Efficient Mutation Analysis by Propagating and Partitioning Infected Execution States

René Just, Michael D. Ernst
Computer Science and Engineering
University of Washington
Seattle, WA, USA
{rjust, mernst}@cs.washington.edu

Gordon Fraser
Department of Computer Science
University of Sheffield
Sheffield, UK
gordon.fraser@sheffield.ac.uk

ABSTRACT

Mutation analysis evaluates a testing technique by measuring how well it detects seeded faults (mutants). Mutation analysis is hampered by inherent scalability problems — a test suite is executed for each of a large number of mutants. Despite numerous optimizations presented in the literature, this scalability issue remains, and this is one of the reasons why mutation analysis is hardly used in practice.

Whereas most previous optimizations attempted to statically reduce the number of executions or their computational overhead, this paper exploits information available only at run time to further reduce the number of executions.

First, *state infection conditions* can reveal — with a single test execution of the unmutated program — which mutants would lead to a different state, thus avoiding unnecessary test executions. Second, determining whether an infected execution state *propagates* can further reduce the number of executions. Mutants that are embedded in compound expressions may infect the state locally without affecting the outcome of the compound expression. Third, those mutants that do infect the state can be *partitioned* based on the resulting infected state — if two mutants lead to the same infected state, only one needs to be executed as the result of the other can be inferred.

We have implemented these optimizations in the Major mutation framework and empirically evaluated them on 14 open source programs. The optimizations reduced the mutation analysis time by 40% on average.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Experimentation

Keywords

Mutation analysis, software testing, dynamic analysis

ISSTA'14, July 21–25, 2014, San Jose, CA, USA
Copyright 2014 ACM 978-1-4503-2645-2/14/07...$15.00
http://dx.doi.org/10.1145/2610384.2610388

1. INTRODUCTION

Mutation analysis is a powerful approach to assess the quality of a test suite and to benchmark testing or debugging techniques. A test suite or testing technique is assumed to be good if it can distinguish a program from many small syntactic variations of this program (*mutants*). The quality of the tests is quantified in the *mutation score*, i.e., the percentage of detected mutants.

Although mutation analysis is well-established in software engineering research, it is not yet widely accepted in practice — one reason is poor scalability. Computing the mutation score for a test suite requires determining, for every mutant, whether the test suite succeeds or fails when run on the mutant. In the worst case each test must be run on each mutant. There are multiple mutants per expression in the program, so even modest programs can lead to significant numbers of mutants.

For a test to fail on a mutant the test needs to reach the mutated code location (reachability), achieve a different execution state than the original program (infection), and this state difference needs to propagate to an observable output (propagation). If any of these conditions does not hold, the test cannot detect the mutant, and hence executing the test on that mutant is unnecessary and wastes CPU cycles. Even if all of these conditions hold, the test's outcome for a certain mutant may be predictable. Different mutants may exhibit identical behavior for the same test, for example if their effects propagate in the same way to an observable output. To make mutation analysis practically relevant, it is desirable to reduce the overall computational costs by avoiding any unnecessary executions.

Researchers have devised several approaches to reducing the number of test executions in mutation analysis. Two approaches to reduce the overall number of mutants, which incur a loss of information, are sampling subsets of mutants or restricting the set of operators that creates the mutants [30]. A lossless approach is to refine mutation operators to avoid those that result in redundant (i.e., semantically identical) mutants [20]. A dynamic approach is to gather information at run time. A lossless example is to perform a prepass to measure mutation coverage — that is, determining which mutations[1] would be reached and executed by which test. Given the mutation coverage information, a test is not executed on a mutant if it does not cover it [11, 21, 28].

This paper takes the dynamic prepass approach a step further and observes the *execution state* during the initial

[1] A *mutation* is the syntactic change within a mutant, which was produced by a mutation operator.

execution of a test suite using an instrumented version of the unmutated program. This enables 3 new optimizations. (1) The program state after the execution of the mutation is said to be infected if it differs from the program state of the original program. If executing a test on a mutant does not lead to an *infected execution state* then executing this test cannot possibly lead to detection of the mutant — the mutant is said to be *test-equivalent* for this test. (2) Even if a test causes a local state infection, the mutant may still be test-equivalent: the local state change may not be observable as a test failure if the mutation is part of a compound expression whose state is not infected. It is therefore beneficial to determine whether the infected state *propagates* to the enclosing expression. (3) If a test leads to an identical infected state on two different mutants, then it is not necessary to execute the test against both mutants as they will lead to the same result. In fact, it is sufficient to execute each test against only one representative of a set of mutants resulting in the same infected state. In our experiments on 14 open source projects, these 3 infection and propagation optimizations reduced the mutation analysis run time by 40% on average compared to the standard mutation coverage (reachability) optimization.

The contributions of this paper are:

- A method to detect test-equivalent mutants by monitoring infected execution states during execution on the unmutated program.
- A method to detect test-equivalent mutants by monitoring propagation of infected execution states in compound expressions during execution on the unmutated program.
- A method to reduce the number of test executions by partitioning mutants with identically infected state.
- An empirical evaluation with 14 open source programs.

The paper is structured as follows. Section 2 describes mutation analysis, and Section 3 introduces the optimization techniques monitoring, propagating, and partitioning state infection. Section 4 describes the implementation of these techniques in the Major mutation framework. Section 5 presents the empirical study. Section 6 discusses related work, and Section 7 concludes the paper and proposes future work.

2. MUTATION ANALYSIS

Mutation analysis systematically injects artificial faults (mutants) into a program under test in order to evaluate a test suite's ability to detect the mutants. The faults to inject are described by mutation operators, which represent a program transformation. These mutation operators are systematically applied to the entire program — in contrast to classical error seeding, where the error injection is led by the intuition of an experienced developer. Each mutation operator thus results in a set of mutants, each differing from the original program by one syntactic change. The restriction to mutants consisting of only one change compared to the original version is justified by the coupling effect [22], which expresses the observation that test suites that are good at detecting small changes in a program are usually also good at detecting larger changes.

The typical application of mutation analysis is to quantify the quality of a test suite in terms of the *mutation score*, which is the percentage of mutants detected by the test suite. A higher mutation score is treated as indicating a better test suite. To determine the mutation score, in principle one needs to execute the entire test suite on every single mutant, as a mutant is only detected (*killed*) if some test fails on it.

2.1 Equivalent Mutants

Mutation analysis suffers from a well-known and much-dreaded problem: although all mutants are syntactically different, they are not necessarily semantically different. An *equivalent* mutant is syntactically different from the original program but semantically identical, i.e., there exists no test that can detect it. Any execution of a test on an equivalent mutant is a waste of CPU time, when trying to calculate a mutation score. An equivalent mutant (and more generally any unkilled mutant) requires a disproportionate amount of time during mutation analysis since every test must be run on it. By contrast, once a mutant is killed, there is no need to execute any further tests on it.

2.2 Test-Equivalent Mutants

A mutant is *test-equivalent for test t*, or is *t-equivalent*, if test t cannot detect the mutant. An equivalent mutant cannot be detected by any test, so it is a test-equivalent mutant for every possible test. A simple example of a test-equivalent mutant is a mutant that is not executed by the corresponding test — a test usually covers only certain parts of a program, and hence only covers (i.e., reaches and executes) mutations applied in these parts. Detecting equivalent mutants is an undecidable problem but the set of test-equivalent mutants can be soundly approximated [2, 3, 12, 15, 22, 23, 27], and that is our general approach as well.

