

UNIVERSIDADE FEDERAL DE SÃO CARLOS– UFSCAR
CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA– CCET
DEPARTAMENTO DE COMPUTAÇÃO– DC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO– PPGCC

Alessandro Viola Pizzoleto

**Investigação de Estratégia para
Redução de Custo do Teste de Mutação
com Apoio de Similaridade entre
Programas**

Alessandro Viola Pizzoleto

**Investigação de Estratégia para
Redução de Custo do Teste de Mutação
com Apoio de Similaridade entre
Programas**

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências Exatas e de Tecnologia da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Doutor em Ciência da Computação.

Área de concentração: Metodologias e Técnicas de Computação

Orientador: Fabiano Cutigi Ferrari

São Carlos

2022

Pizzoleto, Alessandro Viola

Investigação de Estratégia para Redução de Custo do
Teste de Mutação com Apoio de Similaridade entre
Programas / Alessandro Viola Pizzoleto -- 2022.
223f.

Tese (Doutorado) - Universidade Federal de São Carlos,
campus São Carlos, São Carlos

Orientador (a): Fabiano Cutigi Ferrari

Banca Examinadora: Daniel Lucrédio, Auri Marcelo

Rizzo Vincenzi, Simone do Rocio Senger de Souza,

Vinicius Humberto Serapilha Durelli

Bibliografia

1. Análise de mutação . 2. Teste de mutação.. I.

Pizzoleto, Alessandro Viola. II. Título.

Ficha catalográfica desenvolvida pela Secretaria Geral de Informática
(SIn)

DADOS FORNECIDOS PELO AUTOR

Bibliotecário responsável: Arildo Martins - CRB/8 7180



UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Defesa de Tese de Doutorado do candidato Alessandro Viola Pizzoleto, realizada em 16/12/2022.

Comissão Julgadora:

Prof. Dr. Fabiano Cutigi Ferrari (UFSCar)

Prof. Dr. Daniel Lucrédio (UFSCar)

Prof. Dr. Auri Marcelo Rizzo Vincenzi (UFSCar)

Profa. Dra. Simone do Rocio Senger de Souza (USP)

Prof. Dr. Vinicius Humberto Serapilha Durelli (UFSJ)

O Relatório de Defesa assinado pelos membros da Comissão Julgadora encontra-se arquivado junto ao Programa de Pós-Graduação em Ciência da Computação.

Agradecimentos

Agradeço à Deus primeiramente.

Ao meu orientador Prof. Dr. Fabiano Cutifi Ferrari pela paciência, confiança, dedicação e por todo o conhecimento compartilhado.

Ao meu colega de pesquisa Giovanni Guarnieri, por sempre acreditar no projeto e confiar nos conceitos idealizados.

Aos meus colegas de laboratório Daniel, Bento, Luiz, Kathianni pelas conversas, ensinamentos e auxílios no decorrer desta jornada.

Aos professores que colaboraram com minha formação. À Universidade Federal de São Carlos pelo apoio cedido ao longo do doutorado.

Por fim, a todos aqueles que de alguma forma contribuíram para o sucesso dessa jornada.

A todos o meu grande muito obrigado.

“Feliz aquele que transfere o que sabe e aprende o que ensina.”

(Cora Coralina)

“Seja a mudança que você quer ver no mundo.”

(Mahatma Gandhi)

Resumo

Contexto: O critério de teste de mutação destaca-se como importante alternativa para a avaliação tando do software a ser testado quando o conjunto de teste a ser utilizado. Tal critério é considerado eficaz, mas apresenta algumas lacunas que devem ser preenchidas com relação a sua eficiência, devido a seu alto custo de execução. Há um conjunto relativamente alto de técnicas que reduzem seu custo de execução visando sua utilização na indústria, mas seus resultados são poucos generalizáveis. Alguns estudos relacionados a técnicas de redução de custo apresentavam heurísticas cujo objetivo é calcular a similaridades dos programas utilizados, mas não estudado profundamente. **Objetivos:** Neste trabalho apresenta-se uma revisão sistemática da literatura que busca apresentar o estado da arte em teste de mutação, a definição, criação e população de uma base de dados de artefatos, a implementação de processo automatizado para definição do Conjunto Essencial de operadores de mutação para ajudar na redução de custo para execução de teste de mutação. **Método:** A implementação apresentada neste trabalho lida com programas escritos na linguagem Java e inclui um conjunto de variantes que se relacionam a três tipos de abstrações de programas (código-fonte original, código-fonte ofuscado, e métricas de complexidade interna), três estratégias de cálculos de similaridade (*clustering*, funções de distância entre *strings*, e plágio) e uma abordagem de redução de custo de mutação (inspirada nas técnicas de melhores operadores e conjunto essencial). A avaliação apresentada, utilizando 221 classes escritas na linguagem Java, abrange 20 configurações variando-se as técnicas de abstrações de programas e similaridades. **Resultados:** A implementação de uma base de conhecimento de artefatos estruturada, automação da técnica de definição do Conjunto Essencial de Operadores, uma comparação cruzada envolvendo os *clusters* formados que tendem a atingir alta efetividade em prever os melhores operadores de mutação e conjunto essencial para programas não testados com o teste de mutação e uma análise estatística avaliando a semelhança entre os resultados das configurações. **Conclusão:** Considerando-se as configurações selecionadas para os experimentos, os resultados apresentados para as combinações usando a técnica

de Conjuntos Essenciais para a comparação da efetividade destacando as configurações que apresentaram melhores conjuntos de operadores, utilização de novos tamanhos para os clusters de referência, uma análise analítica do resultados associada a uma análise estatística usando a técnica de variância para verificar a equivalência entre os conjuntos de resultados.

Palavras-chave: Análise de Mutação. Teste de Mutação. Redução de Custo. Teste de Software. Escore de Mutação.

Abstract

Context: The mutation test criterion stands out as an important alternative for evaluating both the software to be tested and the test set to be used. This criterion is considered effective, but it has some gaps that must be filled in relation to its efficiency, due to its high execution cost. There is a relatively high set of techniques that reduce their execution costs, aiming at their use in industry, but their results are not generally generalizable. Some studies related to cost reduction techniques presented heuristics whose objective is to calculate the similarities of the programs used, but not deeply studied.

Objective: This work presents a systematic review of the literature that seeks to present the state of the art in mutation testing, the definition, creation and population of a database of artifacts, the implementation of an automated process for defining the Essential Set of mutation operators to help reduce the cost of running mutation testing.

Methodology: The implementation presented in this work deals with programs written in the Java language and includes a set of variants that relate to three types of program abstractions (original source code, obfuscated source code, and internal complexity metrics), three calculation strategies of similarity (*clustering*, distance functions between *strings*, and plagiarism) and a mutation cost reduction approach (inspired by best operators and essential set techniques). The evaluation presented, using 221 classes written in the Java language, covers 20 configurations varying the techniques of program abstractions and similarities.

Results: The implementation of a knowledge base of structured artifacts, automation of the technique of defining the Essential Set of Operators, a cross-comparison involving the *clusters* formed that tend to achieve high effectiveness in predicting the best mutation operators and essential set for untested programs with mutation testing and a statistical analysis evaluating the similarity between configuration results.

Conclusions: Considering the configurations selected for the experiments, the results presented for the combinations using the essential sets technique for the comparison of effectiveness, highlighting the configurations that presented the best set of operators, use of new sizes for the

reference clusters, an analytical analysis of the results associated with a statistical analysis using the variance technique to verify the equivalence between the sets of results.

