Faster Mutation Analysis via Equivalence Modulo States

Bo Wang, Yingfei Xiong, Yangqingwei Shi, Lu Zhang, Dan Hao
Key Laboratory of High Confidence Software Technologies (Peking University), MoE
Institute of Software, EECS, Peking University
Beijing, China 100871
wangbo_15,xiongyf,shiyqw,zhanglucs,haodan@pku.edu.cn

ABSTRACT

Mutation analysis has many applications, such as asserting the quality of test suites and localizing faults. One important bottleneck of mutation analysis is scalability. The latest work explores the possibility of reducing the redundant execution via split-stream execution. However, split-stream execution is only able to remove redundant execution before the first mutated statement.

In this paper we try to also reduce some of the redundant execution after the execution of the first mutated statement. We observe that, although many mutated statements are not equivalent, the execution result of those mutated statements may still be equivalent to the result of the original statement. In other words, the statements are equivalent modulo the current state. In this paper we propose a fast mutation analysis approach, AccMut. AccMut automatically detects the equivalence modulo states among a statement and its mutations, then groups the statements into equivalence classes modulo states, and uses only one process to represent each class. In this way, we can significantly reduce the number of split processes. Our experiments show that our approach can further accelerate mutation analysis on top of split-stream execution with a speedup of 2.56x on average.

CCS CONCEPTS

•Software and its engineering →Software testing and debugging;

KEYWORDS

software testing; mutation testing; dynamic analysis

ACM Reference format:
Bo Wang, Yingfei Xiong, Yangqingwei Shi, Lu Zhang, Dan Hao. 2017. Faster Mutation Analysis via Equivalence Modulo States. In *Proceedings of 26th ACM SIGSOFT International Symposium on Software Testing and Analysis,, Santa Barbara, CA, USA, July 10-14, 2017 (ISSTA'17),* 12 pages.
DOI: 10.1145/3092703.3092714

1 INTRODUCTION

Mutation analysis [5, 18, 24] is a powerful approach for program analysis. The general process of mutation analysis has two steps. First, change the original program with predefined mutation operators and generate a set of mutated programs, called *mutants*. Then, the mutants are executed against a test suite, and information is collected during the execution for various purpose of analysis.

Mutation analysis has many applications. The most representative application is assessing the quality of a test suite. In this application, mutants are treated as seeded faults, and the test suite that detects more mutants is considered better [2, 27]. A test case fails on a mutant is known to *kill* that mutant. There are also many other applications of mutation analysis. For example, recently several papers [4, 20, 41, 49–52, 68] proposed to use mutation analysis for fault localization. Bug fixing techniques in the "generate-and-validate" style [32, 53, 59] have been shown to be a dual of mutation analysis [58]. Mutation analysis is also used for test generation [10, 11, 45, 55, 67], test prioritization [38], program verification [12] and security analysis [37, 42]. In addition, some techniques of variability management, model-based testing and component-based testing are quite close to mutation analysis [6, 7, 21, 39, 47].

However, mutation analysis suffers from one bottleneck: scalability. Since we need to test all mutants for a test suite, the expected analysis time is n times of the expected execution time of the test suite, where n is the number of mutant generated. The number of n depends on the size of the program, but even a mid-size program produces thousands of mutants.

Researchers have realized the problem of scalability, and have proposed many different approaches for accelerating mutation analysis. One of the basic ideas of acceleration is to remove the redundant and unnecessary computations. Mutant schemata [57] avoids the redundancies in compilations. Just et al.'s approach [26] removes redundant mutation executions while Zhang et al.'s approach [65] removes unnecessary mutation executions. The latest work explores split-stream execution [56] to remove redundancies in part of the mutation executions. Given two mutants, the execution of any test before the first mutated statement is redundant. Split-stream execution starts with one process representing the original program, and split a new process when the first mutated statement of a mutant is encountered. In this way, the redundancies before the first mutated statement are removed.

Split-stream execution only removes redundant executions before the first mutated statement. However, executions after the first mutated statement may also be redundant. A typical case is that two statements are equivalent modulo the current state. Given the current state of the program, if the executions of two statements lead to the same new state, we say the two statements are equivalent modulo the state. We observe that, although in general only a small portion of mutated statements are equivalent, there are many more mutated statements that are equivalent modulo the current

Yingfei Xiong is the corresponding author.

ISSTA'17, Santa Barbara, CA, USA

DOI: 10.1145/3092703.3092714

state. For example, given two different side-effect-free Boolean expressions, the probability that they are equivalent is small. However, given a state, the probability that the two expressions produce the same result is 50%, if we assume the evaluation result has an even distribution. Given two mutants and the state before their first different statements, if the two statements are equivalent modulo the state, the execution of the two statements and all following execution until the next different statements are still redundant.

In this paper we propose a novel mutation analysis approach, AccMut, to remove such redundancies. Like split-stream execution, AccMut starts the executions of all mutants in one process. When we encounter a position where at least two mutants have different states, AccMut clusters the mutants based on their equivalence classes modulo the current state, i.e., two mutants are put into the same cluster if their next state after executing the current statement is still the same. Next, AccMut splits the execution into a set of processes, where each process represents one cluster of equivalent mutants. In this way, AccMut can remove part of the redundancies after the first mutated statement. More concretely, there are two more types of reductions compared to split-stream execution. First, when a mutated statement is equivalent to the original statement modulo the current state, we do not necessarily split the execution from the main process. Second, when two mutated statements are equivalent modulo the current state, we do not unnecessarily split two processes for them.

There are some challenges to implement AccMut. First, the cluster process, which is invoked at every location mutation occurs, should be efficient, otherwise the overhead from the clustering may offset the benefit. In our approach we have carefully designed the clustering process so that the time complexity is effectively constant. Second, the process splitting should also be efficient. In our approach we exploit the POSIX system call `fork` to implement the clustering process. This design choice has two benefits. First, it allows us to still compile and execute the mutant, in contrast to existing implementation of split-stream execution that relies on interpreters. As studied by existing work [24, 33], compiler-based mutation analysis is usually much faster than interpreter-based. Second, based on the copy-on-write mechanism of the POSIX `fork`, the splitting process is very fast, causing almost no delay in the execution.

We have evaluated our approach on eleven C programming projects with totally 337122 mutants and 20736 tests. The evaluation shows that, on top of the state-of-art approach for accelerating mutation analysis, our approach further accelerates the analysis, with a speedup of 2.56X on average. We have implemented these optimizations in our tool AccMut for C language.

2 RELATED WORK

In general, the work for accelerating mutation analysis can be divided into lossy approaches and lossless approaches [24].

Lossy Acceleration. A typical lossy approach is weak mutation [22], where a mutation is assumed to be killed by a test if the mutated code changes the current system state. In this way, we do not need to execute any mutant program but only need to test at the mutated points. Recently, weak mutation has been further accelerated by using split-stream execution, which forks new threads on calling mutated method in Java bytecode [9]. However, the results become imprecise as a mutant changing the system state does not necessarily violate the test. In other words, weak mutation only evaluates the capability of the test suite to trigger the fault, but not whether the triggered fault can be propagated to the output and the capability of the test suite to capture the propagated faults. Other lossy approaches include randomly sampling the mutants [60], clustering the mutants and sampling a small number of mutants from each cluster [23], mutant operator selection only adopts a subset of mutation operators [29, 64], select an efficient mutant subset [15, 30, 40, 63], and select tests to execute [65]. Zhang et al.'s work utilizes machine learning to predict the mutant execution results without any execution [62]. Different to lossy approaches, AccMut is a lossless approach, accelerating mutation testing without sacrificing precision.

