Using Non-Redundant Mutation Operators and Test Suite Prioritization to Achieve Efficient and Scalable Mutation Analysis

René Just
Dept. of Computer Science and Engineering
University of Washington
rjust@cs.washington.edu

Gregory M. Kapfhammer
Dept. of Computer Science
Allegheny College
gkapfham@allegheny.edu

Franz Schweiggert
Dept. of Applied Information Processing
Ulm University
franz.schweiggert@uni-ulm.de

***Abstract*—Mutation analysis is a powerful and unbiased technique to assess the quality of input values and test oracles. However, its application domain is still limited due to the fact that it is a time consuming and computationally expensive method, especially when used with large and complex software systems. Addressing these challenges, this paper makes several contributions to significantly improve the efficiency of mutation analysis. First, it investigates the decrease in generated mutants by applying a reduced, yet sufficient, set of mutants for replacing conditional (COR) and relational (ROR) operators. The analysis of ten real-world applications, with 400,000 lines of code and more than 550,000 generated mutants in total, reveals a reduction in the number of mutants created of up to 37% and more than 25% on average. Yet, since the isolated use of non-redundant mutation operators does not ensure that mutation analysis is efficient and scalable, this paper also presents and experimentally evaluates an optimized workflow that exploits the redundancies and runtime differences of test cases to reorder and split the corresponding test suite. Using the same ten open-source applications, an empirical study convincingly demonstrates that the combination of non-redundant operators and prioritization leveraging information about the runtime and mutation coverage of tests reduces the total cost of mutation analysis further by as much as 65%.**

I. Introduction

Mutation analysis is a well-known test adequacy criterion for assessing various testing and debugging techniques, including, for instance, approaches for test oracles, test data generation, or fault localization. Compared with testing techniques that rely on various code coverage criteria, mutation analysis is rather expensive because of the fact that many mutated versions of the analyzed program have to be executed. With regard to the costs of mutation analysis, several approaches have been proposed that try to either reduce the number of generated mutants or to speed-up the analysis process (cf. [8], [14]). However, the runtime of mutation analysis on large and complex software systems is still quite long, sometimes even prohibitively so.

Addressing the challenge of applying mutation analysis to real-world programs, this paper investigates the potential for efficiency improvements due to non-redundant mutation operators. Furthermore, it presents and evaluates an optimized process for mutation analysis that notably reduces its runtime. Overall, this paper makes the following contributions:

- An evaluation of the reduction of generated mutants by employing non-redundant COR and ROR operators on ten real-world applications. Ranging from 3,000 to more than 110,000 lines of code, the analyzed programs collectively contain 400,000 lines of code and more than 550,000 generated mutants. Applying the non-redundant mutation operators reduces the number of generated mutants by 26% to 410,000 mutants in total.
- An investigation of the test suite characteristics of real-world applications that enable runtime improvements.
- A presentation and visualization of an optimized mutation analysis workflow that is based on reordering and splitting test suites to exploit the identified redundancies and runtime differences of test cases.
- An empirical study that evaluates the presented approach on the same ten real-world applications. By utilizing the non-redundant mutation operators, the optimized workflow reduces the total runtime by 30% on average for the remaining set of 410,000 mutants.

Since this paper focuses on the efficiency of mutation analysis, Section II introduces this technique. Section III investigates the savings in terms of generated mutants by applying only a sufficient and non-redundant set of mutations for the ROR and COR mutation operator. Next, Section IV investigates characteristics of existing test suites for real-world applications and Section V presents an approach for significant efficiency improvements by exploiting mutation coverage and test runtime information. Thereafter, Section VI empirically evaluates the approach and discusses threats to validity. Section VII describes related work and Section VIII concludes the paper and presents future work.

II. Background on Mutation Analysis

Originally proposed by Budd [2] and DeMillo et al. [3], mutation analysis seeds small syntactical changes, called *mutants*, into a system under test (SUT). These mutants are generated by applying *mutation operators* that define certain transformations on the SUT where each transformation leads to one mutant. The use of mutants enables the assessment of various testing and debugging techniques with regard to their ability to detect or locate the mutants. Throughout this

paper, we refer to a mutant that is reached and executed by a given test case as *covered* mutant. Intuitively, a test case has to cover a certain mutant in order to be able to detect it. If a test eventually reveals the injected fault caused by the executed mutant, then the corresponding mutant is said to be *killed*. Consequently, a mutant that is not killed is called *live*. Finally, a mutant is deemed to be *equivalent* if it cannot be killed due to semantic equivalence to the original version.

It is important to note that mutation analysis is not feasible without proper tool support. Several mutation testing tools and frameworks have been developed to support a variety of programming languages [8]. This paper employs MAJOR, a mutation testing framework for the Java programming language that uses conditional mutation to reduce the cost of generating and executing mutants [9], [11].

III. Non-Redundant Mutation Operators

Subsumed mutants lead to redundancies in the generated set of mutants and thus affect the efficiency of mutation analysis and misrepresent the mutation score [10]. Offutt et al. [13] and Namin et al. [16] empirically investigated the correlations between mutation operators and statistically determined a subset of mutation operators that is sufficient in terms of the accuracy of the mutation score. Yet, they considered the mutation operators with their original definitions to be atomic — meaning that an operator was either excluded or applied with all defined replacements or insertions.

More recently, Kaminski et al. [12] investigated redundancies in the ROR mutation operator and proposed a non-redundant version for this operator by means of a subsumption hierarchy. Additionally, we revealed redundancies in the COR and UOI mutation operators and showed a sufficient set of mutations for these operators in our prior research [10]. In that previous work, we also examined the reduction of generated mutants in real-world applications by means of a small case study focusing on ROR, COR, and UOI.

Based on the strong evidence given in this past case study, this paper empirically investigates the actual improvement when only applying the non-redundant versions of the mutation operators for a larger set of applications, thus extending and verifying these prior results. To ensure that this paper is self-contained, Sections III-A and III-B briefly survey the prior results on non-redundant operators (cf. [10], [12]).