Keywords: Mutation Analysis. Mutation Testing. Cost Reduction. Software Testing. Mutation Score.

Lista de ilustrações

Figura 1 – Exemplo de cálculo de métrica de distância Jaccard.	38
Figura 2 – Processo geral do SiMut (PIZZOLETO et al., 2020).	61
Figura 3 – Arquitetura da <i>SiMut</i> (GUARNIERI; PIZZOLETEO; FERRARI, 2022).	61
Figura 4 – Composição do grupo de referência G (PIZZOLETO et al., 2020). . . .	62
Figura 5 – Modelo da Base de Dados.	67
Figura 6 – Linha de comando - Evosuite.	70
Figura 7 – Linha de comando - PIT.	70
Figura 8 – Exemplo de um arquivo CSV que representa os resultados de execução da PIT.	72
Figura 9 – Exemplo de um arquivo CSV que representa a killing matrix.	72
Figura 10 – Resultado da Análise de Variância analisando T - Melhores Operadores.	104
Figura 11 – Resultado da Análise de Variância analisando R3 - Melhores Operadores.	105
Figura 12 – Resultado da Análise de Variância analisando T - Operadores Essenciais.	107
Figura 13 – Resultado da Análise de Variância analisando R3 - Operadores Essenciais.	108
Figura 14 – Resultado do Test t considerando-se R1 e R3 - Melhores Operadores. .	109
Figura 15 – Resultado do Test t considerando-se R1 e R3 - Operadores Essenciais. .	110

Lista de tabelas

Tabela 1 – Número de estudos por rodada de busca. A coluna intitulada <i>Sel</i> inclui estudos não-duplicados para a seleção final. As colunas intituladas <i>Rec</i> incluem resultados duplicados de busca.	45
Tabela 2 – Métricas das aplicações incluídas na base.	69
Tabela 3 – Resultados da PIT.	72
Tabela 4 – Lista de configurações selecionadas para os experimentos.	77
Tabela 5 – Trecho dos resultados detalhados, adaptada do trabalho de Guarnieri (2022).	79
Tabela 6 – Categorização dos operadores da PIT.	84
Tabela 7 – Resumo dos resultados considerando os quatro melhores operadores (projeto SimplePrograms).	86
Tabela 8 – Classificação dos resultados (R_i podendo variar de 1 a 35, Melhores Operadores, projeto SimplePrograms).	87
Tabela 9 – Resumo dos resultados considerando os operadores essenciais (projeto SimplePrograms).	88
Tabela 10 – Classificação dos resultados (R_i podendo variar de 1 a 35, Operadores Essenciais, projeto SimplePrograms).	89
Tabela 11 – Resumo dos resultados considerando os quatro melhores operadores (projeto Javassist, clusters tamanho padrão).	91
Tabela 12 – Classificação dos resultados (R_i podendo variar de 1 a 221, Melhores Operadores, projeto Javassist, clusters tamanho padrão).	92
Tabela 13 – Resumo dos resultados considerando os quatro melhores operadores (projeto Javassist, clusters de referência com 10%).	94
Tabela 14 – Classificação dos resultados (R_i podendo variar de 1 a 221, Melhores Operadores, projeto Javassist, clusters de referência com 10%).	95
Tabela 15 – Resumo dos resultados considerando os quatro melhores operadores (projeto Javassist, clusters de referência com 5%).	96

Tabela 16 – Classificação dos resultados (R_i podendo variar de 1 a 221, Melhores Operadores, projeto Javassist, clusters de referência com 5%).	97
Tabela 17 – Resumo dos resultados considerando os operadores essenciais (projeto Javassist, clusters tamanho padrão).	98
Tabela 18 – Classificação dos resultados (R_i podendo variar de 1 a 221, Operadores Essenciais, projeto Javassist, clusters tamanho padrão).	99
Tabela 19 – Resumo dos resultados considerando os operadores essenciais (projeto Javassist, clusters de referência com 10%).	100
Tabela 20 – Classificação dos resultados (R_i podendo variar de 1 a 221, Operadores Essenciais, projeto Javassist, cluster de referência com 10%).	101
Tabela 21 – Resumo dos resultados considerando os operadores essenciais (projeto Javassist, clusters de referência com 5%).	102
Tabela 22 – Classificação dos resultados (R_i podendo variar de 1 a 221, Operadores Essenciais, projeto Javassist, cluster de referência com 5%).	103
Tabela 23 – Resultados de T das execuções com Melhores Operadores.	103
Tabela 24 – Resultados de R3 das execuções com Melhores Operadores.	105
Tabela 25 – Resultados de R3 das execuções com Operadores Essenciais.	106
Tabela 26 – Resultados de R3 das execuções com Operadores Essenciais.	107
Tabela 27 – Resultados de R1 e R3 das execuções com Melhores Operadores	109
Tabela 28 – Resultados de R1 e R3 das execuções com Conjunto Essencial	110

Sumário

1	INTRODUÇÃO	21
1.1	Contexto	21
1.2	Motivação	23
1.3	Objetivos	24
1.4	Organização	25
2	FUNDAMENTAÇÃO TEÓRICA	27
2.1	Teste de Software	27
2.2	Teste de Mutação	30
2.2.1	Técnicas de Redução de Custo do Teste de Mutação	32
2.3	Conceitos de Abstração de Software	33
2.3.1	Métricas Internas	33
2.3.2	Código-Fonte Ofuscado	36
2.4	Conceitos de Similaridade de Software	36
2.4.1	Detecção de Plágio de Código Fonte	37
2.4.2	Métricas de Distância entre Strings	37
2.5	Considerações Finais	41
3	REVISÃO BIBLIOGRÁFICA	43
3.1	Objetivo, Planejamento e Execução da RSL	43
3.2	Resultados da RSL	46
3.2.1	Técnicas de Redução de Custo do Teste de Mutação	46
3.2.2	Métricas Utilizadas em Estudos de Redução de Custo do Teste de Mutação	51
3.2.3	Conclusões da RSL	54
3.3	Outros Trabalhos Relacionados	55
3.4	Considerações Finais	57

4	FRAMEWORK <i>SIMUT</i>	59
4.1	<i>SiMut</i>: Histórico e Visão Geral	59
4.2	Criação de Abstrações e Cálculo de Similaridade entre Programas	60
4.3	Cálculo de Parâmetros de Redução de Custo do Teste de Mutação	62
4.4	Resumo das Contribuições e Limitações	63
5	CRIAÇÃO DE UMA BASE DE CONHECIMENTO	65
5.1	Motivação e Requisitos para a Criação da Base	65
5.2	Processo de Criação da Base de Conhecimento	67
5.2.1	Projetos que Popularam a Base	67
5.3	Procedimentos e Resultados	71
5.4	Resumo das Contribuições e Limitações	73
6	EXPERIMENTOS	75
6.1	Planejamento dos Experimentos	75
6.1.1	Objetivos	76
6.1.2	Questões de Pesquisa	76
6.1.3	Artefatos	77
6.1.4	Modelo de Interpretação dos Resultados	79
6.2	Aplicação da Técnica de Redução de Custo	81
6.2.1	Implementação da Técnica de Conjunto Essenciais	82
6.3	Resultados e Análise	85
6.3.1	Resultados e Análise: Melhores Operadores para SimplePrograms	85
6.3.2	Resultados e Análise: Conjuntos Essenciais para SimplePrograms	87
6.3.3	Resultados e Análises: Melhores Operadores para Javassist	90
6.3.4	Resultados e Análises: Operadores Essenciais para Javassist	94
6.4	Conclusões Preliminares	95
6.5	Análise Estatística	99
6.5.1	Análise Estatística dos Resultados	101
6.6	Considerações Finais	109
7	CONCLUSÃO	113
7.1	Contribuições	114
7.2	Ameaças à validade	115
7.3	Limitações	116
7.4	Trabalhos Futuros	116
	REFERÊNCIAS	119