Lossless Acceleration. A main type of lossless approaches seek to reduce redundant computation in mutation analysis.

Mutation schemata [57] can compile all mutants into a single executable file at a time. Mutants are slight syntactic transformations of the original program, so the most parts of their code are duplicated. Mutation schemata can reduce redundance in compilation.

Split-stream execution [17, 33, 56], as mentioned in the introduction, is the technique that reduces the redundant computation before the first mutated statement. Split-stream execution is first proposed by King and Offutt [33], and then explored by several researchers [17, 56]. The basic idea is to start all mutants execution in one main process, and split the execution into different processes when the first mutated statement is encountered. As discussed, split-stream execution only reduces the redundant computation before the first mutated statements while our approach can reduce redundant computations after those.

Several approaches [43, 48] exist for detecting the equivalence of mutants. Once an equivalent group is detected, only one mutant in the group needs to be executed. Just et al. [26] take a step further to detect equivalence of mutants with respect to one test. Two inequivalent mutants may behave completely the same under one test execution and thus we only need to execute one of them for the test. Compared to these approaches, our approach is more fine-grained as we can reduce the redundancy in part of the test execution. For example, suppose an expression e in a loop is mutated. Under test t, e is evaluated 10 times, where the first 9 times the mutant evaluates to the same value as the original expression, but the 10th evaluation gives a different value. Using either equivalent mutant detection or Just et al.'s approach, the mutant has to be considered as different from the original program and will be fully executed, while in our approach the execution before the 10th evaluation will be shared. In other words, equivalent mutant detection considers absolute equivalence between mutants, Just et al.'s approach considers the equivalence between mutants modulo test input, while our approach considers the equivalence between statements modulo the current state, which is more fine-grained.

Some lossless approaches seek for parallel execution of mutation analysis. Approaches supporting different architectures have been explored, such as MIMD [44] and SIMD [34]. If we view each test-mutant pair as an independent program, we can parallelize their execution on a MIMD machine. On the other hand, if we view each

test as a different input data, we can parallelize the execution of different tests on one mutant on a SIMD machine.

Finally, in the application of evaluating a test, several papers [28, 65, 66] propose to prioritize the tests for each mutation so that this mutation shall be killed quicker. The works are orthogonal to ours and can be used together with AccMut.

3 BASIC FRAMEWORK OF ACCMUT

3.1 Overview

We first describe the redundant execution that AccMut avoids with an example in the following code snippet and in Figure 1. In function `foo`, the line 6 is a computation intensive function without side-effects. The test driver function is `test_foo`, which sets the parameter `a` to `1` and then judges the result. Let us assume that three mutants are generated in the function `foo`. Mutant 1 (M1) changes `a = a + 1` at line 3 into `a = a << 1`. Mutant 2 (M2) and Mutant 3 (M3) change `a = a / 2` at line 5 into `a = a + 2` and `a = a * 2`, respectively.

In standard mutation analysis, we execute all mutants for each test and obtain their testing results. We show the execution of the three mutants in Figure 1(b), (c) and (d), and the execution of the original program in Figure 1(a) as a reference.

```
1  int foo(int a){
2    int i, res;
3    a = a + 1; // M1:a << 1
4    for(i = 0; i < 2; i++){
5      a = a/2; // M2:a + 2, M3:a * 2
6      res += time_consuming(a);
7    }
8    return res;
9  }
10 void test_foo(){
11   assert(foo(1) == RESULT);
12 }
```

The circles represent system states and the states with the same number are the same. The states are captured at the program point listed on the left of Figure 1. The arrows represent the transition of states after the executions of statements. And the length of the arrows means the execution effort during the transition. If two transitions have the same starting state and the same ending state, the two transitions are redundant. To show the redundancy between states and transitions, we use the brightness of circles and the thickness of arrows respectively. The darker a circle is, or the thicker an arrow is, the more redundant the state/transition is. As we can see from Figure 1(b), (c) and (d), there are several redundant transitions among the three mutant executions. First, as the parameter a of foo is set to 1 in this test (State 1), the results of $a = a + 1$ and $a = a << 1$ both equal to 2 (State 2). As a result, the transitions before entering the loop, i.e., transitions to State 1 and transitions between States 1 and 2, are redundant among the three mutants. Second, during the first loop $a = 2$, the states of $a = a + 2$ and $a = a * 2$ are the same, and thus the transitions between States 2 and 5 in M2 and M3 are also redundant. Note that these two transitions involve the call to a time-consuming function which can induce high cost of redundancy, so the length of the arrows during the loop is much longer.

Figure 1(e) exhibits the execution of the mutants in split-stream execution. In split-stream execution, all mutants start as one main process, and are later split from the main process at the first mutated statements. The main process of our example is shown in the first column in Figure 1(e). M1 is split as a new process after State 1, and later M2 and M3 are split as new processes after State 2. We need to keep the main process as more mutants may be split from it. Some of the redundancy is removed in split-stream execution. For example, the transition to State 1 is shared among all the mutants as well as the main process. However, two types of redundancy still exists. First, the transitions between State 1 and 2 are the same between the main process and M1, and thus it is not necessary to split M1 from the main process. Second, although it is necessary to split M2 and M3 after State 2, the transitions between State 2 and State 5, which involve calling the time-consuming function, are still redundant among the two split processes.

Our approach, AccMut, tries to further reduce the redundancy in execution by exploiting the equivalence modulo the current state. An execution of the three mutants in AccMut is shown in Figure 1(f). Different from split-stream execution, AccMut first classifies the next statements in different mutants into equivalent classes modulo the current state, and uses one process to represent each equivalent class. As a result, first, since the mutated statement in M1 is equivalent to the original statement modulo State 1, we would not split a new process for M1. Second, the two mutated statements in M2 and M3 are equivalent modulo State 2, so we split only one process for them. As a result, the redundant transitions in Figure 1(e) are all removed in Figure 1(f).

More concretely, at each state of each process, we conduct a trial execution of the statements and collect their changes to the system state. Then we cluster their changes to the system state into equivalence classes. If the number of equivalence classes is more than one, say n, we split $n - 1$ new processes. Each forked process represents the mutants in one equivalence class, and we apply the change from the equivalence class to the state of the forked process. Finally, the change from the remaining equivalent class is applied to the original process, and the original process now represents only the mutants in the remaining class. This process continues for each process until all processes terminate.

However, in practice it may be expensive to store and cluster the changes, especially when a statement makes a large set of changes. For example, if a statement calls a procedure with a large side effects, i.e., changing many memory locations, it may not be efficient to record the changes of all memory locations and compare the changes from different mutants.

To solve this problem, in AccMut we record abstract changes rather than concrete changes. An abstract change stores often much less information than a concrete change, but nevertheless allows the application of the change to the system state. For example, the change of a procedure call can be represented abstractly by the address of the procedure and all arguments passed to the procedure, rather than all concrete changes produced by the procedure. When we need to apply the change to system, we just actually invoke the method with the arguments. In this way, we record only a small amount of information allowing us to efficiently store and cluster the changes.

