSHOMs are typically composed of only very few first-order mutants. Overall, over 90 % of all SSHOMs and strict-SSHOMs are composed of at most 4 first-order mutants, indicating that subtle interactions are mostly caused by very few first-order mutants. Although we found a few SSHOMs that are up to sixth-order in Chess and Triangle, such cases are rare, especially for strict-SSHOMs. We plot the distribution of orders for both *SSHOMs* and *strict-SSHOMs* in Table 2.

Equivalent Test Failures. In multiple subject systems, many SSHOMs and strict-SSHOMs are composed of first-order mutants that are killed by the same set of test cases (nonstrict-SSHOMs are often killed by the same test cases, whereas strict-SSHOMs necessarily are killed by fewer). In Table 2, we report how many of the

Table 2: Characteristics of SSHOMs and strict-SSHOMs found in our subject systems.

	Order		Equal-Fail Rule		N+1 Rule		Distribution	
Subject	SSHOM	strict-SSHOM	SSHOM	strict-SSHOM	SSHOM	strict-SSHOM	SSHOM	strict-SSHOM
Validator†	-		-	96%	-	99%	-	
Chess†	-		-	76%	-	38%	-	
Monopoly			11%	100%	99%	100%		
Cli			53%	5%	98%	100%		
Triangle			8%	17%	98%	50%		

Order counts the number of constituent first-order mutants; equal-fail and N+1 rule explained in text; distribution: all constituent first-order mutants in the same method (M), multiple methods in the same class (C), two classes (2C), or spread across more than two classes (*).

† for Validator and Chess we omit statistics because we cannot enumerate all possible SSHOMs (too many in Validator and incomplete set in Chess)

SSHOMs and strict-SSHOMs in each project could be found when only combining first-order mutants that are killed by the same test cases, which we name *Equal-Fail SSHOMs*.

Containment Relationships. In addition, we found a common containment pattern: when a (strict-)SSHOM is composed of more than two first-order mutants, it is very likely that a subset of these first-order mutants also forms a (strict-)SSHOM. In other words, an *N+1 Rule*, combining a previously identified (strict-)SSHOM with one further first-order mutant is a promising strategy to identify more (strict-)SSHOMs. In Table 2, we report how many of the (strict-)SSHOMs in each project with more than two constituent first-order mutants could be generated with such a rule.

Proximity. Finally, for most SSHOMs, all constituent first-order mutants are in the same class and often even in the same method, likely because first-order mutants with close proximity have higher chances of data-flow or control-flow interactions. The effect is even more pronounced for strict-SSHOMs. This stronger effect was previously conjectured though not validated [23]. We plot the distributions for all subject systems in Table 2.

Other. We also explored other patterns that may inform search heuristics, such as common combinations of mutation operators (using frequent-itemset mining [2]), but found no additional strong patterns. While we believe a qualitative analysis of the mutants and their characteristics may reveal interesting insights about SSHOMs and whether they more closely mirror realistic human-made faults, such analysis goes beyond our scope of finding SSHOMs efficiently.

5 STEP 3: CHARACTERISTICS-BASED PRIORITIZED SEARCH ($\texttt{search}_{pri}$)

In a final third step, we develop a new search strategy using heuristics based on characteristics found in Step 2, which will be an incomplete, but practical alternative to our $\texttt{search}_{var}$ strategy.

5.1 Search Strategy

Our new search strategy $\texttt{search}_{pri}$ avoids the overhead of variational execution, but instead again evaluates each candidate higher-order mutant by executing the corresponding test suite, one candidate mutant at a time just like $\texttt{search}_{bf}$ and $\texttt{search}_{gen}$. Our key contribution is ordering how we explore candidate mutants to steer the search toward more likely candidates. That is, instead of a naive enumeration of all combinations ($\texttt{search}_{bf}$) or an exploration based on random seeds ($\texttt{search}_{gen}$), we prioritize based on the previously identified typical characteristics of higher-order mutants. Since characteristics for SSHOM and strict-SSHOM do not differ strongly, we develop only a single search strategy.

Conceptually, we calculate a penalty for every candidate higher-order mutant and prioritize those candidates with the lowest penalty. We compute the weighted sum of three factors:

$$penalty = \omega_1 \cdot order + \omega_2 \cdot testDiff - \omega_3 \cdot isN1 \quad (5)$$

First, we assign penalties based on the number of constituent first-order mutants (*order*): a candidate with a higher order receives a larger penalty than a lower-order candidate, thus, prioritizing candidates with lower order that, as our data shows, are more likely to be SSHOMs. Second, we penalize candidates constructed from first-order mutants that do not get killed by the same test cases (*testDiff*, counting the number of test cases that can kill only a subset of all constituent first order mutants), generalizing our *Equivalent Test Failures* insight: if all first-order mutants are killed by the same test cases, the candidate is likely to be an SSHOM, and thus gets a 0 penalty, whereas mutants that are killed by different test cases are less likely to form an SSHOM, and thus is deferred with a higher penalty. Finally, we reduce the penalty of a candidate if the *N+1 Rule* applies (*isN1*, returning 1 or 0); that is, if a candidate can be constructed by adding one more first-order mutant to a known SSHOM, the candidate receives a boost and gets prioritized. By default and for our evaluation, we assign the weights $\omega_1 = 5$, $\omega_2 = 1$, and $\omega_3 = 15$, based on our experience with the subject systems in Section 4.

