Using Conditional Mutation to Increase the Efficiency of Mutation Analysis

René Just
Department of Applied Information Processing
Ulm University
rene.just@uni-ulm.de

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

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

ABSTRACT

Assessing testing strategies and test sets is a crucial part of software testing. Mutation analysis is, among other approaches, a suitable technique for this purpose. However, compared with other methods it is rather time-consuming and applying mutation analysis to large software systems is still problematic. This paper presents a versatile approach, called *conditional mutation*, which increases the efficiency of mutation analysis. This new method significantly reduces the time overhead for generating and executing the mutants. Results are reported for eight investigated programs up to 373,000 lines of code and 406,000 generated mutants. Furthermore, *conditional mutation* has been integrated into the Java 6 Standard Edition compiler. Thus, it is widely applicable and not limited to a certain testing tool or framework.

Categories and Subject Descriptors

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

General Terms

Experimentation, Verification

Keywords

Mutation Analysis, Instrumentation, Compiler-integrated

1. INTRODUCTION

Software testing is an essential part of the software development process. Since software systems are growing in size and complexity, tests should be automated in order to achieve sufficient test results. Automating software tests is a challenging task which concerns not only the automated processing of test cases but also the generation of input values and the evaluation of the resulting outputs by means of an oracle or partial oracle [2, 3]. Considering these aspects, it is clear that an evaluation of the input values as well as the (partial) oracles is necessary in order to achieve reliable results from testing. Mutation analysis is among other approaches suitable for this purpose [1]. However, applying mutation analysis to large software systems is problematic since it is a time-consuming technique.

This paper describes and evaluates *conditional mutation*, an approach to increase the efficiency of mutation analysis. It is based on transforming the abstract syntax tree (AST) to collect all mutants in conjunction with the original program within the resulting assembled code. The name is derived from the conditional statements and expressions which are inserted to encapsulate the mutants. In comparison to prior approaches (e.g., [12, 19, 21]) it is more general because it operates at the source code level on expressions as well as statements. Furthermore, these AST transformations can be integrated into the compiler.

In consideration of this compiler-integrated approach, the runtime to generate and compile the mutants is reduced to a minimum. For instance, the total overhead for compiling 406,000 mutants for the largest investigated program, namely the aspectj compiler, is only 33% compared with the default compile time. This time overhead includes the cost of both mutant generation and compilation and is thus orders of magnitude less than compiling the mutants individually. In addition, the time to run the full corresponding test suites for the instrumented programs is on average only 15% higher than normal testing time in the worst-case scenario associated with executing all of the conditional statements and expressions that support mutation analysis.

The remainder of this paper is structured as follows: Section 2 deals with the basics and outlines the challenges associated with mutation analysis. Related work is discussed in Section 3 and our approach is explained afterwards with implementation details in Section 4. Thereafter, an empirical study with eight software systems is presented to evaluate the approach and the implementation in Section 5. Finally, potential threats to validity are discussed in Section 6 and a conclusion is given in Section 7.

2. MUTATION ANALYSIS

Originally introduced in [4, 7], mutation analysis is a well known technique for assessing the quality of a given test suite or testing strategy. In this approach, faults are systematically seeded into a System Under Test (SUT) and the corresponding test sets or testing strategies are examined with respect to their ability to find the injected faults. These faults are injected methodically, contrary to the classical approach where error seeding is led by the intuition of experienced engineers (cf. [13]). Thus, mutation analysis can be regarded as an unbiased technique.

AST '11, May 23-24, 2011, Waikiki, Honolulu, HI, USA

The way of applying mutation analysis is specified by *mutation operators* and the resulting faulty versions of the SUT are referred to as *mutants*. Examples of mutation operators are the replacement of variables by constants or swapping comparison operators in conditions [11, 12]. Generally, mutation analysis is programming language independent but the mutation operators depend on the chosen language since they have to cover the corresponding syntax and semantics. It also has to be pointed out that mutation analysis is not feasible without appropriate tool support.

Due to the methodical injection of faults, the obtained mutants are reproducible and the method is applicable for an arbitrarily selected SUT. According to the basic hypotheses, namely the *competent programmer hypothesis* and the *coupling effect* [15], a mutant contains only one mutation, thus leading to a large number of mutants when all possible mutation operators are applied to a large or complex SUT.