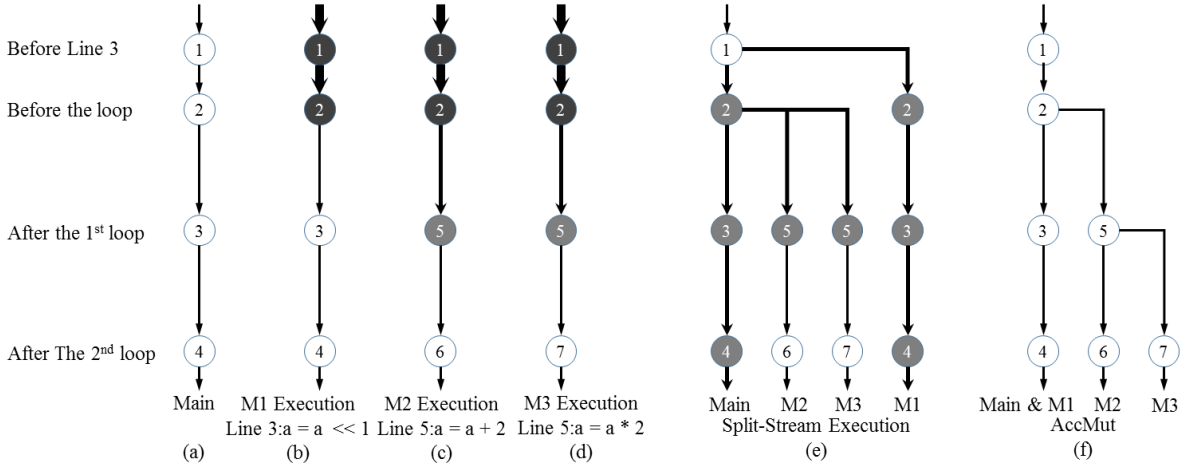

Figure 1: Dynamic Mutation Analysis with Equivalence Analysis

When two abstract changes are the same, applying them to the same system state gives the same new state. However, the inverse is not always true: when two abstract changes are different, they may not always produce different states. For example, invoking a method with a different set of arguments may not necessarily lead to a different system state. In other words, the equivalence relation we computed is conservative: when two statements are equivalent in the computed equivalence relation, they are equivalent modulo the current state; when two statements are inequivalent in the computed equivalence relation, they may still be equivalent modulo the current state.

3.2 Definitions

In this sub section we define a set of concepts and operators that we will use to describe our approach. These definitions abstract away concrete details in mutation operators such that our approach can be generally applicable to different sets of mutation operators. Given a program, a mutation analysis first applies a set of mutation operators to produce mutants from the program. Since the operations can be applied in different granularities in different mutation analyses, e.g., on instruction level, on expression level, or on statement level. We use an abstract concept—location—to represent the unit that a mutation operator applies. Each mutant can be identified by a unique mutant ID.

More concretely, a program can be viewed as a set of locations. A mutation procedure p is a function mapping each location to a set of variants. Each variant v consists of a block of code (denoted as $v.code$) that can be executed and a set of mutant IDs (denoted as $v.I$) that denote the mutants where this variant of code block is enabled. The union of mutant IDs from all variants at any two locations are the same, i.e., $\bigcup_{v \in p(l_1)} v.I = \bigcup_{v \in p(l_2)} v.I$ for any mutation procedure p and any two locations l_1, l_2, and the union represents all mutant IDs in the system. Given any two variants from the same location, their mutation IDs are disjoint, i.e., $v_1, v_2 \in p(l) \Rightarrow v_1.I \cap v_2.I = \emptyset$ for any location l. Given a mutation ID i, the code block $v.code$ at each location l where $i \in v.I \wedge v \in p(l)$ forms a new program, called a mutant.

We use a special set $STATE_p$ to denote all possible states in the execution of program p. Intuitively, a state represents all values stored in memory locations and registers, as well as the states of external resources that may be accessed by the program. The execution of a mutant is represented by a sequence of system states.

A special function ϕ maps each system state to a location, which indicates the next code block to execute. The execution terminates at state s when $\phi(s) = \perp$, meaning the process is finish. Operation `execute` executes a code block on a system state. Given a variable s containing a system state and a code block c, `execute`(s, c) updates the system state in-place. Operation `execute` can be decomposed into two operations `try` and `apply`. Invocation `try`(s, c) executes code block c on system state s, and returns a (potentially abstract) change x describing the changes to the system state, without actually changing s. Invocation `apply`(x, s) applies the change x in-place to a variable s containing a system state. We require that invoking `apply`(`try`$(s, c), s)$ is equivalent to invoking `execute`(s, c). Please note that while $x = y \Rightarrow$ `apply`$(x, s) =$ `apply`(y, s) holds, $x \neq y \Rightarrow$ `apply`$(x, s) \neq$ `apply`(y, s) does not necessarily hold, allowing us to define abstract changes.

To implement AccMut efficiently, we also need three additional operations, as follows. The time complexity of the three operations should be as small as possible, preferably in constant time.

- `fork`. Similar to the POSIX system call `fork()`, this operation splits a child process from the current process.
- `filter_variants`(V, I). This operation filters a set of variants in-place based on a set of mutant IDs, leaving only the variants enabled for the mutants, i.e., V is updated to $\{v \mid v \in V \wedge v.I \cap I \neq \emptyset\}$. The variants are assumed to be at the same location.
- `filter_mutants`(I, V). This operation filters a set of mutant IDs in-place based on a set of variants, leaving only the mutants containing one of the variant, i.e., I is updated to $\{i \mid i \in I \wedge \exists v \in V.i \in v.I\}$. The variants are assumed to be at the same location.

3.3 Standard Mutation Analysis

Based on the definitions, we can build algorithms for mutation analysis. We shall introduce AccMut step by step. We first start with the standard mutation analysis without any acceleration (this subsection), then we extend the algorithm into split-stream execution, and finally we extend split-stream execution into AccMut.

The algorithm for standard mutation analysis is as shown in Algorithm 1. Given all mutant IDs in the system, the algorithm executes them one by one (line 1). The execution of a mutant is a series of state transitions until there is no code block to execute (line 3). At each transition, the system first selects proper variant at the current location (line 4), and then executes the variant (line 5). Finally, necessary information about the execution result is recorded by calling save(s, i) (line 7).