Unlike previously used genetic search strategies, where the exploration order nondeterministically depends on random mutation and crossover in every generation, $\texttt{search}_{pri}$ explores candidates in a deterministic order (lexical order if two candidates have the same priority).

5.2 Implementation

Since we cannot enumerate and sort all possible candidate higher-order mutants for large programs, and even the execution of all first-order mutants may take a long time, we devise an algorithm for $\texttt{search}_{pri}$ that identifies likely candidates in batches, shown in Figure 5. In each batch (configurable, by default one Java package at a time), we enumerate all candidate higher-order mutants up to

```
def findSSHOMs(program P, mutants M, testsuite T,
               maxOrder, maxDist, budget):
  foundSSHOMs = ∅
  # explore the program one fragment at a time
  for (batch ← fragments(P)):
    # identify reachable first-order mutants in fragment
    mutants = reachable(M, batch)
    # run tests on reachable first-order mutants
    fomTestResults = for (m ← mutants) evaluate(T, {m})

    # enumerate candidate SSHOMs up to order and distance bounds
    candidates = enumerateCandidates(mutants, maxOrder, maxDist)
    # compute priorities for each candidate
    priorities = computePriorities(candidates, fomTestResults, {})

    # explore candidates in decreasing priority
    while (candidates ≠ ∅ ∧ within budget):
      candidate = getNext(candidates, priorities)
      candidates -= candidate
      homTestResult = evaluate(T, candidate)
      if (isSSHOM(fomTestResults, homTestResult)):
        foundSSHOMs += candidate
        # update priorities based on N+1 rule
        priorities = computePriorities(candidates, fomTestResults,
                                       foundSSHOMs)
  return foundSSHOMs
```

Figure 5: Characteristics-based prioritized search algorithm.

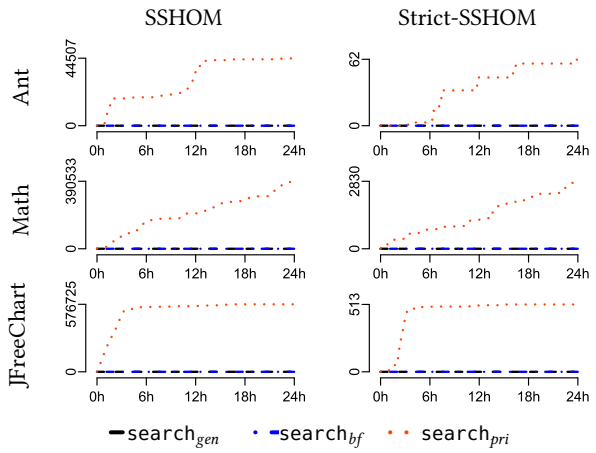


Figure 6: (Strict-)SSHOMs found over time, averaged over 3 executions. Note that time is plotted in a linear scale as SSHOMs are found consistently over time due to batching.

a distance and order bound, then sort these candidates by priority, and finally explore these candidates in order until a (time) budget is reached for that batch. Batching and bounding the search is feasible since the order and distribution characteristics dominate the prioritization anyway and candidates beyond those bounds would be explored only very late. If needed batches could be revisited later with larger bounds to explore more (less likely) candidates.

After batching, our algorithm identifies all first-order mutants defined within the given batch (function `reachable`) and runs the test suite for each of these first-order mutants to identify which tests fail (function `evaluate`). Subsequently, the algorithm enumerates all candidates (function `enumerateCandidates`) up to a given order bound (by default, mutants composed of up to 6 first-order mutants) and up to a given distance bound (by default, up to 4 methods spread across at most 3 classes). We also discard candidates where constituent first-order mutants have no common failing tests because they cannot form SSHOMs according to the definition. Having a manageable set of candidates in the given batch, the algorithm computes priorities (function `computePriorities`) for all candidates using Equation 5 and then explores these candidates in order of decreasing priorities (function `getNext`) until either all candidates are explored or a (time) budget has been reached in that batch (by default, 1 hour per batch). For each candidate, it runs the test suite and compares test results to determine whether a (strict-)SSHOM has been found (function `isSSHOM`); identified SSHOMs are collected and used to recompute priorities based on additional information for the N+1 rule.

5.3 Evaluation

We evaluate how *effective* $\texttt{search}_{pri}$ is at finding (strict-)SSHOMs, and additionally evaluate how it *generalizes* and *scales* to much larger systems than in prior studies on SSHOMs (and used in Sec. 4).

Subject Systems. We evaluate $\texttt{search}_{pri}$ both on the subjects previously used in Section 4 and on a fresh set of much larger subject systems. The comparison against the 5 previously used subject systems allows us to compare effectiveness against the ground truth derived from variational execution, but the results may suffer from overfitting, as we evaluate the search strategy on systems from which the insights that drive its design have been derived.

Hence, we use 3 additional subjects, listed in Table 1 (bottom), after finishing the design of our new search strategy. The new systems are significantly larger, allowing us to explore the different search strategies at a much larger (and possibly more realistic) scale. To select the new subject systems, we collected all research papers published in the last 5 years at ASE, FSE, and ICSE that have the word "mutation" or "mutant" in the title. We then selected the five largest Java systems used, discarding two for which we failed to reliably execute the tests. We did not run $\texttt{search}_{var}$ on these systems, but we still had to exclude some tests or mutants (reported in Table 1), due to technical issues like hard-to-terminate infinite loops.