A. Non-Redundant ROR Operator

The ROR mutation operator replaces all binary relational operators (i.e. ==, !=, <, <=, >, >=) with both all valid alternatives and the special operators `true` and `false`. According to the subsumption hierarchy established by Kaminski et al., the following three replacements per operator are sufficient when mutating relational operators [12]:

<=	⟹	<,==,true	<	⟹	<=,!=,false
>=	⟹	>,==,true	>	⟹	>=,!=,false
==	⟹	<=,>=,true	!=	⟹	<, >,true

Therefore, the sufficient set of three out of seven possible mutations yields a reduction of 57%.

B. Non-Redundant COR Operator

Generally, the COR mutation operator replaces an expression `a <op> b`, where `a` and `b` denote boolean expressions or literals and <op> is one of the logical connectors && or ||. With regard to binary conditional operators, valid mutations belong to one of the following three categories:

1) Apply conditional operator
   - Apply logical connector AND: `a && b`
   - Apply logical connector OR: `a || b`
   - Apply equivalence operator: `a == b`
   - Apply exclusive OR operator: `a != b`
2) Apply special operator
   - Evaluate to left hand side: `lhs`
   - Evaluate to right hand side: `rhs`
   - Always evaluate to true: `true`
   - Always evaluate to false: `false`
3) Insert unary boolean operator
   - Negate left operand: `!a <op> b`
   - Negate right operand : `a <op> !b`
   - Negate expression: `!(a <op> b)`

By employing only the following sufficient four out of ten possible mutations, the reduction of the number of mutants generated for the conditional operators is 60%:

&&	⟹	lhs,rhs,==,false
||	⟹	lhs,rhs,!=,true

C. Empirical Evaluation

Given the sufficient sets for relational and conditional operators, the number of ROR and COR mutants is decreased by 57% and 60%, respectively. However, the total reduction of all generated mutants depends on the ratio of ROR and COR to all other mutants (cf. [10]). In order to investigate the actual decrease of all generated mutants for real-world programs, we analyze the applications described in Table I. It is important to note that this table's data about the test size and runtime reflects the characteristics of the existing JUnit test suites provided and released with the corresponding application. Throughout the analysis, we use the following mutation operators supported by MAJOR:

- Operator Replacement Binary (ORB): Replace all occurrences of arithmetic (AOR), logical (LOR), shift (SOR), conditional (COR), and relational (ROR) operators with all valid alternatives.
- Operator Replacement Unary (ORU): Replace all occurrences of unary operators with all valid alternatives.
- Unary Operator Insertion (UOI): Insert unary boolean operators to negate boolean expressions.
- Literal Value Replacement (LVR): Force literals to take a positive value, a negative value, and zero. Additionally, all reference initializations are replaced by `null`.

Table II shows the decrease in the number of generated mutants when applying the sufficient set of mutations for the ROR, COR, and UOI mutation operators. The exclusion

Table I
INVESTIGATED CASE STUDY APPLICATIONS

	Application	Version	Source files	LOC*	Test runtime*	Test classes	Test methods	Test LOC*	Mutants
trove	GNU Trove	3.0.2	691	116,750	15.2	25	544	13,279	116,991
chart	jFreeChart	1.0.13	585	91,174	27.3	353	2,130	48,026	92,000
itext	iText	5.0.6	408	76,229	8.4	26	75	1,612	160,891
math	Commons Math	2.1	408	39,991	76.2	234	2,169	41,906	81,577
time	Joda-Time	2.0	156	27,139	13.8	123	3,855	51,901	32,380
lang	Commons Lang	3.0.1	99	19,495	14.1	101	2,039	32,699	33,065
jdom	JDOM	2beta4	131	15,163	30.4	78	1,723	22,194	15,616
jaxen	Jaxen	1.1.3	197	12,440	12.1	78	699	8,514	10,247
io	Commons IO	2.0.1	100	7,908	17.4	48	309	13,608	9,901
num4j	Numerics4j	1.3	73	3,647	1.8	63	218	5,273	7,234
total			2,848	409,936		1,181	14,385	239,012	559,902

*Test runtime in seconds and lines of code as reported by sloccount (non-comment and non-blank lines)

Table II
DECREASE IN THE NUMBER OF GENERATED MUTANTS

	All mutants	Reduced Set	Decrease
trove	116,991	72,959	-37.6%
chart	92,000	68,519	-25.5%
itext	160,891	126,781	-21.2%
math	81,577	66,787	-18.1%
time	32,380	23,781	-26.6%
lang	33,065	21,074	-36.3%
jdom	15,616	10,800	-30.8%
jaxen	10,247	7,132	-30.4%
io	9,901	7,319	-26.1%
num4j	7,234	5,437	-24.8%
total	559,902	410,589	-26.7%

of the redundant mutants yields a significant improvement for all applications, with a total decrease of 26.7% and a range between 18.1% for *math* and 37.6% for *trove*. Since redundant mutants also misrepresent the mutation score [10], it is strongly advisable to apply only the non-redundant sets of the ROR and COR mutants. Interestingly, the UOI operator is subsumed by the sufficient set of COR mutants, and hence also redundant. This paper's approach to efficient mutation analysis always uses the non-redundant operators.

IV. Efficient and Scalable Mutation Analysis

By employing the non-redundant versions of the ROR and COR operators, we could decrease the number of generated mutants by almost 27% in total, as shown in Table II. However, the remaining number of mutants, which is 410,000 for all of the investigated applications, is still substantial. Therefore, we focus on further runtime improvements that do not rely on the reduction of mutants. This section makes several motivating observations concerning mutation coverage, the differences in test runtime, and the redundancies in a test suite. Ultimately, these insights lead to an optimized workflow that performs test suite prioritization and splitting.

A. Mutation Coverage

As already mentioned in Section II, a test case has to cover a mutant in order to be able to kill it. Generally, the following three conditions have to be fulfilled to ultimately detect mutants as well as real faults (cf. [17]):

1) Execution: The mutated code must be covered, meaning that it has to be reached and executed.
2) Infection: The execution of the faulty code segment has to change the internal state of the program.
3) Propagation: The infected internal state must be propagated to the output in order to be detectable.

While the first two conditions are necessary, the last one is sufficient to kill a mutant. Besides, the last condition can be generalized to oracles that are not output-based. In this case, the infected internal state has to be propagated to a state that is observable by the test oracle (cf. [7]). An example for such an observable state is the violation of contracts.