If a test case reveals a fault, the corresponding mutant is said to be *killed*. Hence, relating the number of all the killed mutants to the number of generated mutants is an appropriate way to measure the quality of the applied test suite. However, a mutant cannot be killed in certain circumstances. In fact, when there exists no test case that can detect the mutant, it is said to be *equivalent*. Thus, it might be necessary to investigate and remove possibly equivalent mutants manually. Yet, approaches exist to identify some equivalent mutants automatically [17, 19]. Since our focus is the efficient automatic generation and execution of mutants, these techniques are not discussed further in this paper.

3. RELATED WORK

As already mentioned in Section 2, applying all suitable mutation operators to large or complex implementations leads to a huge number of mutants, thus having a considerable impact on the runtime of mutation analysis. Considering the computational costs of mutation analysis, several approaches have been discussed in the literature (cf. [9]). As noted by Offutt and Untch, all techniques and strategies aiming at reducing these costs can be related to one of three categories: do fewer, do smarter, and do faster [18].

Do fewer approaches are sampling and selective techniques that reduce the number of mutants either by decreasing the number of operators or by selecting just a subset of the generated mutants. Offutt et al. investigated the effectiveness of mutation operators and determined that the mutants generated with a smaller subset of sufficient operators are almost as hard to kill as the complete set of mutants [16]. Thus, the reduced set of mutation operators can be applied much more efficiently without a major loss of information.

Do smarter techniques exploit for instance the possibilities of running mutation analysis in a distributed environment [5]. Since every mutant is generated independently, the computation can be parallelized. Another do smarter approach is weak mutation testing [8] where a mutant is said to be killed if its internal state, after executing the mutated code, differs from the original program. Hence, only necessary conditions can be verified by applying weak mutation. Nevertheless, the minor loss of information is proportionate to the considerable decrease in time overhead.

Do faster approaches generally aim at improving the runtime of mutation analysis without using reduction or parallelization. Considering the conventional way of applying mutation analysis, every mutant is a copy of the original program apart from a small syntactical change. According to that fact, compiling every mutant as an independent source file is a substantial time overhead during compilation. In order to alleviate the costs of compiling every mutant, DeMillo et al. proposed a compiler-integrated technique [6]. They modified a compiler to create patches on the machine code level. Thus, the original program was compiled just once and the mutants were created by applying corresponding patches to the compiled original program. However, the effort to implement or adapt the necessary compiler is significant.

With respect to the Java programming language, which uses an intermediate language, bytecode transformation is a similar technique which directly transforms compiled code. While these modifications are usually faster than source code transformations since they obviate additional compilation steps, there are still some drawbacks to this approach. First of all, the bytecode has been simplified or even optimized during the compilation process. Therefore, errors might be injected which could never have been introduced by a programmer at the source code level. Additionally, all semantic information collected during compilation phases, such as building and attributing the abstract syntax tree, has to be gathered redundantly by parsing the bytecode again.

Mutant schemata is another do faster approach which encodes all mutations within generic methods and replaces the original instructions, which shall be mutated, with a call of the corresponding generic method. Accordingly, the necessary time to compile all mutants is reduced. As described by Untch et al., *"A mutant schema has two components, a metamutant and a metaprocedure set"* [21]. The effective mutation of the metamutant is determined at runtime within the generic methods by means of appropriate flags. An example for mutant schemata is the replacement of the built-in arithmetic operators by a generic method `AOP`:

`int a = AOP(b, c);` $\longmapsto$ `int a = b + c;`

`AOP` can now perform a different arithmetic operation at runtime which leads to a mutation of the original statement. Generally, the creation of metamutants can be regarded as a template-based technique. However, the introduction of several indirections, when implemented with method calls [21], implies an additional overhead which may have a significant impact on the runtime of the compiled program.

Higher order mutation is, in addition to these three categories, another approach to generate fewer, but more subtle mutants. Generating mutants by means of the combination of two simple mutants, called first order mutants, is referred to as second order mutation. Accordingly, higher order mutation denotes generally the combination of two or more first order mutants. The computational costs for second and higher order mutation are considerably higher because of the huge number of possible combinations. Nevertheless, recent work has shown that second and higher order mutants exist that are harder to kill than the first order mutants of which they have been generated [10]. Hence, applying these higher order mutants (HOMs) would provide a better assessment for mutation analysis. The problem however is to identify appropriate HOMs in an efficient way. Search-based approaches seem to be a feasible solution for this problem [10].