Measurements. We mirror our previous setup in Section 3.5 and count the number of (strict-)SSHOMs found over time. We collect measurements for $\texttt{search}_{bf}$, $\texttt{search}_{gen}$, and $\texttt{search}_{pri}$. Experiments on the small subject systems were performed on the same AWS EC2 instances (Section 3.5). For the new systems, we collected measurements on Linux machines with 1.30GHz Intel i5 CPU and 16GB memory. When using $\texttt{search}_{pri}$, we used batching for the new larger subject systems, one package at a time, with a 1 hour budget for each package; all other parameters were left at their defaults (described above). For the new subject systems, we ran each measurement for 24 hours, repeated $\texttt{search}_{gen}$ 3 times.

All considered search strategies require executing the test suite repeatedly for each candidate SSHOM. For the larger systems, long test-execution times severely limit the number of mutants we can explore. To minimize the slowdown from test execution that affects all approaches equally, we implement a standard regression test selection technique [60] that only executes test cases that can reach the candidate mutant (technically, we instrument the program to record which test reaches the location of each first-order mutant and only execute tests that reach at least one first-order mutant of a candidate higher-order mutant). We apply this test optimization for all search strategies.

Results. On the small subject systems, as shown in Table 1 and Figure 4, our new search strategy $\texttt{search}_{pri}$ is often very effective, performing at least as well as and usually significantly outperforming both $\texttt{search}_{bf}$ and $\texttt{search}_{gen}$ in all subjects. In a few cases, it even outperforms $\texttt{search}_{var}$: In *Monopoly* it finds almost all higher-order mutants before variational execution finishes running the tests and in *Chess* it finds SSHOMs quickly, not limited by the effort to solve large satisfiability problems.

For the new and larger systems, our results in Table 1 and Figure 6 show that the baseline approaches perform very poorly at this scale. Without being informed by SSHOM characteristics the search in this vast space (e.g., 5 billion candidate combinations of mutation pairs in Math) these approaches find rarely any SSHOMs even when run for a long time. In contrast, $\texttt{search}_{pri}$ finds a significant number of (strict-)SSHOMs in each of these systems: Within 24 hours it explores most batches (91 % of all packages) and has a reasonable precision[3] for finding actual SSHOMs among the tested candidates (60.9 % in *Math*, 29.4 % in *Ant*, and 77.8 % in *JFreeChart*).

We conclude that $\texttt{search}_{pri}$ is an effective search strategy that scales to large systems and generalizes beyond systems from which the characteristics have been collected. While we cannot assess how many SSHOMs we are missing, our strategy is effective at finding a very large number of them in a short amount of time.

6 THREATS TO VALIDITY

External validity might be limited by the specific programs, mutation operators, and test cases. We used common mutation operators and selected subject systems from previous papers to avoid any own sampling bias. From most subject systems, we had to remove some tests or mutations due to technical problems, either engineering limitations of variational execution or issues with memory leaks and infinite loops, which might affect the results to some degree—though we do not expect a systematic bias.

Regarding internal validity, like other studies, our results might be affected by possible mistakes in our implementations or measurements and especially by we reimplemented the existing $\texttt{search}_{gen}$ approach. To mitigate this issue, we verified that the SSHOMs found by $\texttt{search}_{gen}$ and $\texttt{search}_{bf}$ are a strict subset of the ones found by $\texttt{search}_{var}$. For SSHOMs found only by our approach, we additionally verified a sample manually to ensure they are SSHOMs.

To reduce the impact of nondeterminism in performance measurements and genetic search, we report averages across 3 runs, as in previous work [17]. Most differences are large, far exceeding the margins of error from nondeterminism or measurement noise.

7 RELATED WORK

In this section, we focus our discussions on higher-order mutation testing and refer interested readers to a detailed survey for recent advances in mutation testing in general [60].

Approaches for Finding SSHOMs. Early work has investigated different strategies to combine first-order mutants into second-order mutants [36, 49, 52]. Jia and Harman extended this effort to even higher orders using heuristic search looking for certain kinds of valuable higher-order mutants, specifically SSHOMs. They compare a greedy, a hill-climbing, and a genetic algorithm and found that genetic search produces the best results for finding SSHOMs [21, 23]. Since then, higher-order mutation testing has been implemented in different mutation testing tools and frameworks, for different languages [22, 42, 44, 51, 58, 70], usually using some form of heuristic search [21, 23, 43, 59]. Although this work specifically targets SSHOMs, our approach can be generalized to other types of interesting mutants, by updating the way we encode the search as a Boolean satisfiability problem.

Orthogonal to SSHOMs, researchers have recently investigated an interesting type of hard-to-kill mutants called *dominator mutants* [38, 39]. This line of work searches for the hardest-to-kill first-order mutants among a given set by comparing executions with regard to a test suite. Dominator mutants have been shown to be an effective research tool to study existing mutation testing techniques, for example for gauging mutation test completeness [41] and evaluating selective mutation [40]. Just et al. [30] show that program context can be used to approximate dominator mutants, which might also be promising for future search strategies for SSHOMs.