APÊNDICE A	ARTIGO PUBLICADO NO MUTATION'18 . .	139
APÊNDICE B	ARTIGO PUBLICADO NO PERIÓDICO JOURNAL OF SYSTEMS AND SOFTWARE	151
APÊNDICE C	ARTIGO PUBLICADO NO MUTATION'20 . .	191
APÊNDICE D	ARTIGO PUBLICADO NO MUTATION'22 . .	203
APÊNDICE E	ARTIGO PUBLICADO NA TRILHA INSIGHTFUL IDEAS AND EMERGING RESULTS DO SBES'21	215

Capítulo 1

Introdução

Este capítulo apresenta a contextualização do tema abordado, as motivações para a realização da pesquisa, a hipótese de pesquisa e os objetivos deste trabalho. As publicações decorrentes da pesquisa, ocorridas ao longo do doutoramento, compõem o corpo desta tese. Assim, ao final deste capítulo, o corpo da tese é explicado.

1.1 Contexto

A necessidade constante de aumentar a qualidade dos sistemas de software desenvolvidos faz com que empresas produtoras adotem cada vez mais as atividades de Verificação e Validação (V&V) durante todo o processo de desenvolvimento. Uma dessas atividades é o teste de software, que tem o objetivo de revelar defeitos presentes nos artefatos produzidos (MYERS; SANDLER; BADGETT, 2011). Ressalte-se que, mesmo com o avanço nas técnicas de Engenharia de Software e o desenvolvimento de ferramentas de apoio cada vez mais específicas, os defeitos ainda persistem. Por essa razão, o teste de software é utilizado como forma de revelar defeitos existentes. Esses defeitos serão analisados e corrigidos pela equipe responsável pela fase em que o projeto se encontra, de acordo com ciclo de vida adotado para o desenvolvimento do software.

Diversas são as formas de se testar um software. Por exemplo, a técnica de *teste funcional*, também conhecida como teste *caixa-preta*, não considera os detalhes internos da implementação do software. A definição dos casos de testes é feita com o fornecimento dos dados de entrada e as saídas geradas são avaliadas com o objetivo de confirmar a correteza dos artefatos (PRESSMAN, 2016; MYERS; SANDLER; BADGETT, 2011; MERDES et al., 2006; GARVIN; COHEN; DWYER, 2011). Os critérios que compõem essa técnica facilitam a definição dos casos de teste, com a utilização de subconjuntos do

conjunto completo de dados de entrada. Dentre os mais relevantes estão: particionamento da equivalência, análise do valor limite e grafo de causa-efeito (BARBOSA et al., 2005).

Outra técnica muito utilizada é a de *teste estrutural*, que complementa o *teste funcional*. Também chamada de teste *caixa-branca* (MYERS; SANDLER; BADGETT, 2011), tem como requisito inicial o conhecimento da estrutura interna do artefato de software a ser testado, fato que a torna bastante útil, já que permite que sejam projetados casos de teste mais precisos e concentrados nos pontos mais importantes e/ou propensos a causarem defeitos. Essa técnica é composta por alguns critérios dos quais destacam-se: critérios baseados em fluxo de controle (MYERS; SANDLER; BADGETT, 2011) e critérios baseados em fluxo de dados (RAPPS; WEYUKER, 1982; RAPPS; WEYUKER, 1985; MALDONADO, 1991).

Deve-se mencionar ainda, a técnica de *teste baseado em defeitos*, que utiliza como base os defeitos mais comuns introduzidos no código-fonte do programa pelo programador ou projetista (MORELL, 1990). Esses defeitos direcionam a definição de casos de teste eficientes que ajudam a identificar mais facilmente os defeitos existentes (BARBOSA; MALDONADO; VINCENZI, 2001). Exemplos de critérios associados a essa técnica são: semeadura de defeitos e análise de mutantes ou teste de mutação (DEMILLO; LIPTON; SAYWARD, 1978), sendo o segundo amplamente investigado pela comunidade científica.

Além das técnicas e critérios de teste mencionados, há também algumas fases (ou níveis) de teste que devem estar presentes em todo o processo de desenvolvimento de software, as quais são divididas em: (a) *teste de unidade*, no qual cada unidade do software é testada separadamente com o objetivo de isolar as partes e garantir que cada uma delas funcione conforme especificado; (b) *teste de integração*, no qual as unidades do software são testadas de forma combinada com o objetivo de detectar falhas na integração entre elas; e (c) *teste de sistema*, no qual o software é testado considerando-se suas funcionalidades, a integração entre as unidades e a integração com outros sistemas de software. O *teste de sistema* é realizado tanto nos requisitos funcionais, quanto nos não-funcionais (PRESSMAN, 2016; DELAMARO; MALDONADO; JINO, 2007).

O teste de mutação é alvo de investigação deste trabalho, que tem por objetivo definir estratégias que ajudarão na redução do seu custo de aplicação. Mais detalhes serão apresentados no decorrer deste documento. Especificamente em relação ao teste de mutação, tal critério se mostra uma importante alternativa na avaliação tanto do software em si (MATHUR, 2007) quanto do conjunto de testes utilizado para testá-lo. O *teste de mutação* consiste em inserir defeitos simples no software, com o intuito de mostrar que os casos de teste projetados são capazes de revelar esses defeitos. Ressalta-se que os defeitos inseridos têm o objetivo de simulá-los como se fossem inseridos durante o processo de desenvolvimento do software. Quando o *teste de mutação* é executado, obtêm-se alguns benefícios, dentre os quais podem-se destacar: garantir a qualidade do software que está sendo testado e certificar que o conjunto de casos de teste é adequado para revelar de-

feitos que existem ou podem ser inseridos no software original. Em alguns outros casos a mutação realizada pode gerar, na verdade, a versão correta do software (MATHUR, 2007).

Ressalte-se, também, que o *teste de mutação* constitui uma importante base para a condução de estudos experimentais em Engenharia de Software, principalmente para a avaliação de outras técnicas e critérios de seleção de casos de teste (ANDREWS; BRIAND; LABICHE, 2005; JUST et al., 2014). Nesse sentido, dada a necessidade de se testar o software para aumentar a confiança de que ele computa corretamente as funções pretendidas, nota-se, do ponto de vista financeiro, que essa atividade consome uma grande parte do orçamento de projetos de software, girando em torno de 40% dos recursos alocados para o desenvolvimento como um todo (Capgemini Group; Sogeti; HP, 2016). Evidencia-se, então a necessidade de se buscar formas mais econômicas de realizá-lo.

1.2 Motivação

Seguindo sua cronologia, o *teste de mutação* é considerado caro devido ao número de mutantes gerados, à dificuldade para se executar todos os mutantes e, igualmente, a dificuldade de identificar os que são equivalentes ao software original (JIA; HARMAN, 2011; MATEO; USAOLA, 2014; PAPADAKIS et al., 2015). Nesse sentido, observa-se substancial esforço da comunidade científica para reduzir o custo de aplicação desse critério.

