석 사 학 위 논 문
Master's Thesis

세분화된 변이 연산자를 활용한 효과적인 변이 생성 수 절감 기법

Effective Mutant Reduction using
Fine-grained Mutation Operators

2020

판 뒤 록 (Phan, Duy Loc)

한 국 과 학 기 술 원

Korea Advanced Institute of Science and Technology

석 사 학 위 논 문

세분화된 변이 연산자를 활용한 효과적인 변이 생성 수 절감 기법

2020

판 뒤 록

한 국 과 학 기 술 원

전산학부

세분화된 변이 연산자를 활용한 효과적인 변이 생성 수 절감 기법

판 뒤 록

위 논문은 한국과학기술원 석사학위논문으로
학위논문 심사위원회의 심사를 통과하였음

2019년 12월 16일

Effective Mutant Reduction using Fine-grained Mutation Operators

Duy Loc Phan

Advisor: Moonzoo Kim

A dissertation submitted to the faculty of
Korea Advanced Institute of Science and Technology in
partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Daejeon, Korea
December 16, 2019

Approved by

Moonzoo Kim
Associate Professor of School of Computing

The study was conducted in accordance with Code of Research Ethics[1].

[1] Declaration of Ethical Conduct in Research: I, as a graduate student of Korea Advanced Institute of Science and Technology, hereby declare that I have not committed any act that may damage the credibility of my research. This includes, but is not limited to, falsification, thesis written by someone else, distortion of research findings, and plagiarism. I confirm that my thesis contains honest conclusions based on my own careful research under the guidance of my advisor.

MCS
20184174

판뒤록. 세분화된 변이 연산자를 활용한 효과적인 변이 생성 수 절감 기법. 전산학부 . 2020년. 41+iv 쪽. 지도교수: 김문주. (영문 논문)
Duy Loc Phan. Effective Mutant Reduction using Fine-grained Mutation Operators. School of Computing . 2020. 41+iv pages. Advisor: Moonzoo Kim. (Text in English)

Abstract
Although mutation analysis is important for various software analysis tasks, there exists no practical mutation tools for modern, complex, real-world C programs. I have developed MUSIC (MUtation analySIs tool with high Configurability and extensibility) which generates mutants for modern complex real-world C programs. I have conducted a case study on Siemens benchmark programs and a modern real-world C program cURL to compare MUSIC with Milu, Proteum in terms of applicability and number of stillborn (i.e. syntactically illegal) mutants generated. In this case study, MUSIC successfully generates mutants without any stillborn mutants.

Another serious obstacle for mutation analysis is the huge cost of running test suites on a large number of mutants. To resolve this problem, I have proposed a new mutation operator-based mutant reduction technique *REFINER* which applies cost-considerate linear regression (i.e., CLARS) on *fine-grained mutation operators*. Also, I have applied REFINER to predict *hard-to-kill* mutation score which is more valuable to measure test suite quality than commonly used mutation score.

The experiment results show that, while sustaining accurate prediction power to estimate hard-to-kill mutation score, REFINER selects far fewer mutants than CLARS on the traditional mutation operators (i.e., 2.0% vs. 16.5%). Also, REFINER predicts hard-to-kill mutation score 4.5, 4.4, and 4.3 times more accurate than mutant reduction techniques that use random selection, Offutt's four mutation operators selection, and, only SSDL mutation operator respectively.

Keywords Mutation analysis, practical mutation tool, C programs, hard-to-kill mutation score prediction, fine-grained mutation operator, mutant reduction, cost-considerate linear regression

Contents

List of Tables

List of Figures

Chapter 1. Introduction

Software testing is an investigation conducted to confirm the quality, reliability of software. The investigation executes a target software against a set of test cases and checks whether the output meets expected criteria. Unfortunately, if the quality of a test set is bad, passing all the tests cannot guarantee the quality of target software. For example, a bad test set may have only 25% code coverage, and not cover, or test many core functions of the target software. So passing such a test set does not guarantee that the target software operates well as expected. For successful software testing, it is important to evaluate the quality of the test suite.

Mutation analysis [1] has been used in various aspects of software testing, for test suite quality evaluation [1, 10, 13], test case prioritization [2, 48, 49], test generation [3, 4, 47], debugging [5, 6, 37], and its application is getting wider [7, 8]. Mutation analysis generates many variants of a program (called mutants) by applying a set of syntactic code changes (called mutation operators). Then, it runs test suites on the mutants, and analyzes the output difference between the original program and the mutants. A mutant m is killed by a test set T if for some test case $t \in T$, the output of m is different from that of the original program. For test suite quality evaluation, the ratio of the number of killed mutants over the number of generated mutants (called mutation score) can be used as a measure of quality of test suite.

Although mutation analysis is important for various software analysis tasks, there exist few practical mutation tools for C programs. Existing mutation tools for C programs often fail to generate useful mutants of a modern real-world C program. For example, Proteum [33, 34, 35] often fails to generate mutants for modern C programs because it does not support recent C standard later than C89. As another example, Milu [36] generates many stillborn (i.e., syntactically illegal) mutants due to incorrect handling of types including `typedef`, `enum`, `const` type qualifier and an array type.

Another obstacle of mutation analysis is its high runtime cost. For example, a mutation testing tool for C programs MUSIC [9] generates 347,636 mutants for grep-2.0 (5696 LoC) by applying 108 mutation operators. It takes around 483 hours to run the regression test suite of 809 test cases on all generated mutants. As the size and complexity of the target software grows, more mutants are generated, which also increases the time cost of mutation analysis. Considering how frequently real-world software and its test suite are updated, generating and executing a large set of mutants on a large test suite is impractical. Hence, in order to reduce the cost of mutation analysis for test suite quality evaluation, various mutant reduction techniques such as operator-based mutant selection and random selection, have been proposed to select and use a small, representative subset of all generated mutants to calculate mutation score accurately.

1.1 Related Works

1.1.1 Existing Mutation Tools for C Programs

Delamaro et al. [33, 34, 35] developed PROgram TEsting Using Mutant (Proteum), a mutation tool supporting the application of mutation throughout software development. The original version of Proteum supports 75 C mutation operators defined by Agrawal et al. [31], and provides functionalities for executing generated mutants against given test cases. The second version, Proteum/IM 2.0, provides an additional 33 C interface mutation operators, which attempts to simulate integration errors (i.e. errors related to connection between two functions). A drawback of Proteum is that it fails to generate mutants for modern C programs because it does not support recent C standard later than C89.

Jia and Harman [36] developed Milu, a C mutation testing tool designed for both first order and high order mutation testing. Milu, by default, supports 28 C mutation operators and comes with two mode: traditional mode and higher order mode. The traditional mode supports first order mutation testing, in which users can use pre-defined or customized mutation operators. The higher order mode is designed for research on subsuming High Order Mutants. A shortcoming of Milu is that due to incorrect handling of types in C including `typedef`, `enum`, `const` type qualifier and an array type, Milu generates many mutants containing syntactic errors (i.e. stillborn mutants). Generating many stillborn mutants may increase even further the time cost of mutation analysis.

1.1.2 Mutation Operator based Mutant Reduction Techniques

Operator-based mutant selection determines a set of mutation operators, and select only mutants generated by these operators. Offutt et al. [10] identified five out of 22 mutation operators in Mothra as a selective set of mutation operators sufficient for mutation testing. They conducted experiment on 10 programs, generating five test suites per program which kill all mutants generated by the sufficient set of the five operators. They then measured the mutation score of these test suites with respect to all generated mutants. The result shows that all test suites that killed all mutants generated by the five selected operators have an average mutation score of 0.995 with respect to all generated mutants. On average, Offutt et al.ś selective mutation operator set selects 22.4% of generated mutants.

Barbosa et al. [11] proposed six guidelines to identify a sufficient set of mutation operators, which were applied to a set of 27 small C programs. The resulting sufficient set of 10 mutation operators was compared with Offutt et al. [10], Wong et al. [12], and random mutant selection (10%, 20%, 30%, 40%) in terms of mutation score, mutant selection percentage. The sufficient set determines the best mutation score (0.99), and on average, selects 35% of generated mutants.

Namin et al. [13], views the mutant selection problem as an instance of the variable selection problem. They applied Cost-Based Least Angle Regression (CBLARS), to seven Siemens programs and selected the first model with R^2 higher than 0.98 to identify a sufficient set of 28 C mutation operators. For each program, they generated mutants using Proteum's 108 mutation operators, and generated 100 test suites of various sizes. After conducting explanatory experiment on `tcas`, the authors noted that executing all mutants would take too much time to be feasible, so for each of six other programs, they randomly selected 2000 mutants for experiment. The mutation analysis data of seven programs forms training data for CBLARS to generate models. The selected set of 28 mutation operators selects 7.4% of around 15,000 mutants selected for experiment.

Deng et al. [27] conducted an empirical evaluation of Statement Deletion (SSDL) mutation operator for Java programs and found that test sets, which kill all SSDL mutants, achieve high mutation scores with respect to mutants generated by all mutation operators. They conducted experiments on 40 Java classes, whose size ranging from 1 to 26 methods and from 29 to 433 LoC. For each target class, Java mutation tool muJava [1] is employed to generate mutants. The total number of SSDL mutants generated accounted for 19% of all generated mutants. Test suites, which kill all SSDL mutants, are manually generated. Of 40 generated test suites, the average mutation score was 0.92, and the median mutation score was 0.93.

1.1.3 Random Mutant Selection Techniques

Random mutant selection was first proposed by Acree et al. [14] and Budd [15]. Wong and Mathur [16] empirically studied the technique of randomly selecting 10% to 40% mutants generated with 22 mutation operators in Mothra. They evaluated the selected mutants based on the number of selected mutants, the number of the test cases needed to kill all selected mutants, and the mutation score with respect to all non-equivalent mutants. The experiment was conducted using four Fortran programs. In terms of the mutation scores, average mutation score of the test sets that kills all randomly selected mutants with respect to all generated mutants are higher than 0.95. The authors concluded that x% random mutant selection can provide a significant reduction in terms of the number of required test cases and mutants, and suffer a small loss in ability to predict mutation scores.

Despite the simplicity of random mutant selection, Zhang et al. [17] has shown that it is not inferior to other operator-based mutant selection techniques. They compared two random selection techniques with the three operator-based mutant selection techniques – Offutt et al. [10], Namin et al. [13] and Barbosa et al. [11]. The first random selection technique randomly selects a set of mutants whose size is equal to the x% of the mutants selected by an operator-based technique. The second one randomly selects one mutation operator, then randomly selects a mutant generated by that operator. These steps are repeated until the selected mutant set's size is equal to the x% of the mutants selected by an operator-based technique. The two random techniques are evaluated with $x = 50\%$, 75% and 100%. These five techniques were evaluated on seven Siemens programs. For each selected mutant set, 50 test suites that kill all mutants in the mutant set were created, and their mutation score with respect to all mutants were measured. The effectiveness of all five techniques on the seven programs were all above 99%. Also, they found out that at $x = 100\%$, none of the operator-based mutant selection techniques was superior to the random mutant selection techniques.

Zhang et al. [19] combined Offutt et al.'s sufficient set of five mutation operators [10] with eight sampling strategies. The experiment was conducted on 11 Java programs. For each program and sampling strategies, they sampled from the set of mutants selected by Offutt et al.'s sufficient set 20 times with sampling ratio of 5%, 10%, ..., 95%. Two evaluation approaches were employed. The results of the first approach show that the test suites that kill all sampled mutants achieve higher than 98% mutation score with respect to all mutants. In the second approach, 100 test suites were randomly created for each sample and their mutation scores with respect to the sampled mutants and all generated mutants were measured. They then applied linear regression and measured R^2 to see how well mutation scores with respect to the sampled mutants can predict mutation scores with respect to all mutants. At sampling ratio of 5%, the R^2 values were high, ranging from 0.945 to 0.998.

1.1.4 Predictive Mutation Testing

Zhang et al. [51] proposed Predictive Mutation Testing (PMT), an approach to predict mutation testing results without executing mutants. PMT builds a binary classification model, which takes as inputs 15 features of the targeted mutant m and test t, and outputs whether t kills m without executing m against t. The performance of PMT was evaluated under two application scenario (cross-version and cross-project) using nine Java projects, which are widely used in previous software testing research. The following metrics were measured: Precision, Recall, F-measure, Area under ROC curve (AUC) and Absolute Prediction Error.

For cross-project experiment, the authors conducted a nine-fold cross validation experiment to evaluate PMT. For each target program P, mutation testing data of eight other programs were used as training data and data of P was used as testing data. The results show that the Absolute Prediction Error of seven out of nine projects are below 0.1 and all other metrics are above 0.85. In cross-version experiment, for each project, up to ten versions of a project were selected. For each version v, the authors used version $v-1$ as training set, and version v as testing set. The results show the Absolute Prediction Error of 37 out of 39 versions are below 0.05, and almost all other metric values are above 0.9.

The authors compared PMT with traditional mutation testing (using mutation testing tool PIT [52]) in terms of efficiency. The experimental results show that PMT improves the efficiency of mutation testing by up to 151.4 times while incurring less than 0.1 Absolute Prediction Error.

1.2 Thesis Statement

The thesis statement of this dissertation is as follows:

> Fine-grained mutation operators refined from coarse-grained mutation operators can improve mutant reduction technique practically.

Coarse-grained mutation operators refer to the 108 C mutation operators proposed by Agrawal et al. [31] and Delamaro et al. [32]. Fine-grained mutation operator is an idea proposed in this disseration to improve the effectiveness of mutation reduction techniques.

A traditional mutation operator r_i can be refined into fine-grained mutation operators $r_{i,1}, r_{i,2}, ..., r_{i,n}$ whose domains and ranges form partitions of the domain and range of r_i, respectively. For example, traditional mutation operator $OLLN$ (logical operator mutation operator) can be refined into $OLLN_{\&\&\rightarrow||}$, $OLLN_{||\rightarrow\&\&}$. Note that each fine-grained mutation operator $r_{i,j}$ generates a much smaller number of mutants than its traditional mutation operator r_i. Also, r_i and all of its fine-grained mutation operators $r_{i,1}, r_{i,2}, ..., r_{i,n}$ generate the same set of mutants. More details on fine-grained mutation operator and its application will be given in Section 1.3.2 and Chapter 3.

1.3 Proposed Approach

1.3.1 MUSIC: Mutation Analysis Tool with High Configurability and Extensibility

To address the lack of practical mutation tools for C programs, this dissertation proposes *MUtation analySIs tool with high Configurability and extensibility (MUSIC)* to generate mutants for modern complex real-world C programs that consist of multiple source files with complex compilation commands.

MUSIC implements 75 mutation operators defined by Agrawal et al. [31] and 33 interface mutation operators introduced by Delamaro et al. [32]. MUSIC is designed to be extensible for a user to easily make a new mutation operator.

One notable feature of MUSIC is its fine-grained configuration of mutant generation, which can satisfy various purposes of mutation analysis. In other words, MUSIC allows a user to specify a target domain and a range of a mutation operator and a target scope of mutation. For example, MUSIC can apply arithmetic mutation operator (OAAN) to only one of {+,-} and mutate it to only * between Line 100 and Line 200 of `target.c`.

I have applied MUSIC, Milu, and Proteum to seven Siemens benchmark programs [23] and a modern large program cURL [39]. I evaluate these tools in terms of applicability and number of stillborn mutants generated. For both Siemens benchmarks and cURL, MUSIC successfully generates mutants without any manual modification of the target programs and it generates no stillborn mutant. In contrast, Proteum requires manual source code modification to generate mutants for Siemens benchmarks and fails to generate mutants for cURL. Milu generates many stillborn mutants (34.18% and 75.31% of the mutants for Siemens benchmarks and cURL were syntactically illegal, respectively).