Characteristics of SSHOMs. Existing work on SSHOMs mostly discusses the quantity of SSHOMs and the difficulty of finding them [17, 21–23, 43]. For example, Harman et al. [17] discussed how SSHOMs relate to their constituent first-order mutants, but their discussion focuses mainly on test effectiveness and efficiency. Jia and Harman [23] discussed characteristics of a single SSHOM in the Triangle program (also used in our study) but did not explore SSHOM characteristics further. In our work, we can find a complete set of SSHOMs with regard to used tests and first-order mutants, which provides us more data to study what they look like.

Variational Execution. Variational execution was originally developed for information-flow analysis [3] and configuration testing [31, 32, 54]. We previously suggested that variational execution may have additional application scenarios, suggesting mutation testing as explored here as one promising direction [74].

With regard to using variational execution for mutation testing, Devroey et al. [9] are conceptually closest to our work in that they pursue a complete exploration strategy with similarities to lazy configuration exploration in SPLat [33, 53]. However, they explore only traces in state machines without any joining and thus forgo much possible sharing. Their analysis does not distinguish first-order from higher-order mutants and does not identify or analyze SSHOM. Orthogonal to our work, researchers have also used various techniques to speed up traditional mutation testing, such as sampling tests for mutant executions, condensing mutations into a metaprogram, and using advanced execution sharing techniques [24, 26, 60, 73]. At a technical level, Wang et al. [73] are closest to our work in that they look for possible redundant mutant executions by inspecting program state, but forgo potential joining after splitting mutant executions. Since our main goal of using variational execution is to explore the interactions of first-order mutants rather than speed up mutation analysis, we did not perform a performance comparison.

8 CONCLUSIONS

To efficiently find SSHOMs, we proceed in three steps. First, we use variational execution to find *all* SSHOMs in small to medium-sized programs. Second, we analyze the basic characteristics of the identified SSHOMs. Finally, we derive a new prioritized search

[3] We discuss precision with more data in the supplementary material.

strategy based on the characteristics. The prioritized search scales to large systems and is effective (albeit not complete) at finding SSHOMs and outperforms the existing state-of-the-art strategy by far. We hope that the insights and search strategies from this work can support future work in mutation testing.

ACKNOWLEDGMENTS

This work has been supported in part by the NSF (awards 1318808, 1552944, and 1717022), CNPq (Grant 424340/2016-0), CAPES, and FAPEMIG (grant PPM-00651-17). This project was started while Leo participated in the NSF-funded Research Experiences for Undergraduates in Software Engineering (REU-SE) program. We are grateful to all who provided feedback on this work, including Claire Le Goues, Jonathan Aldrich, René Just, the anonymous reviewers, and the audiences of early presentations of this work.

A APPENDIX: TEST SUITE RELEVANCE

As discussed in Sec. 2.3, for all practical search strategies, SSHOMs are identified in terms of used first-order mutants and tests. Different test suites may result in different SSHOMs, though it is conceptually possible to define SSHOMs in terms of an idealized test suite that covers all (possibly infinitely many) executions of a program.

To explore the influence of the test suite, in this appendix, we explore how different (real and ideal) test suites affect the number of (strict-)SSHOMs in a simple program. Given the large number of executions involved and the use of symbolic execution to represent the ideal test suite, we limit our exploration to our smallest subject system, the triangle program.

Real test suites of different sizes. Keeping the first-order mutants fixed, we use $\texttt{search}_{var}$ to compute the set of SSHOMs with regard to different test suites. Instead of the 26 tests used previously, we generate and use a much larger set of 1334 tests for this experiment, by systematically exploring combinations of different inputs for the program: For each of the triangle's three inputs, we consider values from −5 to 5, thus yielding 11^3 tests (easily reaching 100% line and branch coverage). To achieve maximum first-order mutation score, we manually add 3 more tests and verify that the remaining 3 out of 128 first-order mutants are equivalent mutants. From this large test suite, we downsample test suites of different sizes, each time randomly picking a subset of tests; we then identify (strict-)SSHOMs for this test suite with $\texttt{search}_{var}$. We report the median of five executions to account for randomness in selecting tests.

From Figure 7, we can see that the number of SSHOMs heavily depends on the test suite used, as expected (see Section 2). Also, there is a clear trend that the number of SSHOMs decreases as we use more comprehensive tests: the more tests used, the more constrained the search becomes and the fewer SSHOMs remain. In contrast, the number of strict-SSHOMs is low across different test suites, likely due to the fact that they are rare.

Idealized test suite using symbolic execution. Since the number of SSHOMs decreases as more comprehensive tests are used, one could expect that the number of SSHOMs will converge if we consider all possible tests (usually infinitely many). Given a set of first-order mutants, one can consider the *SSHOMs with regard to a given test suite* as an *approximation* of a true set of *SSHOMs with regard to an idealized test suite that includes all possible tests*, similar

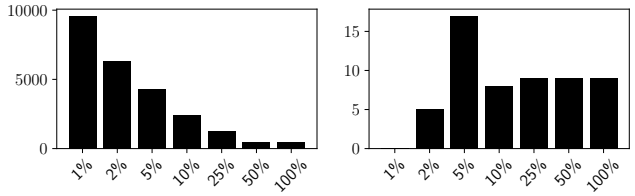


Figure 7: Number of found SSHOMs (left) and strict-SSHOMs (right) using different percentages of tests.

to how comprehensive tests can be used to approximate dominator mutants [30]. One would expect that larger test suites are better approximations of such an idealized test suite. While such a test suite usually does not and cannot exist, for the triangle program we can actually simulate the idealized test suite through symbolic execution. The program is simple enough that formal verification of whether two (mutated) variants are semantically equivalent is decidable and automatable.