Conhecendo as limitações e necessidades, esta proposta de pesquisa tem o objetivo de explorar as técnicas que apresentam resultados significativos quanto à redução de custo para a execução do teste de mutação. Identificadas e analisadas, as técnicas potencialmente econômicas serão aplicadas conjuntamente e os resultados gerados passarão por um processo de análise que verificará o grau de efetividade resultante da combinação.

Ressalte-se que o teste de mutação tem como particularidade se basear em defeitos típicos introduzidos no software durante o desenvolvimento, já identificados e conhecidos (ou seja, já caracterizados), para apoiar a definição dos requisitos de testes que serão derivados para se avaliar o software em teste. Na execução do *teste de mutação*, o programa que recebe os casos de teste é alterado diversas vezes, gerando um conjunto de programas alternativos, também conhecidos como mutantes. Esse processo é executado para que defeitos possam ser inseridos no código fonte do programa original. Sendo assim, a partir deste ponto, fica a cargo do analista de teste definir ou escolher os casos de teste mais adequados para mostrar a diferença de comportamento entre o programa original e o alternativo. Quando se observa essa diferença de comportamento, o mutante é dito estar “morto”.

Para gerar os mutantes, utilizam-se operadores de mutação, que implementam as regras de modificação que representam os defeitos que serão introduzidos no programa original, e que por sua vez geram uma grande quantidade de mutantes a serem analisados.

Quanto maior for a parte do código a sofrer as mutações, maior será a quantidade de mutantes gerados. Devido a isso, a aplicação desse critério de teste fica restrita a programas de médio e pequeno porte, tornando-a inviável de ser aplicada em programas de grande porte devido ao custo de sua execução. Os altos custos em executar o teste de mutação, de maneira bem sucinta, estão diretamente relacionada à quantidade de mutantes que são gerados, o esforço para executar os casos de teste definidos no processo de análise dos mutante que não foram mortos (JIA; HARMAN, 2011; MATEO; USAOLA, 2014; PAPADAKIS et al., 2015).

Via de regra, pesquisas, em geral, sobre redução de custo em teste de mutação descrevem que os resultados estão diretamente relacionados aos artefatos utilizados nos experimentos. Assim, tem-se esta afirmação como mais uma motivação para a realização desta pesquisa e desmistificar essa crença. A fim de contribuir para a redução do custo e viabilizar a adoção deste critério de teste, define-se na próxima seção o objetivo deste. Detalhes serão apresentados ao longo deste documento, conforme descrito na Seção 1.4.

1.3 Objetivos

Apresentados nas seções anteriores o problema e a motivação, e com o foco na necessidade de reduzir o custo do teste de mutação para viabilizar sua aplicação prática, esta pesquisa tem o objetivo de investigar variações de abstrações de programas que analisam programas definidos quanto ao grau de similaridade entre si e corroborar para comprovarmos ou não a hipótese apresentada. Sendo assim, a análise de similaridade entre classes de programas tomada como referência para o estudo, quanto a similaridade pode influenciar na definição dos melhores operadores para criação de mutantes. Espera-se que classes com alto grau de similaridade possam possibilitar o uso de testes de mutação com custo reduzido. Para auxiliar, faz-se uso de duas técnicas de redução de custo: one-op e operadores essenciais.

Para desenvolver este trabalho e, conseqüentemente, alcançar os resultados esperados, foi realizado um planejamento, no qual identificaram questões de pesquisas que ajudam a direcionar os estudos e análise dos resultados:

- ❑ Diferentes abstrações de código podem contribuir para predizer a redução do custo quanto as classes não testadas?
- ❑ Diferentes formas de computar similaridade podem ajudar na identificação de operadores essenciais adequados e que apresentem comprovadamente redução dos custos?
- ❑ Quais técnicas de redução de custo que podem se beneficiar com os resultados das abstrações analisadas?

1.4 Organização

O restante deste trabalho está organizado da seguinte forma: A fundamentação teórica que serviu de embasamento para esta tese é apresentada no Capítulo 2; no Capítulo 3 é apresentado o estado da arte em teste de mutação com informações da Revisão Sistemática da Literatura com a inclusão de uma subseção contendo outros trabalhos relacionados a similaridades, um dos focos desta tese; no Capítulo 4 apresentam-se os detalhes da implementação do *framework SiMut*; no Capítulo 5 detalha-se todo o processo de criação da base de dados, desde a escolha dos artefatos utilizados, processo executado e a base de dados resultante; no Capítulo 6 são apresentados os resultados e análises em cima dos dados obtidos; e, por fim, o Capítulo 7 encerra-se esta tese, apresentando-se os resultados obtidos, as contribuições, as limitações e os possíveis trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Neste capítulo apresenta-se o embasamento teórico para a entendimento deste trabalho. Para atingir os objetivos descritos no capítulo anterior, a fundamentação teórica é direcionada para a utilização da similaridade em benefício da redução de custo do teste de mutação. Com isso, apresentam-se os fundamentos do teste de software e do teste de mutação, os fundamentos de técnicas de abstrações de programas, e descrevem-se algumas técnicas de similaridades entre programas que foram exploradas neste trabalho.

Mais especificamente, na Seção 2.1 deste capítulo são abordados conceitos relacionados à atividade de teste e à importância de sua utilização durante as fases do processo de desenvolvimento e manutenção; na Seção 2.2 descrevem-se conceitos relacionados ao critério Análise de Mutantes, associado à técnica de teste baseada em defeitos. Na Seção 2.2.1 são contextualizadas as técnicas mais investigadas para a redução do custo do teste de mutação. Na Seção 2.3 apresentam-se formas de abstrações de programas, envolvendo ofuscador de código e métricas de internas de códigos, ao passo que na Seção 2.4 apresentam-se as abordagens de similaridades entre programas utilizadas neste trabalho. Por fim, na Seção 2.5 apresentam-se as considerações finais deste capítulo.

2.1 Teste de Software

Durante o processo de desenvolvimento e manutenção de software, podem ser utilizadas diversas técnicas e ferramentas com o propósito de reduzir a quantidade de defeitos. Contudo, ainda não é possível garantir que o software estará livre de tais problemas. As falhas e os defeitos associados podem estar ligados a diversas causas: requisitos mal compreendidos, especificação incorreta, enganos cometidos na codificação, etc.

Nesta pesquisa, as definições utilizadas para os termos “erro”, “defeito” e “falha” procuram seguir o padrão IEEE 610.12-1990 (IEEE, 1990), sendo elas:

- ❑ *Defeito*: consiste em uma divergência entre o produto de software desenvolvido e o produto supostamente correto.
- ❑ *Erro*: caracteriza-se por um estado interno do produto de software que diverge do estado correto, gerado pela execução de um defeito presente nesse produto.
- ❑ *Falha*: manifestação externa de um erro quando esse interfere nos comandos que produzem a saída do software em execução.

Na tentativa de identificar os defeitos existentes, são aplicadas as técnicas de teste de software. Seja um ambiente real ou simulado, um defeito só será detectado caso ocorram falhas no software em execução. Nesse cenário, o teste de software é um processo importante para garantir a qualidade do produto.

A estratégia mais difundida entre as equipes de software consiste na execução incremental dos testes. Essa estratégia parte do nível de unidade,¹ na qual se inicia a execução dos testes, depois avança para a fase de integração, na qual o produto é finalizado.