1.3.2 REFINER: Refined Mutation Operator-based Mutant Reduction

To address the time cost problem of mutation analysis, this dissertation proposes *REFINEd mutation operator based mutant Reduction (REFINER)*, a salient technique to reduce the number of the mutants generated for mutation analysis. REFINER selects a subset of *fine-grained mutation operators* that are refined from the traditional mutation operators, by using cost-considerate least-angle regression (CLARS) [18].

The motivation of REFINER to target refined mutation operators is based on the observation that the traditional mutation operators $r_i s$ may be *similar/redundant* to each other (i.e., a set of mutants M_i generated by applying r_i may be highly correlated with M_j). In other words, many mutants generated by the traditional mutation operators are not useful to learn an accurate and efficient prediction model. In contrast, the correlation between fine-grained mutation operators can be lower than that between the traditional mutation operators (see RQ1 in Section 4.2.1) and the mutants generated by the refined mutation operators can be used to learn an accurate and efficient prediction model.

Also, REFINER focuses to accurately predict *hard-to-kill* mutation score (i.e., mutation score calculated using only hard-to-kill mutants), not mutation score using the whole set of generated mutants (widely performed in literature [13, 17, 19]). For a given mutant set M and a test suite T of a program P, a mutant $m \in M$ is a *hard-to-kill* mutant if only k% of T can kill m with small k. In this disseration, k is set to be 3, 5, and 7. REFINER focuses to accurately predict *hard-to-kill* mutation score because hard-to-kill mutants are valuable to measure the quality of a test suite [20, 21, 22] as easy-to-kill-mutants in the whole set of mutants have severe redundancy and inflict noises to measure the quality of target test suites [50]. To my best knowledge, REFINER is the first mutant reduction technique to predict hard-to-kill mutation score.

To demonstrate the effectiveness of REFINER, I have applied REFINER to the six real-world C programs in SIR [23] to predict mutation score of hard-to-kill mutants. The experiment result shows that

- REFINER selects only 2% of all mutants (in contrast to CLARS on the traditional mutation operators which selects 16.5% of all mutants), and

- REFINER generates a very accurate prediction model (i.e., Mean Squared Error (MSE) between the predicted hard-to-kill mutation scores and real ones is only 0.011 on average over all target programs).
- REFINER predicts hard-to-kill mutation score 4.5, 4.4, and 4.3 times more accurate than mutant reduction techniques that use random selection, Offutt's four mutation operators selection, and only SSDL mutation operator selection, respectively.

1.4 Contributions

The contributions of this disssertation are as follows:

1. I have developed MUSIC which supports 108 C mutation operators, is highly configurable and easy to extend for various mutation analysis purposes targeting modern complex real-world C programs.
    - MUSIC provides 75 mutation operators defined in Agrawal et al. [31] and 33 interface mutation operators [32] which do not generate stillborn mutants.
    - I am upgrading MUSIC to support fine-grained mutation operators.
2. I have performed a case study to evaluate Milu, Proteum and MUSIC on seven C programs in Siemens benchmarks and a modern real-world C program cURL and demonstrated that MUSIC has high applicability and generates no stillborn mutant.
3. The idea and development of fine-grained mutation operators for mutant reduction is salient.
    - Through experiments, I have shown that fine-grained mutation operators are less correlated than the existing traditional mutation operators, which can help REFINER learn a more accurate and efficient prediction model (Section 4.2.1).
4. I have developed REFINER, a new technique that learns an accurate and cost-efficient model of the fine-grained mutation operators to predict hard-to-kill mutation score.
    - Experiments on six real-world C programs shows that REFINER predicts highly accurate hard-to-kill mutation scores (MSE between predicted and real hard-to-kill mutation scores is only 0.011 on average) while using only 2% of all mutants.
5. Through the experiments on the six real-world C programs, I also show that REFINER outperforms the existing mutant reduction techniques.
    - REFINER predicts hard-to-kill mutation score 4.5, 4.4, and 4.3 times more accurate than random selection, Offutt's four mutation operators selection, and only SSDL mutation operator selection, respectively.

1.5 Structure of the Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 describes MUSIC, and compare it with Proteum, Milu. Chapter 3 describes REFINER. Chapter 4 explains the experiment setup to evaluate REFINER compared to other mutant reduction techniques in Section 4.1, reports the experiment results in Section 4.2, and discusses observation from the experiments Section 4.3. Finally, section 5 concludes the dissertaion with future work.

Chapter 2. MUSIC: Mutation Analysis Tool with High Configurability and Extensibility

I have implemented MUSIC based on the modern compiler framework Clang/LLVM 7.0 [43]. MUSIC is written in around 19,923 lines of C++ code, which consists of 292 header and source files. MUSIC is available at https://github.com/swtv-kaist/MUSIC.

Sect. 2.1 describes how a user applies MUSIC to a large complex project conveniently. Sect. 2.2 explains high configurability of MUSIC. Sect. 2.2.1 shows that a user can create his/her own new mutation operators easily with support of MUSIC. Sect. 2.2.2 explains how MUSIC avoids generating stillborn mutants. The component architecture of MUSIC referred through the subsections is shown in Fig. 2.1.

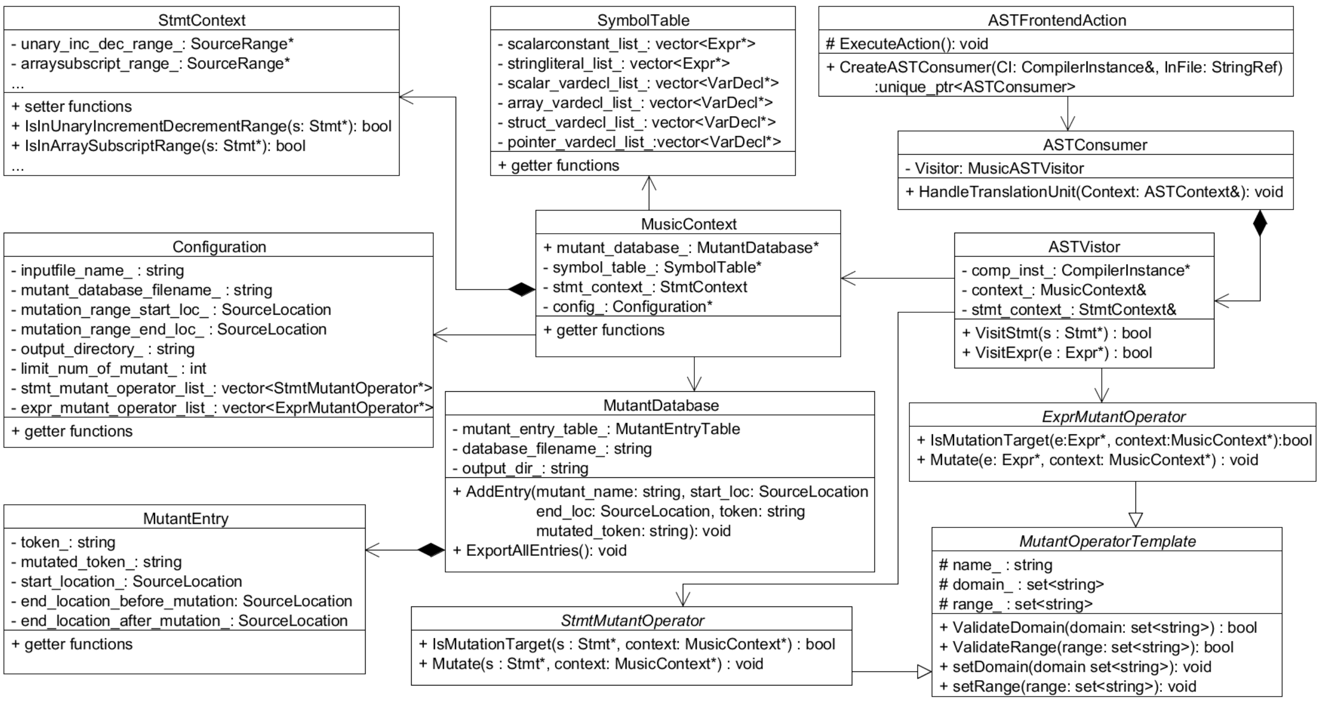

StmtContext
- unary_inc_dec_range_: SourceRange*
- arraysubscript_range_: SourceRange*
...
+ setter functions
+ IsInUnaryIncrementDecrementRange(s: Stmt*): bool
+ IsInArraySubscriptRange(s: Stmt*): bool
...
SymbolTable
- scalarconstant_list_: vector<Expr*>
- stringliteral_list_: vector<Expr*>
- scalar_vardecl_list_: vector<VarDecl*>
- array_vardecl_list_: vector<VarDecl*>
- struct_vardecl_list_: vector<VarDecl*>
- pointer_vardecl_list_:vector<VarDecl*>
+ getter functions
ASTFrontendAction
# ExecuteAction(): void
+ CreateASTConsumer(CI: CompilerInstance&, InFile: StringRef) :unique_ptr<ASTConsumer>
ASTConsumer
- Visitor: MusicASTVisitor
+ HandleTranslationUnit(Context: ASTContext&): void
Configuration
- inputfile_name_ : string
- mutant_database_filename_ : string
- mutation_range_start_loc_ : SourceLocation
- mutation_range_end_loc_ : SourceLocation
- output_directory_ : string
- limit_num_of_mutant_ : int
- stmt_mutant_operator_list_: vector<StmtMutantOperator*>
- expr_mutant_operator_list_: vector<ExprMutantOperator*>
+ getter functions
MusicContext
+ mutant_database_: MutantDatabase*
- symbol_table_: SymbolTable*
- stmt_context_: StmtContext
- config_: Configuration*
+ getter functions
ASTVistor
- comp_inst_: CompilerInstance*
- context_: MusicContext&
- stmt_context_: StmtContext&
+ VisitStmt(s : Stmt*) : bool
+ VisitExpr(e : Expr*) : bool
MutantDatabase
- mutant_entry_table_: MutantEntryTable
- database_filename_: string
- output_dir_: string
+ AddEntry(mutant_name: string, start_loc: SourceLocation end_loc: SourceLocation, token: string mutated_token: string): void
+ ExportAllEntries(): void
ExprMutantOperator
+ IsMutationTarget(e:Expr*, context:MusicContext*):bool
+ Mutate(e: Expr*, context: MusicContext*) : void
MutantEntry
- token_: string
- mutated_token_: string
- start_location_: SourceLocation
- end_location_before_mutation: SourceLocation
- end_location_after_mutation_: SourceLocation
+ getter functions
MutantOperatorTemplate
# name_ : string
# domain_ : set<string>
# range_ : set<string>
+ ValidateDomain(domain: set<string>) : bool
+ ValidateRange(range: set<string>): bool
+ setDomain(domain set<string>): void
+ setRange(range: set<string>): void
StmtMutantOperator
+ IsMutationTarget(s : Stmt*, context: MusicContext*) : bool
+ Mutate(s : Stmt*, context: MusicContext*) : void

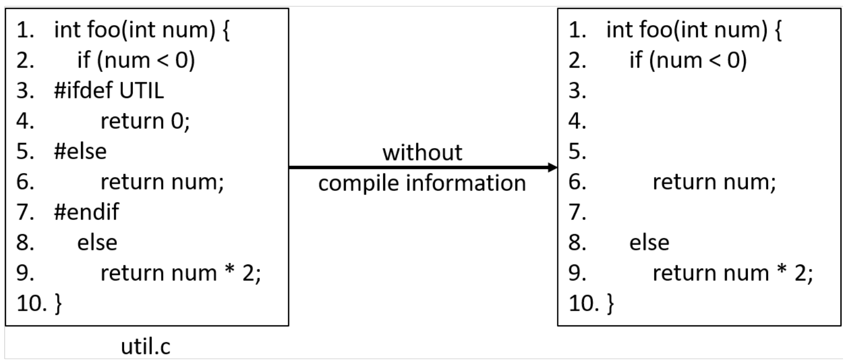
Figure 2.1: Simplified UML diagram of MUSIC

2.1 Applicability

To generate mutants for a large, complex project consisting of many directories and files with file-specific compilation commands, MUSIC utilizes a *compilation database*. This is because mutant generation often depends on specific compilation commands.

For example, `util.c` in the left code of Fig. 2.2 illustrates such situation. Without compilation information, a mutation tool assumes that a flag `UTIL` is *not* defined and generates an AST of the code as shown in the right part in Fig. 2.2, which fails to generate mutants on Line 4 even if a user actually compiles `util.c` with a flag `UTIL` as true. MUSIC can utilize all such compilation information from a given compilation database and a user can apply MUSIC to complex large projects conveniently.

A user can easily generate a compilation database for a project by running CMake [40] with the `-DCMAKE_EXPORT_COMPILE_COMMANDS` flag. Also, Gyp/Ninja [41] or BEAR [42] can be used for the purpose.



```
int foo(int num) {
    if (num < 0)
#ifdef UTIL
        return 0;
#else
        return num;
#endif
    else
        return num * 2;
}
```

```
int foo(int num) {
    if (num < 0)



        return num;

    else
        return num * 2;
}
```

Figure 2.2: Example of code without proper compilation information

A compilation database is a collection of compilation commands for a set of files. MUSIC receives compilation database in a JSON format. Each entry in a compilation database has three fields:

1. A file to which the compilation applies to
2. Compilation commands used
3. A directory in which this command is executed

2.2 Configurability

MUSIC provides mainly four options to selectively generate mutants for multiple C source files as follows:

1. `-m` *mut_op*[:*A*[:*B*]]: to select a mutation operator to apply (e.g., OAAN) and, optionally, a set of target token(s) to replace (e.g., *A* can be {+,*}) and a set of new token(s) to use (e.g., *B* can be {-,/}). *mut_op* can be one of the 108 pre-defined mutation operators as follows:

   (a) 75 mutation operators defined in Agrawal et al. [31]

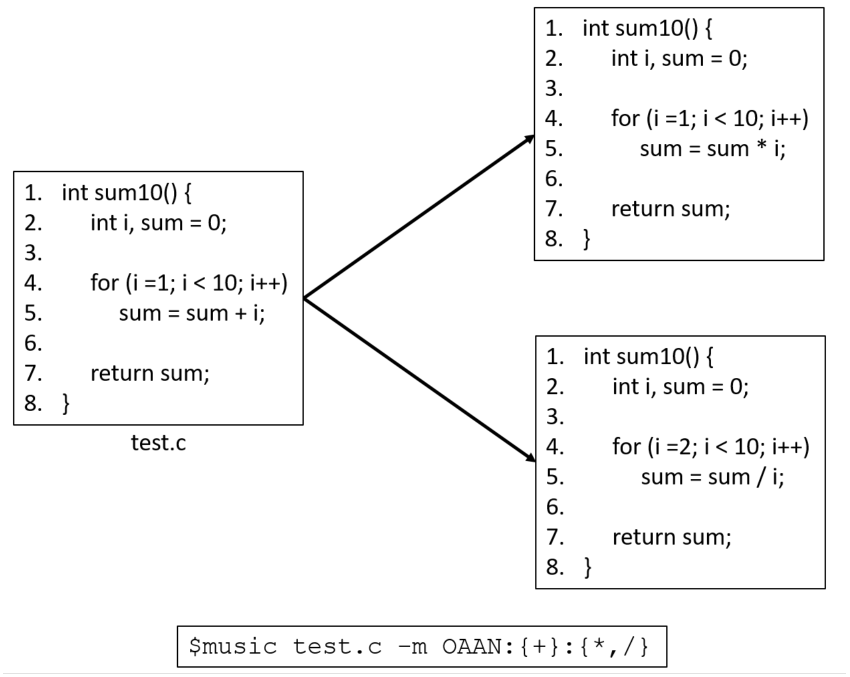
   (b) 33 mutation operators defined by Delamaro et al. [32]

   For example, OAAN mutates arithmetic operators (+, -, *, /, %) to other arithmetic operators. A user can specify a target domain of OAAN as {+} and a target range as {*, /} as shown in Fig. 2.3. Note that specified domain and range of `-m` must be *type-compatible* to a mutation operator (e.g., for OAAN, target domain and range cannot contain < or >>).