Note that a direct implementation of the algorithm effectively executes the program with an interpreter. Another way of implementation is to apply p beforehand, and instrument this algorithm into the target program. This implementation is equivalent to mutant schemata [57], where all mutants are generated into the program to save compilation cost.

```
   Input: p: a mutation procedure
   Data: s: the current system state
1  for each mutant ID i in all mutant IDs do
2  |  s ← the initial system state
3  |  while φ(s) ≠ ⊥ do
4  |  |  {v} ← filter_variants(p(φ(s)), {i})
5  |  |  execute(v.code, s)
6  |  end
7  |  save(s, i)
8  end
```

Algorithm 1: Standard Analysis

```
   Input: p: a mutation procedure
   Data: s: the current system state
   Data: I: a set of mutant IDs represented by the current process
1  I ← all mutant IDs
2  s ← the initial system state
3  while φ(s) ≠ ⊥ do
4  |  proceed(p(φ(s)))
5  end
6  for i ∈ I do
7  |  save(s, i)
8  end
```

Algorithm 2: Main Loop of Split-stream Execution and AccMut

3.4 Mutation Analysis of Split-Stream Execution

The main loop of split-stream execution is in Algorithm 2. There are three differences from standard mutation analysis: (1) for each process, there is a set I indicating the mutant IDs represented by the current process, which is initialized to all mutant IDs; (2) at the end of execution, save() is called for each mutation ID in I; (3) a procedure proceed() is called for state transitions and execution forking.

```
   Input: V: a set of variants at current location
   Data: s: the current system state
   Data: I: a set of mutation IDs represented by the current
            process
1  filter_variants(V, I)
2  if |V| = 1 then
3  |  v ← the only variant in V
4  |  execute (v.code, s)
5  |  return
6  end
7  v′ ← a random variant in V
8  for each v in V where v ≠ v′ do
9  |  pid ← fork()
10 |  if in child process then
11 |  |  filter_mutants(I, {v})
12 |  |  execute (v.code, s)
13 |  |  return //child process directly return
14 |  end
15 end
16 filter_mutants(I, v′)
17 execute (v′.code, s)
```

Algorithm 3: Algorithm of proceed(V) in Split-stream Execution

The procedure of proceed() is shown as Algorithm 3. If there is only one variant to execute, we directly execute the variant and return (lines 2-6). If there are more than one variant, we first select one variant to be represented by the current process (line 7), and then fork a new process for each remaining variant (lines 8-15). When we fork a new process, the new process represents the mutant IDs of the variants in the equivalence class (line 11), and the corresponding variant is executed (line 12). Finally, we update the mutant IDs of the original process and execute the selected variant (line 16-17).

3.5 Mutation Analysis in AccMut

The main loop of AccMut is the same as split-stream execution in Algorithm 2, however the algorithm of proceed() is different. As shown in Algorithm 4, first we will check the number of variants, which is the same as split-stream execution (lines 2-6). The main difference starts from lines 7-10, where we first collect the changes produced by the mutants into set X. Then we cluster the changes into equivalence classes (line 11). The rest of the algorithm has a similar structure to split-stream execution, where each equivalent class corresponds to a variant in split-stream execution. We first select a class that the current process represents (line 12), and then fork a new process for each other cluster (lines 13-22), and finally update the mutant IDs and the state of the current process (lines 23-26).

3.6 Correctness of AccMut

Theorem 3.1. *The algorithms of standard mutation analysis, split-stream execution, and AccMut produce exactly the same sequence of invocations to save(s, i) with the same arguments.*

Input: V: a set of variants at current location
Data: s: the current system state
Data: I: a set of mutation IDs represented by the current process

```
filter_variants(V, I)
if |V| = 1 then
    v ← the only variant in V
    execute (v.code, s)
    return
end
X = ∅
for each v in V do
    X ← X ∪ {try(v.code, s)}
end
𝕏 ← group changes in X into equivalent classes
X_cur ← any one of the equivalent classes in 𝕏
for each equivalence class X in 𝕏 − {X_cur} do
    V ← the variants corresponding to changes in X
    pid ← fork()
    if in child process then
        filter_mutants(I, V)
        x ← a random change in the equivalence class X
        apply(x, s)
        return
    end
end
V ← the variants corresponding to changes in X_cur
filter_mutants(I, V)
x ← a random change in X_cur
apply(x, s)
```

Algorithm 4: Algorithm of proceed(V) in AccMut

Proof Sketch. This can be proved by an induction over the number of state transitions to show that the same sequence of state transitions occurs for each mutant in all three algorithms. When the length is zero, this property trivially holds. When length is k, we can see that each algorithm selects the same variant to execute for each mutant, and thus the property holds for length $k + 1$.

4 IMPLEMENTING ACCMUT FOR FIRST-ORDER MUTATION ANALYSIS ON LLVM IR

Previous section gives the basic framework of our algorithm. In this section we demonstrate how to implement this framework for first-order mutations on LLVM IR. LLVM [36] is a widely used compiler framework where different front-ends exist to translate different high-level languages such as C, C++, Java, into its intermediate representation (IR). To implement the framework, we need to design the mutation operators, and implement operations including `try`, `apply`, `fork`, `large_change`, `filter_variants`, and `filter_mutants`.

Our implementation is available online[1].

[1] https://github.com/wangbo15/accmut

Table 1: Mutation Operators

Name	Description	Example
AOR	Replace arithmetic operator	$a + b \rightarrow a - b$
LOR	Replace logic operator	$a \,\&\, b \rightarrow a \mid b$
ROR	Replace relational operator	$a == b \rightarrow a >= b$
LVR	Replace literal value	$T \rightarrow T + 1$
COR	Replace bit operator	$a \,\&\&\, b \rightarrow a \mid\mid b$
SOR	Replace shift operator	$a >> b \rightarrow a << b$
STDC	Delete a call	$foo() \rightarrow nop$
STDS	Delete a store	$a = 5 \rightarrow nop$
UOI	Insert a unary operation	$b = a \rightarrow a++;\ b = a$
ROV	Replace the operation value	$foo(a, b) \rightarrow foo(b, a)$
ABV	Take absolute value	$foo(a, b) \rightarrow foo(abs(a), b)$

4.1 Mutation Operators

As we mutate on the LLVM IR level, each IR instruction corresponds to a location. Table 1 describe the mutation operators considered in our implementation. These operators are designed by mimicking operators on existing IR-based mutation tools. Since we do not find any mutation tool working on LLVM IR code, we mimic the operators from two mutation tools that work on Java byte code, which takes a similar form to LLVM IR. The two tools we chose are Major [25, 31] and Javalanche [54], both of which are widely used mutation tools for Java. We converted all operators from Major and Javalanche except for two operators from Javalanche manipulating Java concurrency classes such as `Monitor`. These operators cannot be converted because there is no corresponding instruction or functions in LLVM IR level.

4.2 Implementing `fork`

We implement operation `fork` using the POSIX system call `fork`. The system call uses a copy-on-write mechanism [3], where the newly created process shares the same memory space as the original process, and only when a process performed a writing operation, the corresponding memory page is copied and stops to be shared between processes. In this way, the `fork` operation can be completed in constant time.

However, the system call only supports virtual memory access but not IO operations. To support IO operations, we need to reimplement all IO functions with the copy-on-write mechanism. Our current implementation already supports file IO. When a file is opened, we allocate a memory chunk and copy the file into the chunk. Later all IO operations on the file are redirected to the memory chunk, which are already supported by the copy-on-write mechanism of system call `fork`. The memory file can be written back into the disk at the end of the test. This is necessary when a test depends on another test through files, e.g., the first test writes a configuration file while other tests read it. Please note that utilizing POSIX `fork` still leads to several limitations such as being unable to handle multithreaded program [13] or kernel programs [1]. Supporting these programs is future work on implementation.