Since the first condition is necessary, it implies that if a mutant is not covered it cannot be killed. As a consequence, mutants that are not covered can be excluded and marked alive without execution. Utilizing this implication can significantly reduce the number of executions, especially if a test suite exhibits poor mutation coverage. Nevertheless, it is important to note that the mutation coverage has to be determined at runtime. Employing a code coverage tool for this purpose and mapping the covered statements and branches to mutants is feasible, but rather laborious, since code coverage tools are not designed for this purpose.

More advanced mutation analysis systems that encode all mutants within the original program can provide the mutation coverage information at runtime by means of additional code instrumentation (e.g., [6], [11]). MAJOR, the mutation system used in this paper, gathers the mutation coverage information efficiently at runtime. For performance reasons, it only records the mutation coverage if and only if the original, which means the unmutated version of the SUT, is executed (cf. [9], [11]). Due to the overhead incurred by determining the mutation coverage, this feature is disabled during the execution of mutants.

For all of the investigated applications, Table III shows the number of generated and covered mutants plus the ratio of covered-to-generated mutants. The results exhibit a notable divergence between the applications, ranging between 8.2% for *trove* and 94.7% for *num4j*, with a mutation coverage of

Table III
RATIO OF COVERED TO GENERATED MUTANTS

	Generated mutants	Covered mutants	
trove	72,959	6,016	(8.2%)
chart	68,519	35,659	(52.0%)
itext	126,781	16,521	(13.0%)
math	66,787	59,195	(88.6%)
time	23,781	18,971	(79.8%)
lang	21,074	19,112	(90.7%)
jdom	10,800	9,519	(88.1%)
jaxen	7,132	4,419	(62.0%)
io	7,319	4,170	(57.0%)
num4j	5,437	5,149	(94.7%)
total	410,589	178,731	(43.5%)

Table IV
ESTIMATED OVERHEAD IN HOURS FOR EVALUATING UNCOVERED MUTANTS

	Uncovered mutants	Test runtime	Overhead
trove	66,943	15.2 sec	282.6 h
chart	32,860	27.3 sec	249.2 h
itext	110,260	8.4 sec	257.3 h
math	7,592	76.2 sec	160.7 h
time	4,810	13.8 sec	18.4 h
lang	1,962	14.1 sec	7.7 h
jdom	1,281	30.4 sec	10.8 h
jaxen	2,713	12.1 sec	9.1 h
io	3,149	17.4 sec	15.2 h
num4j	288	1.8 sec	0.1 h

43.5% in total. The reason for the extremely low mutation coverage for the *trove* application is that it contains a lot of generated source files, of which not all are tested by the test suite. Overall, the coverage results indicate a considerable potential for runtime improvements by excluding uncovered mutants from the mutation analysis (cf. [15]).

Due to the necessary conditions to detect a fault, a test suite cannot kill a mutant that it does not cover. Hence, a mutation analysis process that does not employ coverage information would have to execute the entire test suite for all of the uncovered mutants. In order to estimate the overhead originating from uncovered mutants, that is running the mutation analysis without coverage information, we use the test suite's runtime and the number of mutants that are not covered by the corresponding test suite. Table IV shows the corresponding results for all applications. In this table, the overhead of the first three applications, which is more than ten days, is huge because of the fact that an enormous number of mutants is not covered. Even though the mutation coverage for the *math* application yields an acceptable ratio of 88.6%, the overhead of 160 hours caused by the long test runtime is still prohibitive. It is important to state that the total runtime of a mutation analysis process would include the estimated overhead and additionally the runtime necessary to analyze all covered mutants. Thus, mutation analysis for large real-world applications is not feasible without mutation coverage information. Therefore, we always exploit this coverage information in the subsequent analyses.

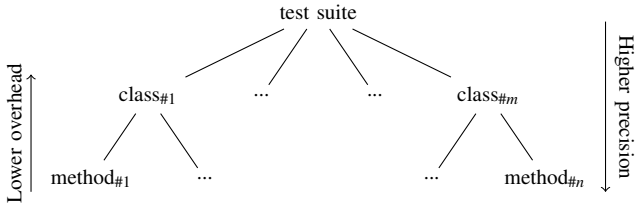

Figure 1. Different levels of granularity in JUnit test suites

B. Precision of the Mutation Coverage

A test suite of JUnit tests is typically a hierarchical composition of test classes containing several test methods. Regarding such a composed test suite, as visualized in Figure 1, there are three different levels of granularity at which the mutation coverage can be measured. The highest one, with a coarse granularity, is the test suite level at which the mutation coverage determines which mutants are covered by the entire test suite. Considering the individual test classes, or even test methods, provides a finer level of granularity, leading to a higher precision in terms of the mutation coverage measure. However, executing test methods independently incurs a much higher overhead caused by additional class loading and, moreover, the instantiation and initialization of the corresponding test classes and the SUT.

Tables V and VI show the differences in the total runtime and the number of covered mutants when executing the test suite at the class and method level. For all applications, the maximum number of mutants covered by a single test class is clearly lower than the number of mutants covered by the entire test suite, as indicated by the sixth column of Table V. Thus, a mutation analysis process should always operate at minimum at the class level. The average numbers of covered mutants in the last column show that certain mutants have to be covered several times since the product of the average number and the number of tests is greater than the total number of covered mutants given in Table III.

At the method level, the number of covered mutants is in turn lower than the class level for almost all of the applications — and yet, the results are divergent. While *io* and *lang* exhibit a significant reduction, the maximum number of covered mutants for *itext* remains unchanged. Moreover, the average coverage per method is even higher for *itext* and *jaxen*, thus indicating that there are a lot of methods that cover numerous mutants. This result again implies a remarkable overlap in terms of the mutation coverage. Generally, the method level provides the most precise mutation coverage information. However, running the test methods independently leads to a higher overhead in terms of runtime, as previously stated. As shown in the fourth column of Table VI, some applications such as *trove*, *math*, *io*, and *num4j* exhibit a moderate overhead compared to the runtime at the class level in Table V. In contrast, *time*, *jdom*, and *jaxen* incur a significant increase in runtime. Hence, the results clearly document the existing tradeoff between precision and runtime overhead.