4. CONDITIONAL MUTATION

The conventional way of generating mutants, as for instance implemented in MuJava [14], is a result of the workflow mentioned in Section 2, with a set of source files and the

convention that each file contains exactly one mutant. This approach to mutant creation incurs a high time overhead because it must repeatedly load and compile every mutant file. In addition, the SUT has to be executed repeatedly with every mutant to determine the mutation score. In terms of mutated Java classes, every corresponding class file has to be loaded to execute the mutated code.

The idea of *conditional mutation* is to accomplish all mutations in the corresponding source file and more precisely in the same entity. This means that all mutants are encoded in the same block or scope and within the same method and class as the original piece of code. Hence, a conditional mutant is a valid substitution of an arbitrary instruction based on conditional evaluations and it contains all mutations as well as the original instruction. Thus, every conditional mutant preserves the scope and visibility within the AST.

Regarding the example in Listing 1, we can distinguish between statements and expressions. Intuitively, every program instruction which is terminated, for example with a semicolon, is a statement. An expression is also an instruction but it represents a value which can be or has to be evaluated within a statement, depending on the corresponding language. For instance, `int a = 3` is a statement which can be used as a single instruction. In contrast, the expression `a * x` represents a value and cannot be used as a statement, in the Java programming language, by adding a terminating semicolon. So, an expression is always part of a surrounding statement with the exception of so-called expression statements. These are expressions like method calls, unary increment/decrement operators, or assignment operators which can be used either as an expression or a statement.

The *conditional mutation* approach aims to retain the basic structure of a program's AST. This means that unnecessary local variables, statements, or blocks must not be inserted. Thus, every expression or statement which is to be mutated has to be replaced with an appropriate expression or statement, respectively. Therefore, conditional statements or conditional expressions are inserted where the `THEN` part contains the mutant and the `ELSE` part the original code. The condition for these conditional expressions or statements may contain an arbitrary expression which determines when the mutant should be triggered. Regarding mutation analysis, where every mutant should be executed, the enumeration of all mutants and the insertion of a global mutant identifier e.g., a global variable `M_NO` is advisable. This variable makes it possible to dynamically choose the mutant to be executed. Thus, the expression of the condition is a comparison of the identifier with the mutant's number.

When compared with mutant schemata, *conditional mutation* is more general since it handles expressions and statements. Furthermore, not every mutation operator can be implemented by the method call approach described in [21]. For instance, if the parameter passing method is call by value, which is used exclusively in Java, the following replacements are not valid because they do not preserve the semantics of the original statement:

- `int a = OP(b);` $\not\longmapsto$ `int a = ++b;`
- `OP(a,b);` $\not\longmapsto$ `a += b;`

These mutations can be represented with conditional mutants since they exist in the same scope as the original expression. Moreover, while conditional statements can easily express faults of omission (e.g., a forgotten `continue` or `break`), method calls cannot represent this type of mutant.

```
1 public int eval(int x){
2     int a = 3, b = 1, y;
3
4     y = a * x;
5
6     y += b;
7
8     return y;
9 }
```

Listing 1: Method with statements and expressions.

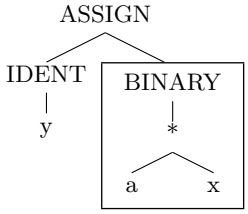

Figure 1: AST subnode of an assignment with a binary expression as right hand side.

Tail-Recursive Algorithm for Conditional Mutation

The proposed algorithm is applicable for both expressions and statements but it is explained based on expressions. In order to apply it to statements, *expr* has to be replaced by *stmt* and a conditional statement *CondStmt* has to be used instead of *CondExpr* within the recursive function (4).

The available mutation operators depend on the expression to be mutated. Thus, the collectivity of all applicable operators can be defined as a set for a certain expression:

$$MOP(expr) = \{mop_1, \cdots, mop_n\}, n \in \mathbb{N} \tag{1}$$

Considering a binary arithmetic expression, examples for the mutation operators mop_i would be the replacement of the arithmetic operator or the permutation of the operands. Next, the syntax tree is traversed and every expression for which at least one mutation operator exists will be replaced by an expression $expr'$:

$$expr' \longleftarrow expr, \forall expr : MOP(expr) \neq \emptyset \tag{2}$$

In order to apply the first k mutation operators given by the set $MOP(expr)$, a recursive algorithm can be defined. In the base case the expression *expr* is replaced by a conditional expression *CondExpr* which contains the condition $cond_1$, the mutant, determined by the evaluation of $mop_1(expr)$, and the original expression *expr*. Any further mutation is encapsulated within a conditional expression which in turn contains the result of the previous mutation step.