4.3 Implementing `try` and `apply`

The key point of implementing `try` and `apply` is to define the format of the abstract changes. Since LLVM IR is in the three-address form, most instructions modify only one memory location with primitive

value. For those instructions, the changes are easy to define: we just need to record the memory location the instruction modifies and the new value the instruction stores at the location.

The only exception is procedure call, as the callee procedure may modify many different locations. As mentioned in the overview section, the abstract change by a procedure call is defined as a tuple including the address of the procedure and the arguments passed to the procedure. Please note that in LLVM IR all passed arguments are well typed. For example, an invocation of the function *void foo*(*int*, *int*) has two mutants, an ROV ($foo(a,b) \rightarrow foo(b,a)$) and an STDC (remove the call). The three variants give three changes, $foo(a,b)$, $foo(b,a)$, and an empty change. The first two changes are equivalent only when $a = b$, and the third change is always different from the first two.

4.4 Implementing filter_variants and filter_mutants

Since `filter_variants` will be performed at every location, and `filter_mutants` will be performed every time we fork a process, it is better to keep the time complexity of the two operations small, preferable $O(1)$. In this sub section we discuss how to implement the two operations for first-order mutation.

The challenges of implementing the two operations is that both operations require the computation of set intersection, and a standard set implementation has a time complexity of $O(n \log n)$, where n is the set size. Since the set may contain all mutant IDs, this method is too costly. To get an efficient implementation, we utilize the fact that the number of variants at each location has a small upper bound u. If the complexities of the operations only depend on u but not the total number of mutants, the complexity is $O(1)$. In other words, we assume that $O(u) = O(u^2) = O(\log u) = \ldots = O(1)$.

More concretely, at each location, there are two types of variants. First, variants generated by applying a mutation operator on the current the location, called *mutant variants*. Such a variant is always enabled for only one mutant ID. Second, the variant of the original instruction, called *original variant*. The variant is enabled for all remaining mutant IDs.

Utilizing this fact, we can design different data structures for different sets. First, for each process, there is a set of mutant IDs that the current process represents. Initially the set contains all mutant IDs, but each time a process is forked into a set of processes, there is at most one process executing the original variant and the sizes of mutation IDs in all other processes are smaller than u. Therefore, we use two different data structures to represent sets of mutation IDs. The mutation IDs of the initial process is represented by a bit vector. Each bit corresponds to a mutant ID, where one indicates this mutant ID is in the set, and zero indicates this mutant ID is not in the set. With a bit vector, operations such as adding, removing, membership query can be finished in $O(1)$ time. Whenever a process is forked, the process executing the original variant inherits the bit vector from its parent, and for all other processes, we create a new linked list for storing the mutant IDs. Since the size of the list is bounded by u, the operations on the list, such as membership query, is effectively $O(1)$.

Second, there is a variable V in Algorithm 4 storing a set of variants. Also for each variant, there is a set of mutant IDs that the variant represents. We represents these sets by treating the two types of variants different. We use a data structure `VariantSet` to store a set of variants. More concretely, a `VariantSet` is a tuple, (`ori_variant`, `ori_included`, `mut_variants`), where `ori_variant` is the code of the original variant, `ori_included` is a Boolean variable indicating whether the original variant is included in the set or not, and `mut_variants` is a linked list of mutant variants. Each mutant variant v consists of a block of code and one mutant ID. To avoid confusion, we shall use $v.i$ to represent the only mutant ID. In this way, we can quickly test whether the original variant is included in the set or not. Also the size of `mut_variants` is bound by u, so the operations on the set is effectively constant time.

The algorithm for implementing `filter_variants` is shown in Algorithm 5. We first filter the mutant variants (lines 3-6). Since all operations have the complexity of $O(1)$ (note that $O(u) = O(1)$) and the loop is bounded by u, this part has the complexity of $O(1)$. Next, we consider whether the original variant should be filtered out or be kept (lines 8-12). Since each mutant variant is enabled for one mutant, if the currently selected variants are fewer than the mutants represented by the current process, there must be remaining mutants and the original variant should also be selected. It is easy to see this part also takes $O(1)$. As a result, the complexity of `filter_variants` is $O(1)$.

The algorithm of `filter_mutants` is shown in Algorithm 6. If the original mutant is included in V, we build the result negatively by removing mutant IDs (lines 1-4), otherwise we build the result positively by adding mutant IDs (lines 6-9). Since all operations and all loops are bound by u, the whole algorithm takes $O(1)$.

```
   Input: V: a set of variants to be filtered
   Input: I: a set of mutation IDs used to filter V
 1 V' ← a new VariantSet
 2 V'.ori_variant ← V.ori_variant
 3 for each v ∈ V.mut_variants do
 4 |   if I.contains(v.i) then
 5 |   |   V'.mut_variants.add(v)
 6 |   end
 7 end
 8 if V'.mut_variants.size < I.size then
 9 |   V'.ori_included = true
10 else
11 |   V'.ori_included = false
12 end
13 V ← V'
```

Algorithm 5: `filter_variants`

4.5 Parallelism Control

A delicate point in our implementation is parallelism control. If a large number of mutants can be generated from a program, we may fork a large number of processes. Too many processes may lead to a large overhead in scheduling these processes. As a result, we limit the number of parallel processes in the implementation. In our current implementation we limit the number of parallel processes to one. There are two reasons for this limit. (1) In the evaluation shown later, this design controls the uncertainty brought by parallel

```
Input: I: a set of mutation IDs to be filtered
Input: V: a set of variants used to filter I
Data: MV: all mutant variants at the current location
1  if V.ori_included then
2      for each v ∈ MV − V.mut_variants do
3          I.remove(v.i)
4      end
5  else
6      I ← a new empty linked list
7      for each v ∈ V.mut_variants do
8          I.add(v.i)
9      end
10 end
```

Algorithm 6: `filter_mutants`

execution, giving a more stable execution time. (2) The parallelism management is simpler: each time we fork a child process, we suspend the parent until the child process exits. Furthermore, though we only allow one parallel process for a test execution, it is still possible to parallel mutation analysis using our tool: we can parallelize the executions of different tests.

5 EVALUATION

Our evaluation aims to answer the research question: How does AccMut perform compared to existing approaches in terms of speedup?

5.1 Subjects

To answer the research questions, we totally collected eleven subjects. The statistics of the selected programs are shown in Table 3. Ten of the subjects are from the SIR repository [8] and one (*vim 7.4*) is from an open source project. We select these subjects because they cover a wide range of application domains from GUI editor to command-line compression, cover both real-world applications and widely-used benchmarks, and cover applications of different sizes from hundreds of lines to hundreds of thousands of lines. Also, many of the subjects have been used in existing studies. For example, *vim 7.4* is the new version of the largest subject (*vim 7.2*) used in Papadakis et al.'s empirical study [48], which is one of the largest empirical study on mutation analysis in recent years. Please note that *vim 7.4* has about 11k more lines of code than *vim 7.2*, making our subject even larger. Since the whole *vim* is too large for evaluation, following Papadakis et al. [48], we selected the largest two components, *eval* and *spell*, as targets of the mutation operators. In Table 3, we list the LOC of the two components in parentheses.

The subjects have total 504482 lines of code, 20736 tests and 337122 mutants upon 27612 IR-level locations.[2] The average of u, namely the average number of mutants per mutated location, is 12.2. The max u of a subject is in the range of 22 to 43. Thus, Algorithm 5 and Algorithm 6 can be bounded in constant complexity generally.