3. PROPAGATING AND PARTITIONING INFECTED EXECUTION STATES

It is our goal to avoid executing a test t for as many t-equivalent mutants as possible. A test that detects a mutant has to fulfill the following three conditions (cf. [32]):

1. The mutation has to be *executed* — that is, the mutated code has to be covered.
2. The execution of the mutation has to *infect* the state — that is, the mutated expression computes a different value than the unmutated expression.
3. The infected state has to *propagate* — that is, other expressions that depend on the mutated one must also compute a different value, eventually including an expression that affects the test oracle.

Existing mutation coverage approaches [21, 28] exploit the first condition (execution). If a test does not execute a certain mutation, then it cannot kill the mutant and it need not be executed on that mutant. This reduces the run-time cost of mutation analysis. We take this approach a step further and suggest three new optimizations that also exploit the latter two conditions to avoid unnecessary test executions and to reduce the run-time cost of mutation analysis.

Our first optimization (Section 3.2) exploits the second condition (infection). In a prepass, it runs the test suite once on an instrumented version of the unmutated program and performs a dynamic analysis. It determines, for each test and each mutation, whether the mutated expression *would* compute a different value when executed by the same test. We refer to the different value as *infected execution state*. Infection is a stronger criterion than coverage for a mutant to be detectable. Hence, the set of mutants with infected execution state I is a subset of covered mutants C: $I \subseteq C$.

Our second optimization (Section 3.3) exploits the third condition (propagation) and utilizes the same prepass as our first optimization. It computes, for each test, each mutation, and each expression that lexically encloses the mutation, whether the enclosing expression *would* compute a different value when executed by the same test. That is, it computes whether the execution state of the enclosing expression is infected due to the infected execution state of the mutation. Propagation is a stronger criterion than infection and therefore the set of mutants with a propagating infected execution state (P) is a subset of all mutants with an infected execution state: $P \subseteq I \subseteq C$. All mutants not in the set P are test-equivalent and do not have to be executed for the test during the mutation analysis. The larger the set difference $C \setminus P$, the greater the improvements of our first two optimizations, compared to mutation coverage.

Our third optimization (Section 3.4) exploits redundancies between mutations and uses the same prepass as the other optimizations. It determines, for each test, whether two mutated expressions *would* lead to an identically infected execution state — that is, whether they *would* evaluate to the same value. If a test leads to an identically infected state for two or more mutants, the test only needs to be executed on one of them during the mutation analysis because the outcome for the other mutants (i.e., detected or test-equivalent) is known to be the same.

3.1 Notation

This section provides notation and definitions used throughout the remainder of the paper.

General notation:

- Test suite T
- Test $t \in T$
- Expression $expr := VAR \mid LIT \mid \langle op, \overline{expr} \rangle$
- Expression operator op
- n-tuple ($n \geq 0$) of expressions $\overline{expr}$
- Expression value $value$
- Expression evaluation $[\![]\!] : expr \mapsto value$
- i-th evaluation of $expr$, executing t: $[\![expr]\!]_i^t = value_i$

An expression is a variable, literal, or n-ary operator with n expressions as arguments. Consider the example `a + b > c`, which contains 5 expressions:

$$\underbrace{\underbrace{\texttt{a}}_{expr_1} + \underbrace{\texttt{b}}_{expr_2}}_{expr_3} \texttt{ > } \underbrace{\texttt{c}}_{expr_4} \qquad \begin{aligned} expr_5 &= \langle op_5, expr_3, expr_4 \rangle \\ &= \langle op_5, \langle op_3, expr_1, expr_2 \rangle, expr_4 \rangle \\ &= \langle >, \langle +, a, b \rangle, c \rangle \end{aligned}$$

All expressions in the source code are enumerated, and distinct occurrences of an expression are treated independently.

3.2 Monitoring Infected Execution States

A test infects the execution state of a mutated expression if the value of that mutated expression differs from the value of the original expression for at least one evaluation. Note that a test might evaluate an expression several times during a single execution (e.g., an expression in a loop body).

DEFINITION 1. *Infected execution state.*
Let t be a test that evaluates an expression $expr_j$ N times. Additionally, $expr'_j$ denotes a mutated version of $expr_j$. Test t infects the execution state of $expr'_j$ at run time iff:

$$\exists i \in [1..N] : [\![expr_j]\!]_i^t \neq [\![expr'_j]\!]_i^t$$

If test t does not infect the execution state of a mutated expression, the corresponding mutant is t-equivalent. Consider the original expression `a + b` and the corresponding mutated expression `a - b`. A test infects the execution state of this mutation if there is at least one evaluation with `b` $\neq 0$. We refer to the process of determining which test infects the execution state of which mutation as *monitoring* infection.

Side Effects

Definition 1 holds in the presence of side effects if the same side effects occur in the original and mutated expression. If the mutation may change the side effects, our analysis conservatively assumes that every test may infect its execution state due to a changed side effect. We apply all the optimizations of this paper only if it is statically provable that the mutation does not change side effects for any test.

As an example, consider the following pairs of original and mutated expressions:

	Original	Mutated	Changes side effects?
1	`if (x > y)`	`if (x >= y)`	no
2	`if (x > y++)`	`if (x >= y++)`	no
3	`if (x > y++)`	`if (true)`	yes
4	`if (x > y)`	`if (x > y++)`	yes
5	`if (flag || foo())`	`if (flag)`	maybe

Statement-level mutation operators such as the statement deletion operator (e.g., removing a method call or an assignment) may also change side effects.

As an optimization, every call of a known pure method (i.e., a method that does not change the program state) is considered to be side-effect-free. Major currently does not perform a purity analysis [14] but rather applies a heuristic — hashCode, toString, and known getter methods are considered pure, and all other methods are considered impure.

3.3 Propagating Infected Execution States

Definition 1 (infected execution state) refers to a local change at the expression level. Such a local difference of the execution state might or might not affect the execution state of a lexically enclosing expression. We say that the infected execution state of an expression *propagates* to a lexically enclosing expression if it leads to an infected execution state of this enclosing expression.

As a motivating example, recall the expression $expr_5$ = `a + b > c`, and suppose its subexpression $expr_3$ = `a + b` is mutated to $expr'_3$ = `a - b`. Suppose that two tests t_1 and t_2 lead to the following results, when evaluating the original and mutated expressions:

Test	Original	Mutated	Infected?
t_1 := `a = 1, b = 2, c = 3`	`a + b` → 3	`a - b` → −1	**infected**
	`a + b > c` → 3 → **false**	`a - b > c` → −1 → **false**	t_1**-equivalent**
t_2 := `a = 3, b = 2, c = 1`	`a + b` → 5	`a - b` → 1	**infected**
	`a + b > c` → 5 → **true**	`a - b > c` → 1 → **false**	**infected**

Both tests, t_1 and t_2, infect the execution state of $expr'_3$, but only the infected state caused by t_2 propagates to $expr_5$. This means that the mutant of $expr'_3$ is t_1-equivalent even though $expr'_3$ fulfills Definition 1. We strengthen our analysis to require, additionally, that an infected execution state has to propagate to lexically enclosing expressions.