To identify SSHOMs with an idealized test suite, we symbolically execute the triangle program with three symbolic variables for the three inputs to compute a symbolic representation of the program output—for which we developed a custom symbolic execution engine. The (possibly infinite) set of tests that distinguishes two programs p_1 and p_2 is the set of values for which $\phi(p_1) \neq \phi(p_2)$ where ϕ computes the symbolic output of the triangle program; if there are no such assignments to the three symbolic inputs, the two programs must be equivalent (as determined by an SMT solver). To check the behavior of a mutation m, we use $\Delta(m) = \big(\phi(p) \neq \phi(p+m)\big)$ to denote the symbolic expression that represents all tests that kill the mutant.

Using symbolic execution and an SMT solver, we can now determine whether a higher-order mutant is an SSHOM with regard to an idealized test suite, by using an SMT solver to solve constraints that encode Equation 1 to determine whether there are any assignments to the symbolic input variables, such that (1) the higher-order mutant fails at least for some tests (i.e., $SAT(\Delta(h))$, (2) that the higher-order mutant fails for those tests where all first-order mutants fail (i.e., $TAUT(\Delta(h) \Rightarrow \bigwedge_i \Delta(m_i))$), and (for strict-SSHOMs) that at least one test input passes for the higher-order mutant but fails for all first-order mutants (i.e., $SAT\big(\neg\Delta(h) \wedge \bigwedge_i \Delta(m_i)\big)$).

In theory, we can use our symbolic analysis to enumerate and verify all valid higher-order mutants in triangle to establish the set of SSHOMs wrt. the idealized test suite. However, we limit this experiment to only combinations of two and three mutants due to the vast search space. Using the idealized test suite of all possible tests, we found 159 second-order and 157 third-order SSHOMs, and 5 and 3 of them are strict-SSHOMs. Interestingly, all 159 and 157 SSHOMs and 2 (of 5) and all 3 strict-SSHOMs were also identified by $\texttt{search}_{var}$ using the original 26 test cases (Sec. 3). Furthermore, 467 of the 965 SSHOMs and 5 of the 6 strict-SSHOMs identified using the 26 tests in Sec. 3 are valid (strict-)SSHOMs with regard to the idealized test suite. That is, *SSHOMs with regard to a given test suite* can indeed be seen as an *approximation* of a true set of *SSHOMs with regard to an idealized test suite.*

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the Accuracy of Spectrum-Based Fault Localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, Windsor, UK, 89–98. https://doi.org/10.1109/TAIC.PART.2007.13

[2] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. 1993. Mining Association Rules Between Sets of Items in Large Databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD '93)*. ACM, New York, NY, USA, 207–216. https://doi.org/10.1145/170035.170072

[3] Thomas H. Austin and Cormac Flanagan. 2012. Multiple Facets for Dynamic Information Flow. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. ACM, New York, NY, USA, 165–178. https://doi.org/10.1145/2103656.2103677

[4] Thomas H. Austin, Jean Yang, Cormac Flanagan, and Armando Solar-Lezama. 2013. Faceted Execution of Policy-Agnostic Programs. In *Proceedings of the Eighth ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS '13)*. ACM, New York, NY, USA, 15–26. https://doi.org/10.1145/2465106.2465121

[5] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and W. Rance Cleaveland (Eds.), Vol. 1579. Springer Berlin Heidelberg, Berlin, Heidelberg, 193–207. https://doi.org/10.1007/3-540-49059-0_14

[6] James Bornholt and Emina Torlak. 2018. Finding Code That Explodes under Symbolic Evaluation. *Proc. ACM Program. Lang.* 2, OOPSLA (Oct. 2018), 149:1–149:26. https://doi.org/10.1145/3276519

[7] Randal E. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.* C-35, 8 (Aug. 1986), 677–691. https://doi.org/10.1109/TC.1986.1676819

[8] Marcelo d'Amorim, Steven Lauterburg, and Darko Marinov. 2008. Delta Execution for Efficient State-Space Exploration of Object-Oriented Programs. *IEEE Transactions on Software Engineering (TSE)* 34, 5 (2008), 597–613. https://doi.org/10.1109/TSE.2008.37

[9] Xavier Devroey, Gilles Perrouin, Mike Papadakis, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. 2016. Featured Model-Based Mutation Analysis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 655–666. https://doi.org/10.1145/2884781.2884821

[10] Jackson Antonio do Prado Lima and Silvia Regina Vergilio. 2019. A Systematic Mapping Study on Higher Order Mutation Testing. *Journal of Systems and Software (JSS)* 154 (2019), 92–109. https://doi.org/10.1016/j.jss.2019.04.031

[11] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. 2019. Empirical Review of Java Program Repair Tools: A Large-Scale Experiment on 2,141 Bugs and 23,551 Repair Attempts. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2019*. ACM Press, Tallinn, Estonia, 302–313. https://doi.org/10.1145/3338906.3338911