Please note that the subjects we use are among the largest in studies of mutation analysis. Table 2 shows the studies of mutation analysis in recent years. As we can see from the table, our largest subject is among the largest in terms of both the lines of code and the number of mutants. Especially, the number of mutants is the highest among all subjects. This is because other studies either use fewer mutation operators, use more coarse-grained operators, or perform sampling on the mutants, which reduce the number of mutants.

[2]LOC is collected by the tool `cloc` (http://cloc.sourceforge.net).

5.2 Procedures

In the experiments we compared AccMut with two controlling techniques: mutant schemata [57] and split-stream execution [17, 33, 56]. However, we cannot find a publicly-available tool that implements both mutant schemata and split-stream execution for C programs. The only state-of-the-art tool that implements split-stream execution on C within our knowledge is MuVM [56]. However, this tool is not publicly available. As a result, we implemented the two techniques by modifying the implementation of our tool. Mutant schemata is implemented as Algorithm 1 and split-stream execution is implemented as Algorithm 2 and Algorithm 3.

In our experiment, we sequentially executed the tests but not parallelized them, in order to obtain more stable results. Since there is at most one forked process running at a time, the execution of mutants was also sequential. Furthermore, we executed each subject using each technique three times and recorded the median execution time.

The experiments are performed on a Ubuntu 16.04 laptop with 2.50 GHz Intel i7-4710MQ CPU and 12GB memory.

5.3 Results

The execution time of the three techniques is shown in Table 4. From the table, we can make the following observations.

- On all subjects AccMut constantly outperforms the other two techniques, suggesting that exploiting the equivalence modulo a state can reduce the more redundant computation over the state-of-the-art techniques, and the benefit outperforms the extra overhead.
- On average, AccMut is 2.56x faster than split-stream execution and 8.95x than mutation schemata. Our approach can significantly boost the performance of mutation analysis over existing approaches.
- Split-stream execution also significantly outperforms mutant schemata, with an average speedup of 3.49x. This result is consistent with an existing study [56].

5.4 Detailed Analysis

To further understand how AccMut achieved the speedups, we performed a detailed analysis of the execution process. First, the main reason our approach outperforms previous approaches is that fewer mutants are executed, and we analyzed how significant the reduction is. Table 5 shows the average number of executed mutants for each approach. Mutation schemata executes all mutants. Split-stream execution executed only mutants covered by a test. AccMut executes much fewer mutants than both approaches, as many covered mutants are still equivalent modulo state.

Besides reducing the mutants executed, AccMut may also introduce extra overheads in trial execution of the instructions, clustering the changes, etc. To further understand how the introduced

Table 2: Summary of the Recent Work on Mutation Analysis

Author(s) [Reference]	Conference	Language	Largest Subject	Max Mutant Number	Sampled Mutants
Just et al. [28]	ISSRE'12	Java	116,750	160,891	No
Gligoric et al. [14]	ISSTA'13	Java	68,218	738	No
Zhang et al. [65]	ISSTA'13	Java	36,910	36,418	Yes
Harman et al. [19]	ASE'14	Java	6,359	7,204	Yes
Just et al. [26]	ISSTA'14	Java	116,750	108,174	No
Yao et al. [61]	ICSE'14	C	35,545	3,838	Yes
Papadakis et al. [48]	ICSE'15	C	362,769 (39,008)	72,432	Yes
Kurtz et al. [35]	FSE'16	C	512	11,080	Yes
Gopinath et al. [16]	ICSE'16	Java	Not given	122,484	Yes
Zhang et al. [62]	ISSTA'16	Java	29,702	28,133	Yes
Papadakis et al. [46]	ISSTA'16	C	20,461	22,023	Yes
This paper	—	C	477,257 (42,073)	173,683	No

Table 3: Subject Programs

Name	LOC	Tests	Mutants	Locations	Avg u	Max u	Description
flex	10334	42	56916	5119	11.1	32	a lexical analyzer generator
gzip	4331	214	37326	3058	12.2	22	a tool for file compression
grep	10102	75	58571	4373	13.4	34	a tool for searching plain-text files
printtokens	475	4130	1862	199	9.4	22	a lexical analyzer
printtokens2	401	4115	2501	207	12.1	22	an other lexical analyzer
replace	512	5542	3000	220	13.6	22	a tool for pattern matching
schedule	292	2650	493	55	9.0	22	a priority scheduler
schedule2	297	2710	1077	121	8.9	22	another priority scheduler
tcas	135	1608	937	73	12.8	35	an aircraft collision avoidance tool
totinfo	346	1052	756	63	12.0	22	a statistics tool
vim 7.4	477257 (42073)	98	173683	14124	12.3	43	a text editor
Total	504482	20736	337122	27612	12.2	—	—

Table 4: Experimental Results

Subjects	AccMut	SSE	MS	SSE/AccMut	MS/AccMut	MS/SSE
flex	12m58s	40m22s	1h26m17s	3.11x	6.65x	2.13x
gzip	50.4s	2m32s	55m19s	3.02x	65.85x	21.84x
grep	2m19s	7m16s	58m56s	3.13x	25.36x	8.10x
printtokens	11m55s	23m36s	2h10m54s	1.94x	10.98x	5.67x
printtokens2	11m35s	38m7s	57m24s	3.30x	4.97x	1.51x
replace	17m27s	41m22s	44m56s	2.37x	2.57x	1.09x
schedule	3m14s	6m20s	8m14s	1.96x	2.54x	1.30x
schedule2	6m40s	13m18s	17m05s	2.00x	2.56x	1.28x
tcas	7.7s	21.2s	16m31s	2.6x	128.7x	46.7x
totinfo	1m55s	4m28s	6m19s	2.33x	3.30x	1.41x
vim 7.4	1m9s	2m10s	3h26m6s	1.88x	179.2x	95.1x
Total	1h10m10s	2h59m52s	10h28m1s	2.56x	8.95x	3.49x

AccMut = Our approach, SSE = Split-Stream Execution [56], MS = Mutant Schemata [57], XXX/YYY = Speed up of YYY over XXX

overheads compared to the saved executions, we measured the number of the original instructions executed and the number of the extra instructions executed in the three approaches. Note that a better measurement here is the time used to execute the instructions, but the LLVM profiler does not work on our implementation because of a signal conflict, so we use the number of instructions to approximate the time. Because of the high cost of tracing the instructions, we only measured the first 100 tests of *tcas* and *printtokens*.

The result is shown in Table 6. As we need to select a variant at each location, the extra instructions executed is much more than the original instructions in all three approaches. AccMut has much higher relative overheads compared to SSE and MS, where 79 extra instructions are executed on average for one original instruction. However, the absolute overheads of AccMut is even lower, as more redundant computations are removed. Please note that, despite the overhead, all three approaches are much faster than plain mutation analysis without any acceleration because of the cost from compiling. Table 7 shows the execution time for mutation schemata and plain mutation analysis for the 100 tests of the two subjects. As we can see, mutation schemata is on average 9.62 times faster.