$$expr' = mut_k, k \in \mathbb{N} \wedge k \leq n \tag{3}$$

$$mut_i = \begin{cases} CondExpr(cond_1, mop_1(expr), expr), & i = 1 \\ CondExpr(cond_i, mop_i(expr), mut_{i-1}), & i > 1 \end{cases} \tag{4}$$

Since function 4 is tail-recursive it can also be implemented as an iterative algorithm if the compiler of the corresponding programming language does not support tail-recursion elimination. By means of an appropriate ordering of the set $MOP(expr)$ in conjunction with the parameter k, sampling strategies or selective mutation can be applied. Exemplary results of using the algorithm are illustrated in Listing 2 and Figure 2. Reconsidering the assignment `y = a * x`,

```
public int eval(int x){
    int a = 3, b = 1, y;

    y = (M_NO==1)? a - x:
        (M_NO==2)? a + x:
                   a * x; // original expr

    if(M_NO==3){
      y -= b;
    }else{
      y += b; // original stmt
    }

    return y;
}
```

Listing 2: Mutated statement and expression.

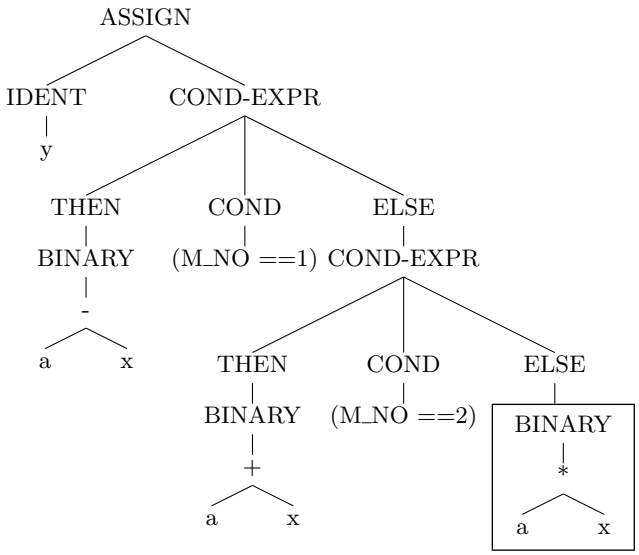

Figure 2: Multiple mutated binary expression as right hand side of an assignment.

```
public int eval(int x){
    int a = 3, b = 1, y;

    y = (M_NO==1)? a - x:
        (M_NO==2)? a + x:
        (M_NO==0 && COVERED(1,2))?
        a * x : a * x; // original expr

    if(M_NO==3){
      y -= b;
    }else{
      if(M_NO==0 && COVERED(3,3)){
          y += b;
      }else{
          y += b; // original stmt
      }
    }

    return y;
}
```

Listing 3: Collecting the coverage information with conditional mutation.

shown in Figure 1, the binary expression which shall be mutated is a subnode of the AST. Hence, just this subnode is replaced in accordance with the *conditional mutation* approach. It has to be pointed out that a possible mutation operator $mop_i \in MOP(expr)$ for an expression $expr \in stmt$ must not occur in the set $MOP(stmt)$ of the surrounding statement. For instance, this means that the following replacement is invalid according to *conditional mutation* since this mutation can be applied at the expression level:

```
if(M_NO==1){ y = a - x; }else{ y = a * x; }
```

This constraint is of particular importance for nested expressions, block statements, and loops because the complete outer expression or statement should not be duplicated.

Regarding the modified AST in Figure 2, the framed node, including its children `a` and `x`, is the original node of Figure 1. Therefore, any further transformations on child nodes, such as replacing `x` by a constant literal, would be applied exclusively on the framed node in order to have exactly one mutant in each `THEN` part. For nested expressions, this condition is crucial for preventing the algorithm from recursively applying transformations on already mutated nodes.

Runtime Optimization with Mutation Coverage

In order to kill a mutant, the following three conditions have to be fulfilled (cf. [22]):

1. The mutated code has to be reached and executed.
2. The mutation has to change the internal state.
3. The change has to be propagated to the output.