2. `-rs` *mut_range_start*: to specify a starting position of a target mutation range (i.e., a triple of a target file name, a line number, a column number)

3. `-re` *mut_range_end*: to specify an ending position of a target mutation range

4. `-l` *max_num*: to limit a maximum number of mutants generated per mutation point and mutation operator



```
int sum10() {
    int i, sum = 0;

    for (i =1; i < 10; i++)
        sum = sum + i;

    return sum;
}
```

test.c

```
int sum10() {
    int i, sum = 0;

    for (i =1; i < 10; i++)
        sum = sum * i;

    return sum;
}
```

```
int sum10() {
    int i, sum = 0;

    for (i =2; i < 10; i++)
        sum = sum / i;

    return sum;
}
```

```
$music test.c -m OAAN:{+}:{*,/}
```

Figure 2.3: Usage of option `-m` to mutate + to * and / by modifying OAAN domain and range

- Some mutation operators may generate many mutants. For example, CCCR mutates constant literals to another constant literals in a target program, which can generate many mutants. If `-l` 10 is given, for each mutation point of CCCR, MUSIC arbitrarily generates at most 10 mutants of CCCR.

2.2.1 Extensibility

One advantage of MUSIC is that it supports a user to create his/her own *new* mutation operators conveniently. A mutation operator of MUSIC is defined as a *rule* to modify a target source file. Such rule specifies a target domain and a range of a mutation operator as a set of tokens such that a mutation operator replaces tokens in a target domain with ones in a target range.

A mutation operator of MUSIC extends `ExprMutantOperator` which mutates C expressions, or `StmtMutantOperator` which mutates C statements (see the bottom right part of Fig. 2.1). These two classes extend an abstract class `MutantOperatorTemplate` which has a mutation operator name, its domain and range, and four utility functions (two for validating domain and range and two setter functions for domain and range). Also each mutation operator implements the following two core functions:

- `IsMutationTarget` function to check whether a current statement/expression should be mutated
- `Mutate` function to actually apply mutation

For example, suppose that a user would like to make a new mutation operator SANL (String mutation operator to Add a New Line character) which mutates `StringLiteral` expressions by adding a newline character '`\n`' at the end of a target string. The domain of SANL is a set of strings in a target source file to mutate. Fig. 2.4 shows how function `SANL::IsMutationTarget` is defined.

```
bool IsMutationTarget(Expr *e, ...) {
  if (!isa<StringLiteral>(e))
    return false;

  if (!user_given_domain_.empty()) {
    return user_given_domain_.find(ConvertToString(e));
  }
  else true; // all strings are targeted
}
```

Figure 2.4: `IsMutationTarget` function for SANL

For `Mutate` function, the goal is to add a new `MutantEntry` to `MutantDatabase`. `MutantEntry` contains a mutation operator name, start and end locations of target a statement/expression, target token(s) to mutate and new tokens to replace target token(s). Figure 2.5 shows how function `SANL::Mutate` can be implemented.

```
void Mutate(Expr *e, MusicContext *context) {
  CompilerInstance *CI = context->comp_inst_;
  SourceLocation start_loc = e->getLocStart();
  SourceLocation end_loc = GetEndLocOfExpr(e, CI);

  string token = ConvertToString(e);
  string new_token = token.substr(token.size()-1)+"\\n\"";

  context->mutant_database_.AddMutantEntry(
      mutation_op_name_, start_loc, end_loc,
      token, new_token);
}
```

Figure 2.5: `Mutate` function for SANL

In addition, MUSIC provides several utility classes to help a user build his/her own mutation operators conveniently. For example, suppose that a user wants to make a new mutation operator SCSR which mutates a string to another string in a target program. MUSIC provides `SymbolTable` class that contains categorized lists of statements/expressions of an entire target source file. For SCSR, a user can utilize `SymbolTable::stringliteral_list_` which is a list of `StringLiteral` expressions in the target source code file (i.e., a user can obtain strings to replace a target string by calling `ConvertToString` on each element of `stringliteral_list_`).

2.2.2 No Stillborn Mutants

MUSIC minimizes number of generated stillborn mutants by utilizing type information. First, MUSIC avoids stillborn mutant generation by utilizing type information of operands of target C operators. For example, Fig. 2.6 shows how MUSIC prohibits generating stillborn mutants. Applying OAAN to mutate + to % on Line 5 will generate a stillborn mutant because % should take only integer operands but the second operand of % (i.e., `f`) is a floating number. MUSIC prevents this mutation by analyzing types of operands in a target expression (i.e., `arr[1]+f`).

```
 1: int foo() {
 2:   float f = -1.0; int a = 1; int arr[2];
 3:   arr[0] = a;
 4:   scanf("%d", &arr[a]);
 5:   int sum = arr[1] + f;
 6:
 7:   if (arr[1] < 0) {
 8:     done:
 9:     return (int) f;
10:   }
11:
12:   if (sum < 0)
13:     goto done;
14:
15:   return sum;
16: }
```

Figure 2.6: Example source code to show how to avoid stillborn mutants

Second, MUSIC avoids stillborn mutant generation by utilizing type information of target variables (including contexts of target variables which are stored in `StmtContext` class). For example, while parsing an expression containing a target variable `a` with unary increment (i.e., `a++`), decrement (i.e., `a--`), address-of (i.e., `&a`) or dereference operator (i.e., `*a`), MUSIC does not mutate a target variable `a` to a constant. For another example, if VSRR (Scalar Variable Replacement) mutates an integer variable `a` (used as an index of an array) to a floating variable `f` at Line 4 of Fig. 2.6, the generated mutant will be syntactically illegal, because an array index must be a integer type. Thus, MUSIC does not mutate `a` to `f`.

Third, MUSIC utilizes information about `goto`, `switch` statements to prevent stillborn mutants violating C syntax. For example, SSDL (Statement Deletion) should not be applied to if-statement on Line 7 of Fig. 2.6 because removal of Line 8 will cause a compile error due to missing target label statement of `goto` at Line 13. For `switch` statements, MUSIC checks all `case` labels' values to prevent stillborn mutants caused by a duplicated case label error.

2.3 Case Study: Siemens Benchmarks and cURL

I evaluate applicability and efficiency (i.e., a number of stillborn mutants generated) of MUSIC by applying MUSIC to the seven Siemens benchmark programs in Software-artifact Infrastructure Repository (SIR) [23] and a large real-world modern C program cURL [39]. Also, I compare MUSIC with Milu and Proteum which are popular mutation tools for C programs.

I target Siemens benchmark programs because they have various C language constructs including integer and floating-point arithmetic, `struct`s, pointers, memory allocations, loops, `switch` statements and complex conditional expressions. For this reason, these programs have been widely studied in testing and debugging literature [44, 45, 46]. Siemens benchmark programs are 312.6 LoC long on average (see the second column of Table 2.1).

cURL is a command line tool and library for transferring data through various network protocols including HTTP, FTP, IMAP, etc. I choose cURL because cURL is a very popular open-source project which has 6,700 stars in GitHub. To maintain mutant generation time reasonably, I build cURL to support only HTTP protocol and perform mutant generation on only the cURL command line tool, not library. cURL with only HTTP protocol support is 12,753 LoC long.

2.3.1 Applicability

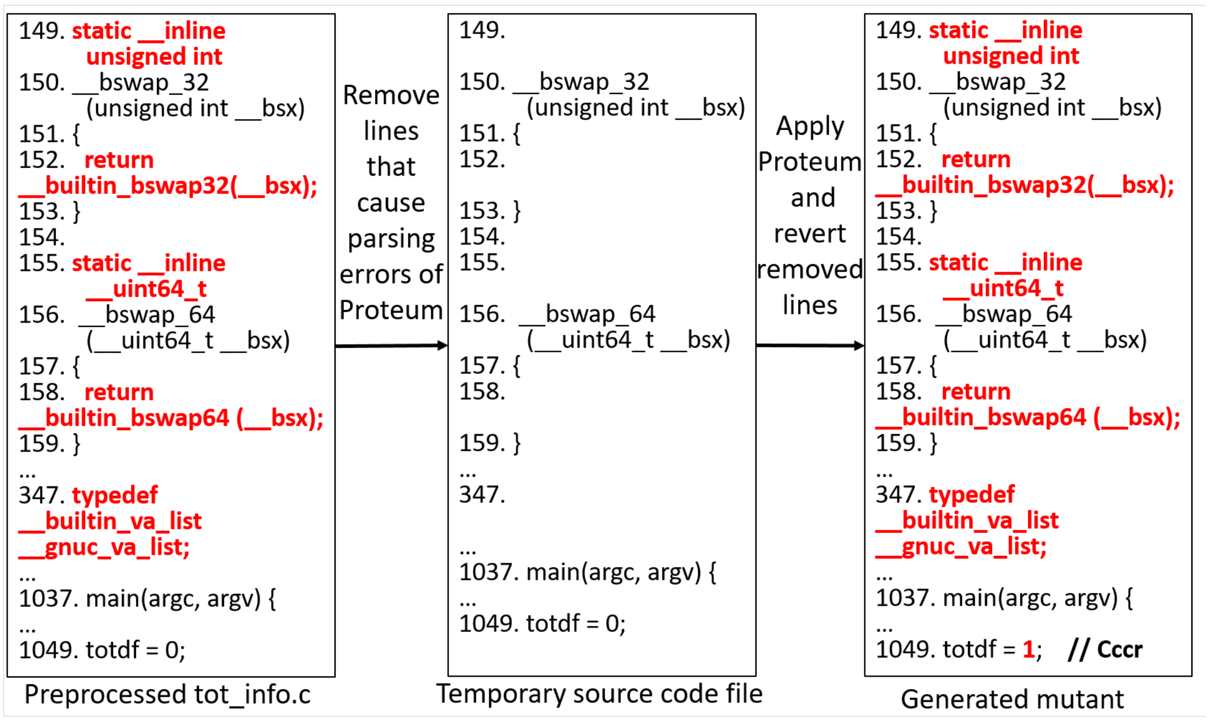
MUSIC clearly shows better applicability than Milu and Proteum. I could apply MUSIC to cURL easily because it takes multiple preprocessed or unpreprocessed C files as input using compilation database (Sect. 2.1). In contrast, Milu and Proteum take only a *single preprocessed* C source file as input. In other words, to apply Milu and Proteum, a user has to manually handle complex compilation information (including macro definitions, header files in separate directories, and so on) for each C file one by one, which causes significant manual overhead for a large project.

Moreover, Proteum often fails to generate mutants for C programs compatible with recent C99 or C11 standards (for example, the system header files of Siemens benchmarks are compatible with C99 standards). A manual workaround for this problem is as follows:

1. A user identifies statement(s) s_f of a target program that make Proteum fail.
2. A user modifies s_f to s' so that Proteum can process a target program without failure.
3. A user generates mutants and then revert s' of every mutant to s_f.

I had to modify five lines for each of `printtokens`, `printtokens2`, `totinfo`, and one line for each of the remaining four Siemens benchmark programs. As an example, Fig. 2.7 shows how I modify the preprocessed source file of `totinfo`. The five lines colored with red in the leftmost box cause parsing errors to Proteum due to inline keyword (Lines 149 and 155), built-in functions (Lines 152 and 158), and built-in type (Line 347). I generate a temporary source file by removing the problematic lines (see the middle box of Fig. 2.7) and apply Proteum to the temporary file. After Proteum generates mutants, I revert the removed lines in each of the mutants (see the right box of Fig. 2.7).

For cURL, Proteum fails to generate a mutant even after I have identified and modified more than 20 lines in the preprocessed `tool_main.c` source file which contains `main()` of cURL. For Milu, I had to manually generate 38 preprocessed source files and apply Milu to each of the preprocessed source files separately. Therefore, MUSIC shows higher applicability to a large real-world modern C program such as cURL than Milu and Proteum.



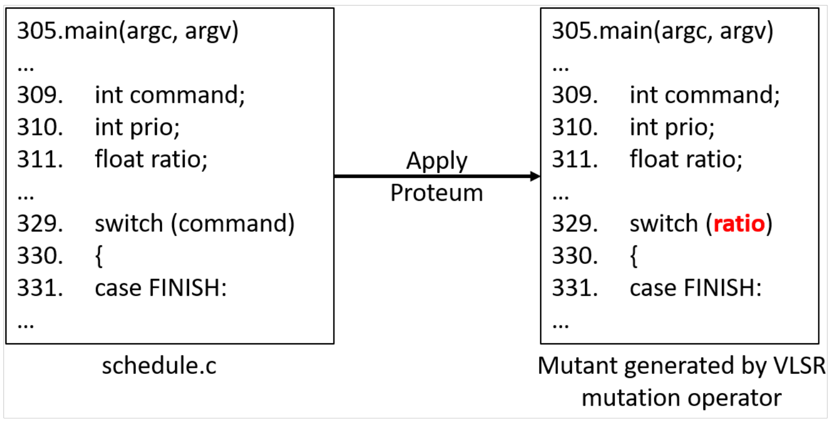
```
149. static __inline
         unsigned int
150. __bswap_32
         (unsigned int __bsx)
151. {
152.   return
__builtin_bswap32(__bsx);
153. }
154.
155. static __inline
         __uint64_t
156.  __bswap_64
         (__uint64_t __bsx)
157. {
158.   return
__builtin_bswap64 (__bsx);
159. }
...
347. typedef
__builtin_va_list
__gnuc_va_list;
...
1037. main(argc, argv) {
...
1049. totdf = 0;
```

```
149.

150. __bswap_32
         (unsigned int __bsx)
151. {
152.

153. }
154.
155.

156.  __bswap_64
         (__uint64_t __bsx)
157. {
158.

159. }
...
347.


...
1037. main(argc, argv) {
...
1049. totdf = 0;
```

```
149. static __inline
         unsigned int
150. __bswap_32
         (unsigned int __bsx)
151. {
152.   return
__builtin_bswap32(__bsx);
153. }
154.
155. static __inline
         __uint64_t
156.  __bswap_64
         (__uint64_t __bsx)
157. {
158.   return
__builtin_bswap64 (__bsx);
159. }
...
347. typedef
__builtin_va_list
__gnuc_va_list;
...
1037. main(argc, argv) {
...
1049. totdf = 1;    // Cccr
```

Remove lines that cause parsing errors of Proteum
Apply Proteum and revert removed lines
Preprocessed tot_info.c
Temporary source code file
Generated mutant

Figure 2.7: Source code modification of tot_info to apply Proteum



```
305.main(argc, argv)
...
309.    int command;
310.    int prio;
311.    float ratio;
...
329.    switch (command)
330.    {
331.    case FINISH:
...
```

```
305.main(argc, argv)
...
309.    int command;
310.    int prio;
311.    float ratio;
...
329.    switch (ratio)
330.    {
331.    case FINISH:
...
```

Figure 2.8: Example of stillborn mutants generated by Proteum

2.3.2 Efficiency

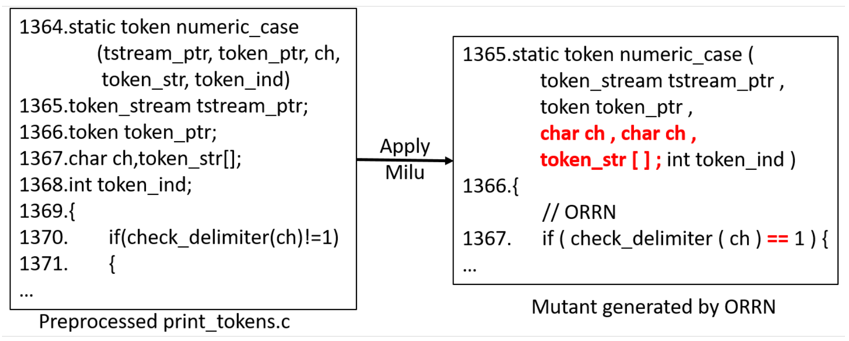
The fifth column of Table 2.1 shows that MUSIC generates no stillborn mutants for Siemens benchmark programs. In contrast, 33.18% and 3.68% of all mutants generated by Milu and Proteum are syntactically illegal and uncompilable, respectively (the sixth column of Table 2.1).