DEFINITION 2. *Propagation of infected execution states. Let t be a test that infects the execution state of a mutated expression $expr'_j$. The infected execution state of $expr'_j$ propagates to a lexically enclosing expression $expr_e$ iff:*

$$\exists i \in [1..N] :$$
$$[\![\langle op_e, ...\langle ..., expr_j ...\rangle ...\rangle]\!]^t_i \neq [\![\langle op_e, ...\langle ..., expr'_j ...\rangle ...\rangle]\!]^t_i$$

Note that $expr'_j$ is the mutated version of the j-th subexpression and the only mutated expression in the entire compound expression. This implies that the propagation of the infected execution state of $expr'_j$ does not need to be further monitored once any enclosing expression $expr_e$ violates this condition — any expression that encloses $expr_e$ can not possibly lead to an infected execution state.

3.4 Partitioning Infected Execution States

Suppose two mutants m_1 and m_2 are mutants of the same expression. A test execution on these two mutants leads to the same overall test result (i.e., both m_1 and m_2 are detected, or both m_1 and m_2 are test-equivalent) if the corresponding mutated expressions evaluate to the same value. To perform optimization based on this observation, we partition the set of mutated expressions M_{expr_j} based on their expression values. For each test t, two mutated expressions are elements of the same partition cell p if they always evaluate to the same expression value during the execution of t.

DEFINITION 3. *Partitioning of infected execution states. Let M_{expr_j} be a set of mutated expressions whose execution states are infected by a test t. Partition P of M_{expr_j} satisfies the following properties:*

$$\bigcup_{p \in P} p = M_{expr_j} \ \wedge \ \forall p_1, p_2 \in P : p_1 \neq p_2 \Rightarrow p_1 \cap p_2 = \emptyset$$
$$\forall expr'_j, expr''_j \in M_{expr_j} :$$
$$expr'_j, expr''_j \in p \Leftrightarrow \forall i : [\![expr'_j]\!]^t_i = [\![expr''_j]\!]^t_i$$

Here is an example in which a test t_3 yields two partition cells for four mutated expressions of $expr_3$:

t_3	$expr_3$	$expr'_3$	$expr''_3$	$expr'''_3$	$expr''''_3$
`a = 1, b = 1`	`a + b`	`a * b`	`a / b`	`a - b`	`a % b`
	`2`	`1`	`1`	`0`	`0`
		p_1		p_2	

Our optimization selects an arbitrary representative mutation from each partition cell $p \in P$. If a test detects the representative mutation of a cell p, all other mutations within p are marked as being detected without execution. If a test does not detect the representative mutation, then the test does not detect any of the other mutations within the cell.

4. IMPLEMENTATION DETAILS

We implemented monitoring, propagating, and partitioning of infected execution states in the Major mutation framework.

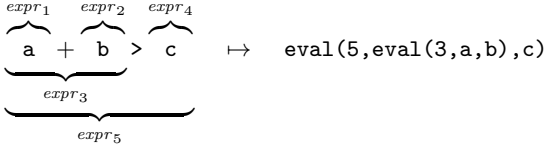
Given a program and a test suite, Major performs a prepass by executing the test suite on an instrumented version of the unmutated program. During this prepass Major determines mutation coverage and identifies test-equivalent mutants. Major ordinarily operates in 4 consecutive phases — 2 for an optimization prepass, and 2 for mutation analysis itself:

1. Major instruments the program under test to measure coverage. (Major extends the OpenJDK Java compiler and operates on the abstract syntax tree (AST).) Major also links in a runtime library to support the instrumentation.
2. Major runs the program's test suite on the instrumented program to determine which mutants are uncovered by which test and thus are test-equivalent.
3. Starting with the original (uninstrumented) version of the program, Major inserts all possible mutants into a single version of the program, each mutant disabled by default. Major also links in a different runtime library to support enabling the mutants.
4. Major performs mutation analysis by running the test suite on the mutated program, activating one mutation at a time. A test is not executed on any test-equivalent mutant, thus speeding up the analysis.

The contribution of this paper is to augment Major's prepass phases in order to identify more test-equivalent mutants. During the prepass, the augmented version of Major determines mutation coverage, monitors infection and propagation, and computes the partitions. The remainder of this section describes our changes to the prepass instrumentation (phase 1) and the prepass analysis (phase 2).

4.1 Prepass Instrumentation

In phase 1, Major instruments the original program by replacing every expression $expr_j$, that is subject to mutation, with a functionally equivalent method call to an instrumentation method in Major's runtime library, named `eval`. For example, the original expression `a + b > c` is replaced by:



The parameters of the `eval` method are the unique expression id and the subexpressions of $expr_j$. This approach ensures compact instrumentation, which is important because of the JVM's limitations on the size of any single method in the original program. While instrumenting the original program, Major also builds the following two mappings:

1. **Expressions**: expr id ↦ original opcode, $\overline{\text{expr id}}$
   (This represents the AST of the original program.)
2. **Mutations**: mutant id ↦ expr id, mutant opcode
   (This represents the set of all mutants of the program.)

Here is an example, for the expression ($expr_5$) above — a subscript e indicates an expression id and m a mutant id:

Expressions
$3_e \mapsto$ `+`, 1_e, 2_e
$5_e \mapsto$ `>`, 3_e, 4_e

Mutations
$1_m \mapsto 3_e$, `-`
$2_m \mapsto 3_e$, `*`
$3_m \mapsto 5_e$, `>=`
$4_m \mapsto 5_e$, `==`

At run time in phase 2, the `eval` method utilizes these mappings to evaluate the expression as usual, but also to perform monitoring, propagation, and partitioning of infected execution states as a side effect. The instrumentation preserves the order and number of expression evaluations, which is necessary to soundly handle expressions with side effects.

Conditional Operators

The semantics of short-circuiting conditional operators has to be preserved when instrumenting conditional expressions. It is not valid to translate `a || b` into `eval(1,a,b)` because `b` is not necessarily evaluated within the original expression — `b` is not evaluated if `a` evaluates to `true`. Furthermore, expression `a` must be evaluated at most once. Therefore, a conditional expression `a` *op* `b`, where *op* denotes the original conditional operator, is translated by means of two method calls `lhs` and `rhs`:

$$\texttt{a}\ \mathit{op}\ \texttt{b} \mapsto \texttt{lhs(22,a)}\ \mathit{op}\ \texttt{rhs(22,b)}$$

Each pair of `lhs` and `rhs` routines communicate via a shared variable. The `lhs` routine evaluates `a` (which might itself contain calls to `eval`) and saves and returns its value. The `rhs` routine executes $\texttt{eval(22,sharedvar}_{22}\texttt{,b)}$.

4.2 Monitoring of Infected Execution States

In phase 2, Major executes the test suite T on the instrumented version of the original program and determines for each test $t \in T$ a set of t-equivalent mutants, which need not be executed for t during the mutation analysis in phase 4.

The `eval` method of the runtime library determines for each test t and each mutation $expr'_j$ of $expr_j$ whether $[\![expr'_j]\!]^t_i = [\![expr_j]\!]^t_i$. If the values differ for some evaluation i, then the execution of t has infected the execution state of $expr'_j$. The runtime library no longer needs to monitor state infection for this pair $\langle t, expr'_j \rangle$, so for efficiency it stops doing so. If a pair $\langle t, expr'_j \rangle$ has not been eliminated by the end of the test execution, then the mutation is t-equivalent.