Conversely, all mutants which cannot be reached and executed cannot be killed under any circumstances. As a consequence, these mutants can be declared as not killed without executing the SUT. With respect to *conditional mutation* where all mutants are encoded together with the original version, the original expression or statement can be replaced again by a conditional expression or statement which additionally collects coverage information. This information should be gathered if and only if the original version is executed (i.e., `M_NO==0`). For this purpose the condition $cond_{cov}$ is inserted which is in turn a concatenation of `M_NO==0` and a method call which collects the coverage information:

$$CondExpr(cond_{cov}, expr, expr) \longleftarrow\!\shortmid expr \tag{5}$$

$$cond_{cov} = (cond_0 \;\&\&\; covered) \tag{6}$$

The `COVERED` method takes, as depicted in Listing 3, a range as parameters in order to efficiently record expressions or statements which are mutated more than once. In addition, lazy evaluation is exploited by using a logical `and` (i.e., `&&`) within the condition and the method always returns `false` in order to fulfill the condition that the right most `else` part contains the original expression or statement.

Support for Higher Order Mutation

Conditional mutation can also be extended to support higher order mutation. The key advantage of having all mutations within one file triggered by a certain condition provides the opportunity to adapt the conditions so that multiple mutants are executed. Depending on the order of higher order mutation, i.e. how many first order mutants should be combined, there are for instance the two following options: Use a bitmask to encode several mutant identifiers within one variable or use an array of mutant identifiers. The bitmask option might be more efficient but it is limited to a small

Table 1: Investigated applications in the empirical study.

Application name	*Number of source files*	*Program LOC**	*Number of mutants*	*Number of test cases*	*Test case LOC**	*Covered mutants*	*Killed mutants*
aspectj	1,975	372,751	406,382	859	17,069	20,144	10,361
apache ant	764	103,679	60,258	1,624	24,178	28,118	21,084
jfreechart	585	91,174	68,782	4,257	48,026	29,485	12,788
itext	395	74,318	124,184	63	1,393	12,793	4,546
java pathfinder	543	47,951	37,331	165	12,567	8,918	4,434
commons math	408	39,991	67,895	2,169	41,907	54,326	44,084
commons lang	85	19,394	25,783	1,937	32,503	21,144	16,153
numerics4j	73	3,647	5,869	219	5,273	4,900	401

*Physical lines of code as reported by sloccount (non-comment and non-blank lines).

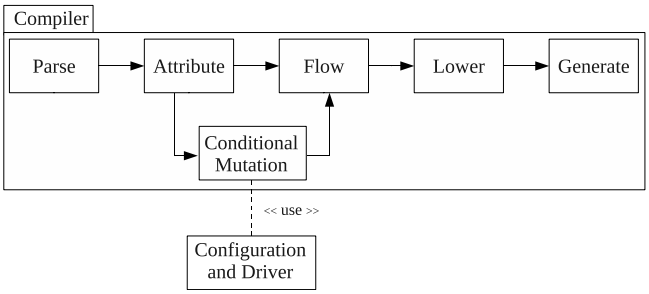

Figure 3: Integration of the conditional mutation approach into the compilation process.

number of identifiers. The concrete number is determined by the trade-off between the maximum number of first order mutants and the level of higher order mutation. Using 4 identifiers with 16 bits each would therefore limit the number of first order mutants to 65,536.

The support for higher order mutation is integrated into the compiler and preliminary runtime results are also promising. However, since the focus of this paper is the design and empirical evaluation of *conditional mutation*, we leave the complete investigation of higher order conditional mutation as future work, which is further discussed in Section 7.

Implementation Details

The process of compiling a source file into intermediate or assembled code can be divided into the following steps:

- Parse: Parse the source code and build the AST.
- Attribute: Add semantic information to the AST.
- Flow: Flow analysis to check for semantic errors.
- Lower: Decompose syntactic sugar and simplify AST.
- Generate: Generate code for the target machine.

Now the question arises as to the best method for integrating *conditional mutation* into the compilation process. Obviously, a parsed AST is necessary to apply a transformation to it. Furthermore, it is advisable to transform the AST before the flow analysis and the lower step for two reasons. On the one hand the code which shall be mutated should not be simplified or desugared previously. On the other hand the mutated code should also be checked by the compiler in order to avoid an incorrect AST and thus invalid code.