An example of a stillborn mutant generated by Proteum is shown in Fig. 2.8. Proteum's VLSR (Local Scalar Variable Replacement) mutates a condition variable `command` on Line 329 to a floating variable `ratio`. Since switch statement cannot take a floating variable, the generated mutant is syntactically illegal.

Fig. 2.9 shows an example of a stillborn mutant generated by Milu. When applying Milu to mutate function `numeric_case` in `printtokens`, all 170 generated mutants were syntactically illegal due to the

Table 2.1: Number of mutants generated by Milu, Proteum and MUSIC on Siemens C Benchmark programs

Target Program	LOC	Mutation Tool	#Gen. Mutants	#Stillborn Mutants	%Stillborn Mutants
printtokens	343	Milu	3077	995	32.34
		Proteum	4273	200	4.68
		MUSIC	11274	0	0.00
printtokens2	355	Milu	2424	523	21.58
		Proteum	4680	162	3.46
		MUSIC	4791	0	0.00
replace	513	Milu	3927	353	8.99
		Proteum	10872	509	4.68
		MUSIC	9925	0	0.00
schedule	296	Milu	1310	640	48.85
		Proteum	2241	103	4.60
		MUSIC	2365	0	0.00
schedule2	263	Milu	1919	915	47.68
		Proteum	2950	114	3.86
		MUSIC	3033	0	0.00
tcas	137	Milu	874	271	31.01
		Proteum	2872	74	2.58
		MUSIC	3415	0	0.00
totinfo	281	Milu	2381	996	41.83
		Proteum	6390	122	1.91
		MUSIC	10486	0	0.00
Average	**312.6**	**Milu**	**2273.1**	**670.4**	**33.18**
		Proteum	**4896.9**	**183.4**	**3.68**
		MUSIC	**6469.9**	**0**	**0.00**



```
1364.static token numeric_case
          (tstream_ptr, token_ptr, ch,
           token_str, token_ind)
1365.token_stream tstream_ptr;
1366.token token_ptr;
1367.char ch,token_str[];
1368.int token_ind;
1369.{
1370.     if(check_delimiter(ch)!=1)
1371.     {
...
```

```
1365.static token numeric_case (
          token_stream tstream_ptr ,
          token token_ptr ,
          char ch , char ch ,
          token_str [ ] ; int token_ind )
1366.{
          // ORRN
1367.     if ( check_delimiter ( ch ) == 1 ) {
...
```

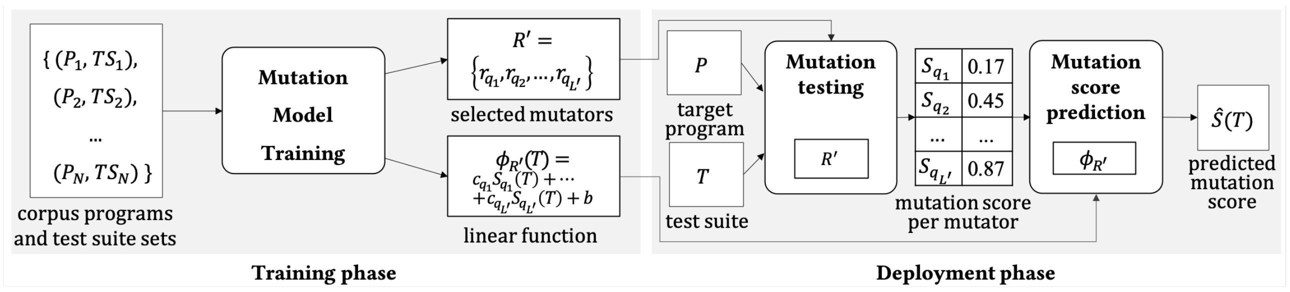
Figure 2.9: Example of stillborn mutants generated by Milu

syntax errors occurred in function definition: redefinition of parameter `ch` and inclusion of semicolon in the list of function parameters.

For cURL, MUSIC generates no stillborn mutant while Milu generates 31,232 ones (i.e., 75.31% of the generated mutants are syntactically illegal). I found that this large number of stillborn mutants is caused by incorrect handling of types including `typedef`, `enum`, `const` type qualifier and an array type.

Chapter 3. REFINER: Refined Mutation Operator-based Mutant Reduction

3.1 Overview



{ (P_1, TS_1), (P_2, TS_2), ... (P_N, TS_N) }
corpus programs and test suite sets
Mutation Model Training
$R' = \{r_{q_1}, r_{q_2}, ..., r_{q_{L'}}\}$
selected mutators
$\phi_{R'}(T) = c_{q_1}S_{q_1}(T) + \cdots + c_{q_{L'}}S_{q_{L'}}(T) + b$
linear function
Training phase
P
target program
T
test suite
Mutation testing
R'
S_{q_1} 0.17
S_{q_2} 0.45
... ...
$S_{q_{L'}}$ 0.87
mutation score per mutator
Mutation score prediction
$\phi_{R'}$
$\hat{S}(T)$
predicted mutation score
Deployment phase

Figure 3.1: The overall process of REFINER

REFINER aims to minimize the number of mutants for predicting hard-to-kill mutation scores of a test suite. REFINER utilizes a set of existing programs and their test suites as corpus to learn a prediction model using a small number of representative mutation operators to predict hard-to-kill mutation scores. Figure 3.1 describes the overall process of REFINER. Given corpus programs (P_i) and a set of test suites for P_i (TS_i), REFINER produces (1) $R' = \{r_{q_1}, r_{q_2}, ..., r_{q_{L'}}\}$, a small subset of the fine-grained mutation operators $R = \{r_1, r_2, ..., r_L\}$ (see Section 3.3) to generate mutants and (2) $\phi_{R'}$, a linear function to compute an expected hard-to-kill mutation score from the mutation testing results using R'. The resulting pair, R' and $\phi_{R'}$, works as a mutation testing model for predicting the hard-to-kill mutation score of a given test suite T of a target program P through the following steps:

- **Step 1.** Apply each mutation operator $r \in R'$ to P for generating a set of mutants M_r .
- **Step 2.** Run each mutant of M_r with the given test suite T.
- **Step 3.** For each mutation operator r, compute a mutation score $S_r(T)$ as the ratio of the number of killed mutants by T in M_r to all mutants in M_r. I refer to $S_r(T)$ as a *mutation score per mutation operator r*.
- **Step 4.** Predict a hard-kill-mutation score of T by evaluating $\phi_{R'}$.

Having given corpus (i.e., pairs of programs and the test suites) as training data, REFINER searches for a combination of fine-grained mutation operators to satisfy the following two objects at the same time:

1. *Accuracy:* a linear combination of mutation scores per mutation operator should accurately predict actual hard-to-kill mutation scores of a test suite, and
2. *Efficiency:* the total number of generated mutants should be significantly smaller than the number of mutants generated by all available mutation operators.

I extend the approach of Namin et al. [13] to the following directions:

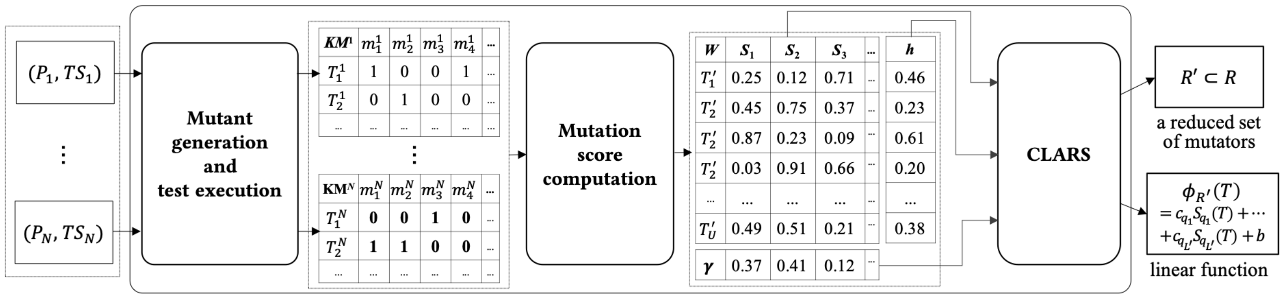
- Develop a set of *fine-grained mutation operators* (see Section 3.3) to reduce mutation testing cost further, and
- Predict the mutation scores of hard-to-kill mutants (instead of all mutants) to reduce mutation testing cost even further

REFINER targets hard-to-kill mutation score because it can assess fault detection capability of a test suite more accurately than the whole set of all mutants [13, 17, 19, 20, 21]. To my best knowledge, REFINER is the first approach to predict hard-to-kill mutation score of a test suite.

The remaining of Chapter 3 is structured as follows. Section 3.2 details how REFINER trains a model using given corpus programs and test suites. Section 3.3 presents fine-grained mutation operators employed by REFINER. Finally, Section 3.4 explains how REFINER works in details with an example.

3.2 Prediction Model Training

3.2.1 Training data generation



(P_1, TS_1)
⋮
(P_N, TS_N)
Mutant generation and test execution
Mutation score computation
CLARS
$R' \subset R$
a reduced set of mutators
$\phi_{R'}(T) = c_{q_1}S_{q_1}(T) + \cdots + c_{q_{L'}}S_{q_{L'}}(T) + b$
linear function

KM^1	m_1^1	m_2^1	m_3^1	m_4^1	...
T_1^1	1	0	0	1	...
T_2^1	0	1	0	0	...
...	...	...	...	...	...

KM^N	m_1^N	m_2^N	m_3^N	m_4^N	...
T_1^N	0	0	1	0	...
T_2^N	1	1	0	0	...
...	...	...	...	...	...

W	S_1	S_2	S_3	...	h
T_1'	0.25	0.12	0.71	...	0.46
T_2'	0.45	0.75	0.37	...	0.23
T_2'	0.87	0.23	0.09	...	0.61
T_2'	0.03	0.91	0.66	...	0.20
...	...	...	...	...	...
T_U'	0.49	0.51	0.21	...	0.38
γ	0.37	0.41	0.12	...	

Figure 3.2: The overall process of the training phase of REFINER

REFINER utilizes a linear regression technique CLARS (Cost-considerate Least Angle Regression) to train a linear model that accurately predicts hard-to-kill mutation score of a test suite based on a set of mutation scores per mutation operators. To train a prediction model, REFINER conducts mutation testing on given corpus programs with given corpus test suites, measures the mutation scores per mutation operator and the hard-to-kill mutation score of each test suite, and feed this data to CLARS.

Figure 3.2 describes how the Prediction Model Training module (shown in Figure 3.1) constructs training data and trains a prediction model using CLARS. For each corpus program P_i with a set of test suites $TS_i = \{T_1^i, T_2^i, ...\}$, REFINER first generates mutants $M^i = \{m_1^i, m_2^i, ...\}$ by employing all fine-grained mutation operators R (see Section 3.3).

From all generated mutants, REFINER eliminates trivially equivalent and duplicated mutants [24] after mutant generation. REFINER identifies an equivalent mutant by checking whether or not the MD5 checksum of the compiled binary executable of a mutant is the same as that of the original target program. In a similar way, two mutants are identified as duplicated mutants if the compiled binary executables of the two mutants have the same MD5 checksum.

Running each mutant m_j^i with each test suite T_k^i, REFINER constructs a killmap $KM^i : M^i \times TS^i \to \{0, 1\}$ of a corpus program P_i, such that $KM^i(m_j^i, T_k^i) = 1$ if and only if mutant m_j^i is killed by test suite

T_k^i. Following previous studies on mutant selection techniques [13, 17], REFINER considers mutants that are never killed by any test suite as equivalent mutants, and eliminate them from the mutant set. Note that, although the number of fine-grained mutation operators in R is greater than the number of the existing coarse-grained mutation operators (i.e., Proteum), R is designed to generate the same set of mutants (see Section 3.3).

From the set of obtained killmaps, REFINER first gives a natural number index to all test suites of all corpus programs, such that test suites can be referred as $T'_1, T'_2, ..., T'_U$. Then, REFINER constructs a vector v_i for each test suite T'_i such that the k-th element of v_i, $v_i[k]$ contains the mutation score per fine-grained mutation operator r_k. REFINER combines vectors v_1 to v_U to construct a $U \times L$ matrix W. In addition, REFINER computes the hard-to-kill mutation score of each test suite and then constructs a column vector h where the k-th row of h, $h[k]$ contains the hard-to-kill mutation score of test suite T'_k. Last, REFINER constructs a cost vector γ where $\gamma[k]$ represents the total number of mutants generated by fine-grained mutation operator r_k over all corpus programs.

3.2.2 Cost-considerate Least Angle Regression (CLARS)

CLARS is a linear regression technique that finds a list of coefficients c_1, c_2, ..., c_n , and a constant (intercept) b, that fit given data of independent variables $X = \{x_1, x_2, ..., x_n\}$, and a dependent variable, y. Unlike conventional linear regression techniques, CLARS additionally receives a cost factor, d_i for each independent variable x_i. In a training for achieving high accuracy (i.e., low error), CLARS selects and uses only a small subset of input variables, $X' \subseteq X$ such that the sum of their cost factors becomes minimized. As a result of training, a coefficient of an independent is assigned with a non-zero value if the corresponding independent variable is selected by CLARS. Otherwise, the coefficient of a non-selected independent variable is assigned with zero (i.e., $X' = \{x_i \in X | c_i \neq 0\}$). When the accuracy objective is to minimize Mean Squared Errors (MSE), CLARS searches for a linear model such that

- minimize $\sum_{i=1}^{n}(x_i \times c_i + b - y)^2/n$, and
- minimize $\sum_{x_i \in X'} d_i$

REFINER feeds W as independent variables, h as dependent variables, and γ as cost factors obtained from the mutation testing results of the corpus programs (see Section 3.2.1) to CLARS. For given data, CLARS produces a coefficient vector C and a constant vector b that minimizes $|(W \times C + b) - h|^2/U$ (i.e, Mean Squared Errors) and, at the same time, minimize the total number of mutants generated by the selected fine-grained mutation operators, $\sum_{r_k \in R'} \gamma[k]$ where $R' = \{r_k \in R | C[k] \neq 0\}$. Note that a fine-grained mutation operators r_k is selected by CLARS when $C[k]$ is not zero. CLARS considers the sum of weights of the non-zero coefficients as the cost of a trained model, and leads a training to minimize both the square sum of error and the cost simultaneously.

Compared to Namin et al. [13], REFINER applies CLARS to minimize the Mean Squared Errors (MSE) of prediction, instead of maximizing R^2. For training an accurate prediction model of hard-to-kill mutation score, I believe that minimizing MSE is a more appropriate training objective than maximizing R^2. Note that, since R^2 measures the linearity of predicted results rather than the accuracies, larger values of R^2 can be obtained even when the predicted values differ considerably from the truth values but the predicted values and the truth values have high linear correlation [25].

3.3 Fine-grained Mutation Operators

I define total 3711 fine-grained mutation operators by refining existing coarse-grained mutation operators. The fine-grained mutation operators are designed to generate the same set of mutants generated by the existing mutation operators. The benefit of the fine-grained mutation operators is that REFINER can explore more diverse combinations of mutation operators, thus, have a higher chance to find more accurate and more efficient prediction model, compared to the existing coarse-grained mutation operators.

