Dynamic Mutation Analysis

Yingfei Xiong, Bo Wang, Yangqingwei Shi, Yingkui Cao, Lu Zhang
Key Laboratory of High Confidence Software Technologies (Peking University), MoE
Institute of Software, EECS, Peking University, Beijing, 100871, China
{xiongyf, shiyqw, zhanglucs}@pku.edu.cn, sa512079@mail.ustc.edu.cn, 289407137@qq.com

ABSTRACT

Mutation analysis has many applications, such as asserting the quality of test suites, fault localization, etc. One important bottleneck of mutation analysis is scalability, and researchers have proposed different techniques to accelerate mutation analysis. A typical way is to remove redundant computations in mutation analysis. However, all redundancy reduction techniques are static, and thus cannot remove redundancy that occurs in part of a mutant execution.

In this paper we propose dynamic mutation analysis, which analyzes mutants during the execution of the program, and forks the execution only when a mutant leads to a new system state. Our experiments on 2713 mutants and 13729 tests show that our approach can further accelerate mutation analysis on top of Just et al's approach, a state-of-the-art approach, with a speedup up to 2.22X.

We propose an abstract model for implementing dynamic mutation analysis for different types of mutation operators, as well as extending it to software product line testing, an area previously independently developed with mutation analysis. Furthermore, by comparing with our algorithm, we identify that Just et al's approach, which was previously believed to be precise, may produce imprecise result. On the other hand, our approach produces precise results with better performance.

1. INTRODUCTION

Mutation analysis [5, 7] is an important technique in program analysis. Given a program, *mutation analysis* changes the program with predefined mutation operators and generates a set of mutated programs, called *mutants*. Then, the mutants are executed under a test suite, and information is collected during the execution for various purposes of analysis.

Mutation analysis has many applications. The most representative application is to assert the quality of a test suite [12]. In this application, the mutated code pieces are considered as injected faults, and the test suit that fails on more mutants are considered to better. 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 [24, 22, 36] proposed to use mutation analysis for fault localization. Furthermore, bug fixing techniques in the "generate-and-validate" style [32, 19, 27] have been shown to be a dual of mutation analysis [31].

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 the mid-size program produces thousands of mutants. As a result, mutation analysis has been adopted only in a limited scope in practice.

Researchers have realized the problem of scalability, and have proposed many different approaches for accelerating mutation analysis. These approaches can be classified as lossy and lossless. Lossy approaches sacrifice precision for efficiency, e.g., by sampling some mutants to run [33] or by predicting the execution result at the middle of execution [10]. Lossless approaches try to remove the redundant computation in mutation analysis, e.g., by integrating mutants into one program to avoid redundancy in compilation [30], or by identifying mutants with equivalent executions and executing only one of them [13]. Existing lossless approaches on reducing redundancies in mutant executions are static: a prepass is first executed to analyze which mutants need to be executed, and then these mutants are executed as in standard mutation analysis.

However, there is one type of redundant computation cannot be removed in static approaches: redundancy in only part of a mutation execution. Given two mutants, the executions of a test before the first mutated statement are completely the same. However, we cannot remove this redundancy in static approaches because, if the executions of the two mutants after the mutated statement are different, each mutant gives a unique execution result and thus cannot be removed beforehand.

In this paper we propose dynamic mutant analysis for lossless acceleration of mutation analysis. Unlike previous approach that analyzes mutants in a prepass, dynamic mutant analysis starts with a process representing all mutants, analyzes mutants during the execution of the program, and forks the execution when a mutant leads to a new system state. This approach has two main advantages.

- The redundancy in part of the execution can be removed because the executions before the mutated statement are shared. Therefore, dynamic mutation analysis leads to a significant speedup in mutation analysis.

- By forking executions only at new system state, dynamic mutation analysis effectively subsumes two state-of-the-art redundancy reduction approaches: mutation schemata [30] and the approach by Just et al. [13], allowing us to further reduce the execution time on top of the state of the art.

By comparing our approach with Just et al.'s approach, we also identify that Just et al.'s approach is not a lossless approach as previously believed, and may produce imprecise results. On the other hand, due to its dynamic nature, dynamic mutation analysis can still perform the reductions in Just et al.'s approach without producing imprecise results. As a matter of fact, if we want to make Just et al.'s approach lossless, we have to introduce a mechanism similar to dynamic mutation testing.

To realize dynamic mutation analysis for different types of mutation operators, we define an abstract model of the infrastructure that supports dynamic mutation analysis. As we shall see later, this abstract model allows us to not only integrate different mutation operators, but also extend dynamic mutation analysis to software product line testing [16, 17, 21], an area previously independently developed with mutation analysis. The extension builds connection between two areas and allows us to compare techniques between them.

Compared with approaches in software product line testing [16, 17, 21], a key novelty in our approach is that it enables the exploit of the POSIX system call `fork` for highly efficient forking of executions, leading to a much more lightweight implementation and much less runtime overhead than the approaches in software product line testing, which use specialized interpreters [17] or model checkers [21].

We have implemented dynamic mutation analysis on C programming language and evaluated our approach on five projects with totally 2713 mutants and 13729 tests. The evaluation shows that, on top of the state-of-art approach for accelerating mutation analysis [13], our approach further accelerates the analysis, with a speedup up to 2.22X.

To sum up, the contributions of this paper are summarized below.

- The novel concept of dynamic mutation analysis, which leads to significant speedup over existing techniques. (Section 3)
- An abstract model for dynamic mutation analysis, enabling the integration of different mutation operators as well as the extension of mutation analysis to other areas. (Section 3.3)
- Two realization techniques of the abstract model, one for first-order mutation testing (Section 3.7) and one for software product line testing (Section 5). The latter builds connection between two previously independently developed areas.
- An experimental evaluation of dynamic mutation analysis that demonstrates its effectiveness. (Section 4)
- The identification of the previously unknown imprecision in Just et al.'s approach [13] by comparing their approach with dynamic mutation analysis. (Section 3.6.3)

2. RELATED WORK

2.1 Acceleration of Mutation Analysis

There has been a lot of work on accelerating mutation analysis. A summary of the work can be found in the survey by Jia and Harman [12]. In general, the work for accelerating mutation analysis can be divided into lossy approaches and lossless approaches. A typical lossy approach is weak mutation [10], where a mutation is assumed to be killed by a test if the mutated code changes the current system state. Other typical lossy approaches include randomly sampling the mutants [33], clustering the mutants and sampling a small number of mutants from each cluster [11], and selecting tests to execute [35]. Different to lossy approaches, dynamic mutation analysis is a lossless approach, accelerating mutation testing without sacrificing precision.