Table V
PRECISION OF MUTATION COVERAGE AND TOTAL RUNTIME AT CLASS LEVEL

	Mutants	Tests	Runtime	Covered per class Min	Max	Avg
trove	6,016	25	15.2 sec	41	2,150	954
chart	35,659	353	27.3 sec	1	4,702	665
itext	16,521	26	8.4 sec	94	9,537	3,906
math	59,195	234	76.2 sec	1	5,957	769
time	18,970	123	13.8 sec	37	6,032	3,011
lang	19,112	101	14.1 sec	1	2,437	310
jdom	9,519	78	30.4 sec	1	3,715	777
jaxen	4,419	78	12.1 sec	1	3,769	1,895
io	4,170	48	17.4 sec	1	2,474	134
num4j	5,149	63	1.8 sec	18	654	195

Table VI
PRECISION OF MUTATION COVERAGE AND TOTAL RUNTIME AT METHOD LEVEL

	Mutants	Tests	Runtime	Covered per method Min	Max	Avg
trove	6,016	544	16.8 sec	2	1,053	516
chart	35,659	2,130	80.2 sec	1	3,599	293
itext	16,521	75	18.3 sec	70	9,537	4,861
math	59,195	2,169	138.8 sec	1	3,606	381
time	18,970	3,855	335.4 sec	1	4,939	1,636
lang	19,112	2,039	43.5 sec	1	780	87
jdom	9,519	1,723	127.1 sec	1	3,418	362
jaxen	4,419	699	60.9 sec	1	2,847	2,156
io	4,170	309	19.7 sec	1	598	68
num4j	5,149	218	3.2 sec	4	654	100

C. Overlap of the Mutation Coverage

The previous section showed that the investigated applications exhibit an overlap in the coverage of mutants. Therefore, we measure this overlap of the individual test classes. Due to the combinatorial explosion of pairwise comparisons between individual test classes, we focus on relating test classes to their encapsulating test suite and define the overlap $O(t_i, T)$ of a certain test class t_i with its corresponding test suite T as follows:

Definition 1. *Overlap* $O(t_i, T) \in [0, 1],\ t_i \in T$

$$O(t_i, T) := \begin{cases} 1, & |Cov(t_i)| = 0 \\ \frac{|Cov(t_i) \cap Cov(T \setminus t_i)|}{|Cov(t_i)|}, & |Cov(t_i)| > 0 \end{cases}$$

In this definition, the set T denotes a test suite containing all its test classes t_x, meaning that $T := \cup t_x$. Without loss of generality, the definition assumes that the test class t_i, of which the overlap is determined, is an element of the set T. Moreover, the operator $|s|$ represents the cardinality of the set s and the function Cov provides the set of mutants covered by the corresponding set of test classes. Intuitively, this overlap metric describes the similarity of a test class to all other test classes within the same test suite.

Figure 2 illustrates the distribution of the overlap for all analyzed applications using a box-and-whisker plot, where the thick line in the middle represents the median. The box itself shows the distribution of the data between the upper and lower quartile, thus including 50% of the data. By excluding the outlier values, the lower and upper whiskers denote the minimum and maximum value, respectively. The extreme values themselves are visualized by means of the circles beyond the whiskers. Within the plot, two exceptional patterns can be identified. On the one hand, there are applications such as *jaxen* and *time*, where even the minimum overlap of the outlier values is at least 50% and the median is almost 100%. On the other hand, the median of the overlap is only 75% for applications such as *num4j* and *lang*. Moreover, these programs contain test classes that have no overlap at all, indicated by an overlap value of 0%.

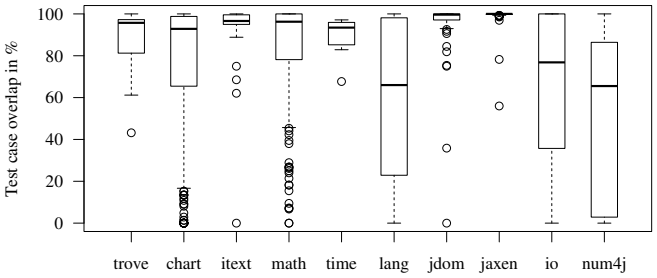

Figure 2. Coverage overlap distribution of the individual test classes related to the corresponding test suite for all investigated applications

Recalling the three conditions for killing a mutant, a high overlap of the mutation coverage does not imply that the test cases within the test suite are highly redundant since the mutation coverage only refers to the execution condition. However, the probability of killing a mutant is much higher if the mutant is covered by several test classes.

D. Runtime of Test Cases

Intuitively, the runtime of a test suite has an essential impact on the total time needed for the mutation analysis because every covered, and yet not killed, mutant has to be evaluated by executing the test suite. As previously mentioned, regarding the test suite at the class or method level leads to more precise mutation coverage information, which reduces the number of mutants that have to be evaluated for a certain test. Nevertheless, a very long-running test case can still result in a prolonged mutation analysis, even for a small number of covered mutants. Hence, the number of covered mutants and the runtime of the individual test cases are the determining factors for the total runtime.

Therefore, we investigate how the individual test classes collectively form the total runtime of the corresponding test suite. The runtime distribution of the individual test classes for all analyzed applications is again visualized by means of box-and-whisker plots in Figure 3. The extremely thin boxes and the short, if existing, whiskers clearly indicate that most of the test classes have a short runtime of less than 1 second. Even though the majority of the test classes have a rather low runtime, there are a few extreme values for which the runtime differs by an order of magnitude. Thus, for all applications, Table VII additionally shows the number of test classes, along with both the cumulative runtime and

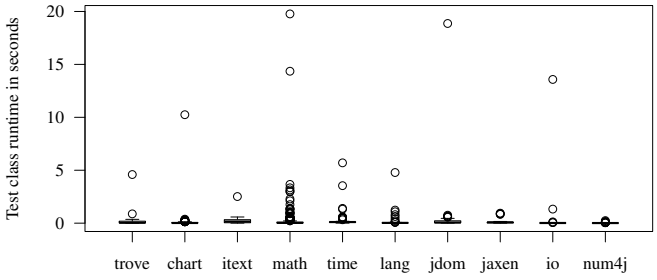

Figure 3. Runtime distribution of the individual test classes

Table VII
CUMULATIVE RUNTIME AND EXTREMUM OF ALL TEST CLASSES

	Test classes	Cumulative*	Extremum*
trove	25	15.2	4.6 (30.3%)
chart	353	27.3	10.2 (37.4%)
itext	78	8.4	2.5 (29.8%)
math	234	76.2	19.8 (26.0%)
time	123	13.8	5.7 (41.3%)
lang	101	14.1	4.8 (34.0%)
jdom	78	30.4	18.9 (62.2%)
jaxen	78	12.1	0.9 (7.4%)
io	48	17.4	13.6 (78.2%)
num4j	63	1.8	0.2 (11.1%)

*Runtimes reported in seconds

the extremum of all test classes. This table demonstrates that the outlier values constitute a substantial proportion, sometimes even most, of the total runtime. For instance, in consideration of a total number of 353 test classes for the *chart* application, the test class with the longest runtime of 10.2 seconds forms more than 37% of the total runtime.