4.3 Propagation of Infected Execution States

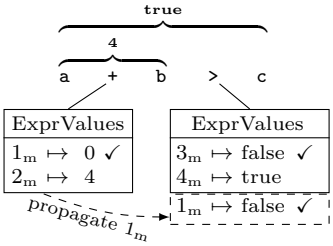
During the prepass analysis of a test t in phase 2, the runtime library keeps track of the propagation of infected execution states within composed expressions. While the runtime library evaluates a composed expression, it maintains a mapping (ExprValues) from mutation id to the value of the currently-executing expression $expr_e$. The mapping contains the ids of mutations of $expr_e$ whose execution state is infected by t and the ids of mutated subexpressions of $expr_e$ whose infected execution state propagated to $expr_e$. As an optimization, a mutation is removed from the mapping as soon as its infected execution state does not (further) propagate. Figure 1 depicts the propagation step and the maintained mapping for a test t_4 and the expression `a + b > c`.

A test t must be executed on a mutant if the execution state of the mutated expression propagates to a toplevel expression (one whose lexical parent is a statement). Major currently does not track propagation across statements. As soon as a mutated expression $expr'_j$ propagates to its toplevel expression for a test t, it need not be tracked any longer, i.e., the runtime library stops monitoring the pair $\langle t, expr'_j \rangle$.

4.4 Partitioning of Infected Execution States

In phase 2, for each test t, the runtime library partitions the mutants with infected execution state as follows:

Figure 1: Example for the propagation of infected execution states within the compound expression `a + b > c` (✓ indicates an infected execution state).



Test	Mutations
t_4 :=	$1_m \mapsto 3_e$, -
a = 2	$2_m \mapsto 3_e$, *
b = 2	$3_m \mapsto 5_e$, <
c = 2	$4_m \mapsto 5_e$, >=

- For each mutated (toplevel) expression, the runtime library maintains a set of mutations with an identically infected execution state — that is, a partition cell containing all mutations that have evaluated to the same value so far.
- Each time t executes a mutated expression (recall that t might evaluate an expression multiple times), these partition cells are updated to retain only those mutants that also had the same result for the new execution. Finally, each cell contains all mutations that evaluated identically with respect to the entire execution of t.
- Each mutation that was not monitored (e.g., those that may change side effects, see Section 3.2) is added individually — that is, such a mutation forms a partition cell of size 1.
- The runtime library selects an arbitrary representative mutant for each partition cell.

The runtime library partitions per test, not per test suite. It would be simpler to construct a single partition for the whole test suite, but it would be less efficient because the partition cells would tend to be smaller. When partitioning per test it is possible that, during mutation analysis in phase 4, a test has partition cells containing only mutants that have already been killed by previously executed tests; such partition cells are skipped by Major. Consider the following example:

Test	Partition	Representative	Executed?	Detected?
t_1	1 2 3 4	1	yes	detected
	5 6	5	yes	not detected
t_2	1	1	no	
	2 3	2	no	
	4 5	4	yes	
	6	6	yes	

Major first runs test t_1. Suppose that t_1 detects mutant 1 but does not detect mutant 5. From the partitioning we know that t_1 also detects mutants 2, 3, and 4. When Major proceeds to test t_2, Major skips the first two partition cells, as all the mutations contained in them have already been detected by t_1. To determine whether t_2 kills mutation 5, Major needs to run t_2 against either mutation 4 or 5 (either is fine; the result of running t_2 against mutation 4 might be different than running t_1 against mutation 4). Major also runs t_2 against mutation 6.

Table 1: Overview of the open source projects chosen for evaluation.

Project	Full Name	Classes	SLOC*	Test classes	Test SLOC*	Mutants	Version	Source
collect	Commons Collections	273	26,323	194	29,055	11,832	3.2.1	http://commons.apache.org
io	Commons IO	104	8,839	95	15,566	6,756	2.4	http://commons.apache.org
lang	Commons Lang	147	19,499	119	33,341	18,887	3.1	http://commons.apache.org
math	Commons Math	536	39,991	266	41,906	55,550	2.1	http://commons.apache.org
itext	IText PDF	592	76,321	26	1,612	108,174	5.0.6	http://itextpdf.com
jaxen	Jaxen	318	21,079	99	8,476	7,179	1.1.3	http://jaxen.codehaus.org
jdom	JDom	161	15,163	88	22,405	10,778	2.0	http://www.jdom.org
chart	JFreeChart	610	91,174	383	48,026	67,097	1.0.13	http://www.jfree.org
time	Joda Time	227	27,139	156	51,901	18,415	2.0	http://joda-time.sourceforge.net
num4j	Numerics4J	86	3,771	64	5,273	5,650	1.3	http://doodleproject.sourceforge.net
poi-main	POI (core)	893	85,427	458	57,599	64,567	3.9	http://poi.apache.org
poi-xml	POI (ooxml)	464	50,589	147	15,388	22,273	3.9	http://poi.apache.org
scratchpad	POI (scratchpad)	699	87,035	176	15,061	71,125	3.9	http://poi.apache.org
trove	GNU Trove	1,594	116,750	28	13,279	71,683	3.0.2	http://trove4j.sourceforge.net
Sum		6,704	669,100	2,299	358,888	539,966		

*SLOC is the number of non-comment, non-empty lines of code, as reported by sloccount (http://www.dwheeler.com/sloccount)

5. EMPIRICAL EVALUATION

We have implemented this paper's optimizations in the Major mutation analysis framework and evaluated them on a set of open source programs to empirically answer the following four research questions:

RQ1: How frequently are mutants covered without leading to state infection?

RQ2: How frequently do infected states not propagate within their lexically enclosing expressions?

RQ3: How frequently do different mutants exhibit an identical infected execution state at run time?

RQ4: How does monitoring, propagating, and partitioning of infected execution states improve the overall efficiency of mutation analysis?

5.1 Subject Programs

We selected software projects that have a decently sized developer-written test suite to make our analysis possible. Furthermore, we selected projects from different application domains (e.g., mathematical, XML processing, data structures, etc.) Table 1 presents statistics on the 14 Java projects that we selected for evaluation.

5.2 Test Suites

To improve generalizability, our evaluation uses four test suites per project (Table 2). Each project already comes with a **developer-written** test suite. We created additional test suites automatically for each project using the EvoSuite [7] test generation tool. EvoSuite supports different criteria when generating test suites, and we used all its non-experimental criteria, i.e., these are the criteria applicable on the chosen projects without leading to errors. We used EvoSuite to generate a test suite of 20 **random** tests for each individual class in each of the projects. We further used EvoSuite to generate one test suite targeting **branch coverage** and one targeting **weak-mutation** testing for each class. Weak-mutation testing means that EvoSuite only tries to achieve state infection, but does not actively optimize propagation. EvoSuite uses its own set of mutation operators working on Java bytecode: arithmetic operator replacement, relational operator replacement, constant replacement, variable replacement, method call deletion, unary operator insertion). EvoSuite was run using its default settings, i.e., each class was tested for up to 10 minutes or 1,000,000 executed statements (whichever limit was hit first). EvoSuite produces JUnit tests that include assertions, which are minimized in EvoSuite using mutation analysis. EvoSuite by default runs Java in "headless mode", which means that classes using GUI components cannot be tested. Furthermore, EvoSuite sometimes produces failing tests (unless using dedicated bytecode instrumentation to prevent this from happening), for example if there are dependencies between tests caused by static state, or assertions based on non-deterministic code (e.g., assertions on values influenced by the system time). We automatically removed all failing test cases from the analysis; on average, 6.6% of the tests produced by EvoSuite were removed.