A main type of lossless approaches seeks to reduce redundant computation in mutation analysis. Mutant schemata [30] reduces redundancy in compilation. Given any two mutants, many parts of their code are duplicated. Compiling these duplicated code leads to redundant computation. Mutation schemata generates one meta mutant that simulates different mutants by setting different runtime parameters, and this meta mutant only needs to be compiled once. Another redundant computation is unnecessary mutant executions. When the mutated code is not covered by a test, it is not necessary to execute this mutant under the test. Several modern mutation tools [28, 14] use a prepass analyze the coverage information of the tests, and avoid execution of the uncovered mutants. Finally, Just et al. [13] takes a step further by identifying, for a test, mutants whose executions are equivalent to the original program and mutants whose executions are the same with each other. Mutants of the former type do not need to be executed and only one mutant in each group of the second type needs to be executed. However, since all these approaches are static, they cannot avoid the redundant computation that constitutes only part of a mutant execution. Furthermore, one contribution of the paper is to identify that Just et al.'s approach is actually lossy, which will be explained in Section 3.6.3.

Furthermore, some lossless approaches seek for parallelizing mutation testing. Parallelizing on different architectures have been explored, including the MIMD architecture [23] and the SIMD architecture [20]. These accelerations are orthogonal to our approach and can be used together.

Finally, recently Zhang et al. [35] propose to prioritize the tests for each mutant so that this mutant shall be killed quicker. In applications such as evaluating a test suite, when a test kills a mutant, we do not need to execute the rest of the tests for that mutant. As a result, the quicker the tests kill the mutant, the faster the mutation analysis. Future work is needed to integrate this kind of reduction into dynamic mutation analysis.

2.2 Software Product Line Testing

A related field is software product line testing. A software product line generates a large number of software products based on a set of configuration options, and an important problem in testing software product lines is how to efficiently determine whether a test passes on all configurations. A typical approach in this area is variability-aware execution, which reduces the cost of testing software product lines by sharing the redundant executions from different products.

Current variability-aware execution approaches are mainly heavyweight, either adopting a specialized interpreter [17, 16] that treats the state from different executions as a meta state with variabilities, or encode the execution semantics of a family of products into a model checker [21, 16].

By viewing mutants as software products generated from a product line, we can apply variability-aware execution to mutation analysis. However, there is an important difference between software product line and mutation analysis: based on the combination of different configuration options, a software product line easily expands to millions or billions of products on a few tens of options [17]; on the other hand, after static reduction, there is usually hundreds or even tens of mutations to execute per test [13]. On the other hand, even the newest variability-execution approach [21] has a slowdown of thousands of times, and thus these approaches are unlikely to accelerate mutation analysis.

Compared with variability-aware execution approaches, dynamic mutation analysis is lightweight in two senses. First, by only forking executions, dynamic mutation analysis causes little runtime overhead, accelerating the analysis even if the number of mutants is small. Second, dynamic mutation analysis is easy to implement, without requiring specialized interpreter or full translation to a model checker.

It would be interesting to know whether dynamic mutation analysis can be applied to software product testing and how it compares to variability-aware execution approaches. One challenge here is how to merge the configuration options and constraints over them into dynamic mutation analysis. In a later section of the paper we shall show how to implement them into our abstract model. However, a full implementation and an experimental comparison with variability-aware execution are beyond the scope of this paper.

2.3 Generate-and-Validate Bug Fixing

Recently, generate-and-validate bug fixing techniques [32, 19, 27] receive a lot of attentions. In a typical generate-and-validate bug fixing approach, a mutation of the program is first generated, and then tests are executed on the mutation to determine whether the bug is fixed. If not, a new mutation is generated, and the process repeats until the bug is fixed. Existing work [31] has shown that generate-and-validate bug fixing is a dual of mutation analysis.

Several approaches have been independently proposed to accelerate the generate-and-validate bug-fixing process, for example, by reusing the previous compilation result [25], prioritizing the test [26, 31], and clustering equivalent mutants [31]. However, within our knowledge, no approach has explored the redundancy occurring in part of a mutant execution, and thus dynamic mutation analysis could potentially further accelerate generate-and-validate bug fixing. This is a future work.

2.4 General Computation Reuse

There are also approaches that reduce redundant computations in general program executions. Typical approaches include self-adjusting computation [1], which automates incremental computation based on the differences between inputs, and computation reuse [3], which allows reusing the execution of a block of code. However, these approaches are not designed for a family of mutated programs, and thus cannot be directly applied to mutation analysis.

On an abstract level, we can view a family of mutated programs as inputs to an interpreter of the respective language, and in this sense self-adjusting computation can be potentially applied to mutation analysis. However, this requires a reimplementation of the interpreter in the language of self-adjusting computation [8], and remains future work to be explored.

3. APPROACH

3.1 Background: System Call fork()

Our approach utilizes on the highly efficient system call `fork` for forking the executions. Procedure `fork()` is a system call in POSIX systems, and is also implemented on other operating systems such as Windows (by Cygwin[1]). When `fork()` is called, a new child process is created, which has exactly the same virtual memory and call stack as the parent process. The main difference between the child process and the parent process is that `fork()` returns 0 in the child process and returns the process ID of the child process in the parent process, allowing us to distinguish whether we are executing the child or the parent.