My conjecture is that the traditional mutation operator r_i might be *similar/redundant* to each other (i.e., a set of mutants M_i generated by applying r_i may be highly correlated with M_j) (see RQ1 in Section 4.2.1)). This means that many of the mutants generated by the traditional mutation operators r_i are not useful to learn an accurate and efficient prediction model. On the other hand, the correlation between fine-grained mutation operators is lower than that between the traditional mutation operators. For example, more than 80% of all pairs of the traditional mutation operators in `make` has high correlation (i.e., Pearson correlation $r \geq 0.9$) while around only 40% of all pairs of the fine-grained mutation operators in `make` has high correlation. Thus, CLARS using the fine-grained mutation operators can learn an accurate and cost-efficient prediction model because the fine-grained mutation operators are more distinct and low-cost than the traditional mutation operators.

The comprehensive set of the traditional mutation operators for C programs consists of 75 mutation operators defined by Agrawal et al. [31] and 33 interface mutation operator defined by Delamaro et al. [32]. A mutation operator can be represented as a multi-valued function that maps source code patterns (i.e., domain) to transformed source code patterns (i.e., range or codomain). Suppose that there exists a traditional mutation operator r_i with domain X_i and range Y_i. Then, r_i maps a source code pattern $x \in X_i$ to one or more instances in Y_i (i.e., $r_i(x) \subseteq Y_i$). For traditional mutation operator r_i, I define a set of fine-grained mutation operators $r_{i,1}, r_{i,2}, ..., r_{i,n}$, such that the domains (ranges) of all refined mutation operators form a partition of the domain (range) of r_i. As a result, I define 3711 fine-grained mutation operators from the 108 traditional ones.

I classify the domains and ranges of the traditional mutation operators into four and five categories, respectively. Then, I define fine-grained mutation operators according to their domain and range categories. The domains of the traditional mutation operators can be classified to the following four categories:

1. *A set of operators*:

   Traditional mutation operators whose domain is in this category replace an operator into another. For example, OAAN (arithmetic operator mutation) has a domain of $\{+, -, *, /, \%\}$ and replaces a single arithmetic operator (e.g., $+$) with another (e.g., $*$).

2. *A set of variables*:

   Traditional mutation operators whose domain is in this category replace a variable into another variable, constant, or expression (e.g., mutating a variable `x` into `++x`). For example, VLSR (scalar references mutation using local scalar references) has a domain of $\{v | v \text{ is a local scalar variable}\}$ and replaces a single local scalar variable (e.g., `x`) with another (e.g., `y`).

3. *A set of constants*:

   Traditional mutation operators whose domain is in this category replace a constant into another

```
/* A simple binary search program */
int main(){
  int array[]={10,20,30,40,50,60,70,80,90,100};
  int n = 10, value=30, low, high, mid, i
  low = 1;
  high = n;
  while (low < high) {
    mid = (low + high)/2;
    if (array[mid] < value)
      low = mid + 1;
    else
      high = mid;
  }

  if ((low <  n) && (array[low] == value)){
    printf("Found, array[%d]=%d\n", low, value);
  else
    printf("Not found\n");}
```

Figure 3.3: An example program

constant, variable, or expression. For example, CLCR (constant replacement using local constants) has a domain of $\{c|c \text{ is a local constant}\}$ and replaces a local constant with another.

4. *A set of statements*:

   Traditional mutation operators whose domain is in this category replace a statement into another. For example, SSDL (statement deletion) has a domain of $\{s|s \text{ is a statement}\}$ and deletes one statement by replacing it with an empty statement (i.e. `;`).

The range of the traditional mutation operators include the four categories of the domain (i.e., operators, variables, constants, and statements) and *expressions* (i.e., traditional mutation operators that replace a variable or a constant with an expression have a set of expressions as a range). For example, in Figure 3.3, IndVarIncDec (Indirect Variable Increment Decrement) replaces a local variable with an expression obtained by adding `++` or `--` before or after the target local variable (e.g., IndVarIncDec mutates `a = b + c` to `a = b + c++` or `a = b + --c`, in which `c` is a local variable).

I define fine-grained mutation operators as follows, according to their domain and range categories:

1. *A set of operators*:

   A fine-grained mutation operator is defined to have its domain and range as a singleton set of operators. For example, the domain and range of OAAN is $\{+, -, *, /, \%\}$ and a fine-grained mutation operator OAAN$_{op1 \rightarrow op2}$ has its domain and range as one of $\{+\}$, $\{-\}$, $\{*\}$, $\{/\}$, and

{%}. One example of OAAN fine-grained mutation operators is $\text{OAAN}_{+\to*}$, which has domain and range as $\{+\}$ and $\{*\}$, respectively.

2. *A set of variables*:

   A fine-grained mutation operator is defined to have its domain and range as one of the n partitions of the entire set of variables as follows. First, I make an ordered list l_v of all target variables in the target program (the variables are sorted in a lexicographical order on the variables' names). Second, I divide l_v into n partitions p_1,...,p_n whose sizes are almost equal (if the size of the domain/range is not divisible by n, p_1 to p_m ($m < n$) has one more element than p_{m+1} to p_n). Finally, I define a fine-grained mutation operator on each of $p_1, ..., p_n$, each of which could be a domain and/or a range.

   Figure 3.3 shows an example program that does a simple binary search. Since the domain and range of VLSR is a set of local scalar variables, VLSR has {`high, i, low, mid, n, value`} as its domain and range. Thus, the domain and range of a fine-grained mutation operator of VLSR with $n = 10$ are one of the following partitions: p_1={`high`}, p_2= {`i`}, p_3={`low`}, p_4={`mid`}, p_5={`n`}, and p_6={`value`}, $p_7 = p_8 = p_9 = p_{10} = \emptyset$. A fine-grained mutation operator $\text{VLSR}_{i\to j}$ has its domain and range as p_i and p_j, respectively. For example, $\text{VLSR}_{1\to2}$ replaces a variable `high` with a variable `i`.

3. *A set of constants*:

   A fine-grained mutation operator on a set of constants is defined as similar to that on a set of variables. I make an ordered list of constants and divide the ordered list into multiple partitions whose sizes are almost equal, on which the fine-grained mutation operators are defined.

   Looking at Fig. 3.3 again, CLCR (Local Constant Replacement) has {`1, 2, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100`} (12 elements) as its domain and range. Thus, the domain and range of a fine-grained mutation operator of CLCR with $n = 10$ are one of the following partitions: p_1={`1, 2`}, p_2= {`10, 20`}, p_3={`30`}, p_4={`40`}, p_5={`50`}, p_6={`60`}, p_7={`70`}, p_8={`80`}, p_9={`90`}, and p_{10}={`100`}. A fine-grained mutation operator $\text{CLCR}_{i\to j}$ has its domain and range as p_i and p_j, respectively. For example, $\text{CLCR}_{1\to2}$ replaces a constant `1` or `2` with another constant `10` or `20`.

4. *A set of statements*: A fine-grained mutation operator on a set of constants is defined as similar to those on a set of variables and a set of constants. I make an ordered list of statements in an ascending order of the statements' line number and divide the ordered list into multiple partitions whose sizes are almost equal, on which the fine-grained mutation operators are defined.

   Again, in Figure 3.3, SSDL has {4, 5, 6, 7, 8, 9, 11, 14, 15, 17} (each number represents a statement at that line) (10 elements in total) as its domain. Thus, the domain of a fine-grained mutation operator of SSDL with $n = 10$ are one of the following partitions: p_1={`4`}, p_2= {`5`}, p_3={`6`}, p_4={`7`}, p_5={`8`}, p_6={`9`}, p_7={`11`}, p_8={`14`}, p_9={`15`}, and p_{10}={`17`}. A fine-grained mutation operator SSDL_i has its domain as p_i. For example, SSDL_1 removes the statement at Line 4 (i.e., `low=1;`).

5. *A set of expressions as a range*: For a fine-grained mutation operator $r_{i,j}$ of a traditional mutation operator o_i that has a set of expressions as its range, $r_{i,j}$ is defined to have its range as a singleton set of C expressions.

Table 3.1: An example of mutation scores of the traditional and and fine-grained mutation operators in the training set

Pro-gram	Test-suite	r_1				r_2				r_3				h
		$r_{1,1}$	$r_{1,2}$	$r_{1,3}$		$r_{2,1}$	$r_{2,2}$	$r_{2,3}$		$r_{3,1}$	$r_{3,2}$	$r_{3,3}$		
P_1	T_1^1	0.20	0.15	0.07	0.14	0.09	0.07	0.12	0.09	0.06	0.00	0.22	0.08	0.10
	T_2^1	0.20	0.15	0.71	0.38	0.27	0.00	0.00	0.07	0.56	0.83	0.78	0.70	0.36
	T_3^1	1.00	0.23	0.71	0.62	0.36	0.20	0.59	0.40	0.38	0.75	1.00	0.65	0.54
P_2	T_1^2	0.20	0.11	0.24	0.18	0.25	0.09	0.05	0.12	0.11	0.24	0.36	0.22	0.17
	T_2^2	0.67	0.47	0.47	0.53	0.44	0.27	0.35	0.34	0.26	0.43	0.64	0.43	0.43
	T_3^2	0.93	0.63	0.59	0.71	0.88	0.68	0.60	0.71	0.74	0.71	0.93	0.78	0.73
# mutants		25	32	31	88	27	37	37	101	35	33	23	91	

Table 3.2: An example of mutation scores of the traditional and and fine-grained mutation operators in the validation set

Pro-gram	Test-suite	r_1				r_2				r_3				h
		$r_{1,1}$	$r_{1,2}$	$r_{1,3}$		$r_{2,1}$	$r_{2,2}$	$r_{2,3}$		$r_{3,1}$	$r_{3,2}$	$r_{3,3}$		
P	T_1	0.00	0.17	0.22	0.14	0.26	0.18	0.33	0.26	0.15	0.09	0.09	0.11	0.17
	T_2	0.65	0.00	0.44	0.36	0.21	0.65	0.67	0.51	0.23	0.18	0.48	0.30	0.38
	T_3	0.05	0.57	0.33	0.33	0.53	0.29	0.71	0.53	0.73	0.32	0.87	0.65	0.50
# mutants		20	23	27	70	19	17	21	57	26	22	23	71	

For example, the domain D of IndVarIncDec is a list of local variables sorted in a lexicographical order of variable names, and the range of IndVarIncDec is {`v++, ++v, v--, --v`} where `v` is a local variable in D. A fine-grained mutation operator IndVarIncDec$_{i\to j}$ has its domain as p_i which is the i^{th} partition of D and its range as one of {`v++`}, {`++v`}, {`v--`}, and {`--v`}. For example, in Figure 3.3, IndVarIncDec$_{1\to 2}$ replaces a variable `high` ($p_1 =$ {`high`}) with `++high`.

3.4 Example

This example shows that REFINER can select fewer mutants and predict a mutation score more accurately than REFINER^{TRD}. Suppose that there are three traditional mutation operators r_1, r_2, and r_3 and refine each traditional mutation operator r_i into three fine-grained mutation operators $r_{i,1}$, $r_{i,2}$, and $r_{i,3}$. REFINER and REFINER^{TRD} learn a linear regression model using P_1 and P_2, and apply the learned model to predict a hard-to-kill mutation score of P using a subset of fine-grained and a subset of traditional mutation operators, respectively.

Table 3.1 shows the mutation scores and the number of generated mutants of the fine-grained and traditional mutation operators for the corpus programs P_1 and P_2. The first and second column show the program and test suite names. The third to fourteenth columns show the mutation score of the corresponding fine-grained or traditional mutation operator (column) using the corresponding test suite (row). The last column shows the mutation score of the hard-to-kill mutants. The last row show the total number of generated mutants of fine-grained and traditional mutation operators for P_1 and P_2. Similarly, Table 3.2 shows the mutation score and the number of generated mutants of the fine-grained and traditional mutation operators for the program P in the validation set.

Applying CLARS with the fine-grained mutation operators achieves a higher accuracy in prediction (0.005 vs. 0.011 in *Mean Squared Error* (MSE) of mutation scores on the selected mutants and all mutants) and selects fewer mutants (62 vs. 141 mutants) than applying CLARS to the traditional ones as shown below:

- REFINER^{TRD} selects r_1, r_3 and generates the following model:

  $$0.81S_1(T) + 0.16S_3(T) - 0.03 \tag{3.1}$$

  where $S_k(T)$ is a mutation score of a test suite T with respect to the mutants generated by a mutation operator r_k. This model selects 141 out of 198 mutants of P, and predicts the hard-to-kill mutation scores of 3 test suites of P to be 0.10, 0.31, 0.34 respectively. The MSE between the predicted mutation scores (0.10, 0.31, 0.34) and actual mutation scores (0.17, 0.38, 0.50) is 0.011.

- REFINER selects $r_{1,1}$, $r_{2,1}$, $r_{3,3}$ and generates the following model:

  $$0.07S_{1,1}(T) + 0.34S_{2,1}(T) + 0.31S_{3,3}(T) + 0.024 \tag{3.2}$$

  This model selects 62 out of 198 mutants of P, and predicts the hard-to-kill mutation scores of 3 test suites of P to be 0.14, 0.29, 0.47 respectively. The MSE between the predicted mutation scores (0.14, 0.29, 0.47) and actual mutation scores (0.17, 0.38, 0.50) is 0.005.

As the example shows, REFINER using fine-grained mutation operators can reduce more mutants and predict mutation score more accurately than the traditional mutation operator-based mutant selection.

Chapter 4. Experiments and Results

4.1 Experiment Setup

4.1.1 Research Questions

I have designed the following research questions to evaluate the effectiveness of REFINER on six SIR programs.

RQ1. Correlation between the fine-grained mutation operators: How much are the refined mutation operators correlated to each other in terms of the mutation scores compared to the coarse-grained mutation operators?

RQ2. Effect of the fine-grained mutation operators on REFINER: How much do the selected fine-grained mutation operators affect the size of M' and the accuracy of the predicted hard-to-kill mutation score compared to the coarse-grained mutation operators?

RQ3. Comparison of REFINER with random mutant selection: With the same number of the selected mutants, how much does REFINER increase the accuracy of the predicted hard-to-kill mutation score compared to the random mutant selection technique?

RQ4. Comparison of REFINER with existing mutation-operator based mutant selection techniques: How much does REFINER reduce the number of selected mutant and increase the accuracy of the predicted hard-to-kill mutation score compared to the existing mutation operator-based mutant selection techniques?

4.1.2 Mutant Selection Techniques to Compare

I have evaluated REFINER and the following mutant selection techniques in the experiments:

- REFINER: For fine-grained mutation operators of REFINER, I extended MUSIC [9] to implement 3711 fine-grained mutation operators refined from the 108 coarse-grained mutation operators. REFINER uses the R package developed by Michael Lerch [18, 26]. REFINER is configured to apply CLARS to the training dataset for 10 minutes, and selected the model with the lowest MSE with respect to the training dataset. The training is stopped after 10 minutes because, after 10 minutes, MSE is almost converged (i.e., MSE difference between models is less than 0.0005).

  To evaluate REFINER with a target SIR program, REFINER uses the other five SIR programs and their test suites as corpus to train a prediction model, and then the trained prediction model is evaluated with the target SIR program (i.e., six-fold cross-validation).

- REFINER^{TRD}: This technique is similar to REFINER but uses the coarse-grained mutation operators instead of the fine-grained mutation operators (REFINER^{TRD} is similar to the method of Namin et al. [13]). I used this technique for answering RQ2.

- RND^{SN}: This technique randomly selects the same number of mutants to the number of mutants generated by REFINER. I used this technique for answering RQ3.