5.3 Mutation Analysis

We performed mutation analysis with the Major mutation framework [16] using its default set of mutation operators. This set includes expression- and statement-level mutation operators (constant replacement, unary operator replacement, binary operator replacement, condition manipulation, statement deletion). We performed mutation analysis for each of the test suites using five different optimization configurations:

Coverage (baseline) A test is executed on a mutant only if the test reaches and executes the mutation. The baseline also includes all default optimizations implemented in Major (e.g., conditional mutation [18], test prioritization [19], and non-redundant mutation operators [20]). Moreover, a test is only executed on undetected mutants, meaning that a mutant is removed from the analysis once it has been detected [19].

Infection A test is executed on a mutant only if the test infects the execution state of the mutant.

Infection+Propagation A test is executed on a mutant only if it infects the execution state of the mutant and the infected state propagates to the toplevel expression.

Infection+Partitioning A partition of infected execution states is built for each mutated expression, and a test is executed on one representative mutant for each partition cell.

Infection+Propagation+Partitioning A partition of infected execution states is built for each toplevel expression, and a test is executed on one representative mutant for each partition cell.

Table 2 reports the number of test executions as well as the overall time for mutation analysis, which includes the time for the prepass. All experiments were run on an off-the-shelf 64-bit Linux machine with Intel i7 CPU and 8GB of memory.

Table 2: Test suites investigated in the empirical study.

Tests denotes the number of individual tests in the test suite, which has *sloc* lines of code and takes *time* seconds to run on the uninstrumented program. *Coverage* gives the ratio of covered to generated mutants, *kill score* gives the mutation kill score (i.e., the ratio of detected to generated mutants — the ratio in parentheses gives the kill score related to the number of covered mutants). *Executions* shows the number of test executions necessary for the mutation analysis when using only the mutation coverage optimization (baseline). The *Overall* row is calculated by considering all tests over all mutants across all 14 projects. Source lines of code (*sloc*) as reported by sloccount (http://www.dwheeler.com/sloccount).

Project	Manual						Random					
	Test suite			Mutation analysis			Test suite			Mutation analysis		
	tests	sloc	time	coverage	kill score	executions	tests	sloc	time	coverage	kill score	executions
collect	1,161	29,055	34	0.84	0.61 (0.72)	12,947	4,277	83,261	8	0.47	0.23 (0.49)	13,881
io	344	15,566	60	0.45	0.34 (0.76)	4,260	1,556	27,810	4	0.36	0.14 (0.39)	3,878
lang	2,047	33,341	32	0.91	0.69 (0.76)	22,077	1,673	37,965	6	0.48	0.19 (0.40)	12,745
math	2,169	41,906	246	0.89	0.72 (0.81)	87,088	5,473	130,123	29	0.42	0.19 (0.46)	58,721
itext	92	1,612	16	0.15	0.12 (0.75)	44,960	6,211	97,790	66	0.33	0.08 (0.25)	84,873
jaxen	634	8,476	13	0.65	0.41 (0.64)	90,600	2,993	61,199	34	0.54	0.18 (0.33)	111,313
jdom	1,638	22,405	55	0.90	0.75 (0.83)	15,687	1,711	29,184	14	0.24	0.11 (0.47)	10,067
chart	2,130	48,026	187	0.52	0.28 (0.53)	71,695	9,106	195,766	57	0.27	0.16 (0.58)	85,646
time	3,855	51,901	146	0.84	0.73 (0.87)	58,331	2,455	44,555	28	0.58	0.30 (0.52)	165,615
num4j	218	5,273	3	0.95	0.65 (0.69)	6,933	1,148	17,020	4	0.74	0.42 (0.57)	7,982
poi-main	1,874	57,599	3	0.18	0.09 (0.49)	20,683	9,497	187,394	43	0.31	0.15 (0.47)	109,457
poi-xml	643	15,388	179	0.76	0.45 (0.60)	45,984	4,947	71,385	88	0.11	0.05 (0.44)	14,638
scratchpad	703	15,061	85	0.42	0.34 (0.81)	90,082	7,197	159,072	41	0.25	0.10 (0.39)	61,770
trove	544	13,279	20	0.08	0.05 (0.64)	18,042	7,045	152,715	61	0.42	0.21 (0.49)	75,348
Overall	18,052	358,888	1080	0.61	0.45 (0.71)	589,369	65,289	1,295,239	481	0.39	0.18 (0.45)	815,934

Project	Branch						Weak					
	Test suite			Mutation analysis			Test suite			Mutation analysis		
	tests	sloc	time	coverage	kill score	executions	tests	sloc	time	coverage	kill score	executions
collect	2,269	25,639	4	0.76	0.45 (0.59)	11,711	2,144	24,834	6	0.76	0.46 (0.61)	11,419
io	697	7,256	3	0.54	0.27 (0.50)	4,500	817	8,655	8	0.66	0.33 (0.50)	5,518
lang	2,500	18,753	5	0.79	0.39 (0.49)	18,100	2,178	17,412	5	0.68	0.36 (0.53)	15,879
math	2,736	29,978	62	0.65	0.32 (0.50)	64,281	2,999	34,078	68	0.65	0.36 (0.55)	65,021
itext	4,526	47,565	76	0.56	0.20 (0.36)	160,909	3,184	34,190	61	0.40	0.23 (0.57)	99,974
jaxen	1,192	14,126	9	0.78	0.28 (0.36)	60,937	937	11,936	39	0.76	0.31 (0.41)	38,447
jdom	1,359	14,406	18	0.45	0.28 (0.61)	11,410	986	10,255	20	0.36	0.21 (0.58)	6,791
chart	6,965	77,348	42	0.50	0.29 (0.57)	80,676	7,114	78,799	65	0.48	0.28 (0.58)	80,042
time	2,708	21,999	19	0.75	0.43 (0.58)	85,543	2,753	21,474	28	0.71	0.45 (0.63)	92,784
num4j	392	4,068	3	0.66	0.41 (0.62)	6,348	507	4,905	12	0.68	0.46 (0.67)	6,961
poi-main	6,445	70,717	64	0.63	0.36 (0.57)	197,059	5,998	67,784	64	0.59	0.35 (0.59)	191,807
poi-xml	637	10,295	9	0.18	0.08 (0.43)	8,096	674	11,039	14	0.20	0.08 (0.42)	8,600
scratchpad	3,428	36,049	31	0.35	0.17 (0.49)	87,241	2,888	32,371	20	0.32	0.16 (0.50)	79,264
trove	11,492	106,084	26	0.63	0.39 (0.61)	87,604	6,207	55,315	33	0.48	0.28 (0.58)	60,636
Overall	47,346	484,283	370	0.59	0.31 (0.52)	884,415	39,386	413,047	443	0.55	0.31 (0.55)	763,143

5.4 Results

We report results in terms of three metrics: filtered test-equivalent mutants (Section 5.4.1), test executions (in greatest detail in Sections 5.4.2–5.4.4), and run time (Section 5.4.5). These measures are correlated but are not the same. Mutants are not linearly related to test executions because a given mutant might be killed by the first test in a suite or survive to the end. Test executions can also differ from run time. Suppose that a suite has many fast unit tests and a few system tests. A technique could eliminate 90% of test executions (those for the fast unit tests) without visibly improving end performance. In our experiments, however, the measures all led to the same conclusions.

5.4.1 Test-Equivalent Mutants

Table 3 summarizes how many of the mutants that were covered by the test suites remain after the prepass optimizations filter out test-equivalent and redundant mutants. Two clear trends are observable. First, each optimization reduces the number of mutants that have to be considered during the mutation analysis. Second, weaker test suites (e.g., random testing) have a higher degree of test-equivalence and redundancy than stronger test suites (e.g., manually written test suites).