The POSIX `fork()` is implemented on a mechanism called copy-on-write [2]. When `fork()` is invoked, the system creates the necessary structures for the new process and assigns a new process ID, but the virtual memory is not copied and shared between the parent process and the child process. Whenever a process subsequently writes on a memory page, the system copies the page to a new area and writes to the new page, and updates the page table of the respective process. The copy-on-write mechanism is integrated into the routine of virtual memory access. As a result, the `fork()` returns very quickly, and the subsequent execution of the forked process is almost as fast as a standard process.

3.2 Overview of Dynamic Mutation Analysis

We first describe the redundant execution in standard mutation analysis that dynamic mutation analysis can avoid. Figure 1(a), (b) and (c) describe the execution of three mutants. The circles represent system states and the states with the same number are the same. The arrows represent the transitions of states after the executions of statements. At state 2, the three mutants execute three different statements, and lead to either state 3 or state 5. Though mutant 1 and mutant 3 executes different statements, the two statements lead to the same system state. This can happen, for example, by evaluating `a+=2` and `a*=2` when `a==2`. At state 3, again mutant 1 and mutant 3 execute different statements, and this time the two statements lead to different states. As a result, state 1, state 2, and the transitions leading to them are completely the same among the three mutants, forming redundant computations. Also, state 5 and the transitions leading to it are the completely same among mutant 1 and mutant 3, also forming a redundant computation. The redundant states are denoted as solid circles or grey circles based on their degrees of redundancy.

Since the three eventually lead to different states, the redundancy happens only in part of the computation and cannot be removed by static approaches. Dynamic mutation analysis removes this kind of redundancy by dynamically forking the execution. In dynamic mutation analysis, we start with one main process that represents all mutants, as

[1]`https://www.cygwin.com/faq.html#faq.api.fork`

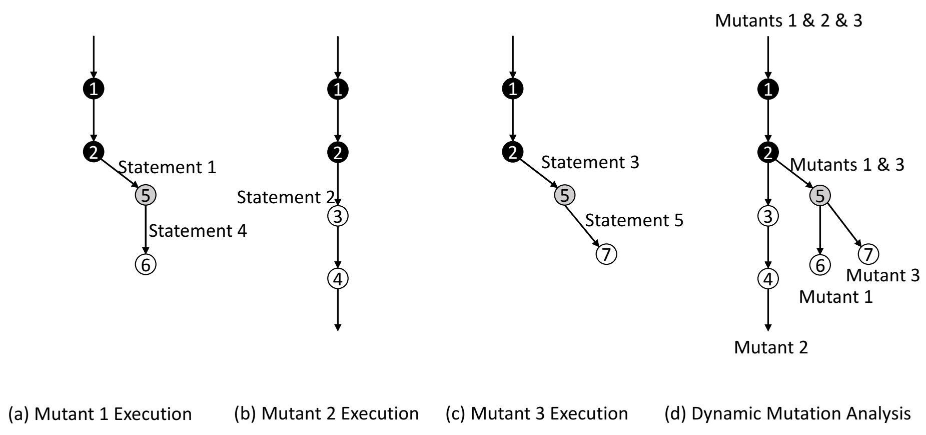

Figure 1: Dynamic Mutation Analysis Example

shown in Figure 1(d). At each state, we check the statements that each represented mutant is going to execute. If the statements are the same, we execute the statement normally. If the statements are different, we analyze their different executions and fork new processes if necessary. At state 2, the three mutants have three different statements to execute, so we need to analyze the executions of the statements.

To perform the analysis, we first 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. Two mutants are in the same equivalent class if their changes lead to the same new system state. In the example, mutant 1 and mutant 3 are clustered into one equivalence class and mutant 2 is in another class.

If number of equivalence class is more than one, say n, we fork $n-1$ 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. In the example, we fork a new process for the equivalent class of mutants 1 and 3, and apply the change from the equivalent class to change the state to state 5. In the original process, we apply the change of the other equivalent class to change the state to state 3, and the original process now represents only mutant 2. Then the two processes continue their execution independently. Again, at state 5, the newly forked process detects that the two mutants need to execute different statements and the two statements lead to different system states, so this process again forked into two processes, each representing one mutant.

From Figure 1(d) we can see that, the transitions leading to states 1, 2, and 5 are only performed once, avoiding the aforementioned redundant computation in standard mutation analysis.

3.3 An Abstract Model to Support Dynamic Mutation Analysis

To support different mutation analyses with different sets of mutation operators, especially to support both first-order and high-order mutation analysis [9], we define an abstract model of the infrastructure to support dynamic mutation analysis. If we need to implement dynamic mutation analysis on a different set of mutation operators, we only need to implement this abstract model.

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 expression level, on statement level, or on block of code level. We use an abstract concept—location—to represent the unit that a mutation operator applies. Also we consider 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.

For example, let consider the following program with two lines of code.

```
1: a = a + 1;
2: b = b + 1;
```

Suppose we have only one mutation operator: replace the basic arithmetic operator (i.e., + - *) in a statement with another basic arithmetic operator. Since this operator applies on statement level, the above program has two locations: line 1 and line 2. At line 1, this operator produces three variants: (a) `a = a + 1`, (b) `a = a - 1`, and (c) `a = a * 1`, where (a) is the original statement while (b) and (c) are mutated statements. Similarly, three variants are produced at line 2: (e) `b = b + 1`, (f) `b = b - 1`, and (g) `b = b * 1`. The four mutated statements at lines 1 and 2 lead to four mutants, with ID 1-4. Then we have (a).I = {3, 4}, (b).I = {1}, (c).I = {2}, (d).I = {1, 2}, (e).I = {3}, (f).I = {4}.

This model also supports high-order mutants. If a high-

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

Algorithm 1: Static Mutation Analysis

order mutant is created by mutating two statements in a program, the two mutants are both enabled for the mutant, and at other locations the original statements are enabled.

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 $\phi(s) = \perp$. 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) update the system state in s 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 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. Note that this execution model implies that, when there is jump statement in any mutant, it only jumps within its own location or to the start of another location, but not the middle of another location. If such a jump statement exists, we need to further divide this location into two locations.

To implement dynamic mutation analysis efficiently, we also need the implementations of the two additional operations, as follows.