E. Visualizing the Overlap and Runtime

In order to examine the correlation between the runtime and the mutation coverage overlap of the tests, we use scatter plots to visualize the overlap in conjunction with the individual test runtime. Due to the large number of analyzed applications, and to avoid a confusing set of diagrams, we focus on the two identified overlap patterns as representatives for all of the chosen applications. Figure 4 shows the plot for *time*, an application representative of those with high overlap, while Figure 5 gives the plot for *num4j* as a representative of those applications with a distinctive distribution in the coverage overlap.

Within the scatter plots, every data point indicates that the mutant on the vertical axis is covered by the test class or test classes on the horizontal axis. Intuitively, the plot visualizes a matrix representing a mutant-covered-by-test-class map. Since the introduced overlap metric is a measure for the similarity of each data line to all others, the plot for *time* clearly reveals that there is indeed a substantial overlap for almost all of the test classes. In contrast to this obvious overlap, the plot for *num4j* reveals that only a fourth of the mutants are overlapped by several test classes.

Besides the overlap, the plots visualize the runtime of the individual test classes. The test runtime in milliseconds is

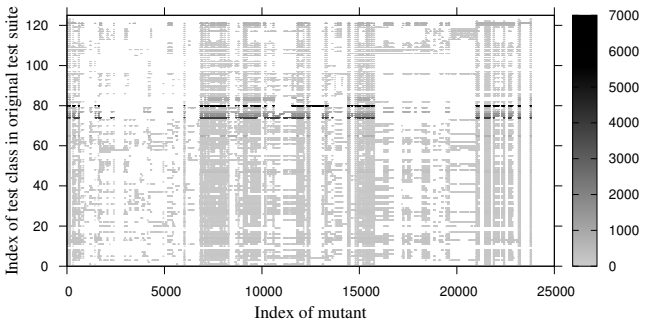

Figure 4. Mutation coverage with corresponding runtime for *time*

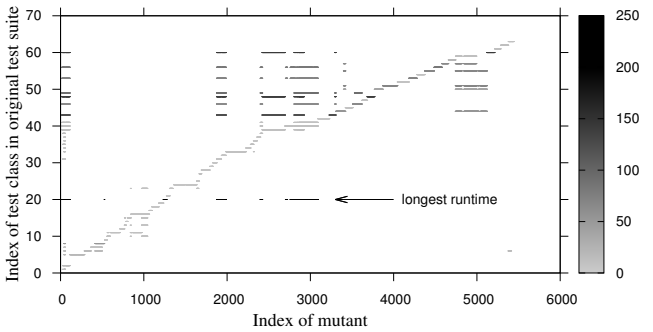

Figure 5. Mutation coverage with corresponding runtime for *num4j*

color coded by means of the gray-scale palette that is aligned to the right of the scatter plot. According to the chosen gray-scale gradient, the darker the color of a data line, the longer is the runtime of the corresponding test class. Both plots manifest that long-running tests, represented by the dark data lines, have a notable overlap with a lot of short-running tests. Moreover, when considering the original ordering of the tests for *num4j*, the test class with the longest runtime is placed before many of the overlapping short-running test classes. As a consequence, the long-running test has to be executed for all covered, and yet not killed, mutants — even though those mutants are also covered by test classes with a much shorter runtime. Since the results from the other applications conform to these observations, the characteristics of these plots suggest that a reordering according to the runtime could significantly improve the mutation analysis process.

V. OPTIMIZED MUTATION ANALYSIS WORKFLOW

We now present an optimized mutation analysis workflow based on the observations and evidence given in the previous Section IV, which can be summarized as follows:

OBS 1. *The mutation coverage is 43.5% on average and ranging between 8.2% and 94.7%.*

OBS 2. *Analyzing the mutation coverage on the class or method level is much more precise. However, executing the test methods independently incurs a significant overhead.*

OBS 3. *Most tests within a test suite have a notable overlap with all remaining tests in terms of mutation coverage.*

OBS 4. *The runtime of individual test classes within a test suite differs, sometimes even by an order of magnitude.*

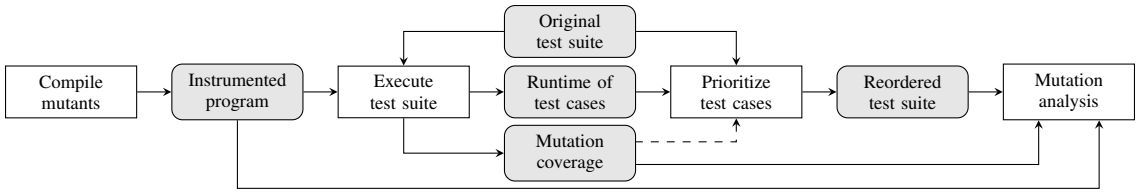

Figure 6. Optimized mutation analysis process that exploits mutation coverage and runtime information of test cases

OBS 5. *Long-running tests have an essential overlap with many short-running tests but the investigated existing JUnit test suites are not ordered according to runtime.*

A. Gather Mutation Coverage Information

Due to the fact that a lot of mutants are not covered by the test suite, we exploit the mutation coverage information provided by MAJOR's driver [11]. A program instrumented by MAJOR reports the coverage information to the driver if and only if the unmutated version is executed and the corresponding flag is enabled. This condition is crucial since gathering the coverage information involves method calls and incurs a notable overhead (cf. [9]). Hence, determining the coverage information during the mutation analysis process would significantly increase the total runtime. Depending on the level of granularity, the coverage information is cached for each test class or test method.