As etapas da estratégia são descritas a seguir:

- ❑ *Teste de Unidade*: nessa fase o teste tem por objetivo garantir que as menores unidades do sistema funcionem corretamente, ou seja, procura-se revelar defeitos de lógica e de implementação nas unidades em teste. O foco nesse nível está na lógica interna do processamento e nas estruturas de dados dentro dos limites de uma unidade do software (PRESSMAN, 2016).
- ❑ *Teste de Integração*: essa fase tem por objetivo testar unidades integradas de software que foram individualmente testadas na fase anterior. Parte-se do pressuposto que a afirmativa “se todos funcionam individualmente, então irão funcionar se colocados em conjunto” não é válida, pois quando colocadas juntas, as unidades podem produzir resultados inesperados ou contrários àquilo que foi especificado. O teste de integração é uma abordagem sistemática utilizada para construir a arquitetura do software, ao mesmo tempo em que conduz testes para descobrir defeitos associados às interfaces (MALDONADO et al., 2000).
- ❑ *Teste de Sistema*: Nessa última fase, o software é incorporado a outros elementos como, por exemplo, hardware, pessoas e informações. A partir desse ponto, vários testes são realizados com o intuito de exercitar e validar o sistema no ambiente de produção. Nota-se que para essa validação “completa” do sistema, são realizados variados tipos de testes de sistema, tais como teste de recuperação, teste de segurança, teste de estresse, e teste de desempenho (PRESSMAN, 2016).

¹ *Unidade*: é considerada como um componente ou porção de software indivisível (IEEE, 1990)

Em cada fase de teste, há quatro atividades que se relacionam e geralmente são implementadas no decorrer do processo de desenvolvimento do software. São elas: (a) planejamento de testes; (b) projetos de casos de teste; (c) execução; e (d) avaliação dos resultados de teste (PRESSMAN, 2016).

Dentre as várias questões relacionadas ao teste de software, a que mais recebe atenção se refere ao tamanho do domínio de entrada, que pode ser muito grande em determinadas funcionalidades, até mesmo infinito. Sendo assim, uma forma de auxiliar os testadores nesta questão é a utilização de critérios de teste, que contribuam na identificação dos subconjuntos do domínio, reduzindo assim a quantidade de casos de teste criados e auxiliando em sua avaliação (FERRARI et al., 2010; LEME et al., 2015).

Os critérios de teste, por sua vez, estão associados a técnicas específicas. A principal diferenciação das técnicas e dos critérios está no tipo de artefato que será utilizado na criação do modelo que representará o software durante o teste. Como exemplo desses artefatos, destacam-se: (a) especificação de requisitos; (b) fluxo de controle e de dados; e (c) modelos e taxonomias de defeitos. Fazendo uso dessa abstração, os critérios de teste guiam a definição dos elementos de software que serão utilizados para se derivarem os requisitos de teste.

Dentre as técnicas mais investigadas pela academia e mais empregadas pela indústria de software, pode-se destacar:

- *Teste funcional*: Nessa técnica, os detalhes internos de implementação não são considerados e, em razão dessa particularidade, este teste é conhecido como teste *caixa-preta* (PRESSMAN, 2016). Em princípio, esta técnica pode detectar todos os defeitos, já que submete todo o programa a todas as entradas possíveis (MYERS; SANDLER; BADGETT, 2011). Entretanto, critérios são adotados para selecionar ou direcionar a criação de conjuntos de teste mais enxuto e, conseqüentemente, tornar o teste mais viável.
- *Teste estrutural*: conhecido como teste *caixa-branca* (MYERS; SANDLER; BADGETT, 2011), esse teste complementa o *Teste Funcional* (PRESSMAN, 2016), mas tem como base o conhecimento da estrutura interna do software a ser testado, já que irá ajudar a definir o melhor conjunto de casos de teste. A partir do teste estrutural é possível derivar casos de teste que, por exemplo: (i) garantam que todos os caminhos independentes de um módulo tenham sido exercitados pelo menos uma vez; (ii) exercitem todas as decisões lógicas; (iii) executam todos os ciclos dentro de seus limites operacionais; e (iv) exercitem todas as possíveis estruturas de dados internas (PRESSMAN, 2016).
- *Teste baseado em defeitos*: Para definir quais estruturas são necessárias para derivar os requisitos de teste, os critérios associados a essa técnica utilizam informações de defeitos comumente presentes nas aplicações de software (MORELL, 1990). Em

linhas gerais, faz-se a introdução de defeitos no software e avalia-se se o conjunto de testes é capaz de detectar a presença dos defeitos introduzidos. O critério mais expoente de teste baseado em defeitos, com mais de quatro décadas de investigação, é a *Análise de Mutantes*, ou *teste de mutação*, que é tema central deste trabalho de doutorado e é descrito na próxima seção.

2.2 Teste de Mutação

A ideia do critério Análise de Mutantes (DEMILLO; LIPTON; SAYWARD, 1978), também conhecido como *teste de mutação*, consiste na criação de várias versões do programa original (tais versões são chamadas *mutantes*), com a realização de pequenas modificações que visam simular possíveis defeitos no software em teste. Para que os mutantes sejam gerados, faz-se uso de operadores de mutação, que encapsulam regras que definem os tipos de modificações realizadas no programa original.

A definição e utilização de bons operadores de mutação podem conduzir a testes eficazes, mas operadores de mutação ruins podem ser ineficazes ou podem gerar um grande número de mutantes equivalentes, cuja definição é apresentada ao longo desta seção. Há duas técnicas que podem ser utilizadas para o desenvolvimento de operadores. Na primeira, os operadores são definidos a partir do uso, quando disponível, de uma taxinomia de defeitos, erros e falhas identificados e classificados durante o processo de desenvolvimento de sistemas de software. Na segunda, os elementos sintáticos da linguagem que está sendo “mutada” são analisados, a fim de que se possa desenvolver operadores aptos a modificá-los.

A definição de operadores de mutação pode estar diretamente relacionada com, por exemplo: (a) paradigmas específicos de programação, como a programação procedimental, Orientada a Objetos (POO), concorrente, e Orientada a Aspectos (POA) (MA; KWON; OFFUTT, 2002; BRADBURY; CORDY; DINGEL, 2006; FERRARI; MALDONADO; RASHID, 2008); (b) domínios específicos de aplicação, tais como aplicações móveis e banco de dados (DENG et al., 2015; TUYA; SUÁREZ-CABAL; RIVA, 2007); e (c) tecnologias específicas, focando-se em operadores para linguagens específicas, tais como: Ada, AspectJ e Java (AGRAWAL et al., 1989; OFFUTT; VOAS; PAYNE, 1996; MA; KWON; OFFUTT, 2002; FERRARI; MALDONADO; RASHID, 2008).

O *teste de mutação* trabalha com duas hipóteses (DEMILLO; LIPTON; SAYWARD, 1978):

- *Hipótese do programador competente*: consiste na suposição de que o programa a ser testado está correto ou muito próximo ao correto. Sendo assim, os defeitos são inseridos no programa de forma a gerar pequenos desvios sintáticos, que podem alterar a semântica do programa e, conseqüentemente, gerar erros de comportamento.

- *Hipótese do efeito de acoplamento*: consiste na suposição de que casos de teste capazes de identificar defeitos simples podem ser utilizados para identificar defeitos complexos, já que se assume que um defeito simples está associado a uma alteração sintática simples e um defeito complexo é composto de um conjunto de defeitos simples.