Table 5: The Average Number of Executed Mutants For Each Test

Subjects	AccMut	SSE	MS	AccMut/SSE	AccMut/MS	SSE/MS
flex	5881.1	16610.7	56916	35.4%	10.3%	29.1%
gzip	434.5	1638.1	37326	26.5%	1.2%	4.4%
grep	1303.2	4014.4	58571	32.5%	2.2 %	6.9%
printtokens	413.5	1019.3	1862	40.6%	22.2%	54.7%
printtokens2	750.4	1724.4	2501	43.5%	30.0%	68.9%
replace	483.7	1484.1	3000	32.6%	16.1%	49.5%
schedule	180.0	405.9	493	44.3%	36.5%	82.3%
schedule2	384.9	844.2	1077	45.6%	35.7%	78.4%
tcas	99.2	434.1	937	22.9%	10.6%	46.3%
totinfo	220.5	566.0	756	39.0%	29.2%	74.9%
vim 7.4	601.0	1472.7	173683	40.8%	0.3%	0.8%
Average	977.5	2746.7	30647.4	35.6%	3.2%	9.0%

Table 6: The Number of Executed Instructions

Instruction Type	AccMut	SSE	MS	AccMut/SSE	AccMut/MS	SSE/MS
Original Program	1,054,174	2,404,487	37,617,433	43.8%	2.9%	6.4%
Extra Cost	83,148,833	96,490,502	177,988,701	86.2%	46.7%	54.2%
Total Executed	84,203,007	98,894,989	215,606,134	85.1%	39.1%	45.9%

Table 7: Execution Time of Mutation Schemata and Plain Mutation Analysis

Subjects	MS	Plain	Plain/MS
tcas	50s	323s	6.46X
printtoken	36s	504s	14.00X
Total	86s	827s	9.62X

5.5 Threats to Validity

The main threat to internal validity is that our implementations may be wrong. To reduce this threat, we manually checked part of the analysis result and found that the result of our approach is consistent with the result of plain mutation analysis.

The main threat to external validity is the mutation operators we used. Using different mutation operators may have a noticeable effect on the performance. AccMut used mutation operators from widely-used tools, and chose subjects from different areas. Thus, our results have a high chance to represent the typical use cases.

The main threat to construct validity is the way we measure performance may be imprecise. To reduce this threat, we performed only sequential but not parallel operations in the experiments, and repeated the experiments three times and report the median results. As a matter of fact, there were only very small differences between the three executions in our experiments, indicating that sequential execution leads to a stable result.

6 LIMITATIONS AND FUTURE WORK

Other IO Operations. One limitation of the current implementation is that it cannot well handle all types of external resources, such as database connections, network commutations, etc. To deal with this problem, we need to implement the copy-on-write mechanism also for more types of external resources. This is a future work on the tool implementation. Nevertheless, this limitation may not be a serious one as well-written tests often use mock objects rather than directly accessing external resources.

Multi-Threaded Programs and Low-Level System Programs. So far we only consider single-threaded and application programs. POSIX `fork()` does not support multi-threaded programs and is not accessible to low-level system programs, such as linux kernel [1]. New mechanisms thus need to be found.

7 CONCLUSION

In this paper, we propose AccMut, which removes redundancies in executions by exploiting the equivalence of statements modulo the current state. The experimental results suggest that our approach can achieve significant speedup over existing reduction approaches.

ACKNOWLEDGMENTS

We acknowledge the fruitful discussion with Rene Just at University of Massachusetts Amherst and Micheal Ernst at University of Washington.

The work is supported by the National Key Research and Development Program under Grant No. 2016YFB1000105, and National Natural Science Foundation of China under Grant No. 61421091, 61529201, 61672045 and 61332010.

REFERENCES

[1] Iftekhar Ahmed, Carlos Jensen, Alex Groce, and Paul E McKenney. 2017. Applying Mutation Analysis on Kernel Test Suites: An Experience Report. In *ICSTW*. 110–115.
[2] J. H. Andrews, L. C. Briand, and Y. Labiche. 2005. Is mutation an appropriate tool for testing experiments?. In *Proc. ICSE*. 402–411.
[3] Randal E Bryant, O'Hallaron David Richard, and O'Hallaron David Richard. 2003. *Computer systems: a programmer's perspective*. Prentice Hall Upper Saddle River.
[4] Thierry Titcheu Chekam, Mike Papadakis, and Yves Le Traon. 2016. Assessing and Comparing Mutation-based Fault Localization Techniques. *arXiv preprint arXiv:1607.05512* (2016).
[5] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. 1978. Hints on test data selection: Help for the practicing programmer. *Computer* 11, 4 (1978), 34–41.

[6] Xavier Devroey, Gilles Perrouin, Maxime Cordy, Mike Papadakis, Axel Legay, and Pierre-Yves Schobbens. 2014. A Variability Perspective of Mutation Analysis. In *FSE*. 841–844.

[7] Xavier Devroey, Gilles Perrouin, Mike Papadakis, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. 2016. Featured model-based mutation analysis. In *ICSE*. 655–666.

[8] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. 2005. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering: An International Journal* 10, 4 (2005), 405–435.

[9] Vinicius HS Durelli, Jeff Offutt, and Marcio E Delamaro. 2012. Toward harnessing high-level language virtual machines for further speeding up weak mutation testing. In *ICST*. 681–690.

[10] Gordon Fraser and Andrea Arcuri. 2015. Achieving scalable mutation-based generation of whole test suites. *Empirical Software Engineering* 20, 3 (2015), 783–812.

[11] Gordon Fraser and Andreas Zeller. 2012. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering* 38, 2 (2012), 278–292.

[12] Juan P Galeotti, Carlo A Furia, Eva May, Gordon Fraser, and Andreas Zeller. 2015. Inferring loop invariants by mutation, dynamic analysis, and static checking. *IEEE Transactions on Software Engineering* 41, 10 (2015), 1019–1037.

[13] Milos Gligoric, Vilas Jagannath, Qingzhou Luo, and Darko Marinov. 2013. Efficient mutation testing of multithreaded code. *Software Testing, Verification and Reliability* 23, 5 (2013), 375–403.

[14] Milos Gligoric, Lingming Zhang, Cristiano Pereira, and Gilles Pokam. 2013. Selective mutation testing for concurrent code. In *Proc. ISSTA*. 224–234.

[15] Rahul Gopinath, Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. 2016. Measuring Effectiveness of Mutant Sets. In *IEEE ICSTW*. 132–141.

[16] Rahul Gopinath, Mohammad Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. 2016. On the limits of mutation reduction strategies. In *ICSE*. 511–522.

[17] Rahul Gopinath, Carlos Jensen, and Alex Groce. 2016. Topsy-Turvy: a smarter and faster parallelization of mutation analysis. In *ICSE*. 740–743.

[18] Richard G Hamlet. 1977. Testing programs with the aid of a compiler. *Software Engineering, IEEE Transactions on* SE-3, 4 (1977), 279–290.

[19] 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 *ASE*. 397–408.

[20] Shin Hong, Byeongcheol Lee, Taehoon Kwak, Yiru Jeon, Bongsuk Ko, Yunho Kim, and Moonzoo Kim. 2015. Mutation-based fault localization for real-world multilingual programs (T). In *ASE*. 464–475.