B. Estimate Test Runtime and Prioritize Test Cases

Given the overlap of the individual tests and the runtime, which differs significantly, the runtime is estimated for every test method and entire class by executing the original version. Attempting to produce a runtime approximation that is as precise as possible, this step executes the original version of the program without enabling the mutation coverage.

Next, based on the runtime results, the tests are sorted in descending order to ensure that the long-running tests will be executed last. This prioritization strategy is based on the assumption that unit tests have no dependencies, and hence the order is irrelevant. Even though this assumption is also specified for unit testing frameworks such as JUnit, we verify that reordering the tests does not break the test suite by executing the unmutated version with the prioritized test suite. In order to increase the confidence in the test's independence, a randomized order of the tests is also executed.

C. Threshold-based Splitting of Test Classes

In light of the tradeoff between precision of the mutation coverage on the one hand and runtime overhead on the other hand, we present two hybrid approaches. Extracting test methods with an exceptionally long runtime or splitting entire long-running test classes seems to be the most promising. Therefore, we define the following two hybrid approaches that represent both kinds of splitting strategies.

Class-hybrid: Extract an individual test method from its corresponding test class if and only if the runtime of the pre-initialized test method is greater than a given threshold th_m.

Method-hybrid: Split a test class into its individual test methods if and only if the total runtime of the pre-initialized test class is greater than a given threshold th_c.

With regard to the threshold parameters that are used in both approaches, we determine an appropriate value by taking into account the average initialization time of a test class for th_m and the mean number of test classes within a test suite for th_c. Additionally, based on the observation that the test suites of all applications exhibit a significant mutation coverage overlap, the overhead of executing a test method separately should not exceed 100% of its runtime. For instance, the average initialization time of a test class ranges between 10 and 50 milliseconds for all applications. Thus, a threshold th_m of 50 milliseconds ensures that an individual test method is not extracted if its runtime is smaller than the initialization time of the enclosing test class.

D. Complete Mutation Analysis Workflow

Integrating the individual steps into a complete workflow leads to the optimized mutation analysis process that is illustrated in Figure 6. Generally, this process consists of three individual but consecutive phases:

1) Mutant generation phase that generates and compiles all mutants into the system under test.
2) Preprocessing step that gathers the mutation coverage and test case runtime information.
3) Mutation analysis with reordered and potentially split test suite employing the mutation coverage.

The dashed line within the diagram indicates that the threshold for the splitting strategy is estimated based on the mutation coverage overlap. It is important to note that we do not calculate the overlap of every test method with its test class but rather use the overlap of the test classes within a corresponding test suite. The splitting strategy may be more effective with an accurate overlap value for test methods within their encapsulating test classes. However, we leave the investigation of this matter open for future work.

VI. Empirical Evaluation

To empirically evaluate the workflow shown in Figure 6, we implemented it in MAJOR's analysis component [9], [11] that extends the Apache ant build system. With regard to the performance evaluation, of particular interest are the runtime improvements due to reordering and splitting and the variation in efficiency due to differences in the coverage overlap and the effectiveness of the test suite. Table VIII

Table VIII
RUNTIMES FOR DIFFERENT PRIORITIZATION AND SPLITTING STRATEGIES

	*Original	*Method-level		*Method-hybrid[1]		*Class-level		*Class-hybrid[2]		Mutation score
trove	107.81	41.68	(-61.3%)	44.93	(-58.3%)	55.96	(-48.1%)	36.89	(-65.8%)	66.6%
chart	608.60	950.55	(56.2%)	564.14	(-7.3%)	270.40	(-55.6%)	309.88	(-49.1%)	36.3%
itext	644.43	1381.51	(114.4%)	1127.25	(74.9%)	627.89	(-2.6%)	674.18	(4.6%)	24.0%
math	793.19	394.60	(-50.3%)	388.39	(-51.0%)	674.73	(-14.9%)	381.10	(-52.0%)	79.1%
time	504.44	1182.61	(134.4%)	559.62	(10.9%)	470.03	(-6.8%)	410.59	(-18.6%)	85.7%
lang	42.75	27.58	(-35.5%)	23.31	(-45.5%)	29.93	(-30.0%)	19.11	(-55.3%)	74.2%
jdom	120.53	135.53	(12.4%)	189.08	(56.9%)	117.42	(-2.6%)	105.01	(-12.9%)	83.4%
jaxen	343.40	1773.15	(416.4%)	1521.79	(343.2%)	338.51	(-1.4%)	357.16	(4.0%)	43.6%
io	5.72	5.64	(-1.5%)	4.11	(-28.1%)	4.83	(-15.5%)	4.35	(-23.9%)	78.0%
num4j	2.54	2.12	(-16.5%)	1.95	(-23.1%)	1.99	(-21.6%)	1.94	(-23.5%)	68.1%
avg			56.9%		27.3%		-19.9%		-29.2%	63.9%

*Runtimes reported in minutes; [1]Threshold th_c = 500 milliseconds; [2]Threshold th_m = 50 milliseconds

reports the runtimes for the complete mutation analysis when employing the different reordering and splitting strategies for all applications. To better visualize the results, the fastest approach is highlighted for every application.

In order to minimize any potential side effects, all analyses were performed on a single machine[1] that did not take advantage of parallelization. Additionally, we measured the real runtime instead of CPU time due to the fact that most analyzed applications are not CPU-bound. Hence, the CPU time is much lower and does not adequately reflect the time needed to perform the entire mutation analysis.

Within the table, Original denotes the mutation analysis of the test suite without any prioritization or splitting. Method-level and Class-level describe the results for sorting and executing the test suite at the method level and class level, respectively. The runtime results for the two suggested approaches, namely Method-hybrid and Class-hybrid, are shown in the corresponding columns. The last column of Table VIII additionally shows the mutation score since the effectiveness of the investigated test suites is also a crucial factor. It is important to consider the mutation score because the prioritization technique is based on the assumption that a test suite kills a certain number of mutants, and hence the number of live mutants decreases over time. Furthermore, a mutant is always killed by the fastest test case that can detect it within the sorted test suite. Thus, if the mutation score is extremely low, reordering will only yield a marginal improvement since the entire test suite has to be executed for the majority of the mutants.