O *teste de mutação*, além de avaliar a qualidade do produto, tem a finalidade de avaliar a qualidade dos casos de teste definidos. Durante a execução de um conjunto de teste, se a saída obtida de um mutante for diferente da saída obtida da execução do programa original em pelo menos um caso de teste, o mutante é considerado *morto*. Caso isso não ocorra, o mutante é considerado *vivo* e pode haver duas possibilidades de análise:

1. o mutante é equivalente ao programa original e deve ser descartado, já que os resultados de todos os testes serão iguais tanto para o mutante quanto para o programa original; ou
2. é necessário melhorar os casos de teste, incluindo tantos quantos forem necessários à obtenção de uma saída diferente para o mutante em comparação com a execução dos mesmos no programa original.

De acordo com os resultados obtidos nos testes e os recursos disponíveis para uma nova execução, o testador tem autonomia para decidir se a atividade de teste continua, ou se será encerrada ao final do ciclo em execução. Os resultados do *teste de mutação* podem ser medidos com a ajuda do *escore de mutação*, que é calculado com a utilização da seguinte fórmula:

$$ms(P, T) = \frac{DM(P, T)}{M(P) - EM(P)}$$

sendo que:

T : conjunto de testes;

P : produto original;

$ms(P, T)$: *mutation score* (escore de mutação);

$DM(P, T)$: número de mutantes mortos por T ;

$M(P)$: número total de mutantes gerados; e

$EM(P)$: número total de mutantes equivalentes identificados.

Destaque-se que, eventualmente, mutantes podem auxiliar na identificação de defeitos no programa original, isto quando a modificação gerada pelo operador de mutação resulta, na verdade, numa versão correta do programa (MATHUR, 2007).

2.2.1 Técnicas de Redução de Custo do Teste de Mutação

Historicamente, o *teste de mutação* é considerado um critério extremamente eficaz, mas caro devido ao número de mutantes gerados e, conseqüentemente, à dificuldade para se executar todos os mutantes e se identificar os que são equivalentes ao software original (JIA; HARMAN, 2011; MATEO; USAOLA, 2014; PAPADAKIS et al., 2015). Nesse sentido, observa-se substancial esforço da comunidade científica para reduzir o custo de aplicação desse critério preservando ao máximo a sua eficácia. As técnicas de redução de custo podem ser agrupadas em diversas categorias tais como: execução parcial dos mutantes, também conhecida como *mutação fraca* (do inglês, *weak mutation*) (HOWDEN, 1982); agrupamento de mutantes, também conhecido como *mutant schema*, que consiste na criação de vários mutantes em um único arquivo de código (UNTCH; OFFUTT; HARROLD, 1993); e redução do número de mutantes gerados, analisados e executados (OFFUTT; ROTHERMEL; ZAPF, 1993).

Além das técnicas apresentadas anteriormente destacam-se outras, como: evitar a geração de mutantes equivalentes, com a definição de métodos para que mutantes equivalentes não sejam gerados (OFFUTT; ROTHERMEL; ZAPF, 1993; OFFUTT et al., 1996; WONG; MATHUR, 1995); detectar automaticamente mutantes equivalentes (OFFUTT; CRAFT, 1994; OFFUTT; PAN, 1996; HIERONS; HARMAN; DANICIC, 1999a; SCHULER; DALLMEIER; ZELLER, 2009); otimização de compilação, que utiliza recursos computacionais para otimizar o processo de compilação dos mutantes (OFFUTT; CRAFT, 1994), etc.

Aliadas a essas categorias, Offutt e Untch (2001) definem três estratégias para classificar as técnicas de redução de custo: “*do Fewer*” (fazer menos), “*do Smarter*” (fazer de forma mais inteligente) e “*do Faster*” (fazer mais rápido). As técnicas que utilizam a estratégia “*do Fewer*” procuram gerar e executar menos mutantes sem que ocorra a perda de eficiência e eficácia da qualidade do teste e de seus resultados. Exemplos de técnicas “*do Fewer*” são: *Mutação Seletiva*, que opta por descartar operadores responsáveis por gerar grandes quantidades de mutantes (OFFUTT; ROTHERMEL; ZAPF, 1993); *Mutação Aleatória*, que utiliza somente uma porcentagem dos mutantes gerados (WONG; MATHUR, 1995); *Mutação Um Operador*, na qual os mutantes são gerados utilizando somente um operador de mutação (UNTCH, 2009), etc.

Nas técnicas que seguem a estratégia “*do Smarter*”, procura-se distribuir os custos de execução em várias máquinas ou em várias execuções ou, ainda, não executar completamente os mutantes. Exemplos de técnicas “*do Smarter*” são: *Mutação Fraca*, aonde o mutante é executado até o ponto do código em que ocorreu a mutação (OFFUTT; LEE, 1994; PAPADAKIS; MALEVRIS; KALLIA, 2010; KINTIS; PAPADAKIS; MALEVRIS, 2010); técnicas baseadas na análise de *fluxo de dados*, as quais indicam quais são os tipos de mutantes que serão gerados (HIERONS; HARMAN; DANICIC, 1999a; OFFUTT; PAN, 1996; OFFUTT; MA; KWON, 2006); e técnicas baseadas na análise de *fluxo de con-*

trole, que usam o fluxo de controle do artefato para analisar características de execução, a fim de identificar qual estrutura deve ser utilizada para que mutantes relevantes sejam gerados (PAPADAKIS; MALEVRIS, 2012; ZHANG et al., 2012; PATRICK; ORIOL; CLARK, 2012; SUN et al., 2016).

No caso da estratégia “*do Faster*”, as técnicas que a seguem procuram apresentar métodos de geração e execução de cada mutante o mais rapidamente possível. Exemplos são: *Otimização da compilação*, em que os recursos computacionais são utilizados para otimizar o processo de compilação e execução dos mutantes (OFFUTT; CRAFT, 1994; PAPADAKIS et al., 2015); *Esquema de Mutantes*, na qual o mutante gerado é parametrizado, permitindo execuções individuais de mutantes que estão todos representados em um único arquivo de código fonte, de acordo com parâmetros pré-definidos e iterativamente atualizados (REALES; POLO, 2012; KIM; MA; KWON, 2012).

No contexto deste trabalho, uma revisão sistemática da literatura foi conduzida como forma de caracterizar o estado da arte em técnicas de redução de custo do teste de mutação. Detalhes da revisão realizada são apresentados no Capítulo 3.

2.3 Conceitos de Abstração de Software

Nesta seção destaca-se a importância das abstrações, tendo em vista que as mesmas têm o objetivo de simplificar o processo de análise de programas complexos. Na prática, essa técnica cria uma relação entre o programa original e um programa abstrato (COUSOT; COUSOT, 1977), que pode ser utilizado na etapa da similaridade do framework definido no contexto deste trabalho. Em particular, apresentam-se a seguir as abstrações utilizadas neste trabalho, sendo elas: métricas internas e código-fonte ofuscado.

2.3.1 Métricas Internas

Métricas de software são medidas de software e a principal proposta delas é ajudar a prever e planejar o desenvolvimento de um software. Além disso, as métricas internas também auxiliam no controle da qualidade do software e no esforço de desenvolvimento (LI, 2000). Complementando este conceito, métricas internas podem servir de apoio para melhorar a qualidade do software, além de também auxiliarem a estimar o custo e o cronograma de projetos de software (CHIDAMBER; KEMERER, 1994).

Métricas de tamanho

Objetivam quantificar o tamanho do programa. Entre essas métricas, as utilizadas neste trabalho são listadas a seguir.

- *Lines of Code*(LOC) - computa a quantidade de linhas de código que o programa possui.
- *Number of Classes* - computa a quantidade de classes que o programa possui.