- 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 $\{v \mid v \in V \land v.I \cap I \neq \emptyset\}$. 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 $\{i \mid i \in I \land \exists v \in V. i \in v.I\}$. The variants are assumed to be at the same location.

3.4 Static Mutation Analysis

Based on the abstract model, we can build an algorithm for traditional static mutation analysis, as shown in Algorithm 1. Given all mutant IDs in the system, the algorithm execute 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 [30], where all mutants are generated into the program to save compilation cost.

```
Input: $p$: a mutation procedure
Data: $s$: the current system state
Data: $I$: a set of mutant IDs represented by the
      current process
1 $I \leftarrow$ all mutant IDs
2 $s \leftarrow$ the initial system state
3 while $\phi(s) \neq \perp$ do
4     proceed($p(\phi(s))$)
5 end
6 for $i \in I$ do
7     save($s, i$)
8 end
```

Algorithm 2: Main Loop of Dynamic Mutation Analysis

3.5 Dynamic Mutation Analysis

Different from static mutation analysis, dynamic mutation analysis starts with a process representing all mutants, and fork the execution when a mutant leads to a different system state. Algorithm 2 shows the main loop of dynamic mutation analysis. There are three differences from static mutation analysis: (1) initially there is a set I containing all mutation 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.

The core part is the algorithm of proceed(), as shown in Algorithm 3. Let us first ignore line 21 in the algorithm. First, if there is only variant to execute, we directly execute the variant and return (lines 2-6). Otherwise, we first try to execute the variants and cluster the changes to the system state in equivalence classes (lines 7-11). Changes in each equivalence class will produce exactly the same new system state when applied to the current system state. If there are more than one equivalence classes, we fork a new process for each extra equivalence class (lines 13-15). When we fork a new process, the new process represents the mutant IDs of the variants in the equivalence class (line 17), and we apply the change to the system state (lines 18-19). Finally, we update the mutant IDs and system state for the current process (lines 24-27).

If there are too many mutants, we may fork a large number of processes. Too many processes may lead to a large overhead in scheduling these processes, and thus we would like to limit the number of parallel processes. To simplify the parallelism management, in this paper we limit the number of parallel processes to one. In other words, the parent process should wait for the child process to terminate once a child process is forked. On the other hand, if we would like have multiple parallel processes to fully utilize modern MIMD machines (e.g., multi-core / many-core systems), we can parallelize the test executions, starting one process for each test.

To implement this limit on processes, we use an invocation to waitpid() at line 21 to wait for the child process right after fork(). The invocation waitpid() will suspend the parent process until the child process terminates. However, a child process may not always terminate, and this may cause the parent process to wait forever. To prevent such cases, we

```
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      $v \leftarrow$ the only variant in $V$
4      execute (v.code, s)
5      return
6  end
7  $X = \emptyset$
8  for each $v$ in $V$ do
9      $X \leftarrow X \cup \{\texttt{try}(v.code, s)\}$
10 end
11 $\mathbb{X} \leftarrow$ group changes in $X$ into equivalent classes
12 $X_{cur} \leftarrow$ any one of the equivalent classes in $\mathbb{X}$
13 for each equivalence class $X$ in $\mathbb{X} - \{X_{cur}\}$ do
14     $V \leftarrow$ the variants corresponding to changes in $X$
15     $pid \leftarrow$ fork()
16     if $pid = 0$ then
           // Child process
17         filter_mutants($I, V$)
18         $x \leftarrow$ a random change in $X$
19         apply($x, s$)
20     else
           // Parent process
21         waitpid($pid$)
22     end
23 end
24 $V \leftarrow$ the variants corresponding to changes in $X_{cur}$
25 filter_mutants($I, V$)
26 $x \leftarrow$ a random change in $X_{cur}$
27 apply($x, s$)
```
Algorithm 3: Algorithm of proceed(s)

set up a timer that terminates a process when it consumes a certain amount of CPU time.

3.6 Discussion

3.6.1 *Equivalence between Static and Dynamic Mutation Analyses*

THEOREM 1. *Algorithm 2 invokes save(s, i) if and only if Algorithm 1 invokes save(s, i) with exact the same parameters.*

PROOF SKETCH. This can be proved by showing that exactly the same sequence of state transitions occurs for both static and dynamic mutation analyses. At each location, we can see that dynamic and static mutation analyses select exactly the same variant to execute. If `try`, `apply`, and change clustering are implemented correctly, the execution should lead to exactly the same state. □

3.6.2 *Comparison with Existing Approaches*

Now we discuss how our approach subsumes two existing reduction techniques for redundant computations. The first one is mutant schemata [30]. As mentioned in related work, mutant schemata generates one meta program that contains all mutants to remove redundancy in compilation. When we execute a mutant, we pass the mutant ID to the meta program, and the program dynamically selects code blocks to execute based on the mutant ID at each location. Our algorithm proceeds the same way as mutant schemata. Function p can be considered as the meta program, and we dynamically selects code blocks from $p(l)$ to execute at each location l. If we implement our algorithm by instrumentation (c.f. Section 4.2), the instrumented program only needs to be compiled once.

The second one is Just et al.'s approach [13]. Given a test, this approach identifies three types of mutants that do not need to be executed for the test: (1) the mutants that are not covered by the test, since these mutants cannot be killed by the tests; (2) the mutants that behave exactly the same as the original program when executed under the test; (3) groups of mutants that behave the same under the test but different from the original program when executed under the test, where only one mutant needs to be executed for each group. The three types of mutants are identified by a prepass that executes the original program. The first type is proposed by existing work [28, 14] and the latter two types are proposed by Just et al.

By forking process only at new system states, dynamic mutation analysis unifies the three types of reductions. For the first one, since all uncovered mutants have the same behavior as the original program, they will be represented by one process and be executed only once. Note that in Just et al.'s approach there is also a prepass that executes the original program. For the second type, the mutants behaving the same as the original program will also be represented by the process. For the third type, a group of mutants that behave the same will be represented by one process in our approach.