[12] Ahmed S. Ghiduk. 2016. Reducing the Number of Higher-Order Mutants with the Aid of Data Flow. *e-Informatica Vol. X* (2016), 2016; ISSN 18977979. https://doi.org/10.5277/E-INF160102

[13] Ahmed S. Ghiduk, Moheb R. Girgis, and Marwa H. Shehata. 2017. Higher Order Mutation Testing: A Systematic Literature Review. *Computer Science Review* 25 (Aug. 2017), 29–48. https://doi.org/10.1016/j.cosrev.2017.06.001

[14] Rahul Gopinath, Carlos Jensen, and Alex Groce. 2017. The Theory of Composite Faults. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Tokyo, Japan, 47–57. https://doi.org/10.1109/ICST.2017.12

[15] Mark Harman, Yue Jia, and William B. Langdon. 2010. A Manifesto for Higher Order Mutation Testing. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*. IEEE, Paris, France, 80–89. https://doi.org/10.1109/ICSTW.2010.13

[16] Mark Harman, Yue Jia, and William B. Langdon. 2011. Strong Higher Order Mutation-Based Test Data Generation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, New York, NY, USA, 212–222. https://doi.org/10.1145/2025113.2025144

[17] Mark Harman, Yue Jia, Pedro Reales Mateo, and Macario Polo. 2014. 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 '14*. ACM Press, Vasteras, Sweden, 397–408. https://doi.org/10.1145/2642937.2643008

[18] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. 1994. Experiments on the Effectiveness of Dataflow-and Control-Flow-Based Test Adequacy Criteria. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, Sorrento, Italy, 191–200. https://doi.org/10.1109/ICSE.1994.296778

[19] Chihiro Iida and Shingo Takada. 2017. Reducing Mutants with Mutant Killable Precondition. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, Tokyo, Japan, 128–133. https://doi.org/10.1109/ICSTW.2017.29

[20] Yue Jia. [n.d.]. *Higher Order Mutation Testing*. Ph.D. Dissertation.

[21] Yue Jia and Mark Harman. 2008. Constructing Subtle Faults Using Higher Order Mutation Testing. In *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, Beijing, China, 249–258. https://doi.org/10.1109/SCAM.2008.36

[22] Yue Jia and Mark Harman. 2008. MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language. In *Testing: Academic Industrial Conference - Practice and Research Techniques (Taic Part 2008)*. IEEE, Windsor, UK, 94–98. https://doi.org/10.1109/TAIC-PART.2008.18

[23] Yue Jia and Mark Harman. 2009. Higher Order Mutation Testing. *Information and Software Technology* 51, 10 (Oct. 2009), 1379–1393. https://doi.org/10.1016/j.infsof.2009.04.016

[24] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering (TSE)* 37, 5 (Sept. 2011), 649–678. https://doi.org/10.1109/TSE.2010.62

[25] James A. Jones and Mary Jean Harrold. 2005. Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. ACM, New York, NY, USA, 273–282. https://doi.org/10.1145/1101908.1101949

[26] René Just, Michael D. Ernst, and Gordon Fraser. 2014. Efficient Mutation Analysis by Propagating and Partitioning Infected Execution States. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*. ACM Press, San Jose, CA, USA, 315–326. https://doi.org/10.1145/2610384.2610388

[27] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. ACM, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[28] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are Mutants a Valid Substitute for Real Faults in Software Testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 654–665. https://doi.org/10.1145/2635868.2635929

[29] René Just, Gregory M. Kapfhammer, and Franz Schweiggert. 2011. Using Conditional Mutation to Increase the Efficiency of Mutation Analysis. In *Proceedings of the 6th International Workshop on Automation of Software Test (AST '11)*. ACM, New York, NY, USA, 50–56. https://doi.org/10.1145/1982595.1982606

[30] René Just, Bob Kurtz, and Paul Ammann. 2017. Inferring Mutant Utility from Program Context. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. ACM, New York, NY, USA, 284–294. https://doi.org/10.1145/3092703.3092732

[31] Christian Kästner, Alexander von Rhein, Sebastian Erdweg, Jonas Pusch, Sven Apel, Tillmann Rendel, and Klaus Ostermann. 2012. Toward Variability-Aware Testing. In *Proceedings of the 4th International Workshop on Feature-Oriented Software Development - FOSD '12*. ACM Press, Dresden, Germany, 1–8. https://doi.org/10.1145/2377816.2377817

[32] Chang Hwan Peter Kim, Sarfraz Khurshid, and Don Batory. 2012. Shared Execution for Efficiently Testing Product Lines. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, Dallas, TX, USA, 221–230. https://doi.org/10.1109/ISSRE.2012.23

[33] Chang Hwan Peter Kim, Darko Marinov, Sarfraz Khurshid, Don Batory, Sabrina Souto, Paulo Barros, and Marcelo d'Amorim. 2013. SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems. In *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*. ACM, New York, NY, USA, 257–267. https://doi.org/10.1145/2491411.2491459

[34] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394. https://doi.org/10.1145/360248.360252

[35] K. N. King and A. Jefferson Offutt. 1991. A Fortran Language System for Mutation-Based Software Testing. *Software: Practice and Experience* 21, 7 (July 1991), 685–718. https://doi.org/10.1002/spe.4380210704