As a consequence, only two options remain for the integration, namely before or after the attribution step. Applying *conditional mutation* after the attribution step is slightly more complex since the additional nodes and subtrees which shall be inserted also have to be attributed. However, an attributed AST provides a lot of semantic information (e.g., type information) which offers more subtle mutations. Overall, the advantages of the second option outweigh the first and hence the *conditional mutation* approach is implemented as an additional, but optional, transformation after the attribution step, as depicted in Figure 3.

Concerning the realized implementation of the approach in the mainstream Java compiler, the optional *conditional mutation* step can be triggered and configured by means of compiler options. Additionally, the global mutant identifier and the method to gather the coverage information are implemented in a separate driver class.

5. EMPIRICAL STUDY

Conditional mutation has been integrated as an optional transformation in the Java Standard Edition (SE) 6 compiler in order to evaluate the approach. Two aspects are of particular interest in this empirical study, namely, the runtime and the memory footprint of the compiler with the enabled *conditional mutation* option and the runtime of the compiled and instrumented programs. For this purpose, we used the eight applications in Table 1 that range from 3,647 to 372,751 lines of code. According to selective mutation [16], a reduced, but sufficient set of mutation operators [20] has been chosen. As a first step, the following operators have been implemented and are configurable via compiler options:

- **ORB** (Operator Replacement Binary): Replace a binary arithmetic(AOR), logical(LOR), relational(ROR), or shift(SOR) operator with all valid alternatives.
- **ORU** (Operator Replacement Unary): Replace a unary operator with all valid alternatives.
- **LVR** (Literal Value Replacement): Replace a literal value by a positive value, a negative value, and zero.

Table 3 illustrates the necessary compile times for applying these operators to the aspectj project. All shown runtimes throughout the empirical study are the median of ten individual runs[1]. We do not report additional descriptive statistics or perform further statistical analysis since the runs are deterministic and the timing results exhibit little, if any, dispersion. In addition to the total runtime and overhead, the last column shows a normalized overhead per 1,000 mutants. A smaller value in this column means less overhead and is thus the better result. Regarding the quantity of 406,382 mutants, the total overhead of 33% for generating and compiling these mutants is almost negligible.

Furthermore, Figure 4 shows the compiler runtimes for all other analyzed projects. The trend lines in this diagram

[1]Reported runtimes are measured on a Linux machine with Intel Centrino CPU, 4GB of RAM, and kernel version 2.6.32-5-amd64.

Table 2: Time and space overhead for applying conditional mutation.

Application name	*Runtime of test suite in sec*				*Memory consumption**			*Size of compiled program**		
	original	*instrumented wcs*		*instrumented wcs+cov*	*original*	*instrumented*		*original*	*instrumented*	
aspectj	4.3	4.8	(11.63%)	5.0 (16.28%)	559	813	(45.44%)	18,368	30,508	(66.09%)
apache ant	331.0	335.0	(1.21%)	346.0 (4.53%)	237	293	(23.63%)	6,976	8,228	(17.95%)
jfreechart	15.0	18.0	(20.00%)	23.0 (53.33%)	220	303	(37.73%)	4,588	5,896	(28.51%)
itext	5.1	5.6	(9.80%)	6.3 (23.53%)	217	325	(49.77%)	4,140	6,580	(58.94%)
java pathfinder	17.0	22.0	(29.41%)	29.0 (70.59%)	182	217	(19.23%)	4,052	5,096	(25.77%)
commons math	67.0	83.0	(23.88%)	98.0 (46.27%)	153	225	(47.06%)	3,124	4,464	(42.89%)
commons lang	10.3	11.8	(14.56%)	14.8 (43.69%)	104	149	(43.27%)	968	1,456	(50.41%)
numerics4j	1.2	1.3	(8.33%)	1.6 (36.44%)	73	90	(23.29%)	408	508	(24.51%)

*Memory consumption of the compiler in MB and size of compiled program in KB.