3.6.3 *Imprecision in Just et al.'s Approach [13]*

Just et al.'s approach obtains the three types of mutants by executing a prepass on the original program. At each location visited in the execution, their approach tries to execute all variants and compare the changes to the system state. If no mutated statement of a mutant is covered, this mutant belongs to the first type. If all mutated statements of a mutant produce the same changes as the original program, the mutant belongs to the second type. If two mutants both have mutated statements at a location that lead to the same state, and the state is different from the next state of the original program, the two mutants are put into one group.

By comparing the mutant executions in our approach, we can identify that their classification of third type is unsound. In our approach, when we fork a process for multiple mutants, the process may be further forked, as shown in 1(d). However, Just et al.'s approach classifies two mutants into a group only by the first time their execution deviates from the original program, and will erroneously classify mutants 1 and 3 into a group and execute only one of them. For a more concrete example, let us consider the following program.

```
1: a = 2;
2: for (int i = 0; i < 2; i += 1) {
3:   a = a / 2;
4:   a = a * 2; }
5: assert(a <= 20);
```

Let us assume two mutants are generated by changing line 3 into `a=a*2`, `a=a+2`, respectively. The first executions of line 3 in the two mutants are the same, and thus the two mutants will be classified into one group by Just et al.'s approach. However, the second executions of line 3 differ in the two mutants. Eventually one leads to the violation of the assertion at line 5, and the other one passes the assertion.

As a result, Just et al.'s approach turns out to be a lossy approach rather than the previously believed lossless approach. While a lossy approach is still useful in applications like mutation testing, it cannot be used in applications that require precise results, such as bug fixing and software product line testing.

To make Just et al.'s approach lossless, we have to apply the prepass also for each equivalence group. If any new equivalence group is discovered during the new prepass, we need to further analyze the new equivalence group. As a result, the process becomes very similar to dynamic mutation analysis.

3.7 Implementing First-Order Mutation

To implement dynamic mutation analysis on a specific set of mutation operators, we need to implement the abstract model. In this section we consider how to implement standard first-order mutation operators. We define standard first-order mutation by the following requirements.

- The mutation operators are applied on statements.
- Each mutant contains only one mutated statement.
- The number of variants at each location is small, and can be considered to have a small upper bound u.
- The total number of mutants, m, may be very large, and is dependent on the size of the program.

The third requirement seems be restrictive for mutation operators work on the source level. If the user wrote a very long statement, there may be a large number of variants generated from this statement. However, in such cases we can always decompose this statement into a set of smaller statement by transforming the original statement into three-address form, and the number of mutants on each decomposed statement would be small.

Most parts of the implementation are direct. The most challenging part is how to efficiently represent a set of mutation IDs and implement the two operations that filter variants and mutation IDs: filter_variants, 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 $O(1)$. However, all the two 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 $O(1)$.

More concretely, we design three types of mutant ID sets.

- `VectorSet`: This type uses a bit vector to implement a set of mutant IDs. 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. This type is used to store the mutant IDS represented by the initial process.
- `ListSet`: This type uses a list to implement a set. Each element in the list is considered to be contained in the set. To ensure the efficiency of the operations, the total number of mutant IDs that can be added in a ListSet is bounded by the constant u. This type is used to store the mutant IDs for mutated statements at each location, as well as the mutant IDs represented by some of the forked processes.
- `DefaultSet`: This type is just a placeholder but does not contain any value. At each location, each mutated statement is enabled in one mutant, and the original statement is enabled in the rest of the mutants. `DefaultSet` represents the rest of mutant IDs.

Both `VectorSet` and `ListSet` have the following member operations to manipulate the set: add, adding an ID to the set; remove, removing an ID from the set; size, returning the number of IDs; contains, testing whether an ID is in the set. Since the size of `ListSet` is bounded by u, it is easy to see that all operations can be implemented in the complexity of $O(1)$. `ListSet` also has a member operation $first$ that returns the first element in the list. Furthermore, the cost to initialize a `VectorSet` is $O(m)$, where m is the total number of mutants, but the cost of initialize the two other types of sets are both $O(1)$.

The algorithm for implement filter_variants is shown in Algorithm 4. Since the input variants are always from the same location, there is one variant with `DefaultSet` and the other variants are using `ListSet`. We need to avoid direct computation with `DefaultSet` because it actually does not contain any value. We first filter the variants with `ListSet` (lines 3-6). Since all operations have the complexity of $O(1)$ and the loop is bounded by u, this part has the complexity of $O(1)$. Note that each variant with `ListSet` has exactly one mutant ID, so we directly use *first* to obtain the ID in line 4. Next, we consider whether the variant with `DefaultSet` should be filtered out or be kept (lines 8-10). Since each variant with `ListSet` is enabled for one mutant, if the currently selected variants is fewer than the mutants represented by the current process, there must be left mutants and the default 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 $O(1)$.

```
   Input: $V$: a set of variants to be filtered
   Input: $I$: a set of mutation IDs used to filter $V$
1  $V' \leftarrow \emptyset$
2  $v' \leftarrow$ the variant in $V$ with DefaultSet
3  for each $v \in V$ where $v \neq v'$ do
4      if $I.contains(v.I.first)$ then
5          $V' \leftarrow V' \cup \{v\}$
6      end
7  end
8  if $V'.size < I.size$ then
9      $V' \leftarrow V' \cup \{v'\}$
10 end
11 $V \leftarrow V'$
```

Algorithm 4: filter_variants

The algorithm of filter_mutants is shown in Algorithm 5. Similar to the previous algorithm, the core idea is to avoid

```
   Input: $I$: a set of mutation IDs to be filtered
   Input: $V$: a set of variants used to filter $I$
   Input: $L$: the current location
 1 if $V$ contains the variant with DefaultSet then
 2     for each $v \in p(L) - V$ do
 3         $I.remove(v.I.first)$
 4     end
 5 else
 6     $I \leftarrow$ a new empty ListSet
 7     for each $v \in V$ do
 8         $I.add(v.I.first)$
 9     end
10 end
```

Algorithm 5: filter_mutants

direct computation with `DefaultSet`. Therefore, if there is `DefaultSet` 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 are bound by u, the whole algorithm takes $O(1)$.