- ❑ *Number of Public Methods*(NPM) - computa a quantidade de métodos públicos de cada classe.

Métricas de Acoplamento

São usadas para medir a qualidade de um projeto orientado a objetos em termos da interdependência entre os subsistemas desse projeto (MARTIN, 1994). Essas métricas são descritas a seguir:

- ❑ *Afferent Couplings* (Ca) - computa a quantidade de classes diferentes que referem-se à classe em análise, através de chamadas de métodos, tipos de retorno, argumentos e exceções.
- ❑ *Efferent Couplings* (Ce) - computa a quantidade de classes diferentes que a classe em análise faz referência, através de chamadas de métodos, tipos de retorno, argumentos e exceções.
- ❑ *Instability* (I) - computa, baseando-se nos valores das métricas Ca e Ce, o grau de instabilidade da classe em análise.

Métricas CK

Estão relacionadas ao desenvolvimento de programas orientados a objetos e foram propostas por Chidamber e Kemerer (1994). Abaixo tem-se a descrição de cada uma delas:

- ❑ *Weighted Methods Per Class* (WMC) - computa a complexidade dos métodos de uma classe. Na prática, esta métrica pode representar a soma da complexidade ciclomática de todos os métodos de uma classe.
- ❑ *Depth of Inheritance Tree* (DIT) - computa a profundidade da árvore de herança para cada classe.
- ❑ *Number of Children* (NOC) - computa o número de filhos de cada classe, medindo o número de descendentes imediatos de cada classe.
- ❑ *Coupling Between Objects* (CBO) - computa o número de classes acopladas a uma outra determinada classe (acoplamento eferentes e acoplamentos aferentes) que podem ocorrer por meio de chamadas de métodos, herança, tipos de retorno e etc.
- ❑ *Response For a Class* (RFC) - computa o número de métodos diferentes que podem ser executados através do objeto da classe em análise, incluindo todos os métodos acessados dentro da hierarquia da classe.

- ❑ *Lack of Cohesion in Methods* (LCOM) - computa a falta de coesão de uma classe. Na prática, essa métrica conta os conjuntos de métodos na classe que não estão relacionados com alguns atributos da mesma classe.

Métricas de Henderson-Sellers

Versão alternativa para a métrica LCOM, a métrica LCOM3 proposta por Henderson-Sellers (1995) computa um valor entre 0 e 2 através da Equação 1.

$$LCOM3 = \frac{(\frac{1}{a} \sum_{j=1}^a \mu(A_j)) - m}{1 - m} \quad (1)$$

Na equação, a representa o número de atributos na classe, m é o número de métodos na classe, e $\mu(A)$ representa o número de métodos que acessam o atributo da classe.

Métricas de QMOOD

Utilizadas para avaliar o nível de coesão de métodos, encapsulamento, agregação e abstração (BANSIYA, 1999; BANSIYA; DAVIS, 2002).

- ❑ *Data Access Metric* (DAM) - computa a razão entre o número de atributos privados (protegidos) e o número total de atributos declarados na classe. O valor da métrica varia entre 0 e 1, sendo desejado o valor máximo 1.
- ❑ *Measure of Aggregation* (MOA) - computa o número de atributos da classe cujos os tipos são classes definidas pelo usuário.
- ❑ *Measure of Functional Abstraction* (MFA) - computa a razão entre o número de métodos herdados por uma classe e o número total de métodos acessíveis pelos métodos membros da classe.
- ❑ *Cohesion Among Methods of Class* (CAM) - computa o relacionamento entre os métodos de uma classe com base na lista de parâmetros dos métodos. A métrica é calculada usando a soma do número de diferentes tipos de parâmetros de método em cada método dividido por uma multiplicação do número de diferentes tipos de parâmetros do método em toda a classe e número de métodos.

Métricas de Tang

Representa um conjunto de três métricas propostas por Tang, Kao e Chen (1999), em um estudo empírico.

- ❑ *Inheritance Coupling* (IC) - computa o número de classes pai às quais uma determinada classe está acoplada. Uma classe é acoplada à sua classe pai se um de seus métodos herdados depender funcionalmente dos métodos novos ou redefinidos na classe.

- ❑ *Coupling Between Methods* (CBM) - computa o número total de métodos novos/redefinidos aos quais todos os métodos herdados estão acoplados.
- ❑ *Average Method Complexity* (AMC) - computa a quantidade média de linhas dos métodos de uma determinada classe.

Métricas de Complexidade Ciclomática

Retorna a quantidade de caminhos independentes que um programa, ou um método de um programa, pode seguir. Foi proposta por McCabe (1976) e é utilizada em testes de caixa branca, pois auxilia na identificação da quantidade mínima de casos de testes que serão necessários para percorrer todos os caminhos de um determinado programa (MYERS; SANDLER; BADGETT, 2011).

2.3.2 Código-Fonte Ofuscado

Um outro tipo de abstração analisada e utilizada para programas está relacionado à utilização de ofuscadores de código-fonte. Este tipo de abstração tem como principal objetivo dificultar a engenharia reversa de programas, que é uma técnica que possibilita a recriação do código-fonte de um determinado programa a partir do código de máquina, ou seja, do executável do programa (LINN; DEBRAY, 2003).

Além de dificultar a engenharia reversa de programas, algumas ferramentas de ofuscamento, como ProGuard,² também possibilitam a redução significativa dos tamanhos dos executáveis dos programas. Essa redução de tamanho ocorre com a utilização de recursos como encurtamento de nomes de variáveis, métodos e classes, e também através de otimização de trechos de códigos, de modo que não comprometa o funcionamento do programa em sua proposta original. Essas otimizações e encurtamentos ocorrem durante o processo de ofuscamento do programa e auxiliam no processo de análise da similaridade apresentado na Seção 2.4.

2.4 Conceitos de Similaridade de Software

Nesta seção apresentam-se os conceito de similaridade entre programas, e as abordagens que estão envolvidas diretamente neste trabalho. Ressalta-se que, neste trabalho, para se definir a similaridade entre programas, devem-se computar as abstrações apresentadas na seção anterior.

O conceito de similaridade se divide em dois tipos: visão sintática e visão semântica, sendo que ambas estão relacionadas a uma visão tática do que seria um “programa” (WALLENSTEIN et al., 2007). A visão sintática, ou representacional, de maneira geral, é

² <<https://www.guardsquare.com/proguard>> - acessado em Dezembro, 2022.

voltada para uma comparação de estrutura e não de significado. Nesse tipo de visão, a comparação é lida com vários níveis de abstrações, como declaração e blocos de códigos. Por outro lado, a similaridade comportamental, ou semântica, refere-se a uma comparação de significado e não de estrutura.

O conceito de similaridade é utilizado em diversos contextos, como: remoção de redundâncias, detecção de plágio, e na diferenciação de programas (WALENSTEIN et al., 2007). Buscando contemplar ambas as formas de similaridade, usou-se métricas de distância de *strings* e algoritmos de *clustering* para identificar a similaridade representacional. Já para a similaridade semântica, utilizou-se a técnica de detecção de plágios. Detalhes são apresentados a seguir.