Table 3: Compiler runtime to generate and compile the mutants for the aspectj project.

aspectj (1,975 files - 372,751 LOC - 18.94 sec)				
Operator	*Mutants*	*Runtime in sec*	*Overhead total*	*Overhead per 1k mutants*
AOR	70,989	21.15	11.67%	0.16%
LOR	81,733	21.71	14.63%	0.18%
ROR	137,297	22.90	20.91%	0.15%
SOR	47,830	20.84	10.03%	0.21%
ORB	337,849	25.02	32.10%	0.10%
ORU	802	19.21	1.43%	1.78%
LVR	67,731	20.83	9.98%	0.15%
ALL	406,382	25.15	32.79%	0.08%

have been computed by means of the gnuplot fit function which uses the method of least mean square error. The gradients of all trend lines are decreasing for a larger number of mutants. That implies that the relative overhead per mutant is decreasing for an increasing number of mutants. In order to avoid obscuring the visualization, Figure 4 does not contain the higher runtimes associated with generating and compiling the many mutants for aspectj. Nevertheless, the trend for the aspectj project is similar to the other investigated programs since the relative overhead per 1,000 mutants is also decreasing for a larger number of mutants.

Besides the time of generating and compiling the mutants, the execution time of the instrumented and compiled programs is also important. In order to determine the overhead associated with the insertion of the conditional statements and expressions, the analyzed applications are executed by means of their test suites. To establish an upper bound on time overhead, we consider the worst-case scenario for conditional mutation when `M_NO==0` and thus every condition has to be evaluated. Furthermore, mutation coverage, as described in Section 4, can be applied to determine the mutants that cannot be killed by the test suite. This information is collected by means of method calls which represent an additional time overhead. Therefore, we also measure the runtime of the test suites with mutation coverage enabled.

The corresponding results for both runtime analyses are depicted in Table 2 where *wcs* and *cov* denote *worst-case scenario* and *coverage*, respectively. The time overhead for the worst-case scenario (*wcs*) ranges from 1.2% for apache ant to 29.4% for java pathfinder. For the worst-case scenario with mutation coverage enabled (*wcs+cov*) the overhead ranges between 4.5% and 70.6%. On average the overhead for *wcs* and *wcs+cov* is 15% and 36%, respectively. Gathering the coverage information leads to a larger overhead due to the additional method calls. Nevertheless, it has to be pointed out that the coverage information is determined only once for an instrumented application with a corresponding test suite and hence the overhead is not crucial. The actual overhead depends on the type of application. For instance, apache ant does a lot of costly file system operations and thus the additional costs for conditional mutation are negligible. In contrast, java pathfinder and commons math represent applications that almost exclusively perform computations, thus explaining why the overhead is more noticeable.

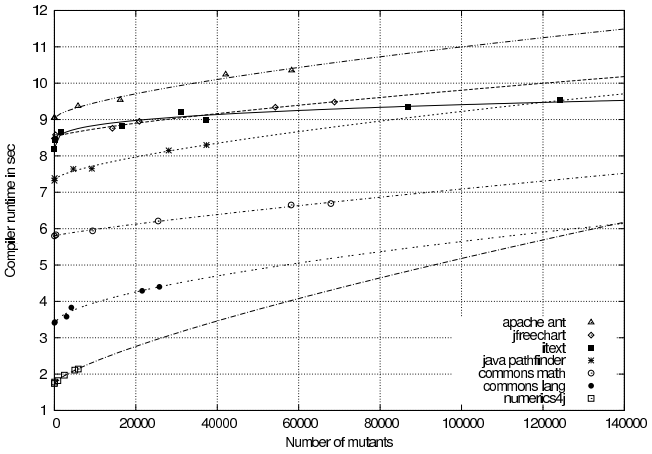

Figure 4: Compiler runtime to generate and compile the mutants for all the projects, excluding aspectj.

Apart from the runtime of the test suites, the space overhead in terms of compiler memory consumption and program size is also considered, as depicted in Table 2. The memory consumption of the compiler ranges from 19.2% to 49.8% and the overhead due to the larger program size varies between 18.0% and 66.1%. Generally, the space overhead is predominantly determined by the ratio of number of mutants to lines of code, as shown in Table 1. Concerning the memory footprint of the compiler, the average overhead is 36.2%. Thus, the enhanced compiler with *conditional mutation* can easily run on commodity workstations.

6. THREATS TO VALIDITY

With regard to the empirical results, some threats to validity have to be considered. The choice of the applied mutation operators could be a threat to internal validity. Different operators may affect the runtime of the compiler. However, the chosen operators are frequently used in the literature and therefore provide comparable results [18, 20].