Table 3: Mutants that remain after the prepass.

Each value is the ratio of non-filtered mutants, left by one of our optimizations, compared to the number of covered mutants.

	Manual	Random	Branch	Weak
Infect	0.90	0.84	0.89	0.90
Infect+Prop	0.88	0.80	0.86	0.87
Infect+Part	0.87	0.79	0.84	0.86
Infect+Prop+Part	0.74	0.66	0.70	0.71

5.4.2 RQ1: Coverage without State Infection

How common is it for a test to cover a mutant without infecting its execution state? The state infection optimization identifies 3–42% of covered mutants as test-equivalent for every test in the test suite (Table 4). These need not be executed during mutation analysis.

The weak-mutation test suites have the lowest number of test-equivalent mutants in most projects, and that is not surprising considering that EvoSuite's weak mutation criterion aims to achieve state infection. The two exceptions are *chart* and *time*, where the weak-mutation test suites lead to the highest number of test-equivalent mutants of all test suites. As *chart* is the second largest project, this also affects the overall measurement for weak-mutation compared to

Table 4: Benefit of infection.
Each value is the ratio of test executions on mutants achieving state infection to test executions on all covered mutants.

Project	Manual	Random	Branch	Weak
collect	0.84	0.62	0.81	0.83
io	0.82	0.63	0.80	0.81
lang	0.82	0.75	0.79	0.79
math	0.80	0.84	0.82	0.85
itext	0.81	0.68	0.91	0.97
jaxen	0.69	0.62	0.58	0.59
jdom	0.79	0.58	0.68	0.81
chart	0.83	0.81	0.89	0.78
time	0.76	0.73	0.77	0.68
num4j	0.88	0.84	0.82	0.84
poi-main	0.84	0.82	0.85	0.86
poi-xml	0.74	0.69	0.79	0.79
scratchpad	0.85	0.72	0.87	0.90
trove	0.83	0.68	0.70	0.75
Overall	0.79	0.73	0.82	0.82

branch-coverage test suites, which have more test-equivalent mutants in all but these two projects.

On the other hand, random tests tend to achieve state infection slightly less often than all other test suites. This pattern of random test suites having more redundancy than more rigorous test suites holds throughout our experiments (see the subsequent sections of the paper). A possible conjecture is that more rigid test criteria result in fewer tests that lead to test-equivalent mutants. This could be explained by the observation that stronger test suites generally cover more mutants than weaker test suites (see Table 2): For example, covering a mutant that is deeply nested in the control flow of the program might require several executions of mutants along the path, and each additional execution of a mutant increases the chances of achieving state infection.

The manually written test suites have fewer test-equivalent mutants than random tests, but overall slightly more than the branch coverage and weak-mutation test suites. This may be explained with the substantially higher number of covered and killed mutants by the manually written tests (see Table 2): For example, if EvoSuite fails to cover large parts of the code in the first place (e.g., due to environmental dependencies or because it is GUI-related code [8]), then any mutants contained in that code would not be covered at all. In contrast, manual testing may easily lead to coverage of such regions of code, yet possibly in an insufficient way such that infection is not achieved.

5.4.3 RQ2: Infection without Propagation

How common is it that a mutant infects the state locally, but the infection does not propagate to its lexically enclosing expressions? More specifically, what is the benefit of the propagation optimization? Table 5 compares test executions for the Infect and Infect+Prop configurations, and Table 6 compares the Infect+Part and Infect+Prop+Part configurations.

In the absence of partitioning (Table 5), 12% of mutations are t-equivalent even though the test t infected its execution state. The *jaxen* project stands out with an extremely high reduction: Only 35% of test executions with coverage actually lead to a propagation of infected execution states!

Table 6 shows that the propagation optimization is synergistic with partitioning: in the presence of partitioning, propagation has an even greater relative benefit. Neither optimization dominates the other.

Table 5: Benefit of propagation (1).
Each value is the ratio of test executions on mutants propagating a state infection to test executions achieving state infection.

Project	Manual	Random	Branch	Weak
collect	0.97	0.92	0.96	0.96
io	0.91	0.93	0.90	0.91
lang	0.96	0.88	0.94	0.94
math	0.92	0.93	0.92	0.93
itext	0.96	0.99	0.91	0.98
jaxen	0.36	0.35	0.37	0.32
jdom	0.90	0.94	0.97	0.87
chart	0.97	0.97	0.93	0.89
time	0.89	0.83	0.81	0.84
num4j	0.97	0.93	0.88	0.93
poi-main	0.94	0.94	0.96	0.95
poi-xml	0.94	0.88	0.91	0.90
scratchpad	0.94	0.97	0.98	0.93
trove	0.99	0.93	0.93	0.94
Overall	0.86	0.85	0.90	0.91

Table 6: Benefit of propagation (2).
Each value is the ratio of test executions on mutants propagating a state infection with partitioning to test executions achieving state infection with partitioning.

Project	Manual	Random	Branch	Weak
collect	0.84	0.78	0.82	0.83
io	0.79	0.80	0.74	0.75
lang	0.74	0.78	0.75	0.76
math	0.73	0.82	0.78	0.79
itext	0.88	0.88	0.88	0.83
jaxen	0.31	0.32	0.34	0.28
jdom	0.76	0.74	0.88	0.87
chart	0.94	0.90	0.92	0.87
time	0.72	0.78	0.86	0.72
num4j	0.80	0.75	0.72	0.74
poi-main	0.86	0.83	0.88	0.89
poi-xml	0.87	0.75	0.81	0.81
scratchpad	0.88	0.91	0.91	0.92
trove	0.84	0.76	0.77	0.79
Overall	0.76	0.76	0.83	0.82

5.4.4 RQ3: Partitioning of Infected States

How much redundancy is caused by tests that lead to the same infected state on the same mutants? The partitioning can be applied using infected states immediately after the mutation, or after the infected state has been propagated, and we evaluated both configurations.

Partitioning reduces test executions by 5% on average (Table 7), compared to state infection. In other words, 5% of the test executions on infected states are redundant as they are identical to the infected states on other executions.

Table 8 summarizes the effect of partitioning *after* checking propagation of infected states. Compared to when partitioning is applied without propagation, the effect is significantly greater (15% average reduction vs. 5%). This reveals that even if state infections are locally different, once they propagate they are more likely to result in a similar infected state. For example, different mutations of an arithmetic expression may lead to different numerical results, but an enclosing relational expression will only either evaluate to true or false, and thus all mutations for which the infected state propagates result in the same state.

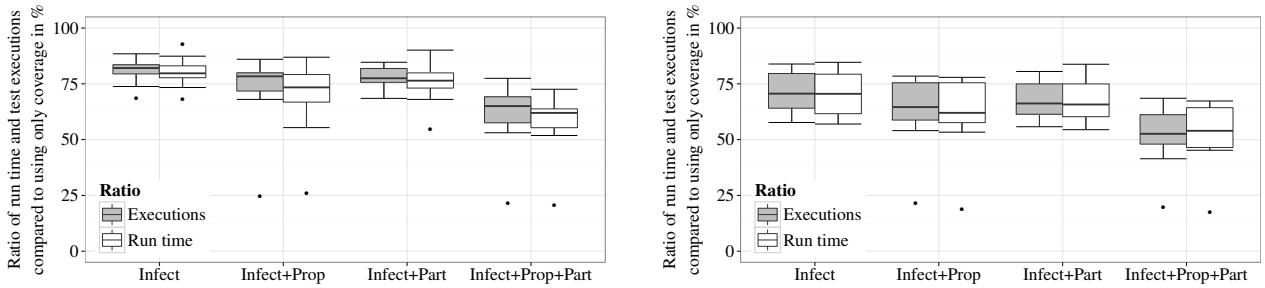
As partitioning has a higher overhead than the other optimizations (partitions need to be updated during test executions) it is important to maximize its effectiveness. Conse-

Table 7: Benefit of partitioning (1).
Each value is the ratio of test executions on partitioned mutants achieving state infection to test executions achieving state infection.