When analyzing the entire test suite at the method level, meaning that every test method is executed independently, the two identified overlap patterns give distinction to the results. The runtime for the applications *itext*, *time*, and *jaxen*, which have a huge overlap, is increasing dramatically due to the incurred overhead. Yet, applications with a lower overhead such as *trove* and *math* yield a considerable runtime decrease of up to 61.3%.

[1]Commodity GNU/Linux workstation with Intel Xeon CPU @2.4GHz, 16GB of RAM, and kernel version 2.6.32-5-amd64.

The Method-hybrid approach reduces the number of individual test methods by only splitting long-running test classes. Due to the reduction, this approach improves the runtime for all applications but cannot compensate for the huge overhead of *itext*, *time*, and *jaxen*.

Sorting the test suite at the class level according to the runtime of the individual test classes yields an improvement for all of the applications due to the existing overlap between test classes and the divergent runtimes of the individual tests. Ranging between 1.4% for *jaxen* and 55.6% for *chart*, sorting at class level yields an average decrease of almost 20%. The improvement for *jaxen* is relatively low for two reasons. The runtime of the individual test classes is very homogeneous with a maximum of only 900 milliseconds and furthermore the mutation score is only 43.6%.

By additionally employing the splitting of the Class-hybrid approach, the runtime can be considerably reduced even further for most of the applications. With a speedup of 65.8% for *trove* and an average improvement of 29.2% for all applications, this approach yields the best results overall. However, the necessary runtime increases by approximately 4% for two applications, namely *itext* and *jaxen*. The reason is again the low mutation score in conjunction with the huge overlap of the individual test classes. For instance, some long-running test methods of *itext* cover exactly the same number of mutants as the entire, enclosing test class. Hence, the extraction of such methods introduces a higher overhead without increasing the precision of the mutation coverage.

For the *math* and *itext* applications, the diagrams in Figure 7 visualize the mutation analysis process using the original test suite and the Class-hybrid approach, which is the most efficient approach for all applications. Within these diagrams, the upper plot illustrates the runtime of the individual test cases and the lower plot depicts the ratio of analyzed-to-covered mutants. This lower-is-better ratio decreases if mutants covered by a certain test are already killed, and hence will not be executed again. Additionally, the width of the boxes exhibits the time needed to execute the corresponding test for all of the covered, and yet not killed, mutants.

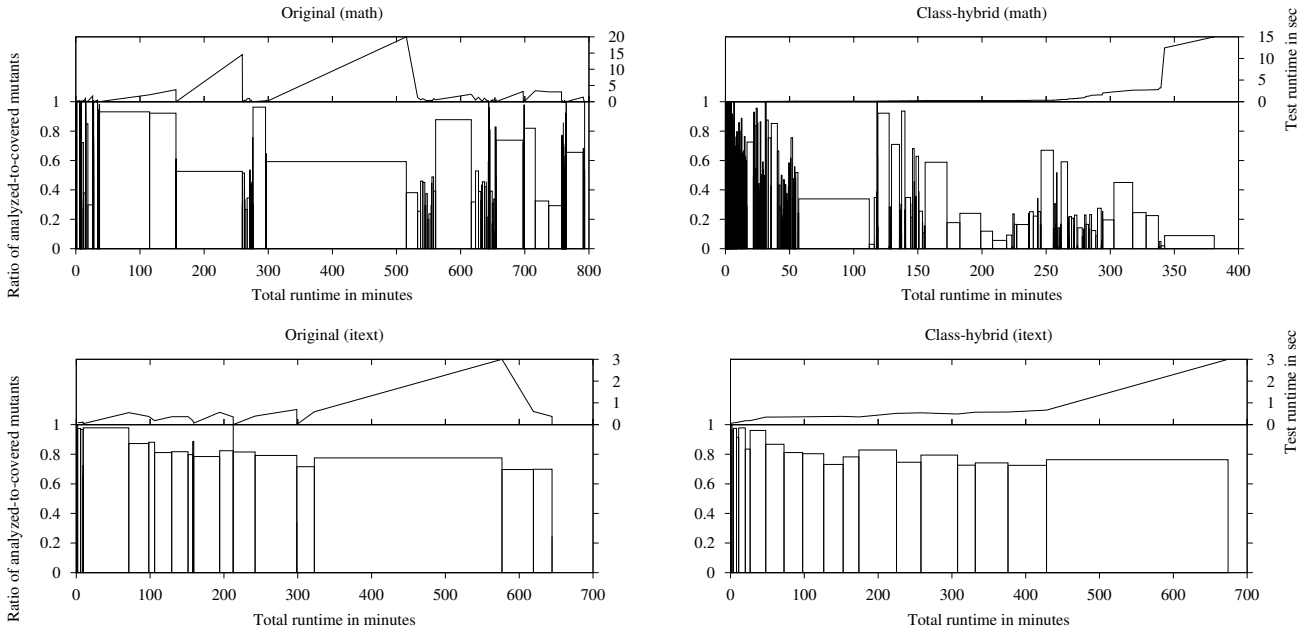

Figure 7. Visualization of the complete mutation analysis process using the Class-hybrid approach

With regard to the original test suite of the *math* application, the test class with the longest runtime is placed in the middle and the time necessary to execute this individual test class for all covered mutants is a considerable proportion of the total runtime of the complete process. Furthermore, the ratio of analyzed-to-covered mutants is still 60%. By means of the Class-hybrid approach, this long-running test class is split, so that the methods with a runtime larger than the threshold are extracted and the resulting set of individual tests is sorted. Accordingly, the last individual test possesses the longest runtime which is, however, with 16 seconds, smaller than the entire long-running test class from which it has been extracted. Given the coverage overlap of the tests and a mutation score of almost 80%, the ratio of analyzed-to-covered mutants is rapidly decreasing, and hence the extracted long-running test method is only executed for approximately 10% of the covered mutants.

An example of an application with both an extremely low mutation score and a better-ordered original test suite is *itext*, for which Figure 7 illustrates the mutation analysis process. The runtime characteristics of the other applications conform to these two examples, and thus are not separately depicted.