A potential threat to external validity might be the representativeness of the selected applications. There is no guarantee that the depicted overheads will remain the same for other programs. Nevertheless, the investigated applications differ considerably in size, complexity, and operation purpose and most of them are widely used. Furthermore, this is to our knowledge the largest study of mutation analysis to date. So, we judge that the reported results are meaningful. Additionally, the *conditional mutation* approach might be more or less efficient in terms of other programming languages and compilers. A replication of this study is thus necessary, especially for languages that do not use intermediate code. This matter is left open for future research.

Defects in the compiler-integrated prototype could be a threat to construct validity, but we controlled this threat by testing our implementation with a developed test suite and by checking the results of several small example programs. Thus, we judge that the implementation worked correctly.

7. CONCLUSIONS AND FUTURE WORK

This paper describes and addresses the challenges associated with increasing the efficiency of mutation analysis. A new method called *conditional mutation* is presented which reduces both the generation time and the execution time. It is, compared with the conventional way, much more efficient and offers the possibility of applying mutation analysis to large software systems. Moreover, mutation testing, where mutants are generated and executed iteratively, becomes more feasible since the generation time is reduced to a minimum. The approach is versatile, programming language independent, and can be integrated within the compiler.

So far, *conditional mutation* has been implemented as an optional transformation in the Java Standard Edition compiler and it has been applied to applications up to 372,751 lines of code. The *conditional mutation* step can be configured via common compiler options. So, this enhanced compiler can be used in any environment based on the Java programming language and hence is not limited to a particular testing framework or tool suite. Furthermore, *conditional mutation* can be easily combined with other do smarter and do fewer approaches since the set of mutation operators is configurable and the mutation analysis can be parallelized and processed in a distributed environment.

Since the support for higher order mutation is also provided by the improved compiler, a comprehensive investigation of higher order conditional mutation is part of our future work. Additionally, the set of applicable mutation operators will be extended and we plan to integrate *conditional mutation* into a C/C++ compiler. Moreover, the approach could be optimized with respect to the runtime by balancing the AST with further conditional expressions and statements. This would decrease the necessary evaluations and thus the runtime overhead. However, the space overhead would increase. Consequently, further empirical studies addressing the efficiency of first and higher order conditional mutation with both new mutation operators and different programming languages is another area for future work.

Finally, we will conduct a study with different tools such as MuJava [12] and Javalanche [19] to compare *conditional mutation* with related techniques. Because the bytecode transformation operates at a different level of abstraction, we will compare our compiler-integrated source-code transformation with the bytecode transformation in order to study the strengths and weaknesses of both approaches.

8. REFERENCES

[1] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 402–411, 2005.

[2] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering*, FOSE '07, pages 85–103, 2007.

[3] R. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 1999.

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

[5] B. Choi, A. Mathur, and B. Pattison. PMothra: scheduling mutants for execution on a hypercube. *Software Engineering Notes*, 14:58–65, 1989.

[6] R. A. DeMillo, E. W. Krauser, and A. P. Mathur. Compiler-integrated program mutation. In *Proceedings of the 5th Annual Computer Software and Applications Conference*, COMPSAC '91, pages 351–356, 1991.

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

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

[9] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. Report TR-09-06, CREST Centre, King's College London, UK, 2009.

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

[11] K. N. King and A. J. Offutt. A Fortran language system for mutation-based software testing. *Software Practice and Experience*, 21(7):685–718, 1991.

[12] Y.-S. Ma, J. Offutt, and Y.-R. Kwon. MuJava: A mutation system for Java. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 827–830, 2006.

[13] H. Mills. On the Statistical Validation of Computer Programs. Technical report, IBM FSD Report, 1970.

[14] MuJava. The official web site of the MuJava project. `http://www.cs.gmu.edu/~offutt/mujava`, 2010.

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

[16] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology*, 5(2):99–118, 1996.

[17] A. J. Offutt and J. Pan. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification, and Reliability*, 7(3):165–192, 1997.

[18] J. Offutt and R. H. Untch. Mutation 2000: Uniting the orthogonal. In *Proceedings of Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, pages 45–55, 2000.

[19] D. Schuler and A. Zeller. (Un-)covering equivalent mutants. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*, ICST '10, pages 45–54, 2010.

[20] A. Siami Namin, J. H. Andrews, and D. J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 351–360, 2008.

[21] R. H. Untch, A. J. Offutt, and M. J. Harrold. Mutation analysis using mutant schemata. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '93, pages 139–148, 1993.

[22] J. M. Voas. PIE: a dynamic failure-based technique. *IEEE Transactions on Software Engineering*, 18:717–727, 1992.