[21] Shan-Shan Hou, Lu Zhang, Tao Xie, Hong Mei, and Jia-Su Sun. 2007. Applying interface-contract mutation in regression testing of component-based software. In *ICSM 2007*. 174–183.

[22] William E. Howden. 1982. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering* SE-8, 4 (1982), 371–379.

[23] Changbin Ji, Zhenyu Chen, Baowen Xu, and Zhihong Zhao. 2009. A Novel Method of Mutation Clustering Based on Domain Analysis.. In *SEKE*. 422–425.

[24] Yue Jia and Mark Harman. 2011. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on* 37, 5 (2011), 649–678.

[25] René Just. 2014. The Major Mutation Framework: Efficient and Scalable Mutation Analysis for Java. In *ISSTA*. ACM, 433–436.

[26] René Just, Michael D Ernst, and Gordon Fraser. 2014. Efficient mutation analysis by propagating and partitioning infected execution states. In *ISSTA*. 315–326.

[27] 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 *FSE*. 654–665.

[28] René Just, Gregory M Kapfhammer, and Franz Schweiggert. 2012. Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis. In *ISSRE*. 11–20.

[29] René Just, Bob Kurtz, and Paul Ammann. 2017. *Customized Program Mutation: Inferring Mutant Utility from Program Context*. Technical Report UM-CS-2017-004. University of Massachusetts, Amherst, Amherst, MA, USA.

[30] René Just and Franz Schweiggert. 2015. Higher accuracy and lower run time: Efficient mutation analysis using non-redundant mutation operators. *Software Testing, Verification and Reliability (JSTVR)* 25, 5-7 (2015), 490–507.

[31] Rene Just, Franz Schweiggert, and Gregory M Kapfhammer. 2011. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *ASE*. 612–615.

[32] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *ICSE '13*. 802–811.

[33] K. N. King and A. Jefferson Offutt. 1991. A Fortran Language System for Mutation-based Software Testing. *Softw. Pract. Exper.* 21, 7 (June 1991), 685–718.

[34] Edward W. Krauser, Aditya P. Mathur, and Vernon J Rego. 1991. High performance software testing on SIMD machines. *IEEE Transactions on Software Engineering* 17, 5 (1991), 403–423.

[35] 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 *FSE*. 571–582.

[36] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*. 75–86.

[37] Thomas Loise, Xavier Devroey, Gilles Perrouin, Mike Papadakis, and Patrick Heymans. 2017. Towards Security-aware Mutation Testing. In *ICSTW*. 97–102.

[38] Yiling Lou, Dan Hao, and Lu Zhang. 2015. Mutation-based test-case prioritization in software evolution. In *ISSRE*. 46–57.

[39] Evan Martin and Tao Xie. 2007. A fault model and mutation testing of access control policies. In *WWW*. 667–676.

[40] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. 2015. Guided mutation testing for javascript web applications. *IEEE Transactions on Software Engineering* 41, 5 (2015), 429–444.

[41] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the mutants: Mutating faulty programs for fault localization. In *ICST*. 153–162.

[42] Tejeddine Mouelhi, Yves Le Traon, and Benoit Baudry. 2007. Mutation analysis for security tests qualification. In *TAICPART-MUTATION 2007*. 233–242.

[43] A Jefferson Offutt and W Michael Craft. 1994. Using compiler optimization techniques to detect equivalent mutants. *Software Testing, Verification and Reliability* 4, 3 (1994), 131–154.

[44] A Jefferson Offutt, Roy P Pargas, Scott V Fichter, and Prashant K Khambekar. 1992. Mutation testing of software using a MIMD computer. In *Proc. ICPP*.

[45] Kai Pan, Xintao Wu, and Tao Xie. 2013. Automatic test generation for mutation testing on database applications. In *Proceedings of the 8th International Workshop on Automation of Software Test*. 111–117.

[46] Mike Papadakis, Christopher Henard, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Threats to the validity of mutation-based test assessment. In *ISSTA*. 354–365.

[47] Mike Papadakis, Christopher Henard, and Yves Le Traon. 2014. Sampling program inputs with mutation analysis: Going beyond combinatorial interaction testing. In *ICST*. 1–10.

[48] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. 2015. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *ICSE*. 936–946.

[49] Mike Papadakis and Yves Le Traon. 2012. Using mutants to locate" unknown" faults. In *ICST*. 691–700.

[50] Mike Papadakis and Yves Le Traon. 2014. Effective fault localization via mutation analysis: A selective mutation approach. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. 1293–1300.

[51] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: mutation-based fault localization. *Software Testing, Verification and Reliability* 25, 5-7 (2015), 605–628.

[52] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating & improving fault localization techniques. In *ICSE*.

[53] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The Strength of Random Search on Automated Program Repair. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. 254–265.

[54] David Schuler and Andreas Zeller. 2009. Javalanche: efficient mutation testing for Java. In *ESEC/FSE*. 297–298.

[55] Francisco Carlos M Souza, Mike Papadakis, Yves Le Traon, and Márcio E Delamaro. 2016. Strong mutation-based test data generation using hill climbing. In *IEEE/ACM SBST*. 45–54.

[56] Susumu Tokumoto, Hiroaki Yoshida, Kazunori Sakamoto, and Shinichi Honiden. 2016. MuVM: Higher Order Mutation Analysis Virtual Machine for C. In *ICST*. 320–329.

[57] Roland H Untch, A Jefferson Offutt, and Mary Jean Harrold. 1993. Mutation analysis using mutant schemata. In *Proc. ISSTA*. 139–148.

[58] W. Weimer, Z.P. Fry, and S. Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *ASE*. 356–366.

[59] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *ICSE '09*. 364–374.

[60] W Eric Wong and Aditya P Mathur. 1995. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software* 31, 3 (1995), 185–196.

[61] Xiangjuan Yao, Mark Harman, and Yue Jia. 2014. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In *ICSE*. 919–930.

[62] Jie Zhang, Ziyi Wang, Lingming Zhang, Dan Hao, Lei Zang, Shiyang Cheng, and Lu Zhang. 2016. Predictive mutation testing. In *ISSTA*. 342–353.

[63] Jie Zhang, Muyao Zhu, Dan Hao, and Lu Zhang. 2014. An empirical study on the scalability of selective mutation testing. In *ISSRE*. 277–287.

[64] Lu Zhang, Shan-Shan Hou, Jun-Jue Hu, Tao Xie, and Hong Mei. 2010. Is operator-based mutant selection superior to random mutant selection?. In *Proc. ICSE*. 435–444.

[65] Lingming Zhang, Darko Marinov, and Sarfraz Khurshid. 2013. Faster mutation testing inspired by test prioritization and reduction. In *Proc. ISSTA*. 235–245.

[66] Lingming Zhang, Darko Marinov, Lu Zhang, and Sarfraz Khurshid. 2012. Regression mutation testing. In *Proc. ISSTA*. 331–341.

[67] Lingming Zhang, Tao Xie, Lu Zhang, Nikolai Tillmann, Jonathan De Halleux, and Hong Mei. 2010. Test generation via dynamic symbolic execution for mutation testing. In *Proc. ICSM*. 1–10.

[68] Lingming Zhang, Lu Zhang, and Sarfraz Khurshid. 2013. Injecting mechanical faults to localize developer faults for evolving software. In *Proc. OOPSLA*. 765–784.