[36] Marinos Kintis, Mike Papadakis, and Nicos Malevris. 2010. Evaluating Mutation Testing Alternatives: A Collateral Experiment. In *2010 Asia Pacific Software Engineering Conference*. IEEE, Sydney, NSW, Australia, 300–309. https://doi.org/10.1109/APSEC.2010.42

[37] Marinos Kintis, Mike Papadakis, and Nicos Malevris. 2015. Employing Second-Order Mutation for Isolating First-Order Equivalent Mutants. *Software Testing, Verification and Reliability* 25, 5-7 (Aug. 2015), 508–535. https://doi.org/10.1002/stvr.1529

[38] Bob Kurtz, Paul Ammann, Marcio E. Delamaro, Jeff Offutt, and Lin Deng. 2014. Mutant Subsumption Graphs. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. IEEE, OH, USA, 176–185. https://doi.org/10.1109/ICSTW.2014.20

[39] Bob Kurtz, Paul Ammann, and Jeff Offutt. 2015. Static Analysis of Mutant Subsumption. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, Graz, Austria, 1–10. https://doi.org/10.1109/ICSTW.2015.7107454

[40] Bob Kurtz, Paul Ammann, Jeff Offutt, Márcio E. Delamaro, Mariet Kurtz, and Nida Gökçe. 2016. Analyzing the Validity of Selective Mutation with Dominator Mutants. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 571–582. https://doi.org/10.1145/2950290.2950322

[41] Bob Kurtz, Paul Ammann, Jeff Offutt, and Mariet Kurtz. 2016. Are We There Yet? How Redundant and Equivalent Mutants Affect Determination of Test Completeness. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, Chicago, IL, USA, 142–151. https://doi.org/10.1109/ICSTW.2016.41

[42] Markus Kusano and Chao Wang. 2013. CCmutator: A Mutation Generator for Concurrency Constructs in Multithreaded C/C++ Applications. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Silicon Valley, CA, USA, 722–725. https://doi.org/10.1109/ASE.2013.6693142

[43] William B. Langdon, Mark Harman, and Yue Jia. 2010. Efficient Multi-Objective Higher Order Mutation Testing with Genetic Programming. *Journal of Systems and Software* 83, 12 (Dec. 2010), 2416–2430. https://doi.org/10.1016/j.jss.2010.07.027

[44] Duc Le, Mohammad Amin Alipour, Rahul Gopinath, and Alex Groce. 2014. MuCheck: An Extensible Tool for Mutation Testing of Haskell Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. ACM, New York, NY, USA, 429–432. https://doi.org/10.1145/2610384.2628052

[45] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, Zurich, Switzerland, 3–13. https://doi.org/10.1109/ICSE.2012.6227211

[46] Claire Le Goues, Stephanie Forrest, and Westley Weimer. 2013. Current Challenges in Automatic Software Repair. *Software quality journal* 21, 3 (2013), 421–443. https://doi.org/10.1007/s11219-013-9208-0

[47] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff. 2005. SOBER: Statistical Model-Based Bug Localization. *Proceedings of the International Symposium Foundations of Software Engineering (FSE)* 30, 5 (2005), 286–295. https://doi.org/10.1145/1095430.1081753

[48] Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. 2019. Bears: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Hangzhou, China, 468–478. https://doi.org/10.1109/SANER.2019.8667991

[49] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Józala. 2014. Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation. *IEEE Transactions on Software Engineering* 40, 1 (Jan. 2014), 23–42. https://doi.org/10.1109/TSE.2013.44

[50] Lech Madeyski and Norbert Radyk. 2010. Judy – a Mutation Testing Tool for Java. *IET Software* 4, 1 (2010), 32. https://doi.org/10.1049/iet-sen.2008.0038

[51] Sonal Mahajan and William G.J. Halfond. 2014. Finding HTML Presentation Failures Using Image Comparison Techniques. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. ACM, New York, NY, USA, 91–96. https://doi.org/10.1145/2642937.2642966

[52] Pedro Reales Mateo, Macario Polo Usaola, and José Luis Fernández Alemán. 2013. Validating Second-Order Mutation at System Level. *IEEE Transactions on Software Engineering* 39, 4 (April 2013), 570–587. https://doi.org/10.1109/TSE.2012.39

[53] Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. 2016. On Essential Configuration Complexity: Measuring Interactions in Highly-Configurable Systems. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 483–494. https://doi.org/10.1145/2970276.2970322

[54] Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. 2014. Exploring Variability-Aware Execution for Testing Plugin-Based Web Applications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 907–918. https://doi.org/10.1145/2568225.2568300

[55] A. Jefferson Offutt. 1992. Investigations of the Software Testing Coupling Effect. *ACM Transactions on Software Engineering and Methodology* 1, 1 (Jan. 1992), 5–20. https://doi.org/10.1145/125489.125473

[56] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. 1996. An Experimental Determination of Sufficient Mutant Operators. *ACM Transactions on Software Engineering and Methodology* 5, 2 (April 1996), 99–118. https://doi.org/10.1145/227607.227610

[57] A. Jefferson Offutt, Gregg Rothermel, and Christian Zapf. 1993. An Experimental Evaluation of Selective Mutation. In *Proceedings of 1993 15th International Conference on Software Engineering*. IEEE, Baltimore, MD, USA, 100–107. https://doi.org/10.1109/ICSE.1993.346062