4. EVALUATION

Our evaluation aims to answer the following research question: how much does dynamic mutation analysis boost the performance of mutation analysis?

4.1 Performance Analysis

First we theoretically analyze the performance of dynamic mutation analysis. Dynamic mutation analysis saves some redundant computations by perform some extra runtime analysis. We analyze the two aspects respectively.

Runtime Overheads. The runtime overheads mainly come from the procedure proceed(s), which replaces the direct execution of code blocks in standard mutation analysis. Besides executing code blocks, proceed(s) mainly performed three tasks: (1) select variants to execute based on the current mutant ID, (2) clustering the changes, (3) fork new processes. In previous sections we have seen that the operations and loops performed for the three tasks are either $O(1)$ time or be bounded by constant u, so the extra runtime overheads take constant time.

The first two tasks are performed also in existing acceleration approaches. The first task is performed in mutant schemata [30]. The second task is performed in Just et al.'s approach [13]. Though Just et al.'s approach performs this step in a prepass and our approach performs the analysis dynamically, the amount of computation is similar. The studies of existing approaches [13, 30] have found that the extra overheads in both techniques are significantly outweighed by the benefits. Note that Just et al.s approach is lossy, and thus the benefit for a lossless analysis should be smaller than the reported benefit [13], but the difference should be small as conditions for triggering the imprecision is strong.

The only extra task over the state-of-the-art techniques is process forking, and this cost is very small by utilizing the POSIX system call fork(). Also, fork() performs at most m times, where m is the number of mutants. Thus, the performance of fork() is not very critical compared with, for example, the first task that is performed at each invocation to proceed().

Reduced Computation. Here we consider only the reduction over state-of-the-art Just et al.'s approach [13]. For each mutant, the execution before the mutated statement is shared, so expected reduction in computation is the expected position of the first state deviation caused by the mutated statement in a test execution. Now let us assume that each statement takes a constant time to execute, and the mutated statements are evenly distributed among the program. Then the expected position of the mutated statement in a program is 50%. If the program does not contain any branch or loop, the expected computation reduction is also 50%.

However, the above assumptions usually do not hold in practice. If the program has a small, time-consuming loop near the beginning of the execution, the reduction rate is likely to be higher than 50%. On the other hand, if such a loop is near the end of the execution, the reduction is likely to be lower than 50%. However, it is difficult to get a precise value from theoretical analysis, so we evaluate the actual performance boost by experiments in Section 4.3.

Table 1: Mutation Operators

Name	Description	Example
AOR	Replace arithmetic operator	`a+b` $\rightarrow$ `a-b`
ROR	Replace relational operator	`a==b` $\rightarrow$ `a>=b`
LVR	Replace literal value	$0 \rightarrow 1$

4.2 Implementation

We have implemented our approach for C mutation analysis on top of LLVM, a widely-used compiler framework. LLVM first translates C programs into Intermediate Representation (IR) code, and then compiles the IR code into executables. We also design the mutation operator at the IR level, as mutating at IR is noticeably simpler than mutating at the source level.

Table 1 describe the mutation operators used in our implementation. The mutation operators are designed by migrating the operators used in the Major framework [14]. The number of operators in our implementation is smaller because of the following two reasons. (1) Some operators are designed for mutating Java source code, and are not needed at IR level. For example, in Major framework there is an operator COR that changes conditional operators, but at IR level it is equal to ROR because Boolean variables are compiled into integers. (2) Some operators change code elements that do not appear in our experiment subjects. For example, in Major framework there is an operator LOR that replaces bit operators, but we observe almost no bit operators in our evaluation subject.

To enhance performance, our implementation uses a prepass to instrument our algorithm into the target program. More concretely, our approach first applies mutation operators to the IR program to generate variants, and for each variant, we generate three versions of code: the original code block (i.e., $c = a + b$), the trial execution version (i.e., $c1 = a + b$), and the change application version (i.e., $c = c1$). At each location in the target program, the proceed() is first specialized using the three versions of variants, and then specialized version of proceed() is used to replace the original code block at the position.

For comparison, we also implemented two reference techniques: the approach by Just et al. [13] and plain mutation

Table 2: Subject Programs

Name	Lines of Code	Tests	Mutants
grep	10068	339	953
print_tokens	726	4130	390
replace	564	5542	864
schedule	412	2650	137
tcas	173	1608	369
Total	11943	13729	2713

analysis with only mutant schemata [30]. Just et al.'s approach was originally implemented in Java using Major [14]. We took their code and migrated the code to C using LLVM. Since their code already contains mutant schemata, we implement the plain mutation analysis by simply removing the reduction component.

Our implementations and all experimental data are available as an opens source project[2].

4.3 Experiments

4.3.1 Subjects

To answer this research question, we collected 5 subjects from the SIR repository (software artifact infrastructure repository[3]). SIR is widely used in existing studies in evaluating mutation analysis techniques [34, 29, 24]. The statistics of the subjects are shown in Table 2. These subjects are selected because (1) their sizes or the numbers of tests are not very large, allowing us to finish the second reference technique, the plain mutation analysis, with a reasonable amount of time; (2) the subjects are from different domains. In particular, grep is Unix utility for searching text, schedule is a priority scheduler, print_tokens is a small lexical analyzer, tcas is an aircraft collision avoidance system, and replace is a pattern matching and substitution tool. Furthermore, the sizes of the subjects are comparable to those used in existing studies [34, 29, 24]. These subjects come with in total 13729 tests, and our mutation operators generated in total 2713 mutants on the subjects.

4.3.2 Procedures

For each subjects, we used three techniques to perform the mutation analysis: our technique, Just et al.'s approach [13], and a plain mutation analysis with only mutant schemata [30]. Note that mutant schemata is also adopted in the implementation of Just et al.'s approach, and our approach effectively subsumes mutant schemata and the reductions in Just et al.'s approach, so the comparison shows the extra benefits brought by each approach. The mutation analysis determines, for each pair of mutant and test, whether the test kills the mutant.

We recorded the analysis time per each subject and each technique. In our experiment, we sequentially executed the tests but not parallelized them, so as to obtain more stable results. Since there was at most one forked process running at a time, the executions of mutants were also sequential.