Table 4.1: Statistics of SIR target programs

Target pgms.	LoC	#tests	#generated mutants	#used mutants	#hard-to-kill k% mutants $k = 3$	$k = 5$	$k = 7$
flex	7254	567	1968398	1146413	312561	337336	347732
grep	5696	809	347636	187913	78929	86532	95984
gzip	3040	208	739923	371289	107033	114904	198787
make	9820	1006	585224	282328	169702	172148	173035
sed	3980	360	174759	72187	11064	16290	20264
space	5489	13585	183444	108598	55377	61075	65137
Avg.	5879.8	2755.8	666564.0	361454.7	122444.3	131380.8	150156.5

- E-SELECTIVE: This technique uses the mutants generated by only four C mutation operators OAAN, ORRN, OLLN and OLNG to predict mutation score. These four C operators corresponds to Offutt et al.'s sufficient set of 5 Fortran mutation operators. Only four out of five has a corresponding C mutation operators, indicated in parenthesis in the list below:

  1. ABS: Absolute Value
  2. AOR: Arithmetic Operator Replacement (C: OAAN)
  3. LCR: Logical Connector Replacement (C: OLLN)
  4. ROR: Relational Operator Replacement (C: ORRN)
  5. UOI: Unary Operator Insertion (C: OLNG)

  E-SELECTIVE was used for answering RQ4.

- SSDL: This technique uses only mutants generated by applying SSDL mutation operator to predict mutation score. SSDL has been studied in previous researches as a cost-effective mutant selection approach [27, 28, 29]. This technique was used for answering RQ4.

4.1.3 Target Programs

Table 4.1 shows the information on the six target C programs in the SIR benchmark [23] used in the experiments, including their names, executable LoC, numbers of test cases and numbers of mutants generated by the 108 coarse-grained mutation operators. Each target program has 5879.8 LoC, 2755.8 test cases, and 666564.0 generated mutants on average. I chose SIR programs as target programs because they use various C language features such as integer and floating-point arithmetic, pointer arithmetic, loops, `switch-case`, and `structs`.

4.1.4 Test Suite Generation

For each of the six SIR programs, I generated 1000 test suites. To generate a test suite, I randomly selected a test case from the pool of the test cases provided in the SIR benchmark and added to the test suite until it reached the chosen size. For each program P and its test suite T (provided in SIR), let $|T|$ be the size of T. For each test suite size of 1% $\times$ $|T|$, 3% $\times$ $|T|$, 5% $\times$ $|T|$, ..., 99% $\times$ $|T|$, I generated 20 test suites. In other words, I generated 20 test suites having 1% $\times$ $|T|$ test cases, 20 test suites having

$3\% \times |T|$ test cases, and so on. I chose these test suite sizes because I want to generate test suites that achieve various range of hard-to-kill mutation score from 0.0 to 1.0.

4.1.5 Hard-to-kill Mutation Scores

I identified hard-to-kill mutants of the six SIR programs and measured hard-to-kill scores of all generated test suites. Similar to the training generation process of REFINER (see Section 3.2.1), I generated all possible mutants by applying all coarse-grained mutation operators of MUSIC, and then eliminate trivially equivalent, duplicated mutants, and equivalent mutants with respect to the given test suites. In total, out of 3,999,384 mutants generated by MUSIC, I eliminated 962,300 equivalent mutants and 868,356 redundant mutants.

I referred to the remaining mutants as used mutants and reported the number of the used mutants for each program in the fifth column of Table 4.1. For each program, the used mutants which are killed by less than 3%, 5%, and 7% of available test suites are represented as hard-to-kill 3%, hard-to-kill 5%, hard-to-kill 7% mutants respectively. The number of the hard-to-kill mutants for each program is showed in the sixth to the last column of Table 4.1.

4.1.6 Measurement

To evaluate the mutant selection techniques, I measure the following two items:

- Number of the selected mutants (either by REFINER, random mutant selection, Offutt et al.'s selective mutation operators or SSDL)
- Mean Square Error (MSE) between actual hard-to-kill mutation score of each test suite and the mutation score measured by a mutant selection technique (i.e., predicted hard-to-kill mutation score by the technique).

To reduce the variance of random selection, I repeated the experiment that uses random mutant selection techniques 30 times and reported the average value of the results.

4.2 Experiment Results

This section reports and discusses the experiment results to answer the research questions with regard to $\mathcal{M}$ as a set of mutants generated by all the coarse-grained mutation operators.

4.2.1 RQ1: Correlation between Fine-grained mutation operators

The experiment results in Figure 4.1– 4.6 show that the coarse-grained mutation operators are more highly correlated to each other than the fine-grained mutation operators , which supports my conjecture in Section 3.3. As a representative case, Figure 4.4 shows the Pearson correlation distribution of the pairs of the coarse-grained mutation operators (left of Figure 4.4, and the fine-grained mutation operators (right of Figure 4.4) for `make`. In case of `make`, around 80% of the coarse-grained mutation operator pairs are highly correlated to each other in terms of mutation scores (Pearson's Correlation ≥ 0.90), while only 40% of fine-grained mutation operator pairs are highly correlated to each other. The other five target programs show the similar distribution. Thus, I can conclude that the fine-grained mutation operators are more distinct than the traditional mutation operators, which can help REFINER generate more accurate and efficient models.

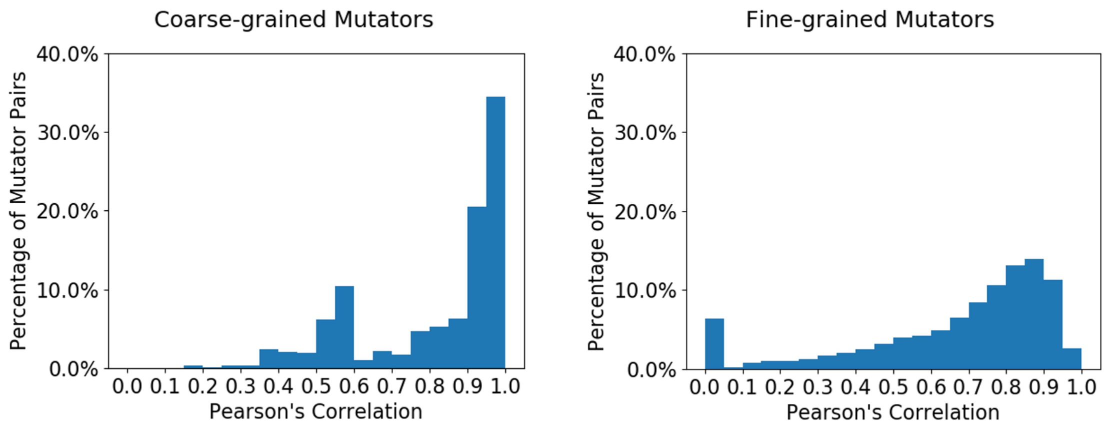


Figure 4.1: Pearson correlation distribution of the pairs of the coarse-grained mutaion operators (left) and the fine-grained mutaion operators (right) of `flex`

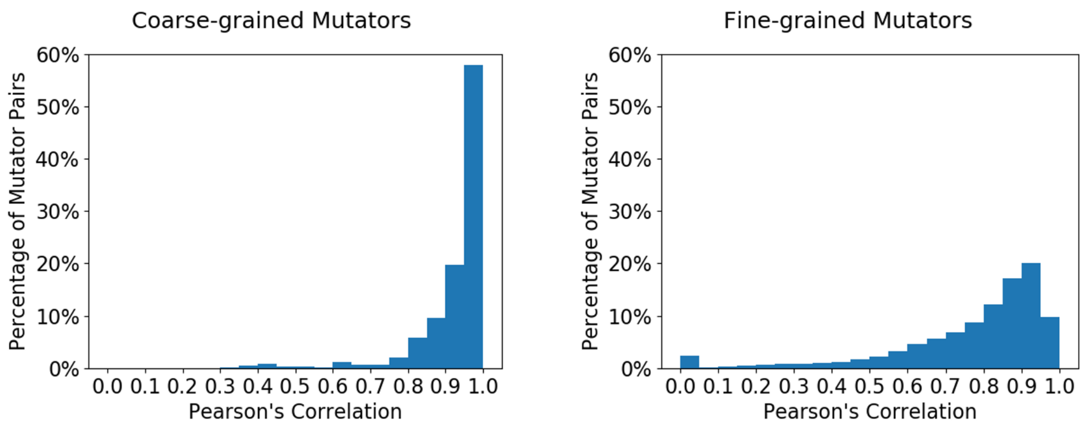


Figure 4.2: Pearson correlation distribution of the pairs of the coarse-grained mutaion operators (left) and the fine-grained mutaion operators (right) of `grep`

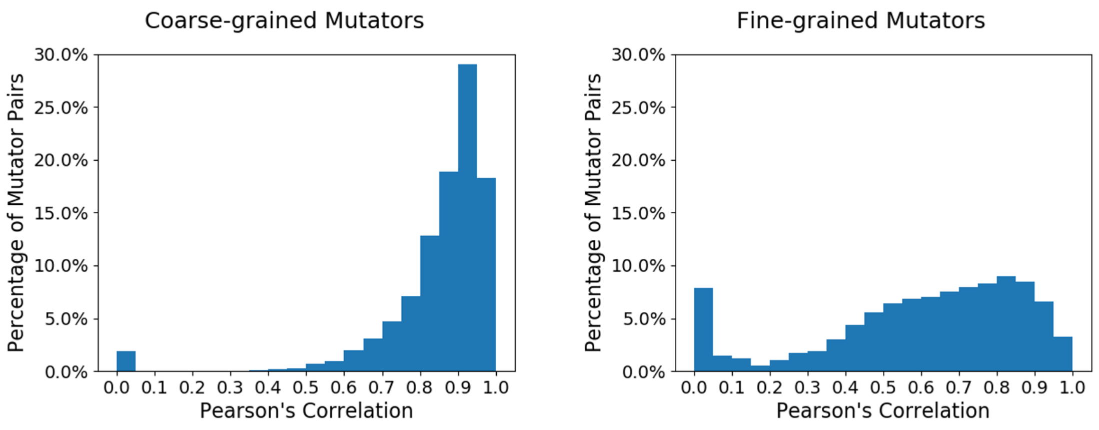


Figure 4.3: Pearson correlation distribution of the pairs of the coarse-grained mutaion operators (left) and the fine-grained mutaion operators (right) of `gzip`

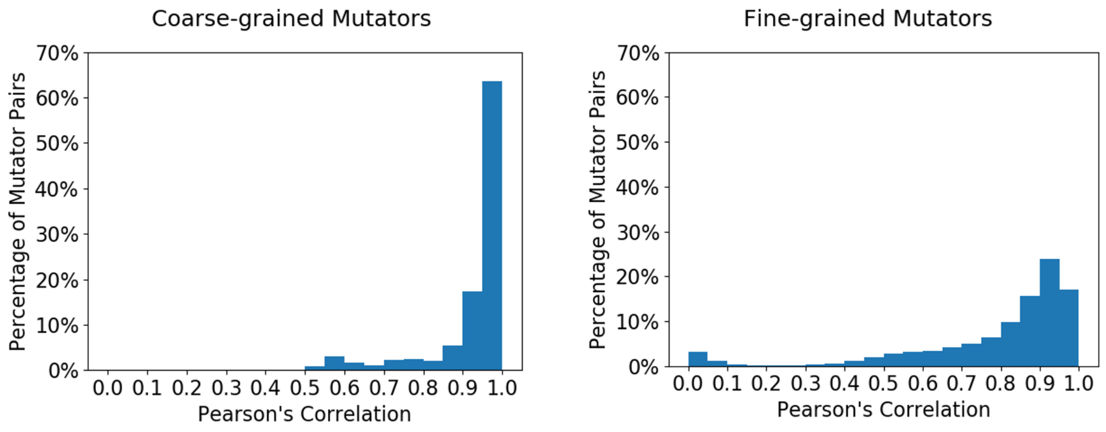


Figure 4.4: Pearson correlation distribution of the pairs of the coarse-grained mutaion operators (left) and the fine-grained mutaion operators (right) of `make`

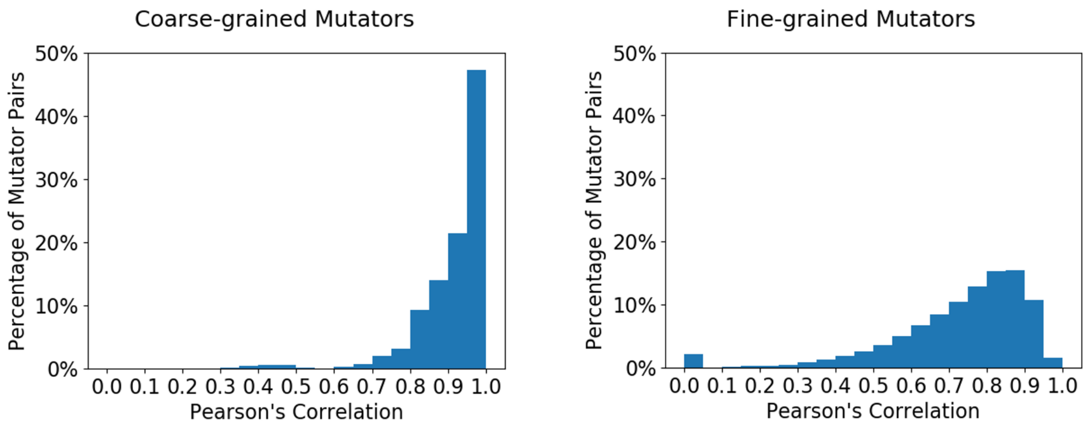


Figure 4.5: Pearson correlation distribution of the pairs of the coarse-grained mutaion operators (left) and the fine-grained mutaion operators (right) of `sed`

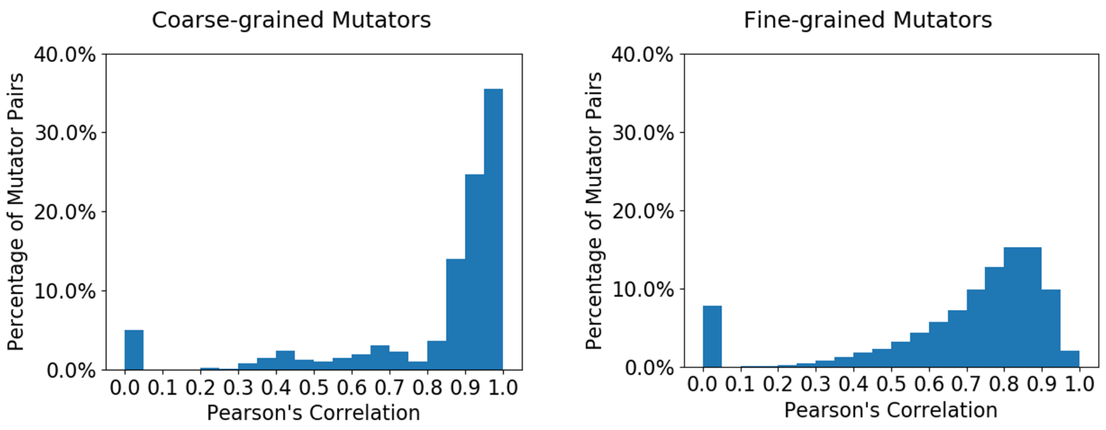


Figure 4.6: Pearson correlation distribution of the pairs of the coarse-grained mutaion operators (left) and the fine-grained mutaion operators (right) of `space`

Table 4.2: The number and ratio of the mutants selected by, and MSE of REFINER^{TRD} and REFINER

	Targets	REFINER^{TRD}			REFINER		
		#selected mutants	%selected mutants	MSE	#selected mutants	%selected mutants	MSE
Hard-to-kill 3%	flex	195062	17.0%	0.049	9062	0.8%	0.072
	grep	55672	29.6%	0.008	3846	2.0%	0.037
	gzip	18882	5.1%	0.040	912	0.2%	0.034
	make	59272	21.0%	0.129	6777	2.4%	0.035
	sed	1363	1.9%	0.016	1810	2.5%	0.057
	space	57353	52.8%	0.010	3821	3.5%	0.005
	Avg.	64600.7	17.9%	0.042	4371.3	1.2%	0.040
Hard-to-kill 5%	flex	9718	0.8%	0.016	9806	0.9%	0.013
	grep	30000	16.0%	0.008	5172	2.8%	0.007
	gzip	13593	3.7%	0.053	1285	0.3%	0.025
	make	39973	14.2%	0.073	7328	2.6%	0.002
	sed	19028	26.4%	0.043	1262	1.7%	0.015
	space	37944	34.9%	0.012	3454	3.2%	0.002
	Avg.	25042.7	6.9%	0.034	4717.8	1.3%	0.011
Hard-to-kill 7%	flex	46440	4.1%	0.031	20808	1.8%	0.018
	grep	97274	51.8%	0.002	4354	2.3%	0.008
	gzip	1052	0.3%	0.014	1412	0.4%	0.006
	make	17468	6.2%	0.001	3829	1.4%	0.015
	sed	2880	4.0%	0.007	1733	2.4%	0.013
	space	1407	1.3%	0.031	3662	3.4%	0.002
	Avg.	27753.5	7.7%	0.014	5966.3	1.7%	0.010

4.2.2 RQ2: Effect of the Fine-grained mutation operators of REFINER

Table 4.2 shows the experiment results of REFINER and REFINER^{TRD} for predicting hard-to-kill 3%, 5% and 7% mutation scores. The first column shows category of hard-to-kill mutants, and the second column shows the names of target programs. The third to the fifth columns and the sixth to the last columns show the numbers of the selected mutants, the ratios of the selected mutants over $\mathcal{M}$, and MSE of the hard-to-kill mutation scores on $\mathcal{M}$ and on the mutants selected by REFINER^{TRD} and REFINER, respectively. The average MSE with respect to training dataset of the selected models in cross validation experiments is 0.0001 and 0.0005 for REFINER and REFINER^{TRD}, respectively.