[58] Elmahdi Omar, Sudipto Ghosh, and Darrell Whitley. 2014. HOMAJ: A Tool for Higher Order Mutation Testing in AspectJ and Java. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. IEEE, Cleveland, OH, USA, 165–170. https://doi.org/10.1109/ICSTW.2014.19

[59] Elmahdi Omar, Sudipto Ghosh, and Darrell Whitley. 2017. Subtle Higher Order Mutants. *Inf. Softw. Technol.* 81, C (Jan. 2017), 3–18. https://doi.org/10.1016/j.infsof.2016.01.016

[60] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation Testing Advances: An Analysis and Survey. In *Advances in Computers*. Vol. 112. Elsevier, 275–378. https://doi.org/10.1016/bs.adcom.2018.03.015

[61] Mike Papadakis and Nicos Malevris. 2010. An Empirical Evaluation of the First and Second Order Mutation Testing Strategies. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*. IEEE, Paris, France, 90–99. https://doi.org/10.1109/ICSTW.2010.50

[62] Spencer Pearson, Jose Campos, Rene Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and Improving Fault Localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, Buenos Aires, 609–620. https://doi.org/10.1109/ICSE.2017.62

[63] Goran Petrović and Marko Ivanković. 2018. State of Mutation Testing at Google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '18)*. ACM, New York, NY, USA, 163–171. https://doi.org/10.1145/3183519.3183521

[64] Goran Petrović, Marko Ivanković, Bob Kurtz, Paul Ammann, and René Just. 2018. An Industrial Application of Mutation Testing: Lessons, Challenges, and Research Directions. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, Vasteras, Sweden, 47–53. https://doi.org/10.1109/ICSTW.2018.00027

[65] Macario Polo, Mario Piattini, and Ignacio García-Rodríguez. 2009. Decreasing the Cost of Mutation Testing with Second-Order Mutants. *Softw. Test. Verif. Reliab.* 19, 2 (June 2009), 111–131. https://doi.org/10.1002/stvr.v19:2

[66] Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S. Foster, and Adam Porter. 2010. Using Symbolic Evaluation to Understand Behavior in Configurable Software Systems. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, New York, NY, USA, 445–454. https://doi.org/10.1145/1806799.1806864

[67] Manos Renieres and Steven P. Reiss. 2003. Fault Localization with Nearest Neighbor Queries. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.* IEEE, Montreal, Que., Canada, 30–39. https://doi.org/10.1109/ASE.2003.1240292

[68] Thomas Schmitz, Dustin Rhodes, Thomas H. Austin, Kenneth Knowles, and Cormac Flanagan. 2016. Faceted Dynamic Information Flow via Control and Data Monads. In *Proceedings of the 5th International Conference on Principles of Security and Trust - Volume 9635*. Springer-Verlag New York, Inc., New York, NY, USA, 3–23. https://doi.org/10.1007/978-3-662-49635-0_1

[69] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. 2015. MultiSE: Multi-Path Symbolic Execution Using Value Summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*. ACM Press, Bergamo, Italy, 842–853. https://doi.org/10.1145/2786805.2786830

[70] Susumu Tokumoto, Hiroaki Yoshida, Kazunori Sakamoto, and Shinichi Honiden. 2016. MuVM: Higher Order Mutation Analysis Virtual Machine for C. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Chicago, IL, USA, 320–329. https://doi.org/10.1109/ICST.2016.18

[71] David A. Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T. Devanbu, Bogdan Vasilescu, and Cindy Rubio-Gonzalez. 2019. BugSwarm: Mining and Continuously Growing a Dataset of Reproducible Failures and Fixes. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, Montreal, QC, Canada, 339–349. https://doi.org/10.1109/ICSE.2019.00048

[72] Roland H. Untch, A. Jefferson Offutt, and Mary Jean Harrold. 1993. Mutation Analysis Using Mutant Schemata. In *Proceedings of the 1993 International Symposium on Software Testing and Analysis - ISSTA '93*. ACM Press, Cambridge, Massachusetts, United States, 139–148. https://doi.org/10.1145/154183.154265

[73] Bo Wang, Yingfei Xiong, Yangqingwei Shi, Lu Zhang, and Dan Hao. 2017. Faster Mutation Analysis via Equivalence Modulo States. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. ACM, New York, NY, USA, 295–306. https://doi.org/10.1145/3092703.3092714

[74] Chu-Pan Wong, Jens Meinicke, and Christian Kästner. 2018. Beyond Testing Configurable Systems: Applying Variational Execution to Automatic Program Repair and Higher Order Mutation Testing. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, New York, NY, USA, 749–753. https://doi.org/10.1145/3236024.3264837

[75] Chu-Pan Wong, Jens Meinicke, Lukas Lazarek, and Christian Kästner. 2018. Faster Variational Execution with Transparent Bytecode Transformation. *Proc. ACM Program. Lang.* 2, OOPSLA (Oct. 2018), 117:1–117:30. https://doi.org/10.1145/3276487

[76] Hao Zhong and Zhendong Su. 2015. An Empirical Study on Real Bug Fixes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE, Piscataway, NJ, USA, 913–923. https://doi.org/10.1109/ICSE.2015.101