As with every empirical study, it is crucial to discuss the *threats to validity*. The representativeness of the chosen applications might be a potential threat to external validity. However, we controlled this threat by examining programs that differ significantly in their size, complexity, and operation purpose. Starting with an initial set of the five applications *math*, *time*, *lang*, *io*, and *num4j*, we improved the generalizability by successively adding new applications with varying operation purposes and originations from different developer communities. While adding more programs to the empirical study, we could identify the two exceptional overlap patterns to which all of the investigated applications can be related. Therefore, we judge that the reported results are meaningful and indeed transferable to other applications.

Relying on the sufficient set of mutation operators could be a threat to internal validity. Different or additional operators may affect the improvement results due to differences in the mutation coverage overlap and the mutation score. However, the applied operators are frequently used in the literature, and thus provide comparable results [14], [16]. Moreover, the investigated test suites exhibit a significant divergence with regard to the mutation score and therefore the study also examines boundary cases.

A threat to construct validity could be a defect in the chosen mutation testing framework. Since we employed the framework to conduct previous experiments [9], [10] without encountering any problems and we also verified the results for example programs in a manual fashion, we judge that the implementation worked correctly.

VII. Related Work

Considering the efficiency of mutation analysis, several approaches proposed in the literature belong to one of three categories: *do fewer*, *do smarter*, and *do faster* [8], [14]. *Do fewer* approaches employ selective or sampling strategies to reduce the number of generated mutants by either (randomly) selecting a subset of all generated mutants or by reducing the number of applied mutation operators. Namin et al. [16], as well as Offutt et al. [13], determined a sufficient set of mutation operators that can be applied without a major loss of information.

Concerning this relationship between reduction and accuracy, the utilization of non-redundant mutation operators is

a special case of a *do fewer* approach since the exclusion of subsumed mutants even increases the accuracy of the mutation score (cf. [10]). As already stated in Section III, Kaminski et al. established a subsumption hierarchy for a non-redundant ROR mutation operator [12] and we identified a non-redundant set of mutations for the COR and UOI operators in our prior work [10].

While *do smarter* methods exploit distributed or multi-core systems to parallelize the mutation analysis, *do faster* approaches aim at improving the efficiency without reduction or parallelization. Even though this paper's optimized workflow belongs to the group of *do faster* techniques, it can nevertheless be combined with sampling or selective approaches and it also can be easily parallelized to fully utilize the computational power of current machines.

To the best of our knowledge, this paper is the first to employ test suite prioritization to achieve efficient and scalable mutation analysis. However, a wide variety of software testing techniques leverage mutation analysis. For instance, Elbaum et al. use mutation testing tools to support the reordering of a test suite according to the fault exposing potential of a test case [5]. Finally, both Andrews et al. and Do and Rothermel use mutation analysis to empirically evaluate test suite prioritization techniques [1], [4].

VIII. Conclusions and Future Work

This paper makes several contributions that address the challenges associated with achieving efficient and scalable mutation analysis. By leveraging existing definitions of non-redundant COR and ROR mutation operators, an empirical evaluation shows that the reduced, yet sufficient, sets of COR and ROR mutants also result in a significantly decreased total number of generated mutants. Drawing on several observations concerning the characteristics of real-world test suites, this paper also presents an optimized mutation analysis process that exploits mutation coverage and test runtime information. Empirically evaluated on ten open-source applications with 410,000 lines of code and 550,000 generated mutants in total, the suggested approach reduces the runtime by up to 65% and 30% on average. Overall, this paper convincingly demonstrates that mutation analysis is indeed applicable to large programs, and hence ready for a transfer and a wider integration into industry.

Since the proposed splitting approaches are parameterized with a threshold that is currently determined with a heuristic based on the test class initialization time, we will conduct further experiments to compare our approaches with the optimally sorted test suite. It is important to note that the identification of this suite implies a tremendous computational expense since every test case has to be executed for every covered mutant to identify the fastest test cases that not only cover but also kill the mutants. Additionally, we plan as part of our future work to implement a splitting approach that also takes into account the mutation coverage overlap at the method level. A splitting technique that extracts long-running test methods, when the precision of the mutation coverage adequately increases, might be more sensitive to test suites that exhibit a substantial overlap.

Because of the promising results achieved by employing the non-redundant versions of the COR and ROR mutation operators, we are currently investigating redundancies in the mutation set for replacing arithmetic operators (AOR). Since AOR mutants also constitute a significant proportion of all generated mutants, a reduction would noticeably increase the efficiency of mutation analysis. Combining the techniques presented in this paper with our future work will result in a complete, efficient, scalable, and thoroughly-studied framework for the mutation analysis of Java programs.

References

[1] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proc. of the 27th ICSE*, 2005.

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

[3] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Comp.*, 11(4), 1978.

[4] H. Do and G. Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Trans. on Soft. Engin.*, 32(9), Sept. 2006.

[5] S. Elbaum, A. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Trans. on Soft. Engin.*, 28(2), 2002.

[6] G. Fraser and A. Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proc. of the 8th ESEC/FSE*. ACM, 2011.

[7] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. In *Proc. of the 19th ISSTA*, 2010.

[8] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. on Soft. Engin.*, 37(5), 2011.

[9] R. Just, G. M. Kapfhammer, and F. Schweiggert. Using conditional mutation to increase the efficiency of mutation analysis. In *Proc. of the 6th AST*, 2011.

[10] R. Just, G. M. Kapfhammer, and F. Schweiggert. Do redundant mutants affect the effectiveness and efficiency of mutation analysis? In *Proc. of the 7th Mutation*, 2012.

[11] R. Just, F. Schweiggert, and G. M. Kapfhammer. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *Proc. of the 26th ASE*, 2011.

[12] G. Kaminski, P. Ammann, and J. Offutt. Better predicate testing. In *Proc. of the 6th AST*, 2011.

[13] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. on Soft. Engin. and Method.*, 5(2), 1996.

[14] J. Offutt and R. H. Untch. Mutation 2000: Uniting the orthogonal. In *Proc. of Mutation 2000*, 2000.

[15] D. Schuler, V. Dallmeier, and A. Zeller. Efficient mutation testing by checking invariant violations. In *Proc. of the 18th ISSTA*, 2009.

[16] A. Siami Namin, J. H. Andrews, and D. J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Proc. of the 30th ICSE*, 2008.

[17] J. M. Voas. PIE: a dynamic failure-based technique. *IEEE Trans. on Soft. Engin.*, 18, 1992.