Project	Manual	Random	Branch	Weak
collect	0.95	0.95	0.95	0.95
io	0.93	0.96	0.94	0.94
lang	0.94	0.94	0.94	0.94
math	0.97	0.92	0.91	0.91
itext	0.95	0.93	0.91	0.97
jaxen	1.00	0.99	0.99	0.99
jdom	0.95	0.97	0.90	0.86
chart	1.00	0.94	0.91	0.85
time	0.97	0.84	0.71	0.96
num4j	0.96	0.96	0.94	0.97
poi-main	0.96	0.95	0.98	0.98
poi-xml	0.99	0.97	0.96	0.97
scratchpad	0.97	0.94	0.99	0.98
trove	1.00	0.96	0.95	0.96
Overall	0.97	0.93	0.93	0.95

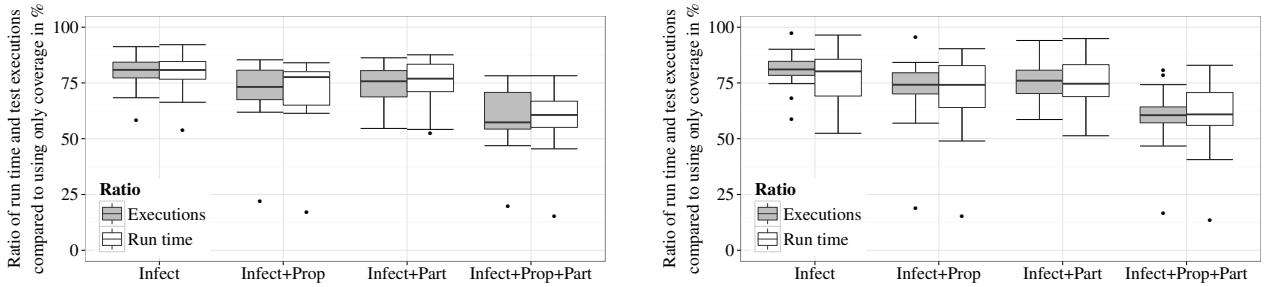
Table 8: Benefit of partitioning (2).
Each value is the ratio of test executions on partitioned mutants achieving propagated state infection to test executions leading to propagated state infection.

Project	Manual	Random	Branch	Weak
collect	0.82	0.80	0.82	0.82
io	0.81	0.83	0.78	0.77
lang	0.73	0.84	0.75	0.77
math	0.78	0.81	0.77	0.78
itext	0.87	0.83	0.88	0.82
jaxen	0.87	0.92	0.90	0.87
jdom	0.81	0.77	0.82	0.87
chart	0.97	0.87	0.91	0.83
time	0.78	0.80	0.76	0.82
num4j	0.78	0.77	0.77	0.77
poi-main	0.88	0.84	0.90	0.91
poi-xml	0.92	0.82	0.86	0.87
scratchpad	0.91	0.88	0.92	0.96
trove	0.84	0.79	0.79	0.81
Overall	0.86	0.83	0.86	0.86



(a) Manually written test suites T_{orig}

(b) Generated test suites T_{random}



(c) Generated test suites T_{branch}

(d) Generated test suites T_{weak}

Figure 2: Ratio of total run time and test executions compared to using only coverage in %.
The Wilcoxon signed rank test showed that the differences in the medians of total run time and test executions are statistically significant at the 1% level. The differences are statistically significant for all test suites and all configurations compared to using only coverage.

quently, these results suggest that partitioning should only be applied in conjunction with propagation.

5.4.5 RQ4: Efficiency Improvement

The results so far have shown that there is a significant number of test-equivalent mutants and redundant test executions, which can be avoided using the presented optimizations. How do these optimizations affect the efficiency of the mutation analysis?

To quantify the efficiency improvement, we measure the total time taken by Major for the complete analysis, which is the data most relevant to a user. The total time includes the run time of the prepass, which is several orders of magnitude smaller than the run time of the mutation analysis.

Figure 2 illustrates the improvement in terms of run time and number of test executions. The median reduction in run time is very similar to the median reduction of test executions. The reduction in run time is slightly higher than the reduction of test executions for most configurations, except for the branch coverage test suites. The inter-quartile range is generally largest for the random test suites, which is likely because random tests are not minimized like the other generated tests, and thus vary more in length and execution time. For the weak mutation test suites the inter-quartile range is larger for the time reduction than for the test executions, suggesting that there is less variation in the amount of redundancy in the tests than in their execution time.

Between the individual configurations the time reduction mirrors the results obtained for the test executions: Restricting test executions to cases with state infection reduces the

time of mutation analysis by around 20%. Propagation is slightly more effective than partitioning, and each is only a slight improvement over infection. The redundancy exploited by these two optimizations is orthogonal: the combination is greater than the sum of its parts. Overall the three optimizations reduce mutation analysis time by 40% over the coverage optimization alone.

5.5 Threats to Validity

Threats to *internal validity* might come from how the empirical study was carried out. To reduce the probability of having faults in our mutation analysis tool or experimental setup, it has been carefully tested; however, it is well known that testing alone cannot prove the absence of defects. Randomized algorithms are affected by chance, which means that the test suites produced by EvoSuite may differ at each run. However, there are more than 6,000 classes in the projects we used for evaluation, which reduces the chances of this affecting the results.

To cope with possible threats to *external validity*, we selected a variety of large and well-known open source projects, which differ in size, operation purpose, and test suite quality. Furthermore, we used test suites provided with the evaluation projects as well as automatically generated test suites using the EvoSuite tool in order to avoid that results are biased by the test suites provided with the evaluation projects.

Threats to *construct validity* treat the definition of the performance of a mutation analysis optimization. As the actual performance of a mutation analysis tool is dependent on the implementation as well as the tests at hand, we explicitly measured the overhead of the techniques as well as the reduction in terms of number of avoided test executions.

6. RELATED WORK

A commonly-used categorization identifies three main strategies to improve the scalability of mutation analysis: 1) "do fewer", e.g., mutant sampling [1,4], selective mutation [25], 2) "do smarter", e.g., parallelization [6], weak mutation [13], non-redundant definitions of mutation operators [19], and 3) "do faster", e.g., mutant schema [31], mutation of bytecode instead of source code [24]. All these techniques are independent of the program under test, and all techniques can be used together with the optimizations presented in this paper.

In contrast, *run-time* optimizations require knowledge of the program under test, which is acquired by executing the test suite on the original program before executing it on any mutants. An effective optimization of this category is to first determine which test covers (i.e., reaches and executes) which mutant, and then for each mutant to only execute tests that cover the mutant. This is implemented in mutation tools such as Javalanche [27], Major [19], and Certitude [11]. Implementation is straightforward; for example, mutant schemata can easily be extended to collect the necessary coverage information [18].