The experiment results in Table 4.2 show that REFINER using fine-grained mutation operators selects much fewer mutants and achieves more accurate prediction of hard-to-kill mutation scores than REFINER^{TRD}. For predicting hard-to-kill 3% mutation scores, REFINER selects 4371.3 mutants (i.e., 1.2% of $\mathcal{M}$) with $MSE = 0.040$, on average, while REFINER^{TRD} selects 64600.7 mutants (i.e., 17.9% of $\mathcal{M}$) with $MSE = 0.042$, on average. REFINER selects 93.2% (=4371.3/(64600.7-4371.3)) less mutants and achieves 4.9% lower MSE than REFINER^{TRD}. Similarly, for predicting hard-to-kill 5% and 7% mutation scores, REFINER selects 81.1% and 77.9% less mutants and achieves 67.6% and 28.6% lower MSE than REFINER^{TRD}, respectively.

Table 4.3: The number and ratio of the mutants selected by, and MSE of RND^{SN} and REFINER

	Targets	RND^{SN} #selected mutants	RND^{SN} %selected mutants	RND^{SN} MSE	REFINER #selected mutants	REFINER %selected mutants	REFINER MSE
Hard-to-kill 3%	flex	9062	0.8%	0.060	9062	0.8%	0.072
	grep	3846	2.0%	0.046	3846	2.0%	0.037
	gzip	912	0.2%	0.132	912	0.2%	0.034
	make	6777	2.4%	0.045	6777	2.4%	0.035
	sed	1810	2.5%	0.096	1810	2.5%	0.057
	space	3821	3.5%	0.010	3821	3.5%	0.005
	Avg.	4371.3	1.2%	0.065	4371.3	1.2%	0.040
Hard-to-kill 5%	flex	9806	0.9%	0.052	9806	0.9%	0.013
	grep	5172	2.8%	0.035	5172	2.8%	0.007
	gzip	1285	0.3%	0.116	1285	0.3%	0.025
	make	7328	2.6%	0.043	7328	2.6%	0.002
	sed	1262	1.7%	0.049	1262	1.7%	0.015
	space	3454	3.2%	0.007	3454	3.2%	0.002
	Avg.	4717.8	1.3%	0.050	4717.8	1.3%	0.011
Hard-to-kill 7%	flex	20808	1.8%	0.048	20808	1.8%	0.018
	grep	4354	2.3%	0.024	4354	2.3%	0.008
	gzip	1412	0.4%	0.029	1412	0.4%	0.006
	make	3829	1.4%	0.027	3829	1.4%	0.015
	sed	1733	2.4%	0.031	1733	2.4%	0.013
	space	3662	3.4%	0.005	3662	3.4%	0.002
	Avg.	5966.3	1.7%	0.027	5966.3	1.7%	0.010

4.2.3 RQ3. Comparison of REFINER with Random Mutant Selection

Compared to RND^{SN}, REFINER achieves much more accurate prediction of hard-to-kill mutation scores while using the exact same number of mutants as shown in Table 4.3. In Table 4.3, the first column shows category of hard-to-kill mutants, and the second column shows the names of target programs. The third to the fifth columns show the results of RND^{SN}, and the sixth to last columns show the results of REFINER. For each technique, Table 4.3 shows the the numbers of the selected mutants, the ratios of the selected mutants over $\mathcal{M}$, and MSE of hard-to-kill mutation scores on $\mathcal{M}$ and the mutants selected by the technique.

REFINER achieves 1.6 times (=0.065/0.040), 4.5 times (=0.050/0.011), and 2.7 times (=0.027/0.010) lower MSE than RND^{SN} while selecting the exact same number of mutants to predict hard-to-kill 3%, 5%, 7% mutation scores respectively. Figure 4.7 shows the plot of predicted hard-to-kill 5% mutation scores of RND^{SN} (Triangle) and REFINER (Circle) against the actual hard-to-kill 5% mutation scores for test suites of six SIR programs. The closer a point is to the $y = x$ line, the more accurate the prediction is. Hard-to-kill mutation scores predicted using RND^{SN} mutants tend to overestimate the actual hard-to-kill mutation scores of test suites, while REFINER's predictions are consistently closer to the actual hard-to-kill mutation scores.

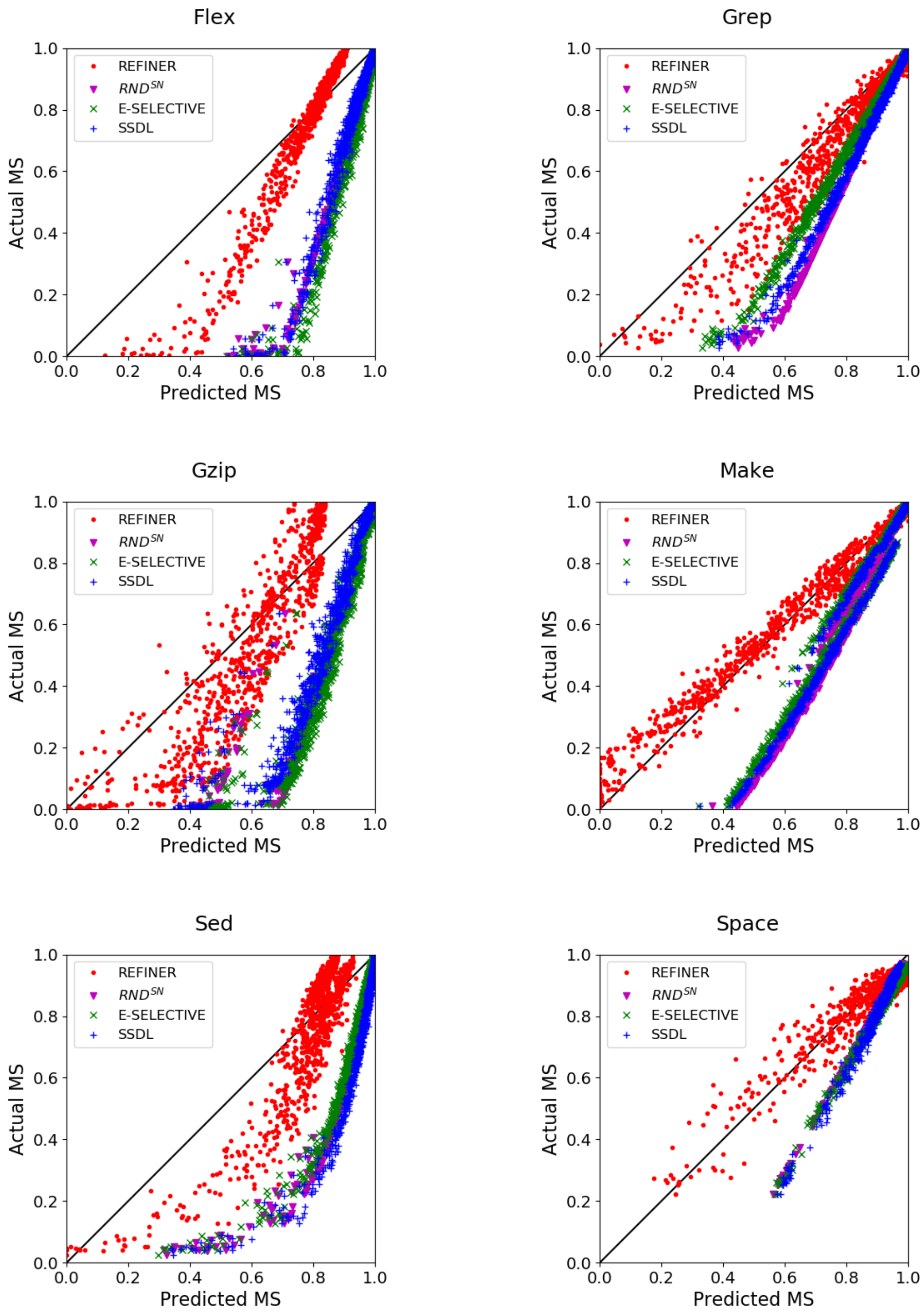


Figure 4.7: Actual vs Predicted Hard-to-kill 5% Mutation score of SSDL, E-SELECTIVE, RND^{SN} and REFINER in six SIR programs

Table 4.4: The number and ratio of the mutants selected by, and MSE of E-SELECTIVE, SSDL, and REFINER

	Targets	E-SELECTIVE			SSDL			REFINER		
		#Sel. Muts	%Sel. Muts.	MSE	#Sel. muts	%Sel. Muts.	MSE	#Sel. Muts	%Sel. Muts	MSE
Hard-to-kill 3%	flex	1820	0.2%	0.068	1372	0.1%	0.059	9062	0.8%	0.072
	grep	3315	1.8%	0.028	1604	0.9%	0.040	3846	2.0%	0.037
	gzip	1989	0.5%	0.140	852	0.2%	0.119	912	0.2%	0.034
	make	4385	1.6%	0.037	2630	0.9%	0.042	6777	2.4%	0.035
	sed	1912	2.6%	0.091	972	1.3%	0.105	1810	2.5%	0.057
	space	2564	2.4%	0.011	2074	1.9%	0.011	3821	3.5%	0.005
	Avg.	2664.2	0.7%	0.063	1584.0	0.4%	0.063	4371.3	1.2%	0.040
Hard-to-kill 5%	flex	1820	0.2%	0.059	1372	0.1%	0.051	9806	0.9%	0.013
	grep	3315	1.8%	0.020	1604	0.9%	0.030	5172	2.8%	0.007
	gzip	1989	0.5%	0.119	852	0.2%	0.100	1285	0.3%	0.025
	make	4385	1.6%	0.035	2630	0.9%	0.040	7328	2.6%	0.002
	sed	1912	2.6%	0.045	972	1.3%	0.055	1262	1.7%	0.015
	space	2564	2.4%	0.008	2074	1.9%	0.008	3454	3.2%	0.002
	Avg.	2664.2	0.7%	0.048	1584.0	0.4%	0.047	4717.8	1.3%	0.011
Hard-to-kill 7%	flex	1820	0.2%	0.054	1372	0.1%	0.046	20808	1.8%	0.018
	grep	3315	1.8%	0.012	1604	0.9%	0.020	4354	2.3%	0.008
	gzip	1989	0.5%	0.032	852	0.2%	0.022	1412	0.4%	0.006
	make	4385	1.6%	0.033	2630	0.9%	0.037	3829	1.4%	0.015
	sed	1912	2.6%	0.028	972	1.3%	0.036	1733	2.4%	0.013
	space	2564	2.4%	0.006	2074	1.9%	0.006	3662	3.4%	0.002
	Avg.	2664.2	0.7%	0.027	1584.0	0.4%	0.028	5966.3	1.7%	0.010

4.2.4 RQ4. Comparison of REFINER with Existing Operator-based Mutant Selection Techniques

The experiment results in Table 4.4 show that, REFINER can predict hard-to-kill mutation scores more accurately than SSDL and E-SELECTIVE (i.e., REFINER achieves lower MSE than SSDL and E-SELECTIVE). The first column shows category of hard-to-kill mutants, and the second column shows the names of target programs. The third to the fifth columns show the results of E-SELECTIVE, the sixth to the eighth columns show the results of SSDL, and the ninth to last columns show the results of REFINER. For each technique, Table 4.4 shows the the numbers of the selected mutants, the ratios of the selected mutants over $\mathcal{M}$, and MSE of Hard-to-kill mutation scores on $\mathcal{M}$ and the mutants selected by the technique.

In terms of MSE, E-SELECTIVE and SSDL performs similarly. On average, E-SELECTIVE achieves $MSE = 0.063, 0.048, 0.027$ and SSDL achieves $MSE = 0.063, 0.047, 0.028$ in predicting hard-to-kill 3%, 5%, 7% mutation scores, respectively. REFINER achieves 1.6 times (=0.063/0.040), 4.3 times (=0.047/0.011), 2.7 times (=0.027/0.010) more accurate prediction of the hard-to-kill 3%, 5%, 7% mutation score than E-SELECTIVE and SSDL. In terms of the number of selected mutants, on average, REFINER selects 1.9 times more mutants than E-SELECTIVE, and selects 3.2 times more mutants than SSDL. It is worth noting that REFINER achieves higher accuracy not just by selecting more mutants than E-SELECTIVE and SSDL but by selecting mutants carefully using CLARS. Selecting 1% random mutants achieves 0.0482 of MSE for hard-to-kill mutation score prediction, and selecting 25% random mutants achieves 0.0477 of MSE. It means that selecting 25 times more mutants without careful selection does not increase accuracy of predicting hard-to-kill mutation score.

Figure 4.7 shows the plot of predicted hard-to-kill 5% mutation scores of E-SELECTIVE ($\times$ Sign), SSDL ($+$ Sign), and REFINER (Circle) against the actual hard-to-kill 5% mutation scores for test suites of `make` and `sed` respectively. The figures demonstrate that hard-to-kill mutation scores predicted using E-SELECTIVE and SSDL tend to be higher than the actual hard-to-kill mutation scores of test suites, similar to RND^{SN}. On the other hand, the hard-to-kill mutation scores predicted by REFINER are closer to actual hard-to-kill mutation score, compared to the other two techniques.

4.3 Discussion

4.3.1 Advantage of the Fine-grained Mutation Operators

I demonstrate that each of the fine-grained mutation operator of REFINER generates less redundant set of mutants than the traditional coarse-grained mutation operators (see Section 4.2.1). Thus, the fine-grained mutation operators can provide better diversity to various mutation analysis techniques (e.g., mutation-based fault localization, mutation-based test prioritization [30], and mutation-based test generation [4]) such that the techniques can find specific mutation strategies customized for a specific target domain/application effectively. For instance, fine-grained mutation operators can be used for incremental mutation testing strategies for mutation-based fault localization to select/prioritize mutants in more cost-effective ways.