Since some mutants may execute forever, we also need to set the timeout value for killing a mutant if it takes more CPU time than the timeout value. To find a proper timeout value, we randomly sampled 300 tests from each mutant, ran the sampled tests, and recorded their CPU time. Then we set the timeout value as five times of the average CPU time of a test execution. The final timeout value we used in the experiments was 5ms.

The experiments were carried out on an Ubuntu 14.10 laptop with Intel i7-4710MQ 2.5GHz CPU and 12G memory.

4.3.3 Results

The result of the experiments is shown in Table 3. From the table we make the following observations.

- On all subjects our approach constantly outperformed the other two techniques. This result is consistent with our theoretical analysis, where the reduced redundant computations are much more significant than the introduced runtime overhead.
- Our approach had significant speedups on most subjects over existing techniques. The average speedup over Just et al.'s approach was 1.929X and the maximum was 2.220X. The average speedup over mutation schemata was 4.775X and the maximum was 17.422X. Note that Just et al.'s approach can be lossy, while our approach is lossless.
- The speedup of our approach over Just et al.'s approach was small on subject tcas. We further investigated the code of tcas, and found that most tests in tcas are very small and return quickly. Since tcas is a command line tool, each test executes in a separate process. As a result, the time spent on creating and destroying processes was significantly larger than the time executing tests. Since our approach only accelerates test execution, we cannot achieve a high speedup.
- Just et al.'s approach significantly outperformed the plain mutation analysis. This result is consistent with that in the original paper [13] and indicates that our implementation of Just et al.'s approach is most likely consistent with theirs.

4.4 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. Furthermore, the similar speedups of Just et al.'s approach in our experiments and the original paper indicate that our implementation of Just et al.'s approach is likely to be consistent with theirs.

The main threat to external validity is the mutation operators we used in our experiments. Though we migrated mutation operators from a Java mutation analysis framework, some existing C mutation analysis tool [4] provides a much larger number of mutation operators. Using a large set of mutation operators are likely to give a more even distribution of mutated statements over the program, potentially reducing the variance of the result.

The main threat to construct validity is the way we measure performance may be imprecise. To reduce this threat, we perform only sequential but not parallel operations in the experiments, in order to get a more stable result.

5. EXTENSION: TESTING SOFTWARE PRODUCT LINES

[2]`https://github.com/wangbo15/accmut`

[3]available at `http://sir.unl.edu/portal/index.html`.

Table 3: Experimental Results

Subjects	DMA	Just	MS	DMA vs. Just	DMA vs. MS	Just vs. MS
grep	2m23.15s	2m46.06s	8m43.71s	1.160X	3.658X	3.154X
print_token	20m22.79s	45m14.68s	101m0.74s	2.220X	4.956X	2.233X
replace	11m59.59s	19m13.21s	55m17.08s	1.603X	4.610X	2.877X
schedule	7.38s	9.13s	23.31s	1.237X	3.159X	2.553X
tcas	5.38s	5.48s	1m33.59s	1.020X	17.422X	17.088X
Total/Average	34m58.28s	67m28.56s	166m58.42s	1.929X	4.775X	2.475X

DMA = Dynamic mutation analysis, Just = Just et al.'s approach [13], MS = Mutant schemata [30]
DMA vs. Just = Speedup of DMA over Just DMA vs. MS = Speedup of DMA over MS Just vs. MS = Speedup of Just over MS

```
1: #if A && !B
2:   x++;
3: #else
4:   x--;
5: #endif
```

(a) Example Variants

```
#if A || B
...
#endif
#if A
...
#endif
```

(b) SPLat Example

Figure 2: Software Product Line Examples

```
   Input: $V$: a set of variants to be filtered
   Input: $I$: a set of mutation IDs used to filter $V$
1  $V' \leftarrow \emptyset$
2  for each $v \in V$ do
3      if $satisfiable(I \wedge v.I)$ then
4          $V' \leftarrow V' \cup \{v\}$
5      end
6  end
7  $V \leftarrow V'$
```

Algorithm 6: filter_variants for software product lines

The abstract model also allows us to extend dynamic mutation analysis to software product line testing. In a typical software product line, there is a set of Boolean configuration options, and at some location of the program different variants may be selected based on the assignment to the features, such as the example in Figure 2(a). In this example, `A` and `B` are configuration options. Based on the value of `A` and `B`, either `x++` or `x--` is left in the code. We can assume that in both branches of `#if` statement there are only complete statements, since existing techniques can make this conversion if there are partial statements [15, 6].

Each assignment to the configuration options is known as a configuration. The problem of testing a product line is to run the tests on all configurations.

Software product line fits well into our model. Each configuration is a mutant ID. A `#if` statement defines a location and a set of variants. The mutants that a variant is enabled for are defined by the conditional expression in the corresponding if statement. In Figure 2(a), lines 1-5 together form a location, while line 2 and line 4 are two variants. Line 2 is enabled in configurations satisfying $\texttt{A} \wedge \neg\texttt{B}$ while line 4 is enabled in configurations satisfying $\neg(\texttt{A} \wedge \neg\texttt{B})$

The core part of the implementation is still how to represent sets of mutant IDs and implement the two operations, filter_variants, and filter_mutants. Here we represent a set of mutant ID directly by a logic expression, where the free variables in the expression are configuration options in the software product line. Any assignment that satisfies the logic expression is considered to be in the set.

The algorithm for operation filter_variants is shown in Algorithm 6. Here we rely on SAT solvers to check the satisfiability of a formula, represented by procedure $satisfiable$ in line 3. If the conjunction of I and $v.I$ is satisfiable, the intersection of the two sets is not empty, and v should be kept in the result.

Operation filter_mutants(I, V) is implemented by changing the logic expressions. This operation will update I to expression $I \wedge v_1.I \wedge \ldots \wedge v_n.I$, where $V = \{v_1, \ldots, v_n\}$.