A further run-time optimization is to record execution times of tests, and then to prioritize tests such that quicker tests are executed first [19]. The FaMT [33] technique (Faster Mutation Testing) takes a similar approach by prioritizing tests for every mutant, and by executing only the tests estimated to be most likely to detect a mutant. Again, such optimizations are orthogonal to the optimizations presented in this paper and are thus well-suited to be combined.

The idea of state infection conditions goes back to DeMillo et al. [5], who identified reachability, infection, and propagation as the three necessary conditions to kill a mutant. Their state infection conditions have since been used to drive test generation (e.g., [5, 10, 26, 34]). In this context, the conditions have also been used to avoid redundant test executions during fitness evaluations in a search-based test generation approach [9]. In contrast, our approach uses state infection during mutation analysis. It monitors state infection and avoids executing tests on mutants if they do not infect the state or if the infected state does not propagate. Moreover, it also avoids redundant test executions by observing and partitioning the actual state infections.

Schuler and Zeller have shown that the likelihood of a mutant being equivalent is related to the *impact* that it causes in terms of violations of invariants [28] and changed code coverage [29]. By measuring state infection and propagation we essentially determine which mutants are guaranteed to have no impact. However, our approach determines this from a single test execution on an instrumented program, without requiring test executions on any mutants.

7. CONCLUSIONS AND FUTURE WORK

Mutation analysis provides a quality metric for existing test suites and can guide test generation, but its applicability is limited by inherent scalability issues. The three optimizations presented in this paper significantly improve scalability, thus taking mutation analysis a big step further towards practical applicability.

There is still potential to further improve the performance of mutation analysis. For example, we exploit information about propagation in complex expressions, but even if an infected state propagates beyond its enclosing expressions, it may not propagate to an observable output. More advanced propagation information could identify further test-equivalent mutants and would provide more opportunity to partition mutants based on their infected states. A challenge is to derive such information from a single execution *before* executing a test on the mutant, like in this paper.

Besides scalability, the second major limitation of mutation analysis are mutants that are equivalent not only for a particular test, but for all possible tests. It has been shown that there is a relation between equivalence and to what extent state infection propagates and affects control flow [29]. Potentially, test-equivalence offers a means to predict equivalence without requiring to execute a test on a mutant (cf. [17]).

These considerations are rooted in a scenario where a developer would use a mutation analysis tool to analyze an existing test suite or to decide which test should be written next. However, the presented optimizations might actually be most useful when implemented in a scenario of automated test generation, where scalability of mutation as a test criterion is even more important than for regular mutation analysis. Future work should consider how test-equivalence and partitioning can be exploited in that scenario.

The Major mutation framework and the subject programs used in the evaluation are publicly available at:

`http://mutation-testing.org`

8. ACKNOWLEDGMENTS

This material is based on research sponsored by DARPA under agreement number FA8750-12-2-0107. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

9. REFERENCES

[1] A. T. Acree. *On Mutation*. PhD thesis, Georgia Institute of Technology, 1980.

[2] K. Adamopoulos, M. Harman, and R. M. Hierons. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, volume 3103 of *LNCS*, pages 1338–1349, 2004.

[3] D. Baldwin and F. G. Sayward. Heuristics for determining equivalence of program mutations. Technical Report 276, Yale University, 1979.

[4] T. A. Budd. *Mutation Analysis of Program Test Data*. Phd thesis, Yale University, 1980.

[5] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering (TSE)*, 17(9):900–910, 1991.

[6] V. N. Fleyshgakker and S. N. Weiss. Efficient mutation analysis: A new approach. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 185–195, 1994.

[7] G. Fraser and A. Arcuri. EvoSuite: Automatic test suite generation for object-oriented software. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 416–419, 2011.

[8] G. Fraser and A. Arcuri. Sound empirical evidence in software testing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 178–188, 2012.

[9] G. Fraser and A. Arcuri. Efficient mutation testing using whole test suite generation. *Empirical Software Engineering (ESEM)*, 2014. To appear.

[10] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering (TSE)*, 28(2):278–292, 2012.

[11] M. Hampton and S. Petithomme. Leveraging a commercial mutation analysis tool for research. In *Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART)*, pages 203–209, 2007.

[12] R. M. Hierons, M. Harman, and S. Danicic. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability (JSTVR)*, 9(4):233–262, 1999.

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

[14] W. Huang, A. Milanova, W. Dietl, and M. D. Ernst. ReIm & ReImInfer: Checking and inference of reference immutability and method purity. In *Proceedings of Object-Oriented Programming, Systems, Languages & Applications*, pages 879–896, 2012.

[15] Y. Jia and M. Harman. Higher order mutation testing. *Information and Software Technology (IST)*, 51(10):1379–1393, 2009.

[16] R. Just. The Major mutation framework: Efficient and scalable mutation analysis for Java. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2014. To appear.

[17] R. Just, M. D. Ernst, and G. Fraser. Using state infection conditions to detect equivalent mutants and speed up mutation analysis. In *Proceedings of the Dagstuhl Seminar 13021: Symbolic Methods in Testing*, volume abs/1303.2784, 2013.

[18] R. Just, G. M. Kapfhammer, and F. Schweiggert. Using conditional mutation to increase the efficiency of mutation analysis. In *Proceedings of the International Workshop on Automation of Software Test (AST)*, pages 50–56, 2011.

[19] R. Just, G. M. Kapfhammer, and F. Schweiggert. Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 11–20, 2012.

[20] R. Just and F. Schweiggert. Higher accuracy and lower runtime: Efficient mutation analysis using non-redundant mutation operators. *Software Testing, Verification and Reliability (JSTVR)*, 2014. To appear.

[21] R. Just, F. Schweiggert, and G. M. Kapfhammer. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 612–615, 2011.

[22] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(1):5–20, 1992.

[23] A. J. Offutt and W. M. Craft. Using compiler optimization techniques to detect equivalent mutants. *Software Testing, Verification and Reliability (JSTVR)*, 4(3):131–154, 1994.

[24] A. J. Offutt, Y.-S. Ma, and Y.-R. Kwon. An Experimental Mutation System for Java. *ACM Software Engineering Notes*, 29(5):1–4, September 2004.

[25] A. J. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 100–107, 1993.

[26] M. Papadakis and N. Malevris. Automatic mutation test case generation via dynamic symbolic execution. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 121–130, 2010.

[27] D. Schuler, V. Dallmeier, and A. Zeller. Efficient mutation testing by checking invariant violations. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 69–80, 2009.

[28] D. Schuler and A. Zeller. Javalanche: Efficient mutation testing for Java. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 297–298, 2009.

[29] D. Schuler and A. Zeller. (Un-)covering equivalent mutants. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 45–54, 2010.

[30] A. Siami Namin, J. H. Andrews, and D. J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 351–360, 2008.

[31] R. H. Untch, A. J. Offutt, and M. J. Harrold. Mutation

analysis using mutant schemata. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 139–148, 1993.

[32] J. M. Voas. PIE: A dynamic failure-based technique. *IEEE Transactions on Software Engineering (TSE)*, 18(8):717–727, 1992.

[33] L. Zhang, D. Marinov, and S. Khurshid. Faster mutation testing inspired by test prioritization and reduction. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 235–245, 2013.

[34] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. de Halleux, and H. Mei. Test generation via dynamic symbolic execution for mutation testing. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 1–10, 2010.