4.3.2 Accurate Prediction of Hard-to-Kill Mutation Scores even with Low-quality Test Suites

From the experiment results, I have observed that for the low-quality test suites (i.e., test suites whose actual hard-to-kill mutation score is low), REFINER predicts hard-to-kill mutation scores much more accurately than RND^{SN}, E-SELECTIVE, and SSDL (see Figure 4.7). For example of `make`'s test suites whose actual hard-to-kill 5% mutation score is lower than 0.2, MSE of REFINER's predicted hard-to-kill mutation score is 0.033 while MSE of E-SELECTIVE (the second most accurate technique for the low-quality test suite of `sed`) is 0.187 which is 5.7 times higher than that of $REFINER$. This is because RND^{SN}, E-SELECTIVE, and SSDL techniques selects many easy-to-kill mutants which make the prediction of hard-to-kill score inaccurate. In practice, REFINER predicts the hard-to-kill mutation score much more accurately than RND^{SN}, E-SELECTIVE, and SSDL. This is because most test suites in real-world are of low quality and, consequently, their actual predication accuracy of hard-to-kill mutation score will be low.

4.3.3 Advantage of Learning-based Model to Predict Mutation Scores

REFINER generates an accurate and efficient prediction model of hard-to-kill scores for each program. REFINER selects 198.3 fine-grained mutation operators to construct the prediction models on average. REFINER selects 149 fine-grained mutation operators for `sed` (the smallest model in terms of the selected mutation operators) and 273 fine-grained mutation operators for `flex` (the largest one). I do not observe a single fine-grained mutation operator that has the largest size of the effect (i.e., the absolute value of the coefficient) for all six target programs.

One interesting observation is that no fine-grained mutation operators refined from the four coarse-grained mutation operators in E-SELECTIVE is in the top 10 mutation operators in terms of the size of effect of the mutation operator in the models. Also, no fine-grained mutation operator refined from SSDL is included in the six models.

These observations imply that, for predicting hard-to-kill mutation scores accurately and efficiently, learning-based approach is more appropriate than selecting specific mutation operators.

Chapter 5. Conclusion and Future Work

5.1 Conclusion

In this dissertation, I have developed MUSIC, a configurable and extensible mutation tool for C programs; and REFINER, a fine-grained mutation operator-based mutant reduction technique to predict hard-to-kill mutation score accurately and efficiently.

MUSIC is developed as a solution to the lack of practical mutation testing tool for modern, complex, real-world C programs. Through a case study on Siemens benchmark programs and a large real-world modern C program cURL, I have demonstrated that MUSIC has much higher applicability and generates no stillborn mutant comparing to two famous mutation tool for C programs, Proteum and Milu.

To address the large cost problem of mutation analysis, I have proposed REFINER. A salient idea of REFINER is to define and utilize fine-grained mutation operators in mutation operator-based mutant reduction techniques to accurately and efficiently predict the hard-to-kill mutation score. This idea is based on the observation that the existing coarse-grained mutation operators is redundant/similar to each other in a large degree, and fine-grained mutation operators are less correlated, which can help REFINER produce models accurately and efficiently predicting the hard-to-kill mutation score. I have evaluated REFINER and other mutant reduction techniques through experiments on the six SIR programs and the experiment results show that REFINER selects less than 2.0% of all mutants on average and achieves the lowest prediction error in terms of MSE compared to REFINER^{TRD}, RND^{SN}, E-SELECTIVE, and SSDL.

5.2 Future Work

As future work, for MUSIC, I plan to apply MUSIC to more C projects to evaluate its applicability further. Also, since mutation analysis is actively used for various software analysis tasks, I plan to provide more diverse mutation operators in MUSIC. Furthermore, I plan to find other ways to ensure that MUSIC never generates stillborn mutants. I am upgrading MUSIC to support fine-grained mutation operators.

For REFINER, I plan to develop program semantics-based mutant reduction heuristics and combine these heuristics with REFINER to improve mutant reduction furthermore. In addition, I plan to apply REFINER to select mutants for mutation-based fault localization, and test case generation. In fault localization, the effectiveness of REFINER can be measured in terms of accuracy of fault detection. In test case generation, the effectiveness of REFINER can be measured in terms of code coverage of generated test suites or number of bugs detected.

Bibliography

[1] Jia Y, Harman M. An analysis and survey of the development of mutation testing. IEEE transactions on software engineering. 2010 Jun 17;37(5):649-78.

[2] Do H, Rothermel G. On the use of mutation faults in empirical assessments of test case prioritization techniques. IEEE Transactions on Software Engineering. 2006 Oct 9;32(9):733-52.

[3] Fraser G, Zeller A. Mutation-driven generation of unit tests and oracles. IEEE Transactions on Software Engineering. 2011 Sep 15;38(2):278-92.

[4] Kim Y, Hong S, Ko B, Phan DL, Kim M. Invasive software testing: Mutating target programs to diversify test exploration for high test coverage. In 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST) 2018 Apr 9 (pp. 239-249). IEEE.

[5] Moon S, Kim Y, Kim M, Yoo S. Ask the mutants: Mutating faulty programs for fault localization. In 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation 2014 Mar 31 (pp. 153-162). IEEE.

[6] Papadakis M, Le Traon Y. Metallaxis-FL: mutation-based fault localization. Software Testing, Verification and Reliability. 2015 Aug;25(5-7):605-28.

[7] Papadakis M, Kintis M, Zhang J, Jia Y, Le Traon Y, Harman M. Mutation testing advances: an analysis and survey. InAdvances in Computers 2019 Jan 1 (Vol. 112, pp. 275-378). Elsevier.

[8] Jia Y, Harman M. An analysis and survey of the development of mutation testing. IEEE transactions on software engineering. 2010 Jun 17;37(5):649-78.

[9] Phan DL, Kim Y, Kim M. MUSIC: Mutation Analysis Tool with High Configurability and Extensibility. In2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW) 2018 Apr 9 (pp. 40-46). IEEE.

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

[11] Barbosa EF, Maldonado JC, Vincenzi AM. Toward the determination of sufficient mutant operators for C. Software Testing, Verification and Reliability. 2001 Jun;11(2):113-36.

[12] Wong WE, Maldonado JC, Delamaro ME. Reduncing the Cost of Regression Testing by Using Selective Mutation. InAnais VIII Conferência Internacional de Tecnologia de Software: Qualidade de Software 1997 Jun.

[13] Siami Namin A, Andrews JH, Murdoch DJ. Sufficient mutation operators for measuring test effectiveness. InProceedings of the 30th international conference on Software engineering 2008 May 15 (pp. 351-360). ACM.

[14] Acree AT, Budd TA, DeMillo RA, Lipton RJ, Sayward FG. Mutation Analysis. GEORGIA INST OF TECH ATLANTA SCHOOL OF INFORMATION AND COMPUTER SCIENCE; 1979 Sep.

[15] Budd TA. MUTATION ANALYSIS OF PROGRAM TEST DATA.

[16] Wong WE, Mathur AP. Reducing the cost of mutation testing: An empirical study. Journal of Systems and Software. 1995 Dec 1;31(3):185-96.

[17] Zhang L, Hou SS, Hu JJ, Xie T, Mei H. Is operator-based mutant selection superior to random mutant selection?. InProceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1 2010 May 1 (pp. 435-444). ACM.

[18] Lerch MD. Statistics in the presence of cost: cost-considerate variable selection and MCMC convergence diagnostics (Doctoral dissertation, Montana State University-Bozeman, College of Letters & Science).

[19] Zhang L, Gligoric M, Marinov D, Khurshid S. Operator-based and random mutant selection: Better together. InProceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering 2013 Nov 11 (pp. 92-102). IEEE Press.

[20] Yao X, Harman M, Jia Y. A study of equivalent and stubborn mutation operators using human analysis of equivalence. InProceedings of the 36th International Conference on Software Engineering 2014 May 31 (pp. 919-930). ACM.

[21] Papadakis M, Chekam TT, Le Traon Y. Mutant quality indicators. In2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW) 2018 Apr 9 (pp. 32-39). IEEE.

[22] Visser W. What makes killing a mutant hard. InProceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering 2016 Aug 25 (pp. 39-44). ACM.

[23] Do H, Elbaum S, Rothermel G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empirical Software Engineering. 2005 Oct 1;10(4):405-35.

[24] Kintis M, Papadakis M, Jia Y, Malevris N, Le Traon Y, Harman M. Detecting trivial mutant equivalences via compiler optimisations. IEEE Transactions on Software Engineering. 2017 Mar 20;44(4):308-33.

[25] Legates DR, McCabe Jr GJ. Evaluating the use of "goodness-of-fit" measures in hydrologic and hydroclimatic model validation. Water resources research. 1999 Jan;35(1):233-41.

[26] https://github.com/mdlerch/ccs

[27] Deng L, Offutt J, Li N. Empirical evaluation of the statement deletion mutation operator. In2013 IEEE Sixth International Conference on Software Testing, Verification and Validation 2013 Mar 18 (pp. 84-93). IEEE.

[28] Delamaro ME, Deng L, Durelli VH, Li N, Offutt J. Experimental evaluation of SDL and one-op mutation for C. In2014 IEEE Seventh International Conference on Software Testing, Verification and Validation 2014 Mar 31 (pp. 203-212). IEEE.

[29] Untch RH. On reduced neighborhood mutation analysis using a single mutagenic operator. InProceedings of the 47th Annual Southeast Regional Conference 2009 Mar 19 (p. 71). ACM.

[30] Shin D, Yoo S, Papadakis M, Bae DH. Empirical evaluation of mutation-based test case prioritization techniques. Software Testing, Verification and Reliability. 2019 Jan;29(1-2):e1695.

[31] Agrawal H, DeMillo R, Hathaway R, Hsu W, Hsu W, Krauser EW, Martin RJ, Mathur AP, Spafford E. Design of mutant operators for the C programming language. Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue University; 1989 Mar 20.

[32] Delamaro ME, Maidonado JC, Mathur AP. Interface mutation: An approach for integration testing. IEEE transactions on software engineering. 2001 Mar;27(3):228-47.

[33] Delamaro ME, Maldonado JC, Mathur AP. Proteum-a tool for the assessment of test adequacy for c programs user's guide. InPCS 1996 Apr (Vol. 96, pp. 79-95).

[34] Maldonado JC, Delamaro ME, Fabbri SC, da Silva Simão A, Sugeta T, Vincenzi AM, Masiero PC. Proteum: A family of tools to support specification and program testing based on mutation. InMutation testing for the new century 2001 (pp. 113-116). Springer, Boston, MA.

[35] Delamaro ME, Maldonado JC, Vincenzi AM. Proteum/IM 2.0: An integrated mutation testing environment. InMutation testing for the new century 2001 (pp. 91-101). Springer, Boston, MA.

[36] Jia Y, Harman M. MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language. InTesting: Academic & Industrial Conference-Practice and Research Techniques (taic part 2008) 2008 Aug 29 (pp. 94-98). IEEE.

[37] Hong S, Lee B, Kwak T, Jeon Y, Ko B, Kim Y, Kim M. Mutation-based fault localization for real-world multilingual programs (T). In2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) 2015 Nov 9 (pp. 464-475). IEEE.

[38] Hong S, Kwak T, Lee B, Jeon Y, Ko B, Kim Y, Kim M. MUSEUM: Debugging real-world multilingual programs using mutation analysis. Information and Software Technology. 2017 Feb 1;82:80-95.

[39] https://curl.haxx.se/

[40] https://cmake.org/

[41] https://gyp.gsrc.io/

[42] https://github.com/rizsotto/Bear

[43] Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation. InProceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization 2004 Mar 20 (p. 75). IEEE Computer Society.

[44] Abreu R, Zoeteweij P, Van Gemund AJ. Spectrum-based multiple fault localization. InProceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering 2009 Nov 16 (pp. 88-99). IEEE Computer Society.

[45] Lo D, Cheng H, Han J, Khoo SC, Sun C. Classification of software behaviors for failure detection: a discriminative pattern mining approach. InProceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining 2009 Jun 28 (pp. 557-566). ACM.

[46] Abreu R, González A, Zoeteweij P, van Gemund AJ. Automatic software fault localization using generic program invariants. InProceedings of the 2008 ACM symposium on Applied computing 2008 Mar 16 (pp. 712-717). ACM.

[47] Papadakis M, Malevris N. Mutation based test case generation via a path selection strategy. Information and Software Technology. 2012 Sep 1;54(9):915-32.

[48] Lou Y, Hao D, Zhang L. Mutation-based test-case prioritization in software evolution. In2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE) 2015 Nov 2 (pp. 46-57). IEEE.

[49] Zhang L, Marinov D, Khurshid S. Faster mutation testing inspired by test prioritization and reduction. InProceedings of the 2013 International Symposium on Software Testing and Analysis 2013 Jul 15 (pp. 235-245). ACM.

[50] Kurtz B, Ammann P, Offutt J, Kurtz M. Are we there yet? How redundant and equivalent mutants affect determination of test completeness. In2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW) 2016 Apr 11 (pp. 142-151). IEEE.

[51] Zhang J, Zhang L, Harman M, Hao D, Jia Y, Zhang L. Predictive mutation testing. IEEE Transactions on Software Engineering. 2018 Feb 28.

[52] Coles H, Laurent T, Henard C, Papadakis M, Ventresque A. Pit: a practical mutation testing tool for java. InProceedings of the 25th International Symposium on Software Testing and Analysis 2016 Jul 18 (pp. 449-452). ACM.

Acknowledgment

First and foremost, I wish to express my sincere gratitude to my advisor Professor Moonzoo Kim, for giving me the opportunity to do research at SWTV lab and providing invaluable guidance throughout my time as an undergraduate and master student at KAIST. His patience, motivational speech, immense knowledge and experience have deeply inspired me. It was a great honor to work and study under his guidance. Besides my advisor, I would like to thank the rest of my thesis committee, Professor Shin Yoo and Professor In-Young Ko, for their insightful comments and questions.

My sincere thanks goes to Dr. Yunho Kim for the stimulating discussions and for providing valuable advices and help with my research. Also, I would like to pay special regards to members at SWTV lab - Hyunsu Lim, Kunwoo Park, Ahcheong Lee, Yang Zidong, Junho Yeo, Lee Dong Hee and Joonwang Ji. Thank you so much for your friendship and wonderful support during this challenging period.

I am extremely grateful to my parents for their love, support and sacrifices for my education and my future. Last but definitely not least, I am very much indebted to my girlfriend, Chi, my friends, Huan, Doyin, Anta, Asad, and Faris for their help, encouragement and for making my life at KAIST enjoyable.

Curriculum Vitae

Name	:	Phan Duy Loc
Date of Birth	:	March 15, 1995
E-mail	:	locphan1503@gmail.com

Educations

2011. 2. – 2014. 2.	Korea Science Academy of KAIST
2014. 2. – 2018. 2.	Korea Advanced Institute of Science and Technology Bachelor of Science in School of Computing
2018. 2. – 2020. 2.	Korea Advanced Institute of Science and Technology Master of Science in School of Computing

Publications

1. Kim Y, Hong S, Ko B, **Phan DL**, Kim M. Invasive software testing: Mutating target programs to diversify test exploration for high test coverage. In 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST) 2018 Apr 9 (pp. 239-249). IEEE. **(Distinguished paper award)**
2. **Phan DL**, Kim Y, Kim M. MUSIC: Mutation Analysis Tool with High Configurability and Extensibility. In2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW) 2018 Apr 9 (pp. 40-46). IEEE.
3. **L. Phan**, B. Ko, Y. Kim, and M.Kim, COMUT: A Configurable Mutant Generation Tool for C programs for effective and efficient mutation analysis, Korea Software Congress (KSC), Dec 20-22, 2017 **(Best paper award)**