Interesting, this extension actually subsumes SPLat [18], a state-of-art static reduction techniques for testing software product line. SPLat records the configuration options that are accessed during the test execution, and ignore the configurations that only differ on the unaccessed options. On the other hand, our approach forks a process only when a variant leads to a different system state, and would not fork any process for mutants that only differ in unaccessed options. Furthermore, besides the redundancy in part of a mutant execution, our approach also remove more entirely redundant mutant executions than SPLat. For example, let us consider the program in Figure 2(b). SPLat will execute configurations {A=1, B=1} and {A=1, B=0} as both options are accessed during test execution, but the two configurations lead to exactly the same program. On the other hand, our approach will use one process to represent the two configurations and execute them only once.

6. CONCLUSION AND LIMITATION

In this paper we propose dynamic mutation analysis, which analyzes the equivalence of executions at runtime and forks new process only at new system states. The experimental results suggest that dynamic mutation analysis can achieve significant speedup over existing static reduction approaches. Furthermore, we propose an abstract model for dynamic mutation analysis that views mutants as variants in different locations. This model allows us to extend the approach to software product line testing. By comparing with our algorithm, we also identify that a previously believed lossless approach is actually lossy.

One limitation of current implementation is that it cannot well handle external resources, such as database connections, files, etc. This is because the copy-on-write mechanism of fork() only work for memory objects but not external resources. To deal with this problem, we need to implement the copy-on-write mechanism also for external resources. Nevertheless, this limitation may not be a serious one as well-written tests often use mock objects rather than directly accessing external objects.

7. REFERENCES

[1] U. A. Acar. Self-adjusting computation:(an overview). In *PEPM*, pages 1–6, 2009.

[2] R. E. Bryant, O. David Richard, and O. David Richard. *Computer systems: a programmer's perspective*. Prentice Hall Upper Saddle River, 2003.

[3] D. Conners and W.-m. W. Hwu. Compiler-directed dynamic computation reuse: rationale and initial results. In *MICRO*, pages 158–169, 1999.

[4] M. E. Delamaro, J. C. Maldonado, and A. Mathur. Proteum-a tool for the assessment of test adequacy for C programs user's guide. In *PCS*, pages 79–95, 1996.

[5] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[6] P. Gazzillo and R. Grimm. Superc: Parsing all of c by taming the preprocessor. In *PLDI*, pages 323–334, 2012.

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

[8] M. A. Hammer, U. A. Acar, and Y. Chen. Ceal: a c-based language for self-adjusting computation. In *PLDI*, pages 25–37. ACM, 2009.

[9] M. Harman, Y. Jia, and W. B. Langdon. Strong higher order mutation-based test data generation. In *Proc. FSE*, pages 212–222, 2011.

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

[11] C. Ji, Z. Chen, B. Xu, and Z. Zhao. A novel method of mutation clustering based on domain analysis. In *SEKE*, pages 422–425, 2009.

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

[13] R. Just, M. D. Ernst, and G. Fraser. Efficient mutation analysis by propagating and partitioning infected execution states. In *ISSTA*, pages 315–326. ACM, 2014.

[14] R. Just, F. Schweiggert, and G. M. Kapfhammer. Major: An efficient and extensible tool for mutation analysis in a java compiler. In *ASE*, pages 612–615, 2011.

[15] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *OOPSLA*, pages 805–824, 2011.

[16] C. Kästner, A. von Rhein, S. Erdweg, J. Pusch, S. Apel, T. Rendel, and K. Ostermann. Toward variability-aware testing. In *FOSD*, pages 1–8, 2012.

[17] C. H. P. Kim, S. Khurshid, and D. Batory. Shared execution for efficiently testing product lines. In *ISSRE*, pages 221–230. IEEE, 2012.

[18] C. H. P. Kim, D. Marinov, S. Khurshid, D. Batory, S. Souto, P. Barros, and M. d'Amorim. Splat: Lightweight dynamic analysis for reducing combinatorics in testing configurable systems. In *FSE*, pages 257–267, 2013.

[19] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *ICSE '13*, pages 802–811, 2013.

[20] E. W. Krauser, A. P. Mathur, and V. J. Rego. High performance software testing on simd machines. *IEEE Transactions on Software Engineering*, 17(5):403–423, 1991.

[21] J. Meinicke. Varexj: A variability-aware interpreter for java applications, 2014.

[22] S. Moon, Y. Kim, M. Kim, and S. Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *ICST*, pages 153–162, 2014.

[23] A. J. Offutt, R. P. Pargas, S. V. Fichter, and P. K. Khambekar. Mutation testing of software using a mimd computer. In *Proc. ICPP*, 1992.

[24] M. Papadakis and Y. Le Traon. Using mutants to locate" unknown" faults. In *ICST*, pages 691–700, 2012.

[25] Y. Qi, X. Mao, and Y. Lei. Making automatic repair for large-scale programs more efficient using weak recompilation. In *ICSM*, pages 254–263. IEEE, 2012.

[26] Y. Qi, X. Mao, and Y. Lei. Efficient automated program repair through fault-recorded testing prioritization. In *ICSM*, pages 180–189. IEEE, 2013.

[27] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 254–265, 2014.

[28] D. Schuler and A. Zeller. Javalanche: efficient mutation testing for java. In *ESEC/FSE*, pages 297–298, 2009.

[29] A. Siami Namin, J. H. Andrews, and D. J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Proc. ICSE*, pages 351–360, 2008.

[30] R. H. Untch, A. J. Offutt, and M. J. Harrold. Mutation analysis using mutant schemata. In *Proc. ISSTA*, pages 139–148, 1993.

[31] W. Weimer, Z. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 356–366, 2013.

[32] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ICSE '09*, pages 364–374, 2009.

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

[34] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei. Is operator-based mutant selection superior to random mutant selection? In *Proc. ICSE*, pages 435–444, 2010.

[35] L. Zhang, D. Marinov, and S. Khurshid. Faster mutation testing inspired by test prioritization and reduction. In *Proc. ISSTA*, pages 235–245, 2013.

[36] L. Zhang, L. Zhang, and S. Khurshid. Injecting mechanical faults to localize developer faults for evolving software. In *Proc. OOPSLA*, pages 765–784, 2013.