2.4.1 Detecção de Plágio de Código Fonte

Esta técnica foi utilizada como alternativa viável para identificar similaridades (“plágio”) entre programas. Vários pesquisadores (SCHLEIMER; WILKERSON; AIKEN, 2003; PRECHELT et al., 2002) buscam formas para auxiliar na identificação de plágio entre programas, tendo em vista que o processo manual de identificação é complexo e demorado. Há diversas ferramentas que buscam automatizar este processo. Destaca-se, aqui, a ferramenta JPlag (PRECHELT et al., 2002), a qual foi empregada neste trabalho. De maneira geral, ao término da execução dessas ferramentas, computa-se um percentual de similaridade entre os programas envolvidos na execução.

Optou-se por utilizar esta ferramenta de detecção de plágio de código fonte como uma das abordagens, por parecer promissora no processo de identificar similaridades entre programas.

2.4.2 Métricas de Distância entre Strings

Este tipo de métrica determina um valor numérico que quantifica a distância entre duas *strings*. Neste trabalho, utilizou-se esta opção como abordagem de similaridades entre códigos fontes de programas, tanto original quanto ofuscado, para identificar o quão similares são. Optou-se por estas métricas por serem amplamente utilizadas e por serem computadas com uso de ferramenta empregado nesta pesquisa. Na sequência conceituam-se as métricas de distância entre *strings*.

Jaccard

Retorna um valor entre 0 e 1 que representa a distância entre dois conjuntos de *strings* A e B , onde quanto mais próximo de 0, mais similares são os dois conjuntos e, quanto mais próximo de 1, mais distantes são os dois conjuntos. A métrica foi proposta por Jaccard (1901) e é dada pela Equação 2. Na Figura 1 apresenta-se um pequeno exemplo de como o cálculo de métrica de distância Jaccard é realizado para dois conjuntos de comandos.

Observa-se que a interseção entre os dois conjuntos é igual a 2, enquanto que a junção dos elementos dos dois conjuntos é igual 5. Aplicando a Equação 2, o resultado final é de 0.6; deste modo, a distância entre os dois conjuntos de comandos, em uma escala de 0 a 1, é de 0.6.

$$d_J(A, B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|} \quad (2)$$

Figura 1 – Exemplo de cálculo de métrica de distância Jaccard.

$$\begin{aligned} A &= (\text{"printf"}, \text{"int"}, \text{"double"}, \text{"for"}) \\ B &= (\text{"double"}, \text{"for"}, \text{"if"}) \\ A \cup B &= (\text{"double"}, \text{"for"}, \text{"if"}, \text{"int"}, \text{"printf"}) \\ A \cap B &= (\text{"double"}, \text{"for"}) \\ d_J(A, B) &= \frac{|A \cup B| - |A \cap B|}{|A \cup B|} \\ d_J(A, B) &= \frac{5 - 2}{5} = \frac{3}{5} = 0.6 \end{aligned}$$

Na prática, cada código fonte de programa é representado por um conjunto de *strings* de símbolos ou comandos do código fonte. Aplicando-se a equação da métrica de distância Jaccard, é então computada a distância entre dois programas.

Levenshtein

Tem como objetivo computar o número mínimo de modificações (inserções, remoções e substituições) necessárias para transformar uma *string* em uma outra *string* (LEVENSHTein et al., 1966). Quanto menor o valor da métrica, mais similares são as duas *strings*.

Normalised Compression Distance

É uma métrica de distância dada pela Equação 3, em que x e y são duas *strings*, $C(x)$ é o tamanho da *string* x comprimida, $C(y)$ é o tamanho da *string* y comprimida, e $C(xy)$ é o tamanho da concatenação das *strings* x e y após o processo de compressão (CILIBRASI; VITÁNYI, 2005; FELDT et al., 2016). A função *min* retorna o menor tamanho entre as *strings* comprimidas. Por fim, a função *max* retorna o maior tamanho entre as *strings* comprimidas.

$$NCD(x, y) = \frac{C(xy) - \min \{C(x), C(y)\}}{\max \{C(x), C(y)\}} \quad (3)$$

O resultado da métrica de distância NCD é um valor entre 0 e 1, onde quanto mais próximo de 0, mais parecidas são as *strings* e, quanto mais próximo de 1, mais diferentes são as *strings*.

Algoritmos de Agrupamento

Estes algoritmos têm como objetivo, dada uma entrada de várias instâncias, formar grupos similares automaticamente. Para isso, utilizam algumas características já processadas e identificadas. Nos experimentos realizados neste trabalho, essas características foram representadas como: (i) um conjunto de resultados de métricas internas computadas para cada programa, e (ii) um conjunto de contagem de ocorrências de cada palavra presente no código fonte de cada programa.

K-Means (Agrupamento)

É o algoritmo que faz uso de um método iterativo que particiona um conjunto de instâncias em uma quantidade específica K de grupos de instâncias similares. Para isso, foi proposto com alguns passos de execução que são descritos a seguir:

- Definição da quantidade K desejada, onde K representa quantos *clusters* se deseja dividir as instâncias ao final da execução do algoritmo.
- Para cada *cluster* definido, é posicionado aleatoriamente um centroide em relação ao conjunto de instâncias. Esses centroides servem como referência para a formação dos *clusters* ao final da execução do algoritmo.
- Na formação dos *clusters*, cada instância é alocada no *cluster* com centroide mais próximo dela. Os algoritmos de distância Euclidiana e Manhattan, descritos na sequência, apresentam informações sobre as distâncias utilizadas neste trabalho para essa etapa.
- Cada centroide é reposicionado no centro do *cluster* correspondente, com base nas instâncias alocadas no mesmo.
- As duas últimas etapas descritas acima são executadas de forma iterativa enquanto houver modificações de alocação de instância na execução. Caso não ocorra novas modificações, então o algoritmo de *clustering* K-Means é dado como concluído.

X-Means (Agrupamento)

O algoritmo de *clustering* X-Means (PELLEG; MOORE, 2000) é uma variação do algoritmo K-Means. Na prática, inicia a execução com a menor quantidade de *clusters* K com base em um intervalo determinado. O algoritmo X-Means adiciona novos centroides à medida em que eles são necessários, até chegar na quantidade máxima definida no intervalo. Durante toda a execução do algoritmo, o conjunto de centroides que atinge a melhor pontuação é registrado, e este é o que finalmente é utilizado. Durante esse processo, a cada variação de quantidade de K , duas operações são executadas: *melhorar os parâmetros* e *melhorar a estrutura*.

Melhorar os parâmetros consiste na execução do algoritmo de K-Means original até a convergência, ou seja, até onde não ocorra nenhuma nova modificação de alocação de instâncias. Na operação *melhorar estrutura* o objetivo é checar se, e onde, os novos centroides devem aparecer; esse processo ocorre dividindo-se os *clusters* em dois novos *subclusters* através do K-Means. Cada divisão efetuada é avaliada e as divisões que geraram as melhores partições são mantidas, de forma a não gerar mais que a quantidade máxima de grupos do intervalo definido.

Em um cenário no qual nenhum *subcluster* é mantido, divide-se uma proporção dos *clusters* formados inicialmente, geralmente na metade, a partir das divisões que resultem nas melhores avaliações. Deste modo, mantém-se tanto os *clusters* bem distribuídos, quanto os novos possíveis *subclusters* com boa cobertura de instâncias não tão bem representadas pelo *cluster* inicial. Para avaliar a permanência ou não dos *clusters* e dos *subclusters* formados durante a execução do X-Means, utiliza-se o cálculo de *Bayesian Information Criterion* (BIC), da área da estatística, que indica o melhor modelo entre um conjunto limitado de modelos.

O algoritmo X-Means foi proposto para resolver alguns problemas apresentado pelo algoritmo K-Means, dentre os quais destacam-se: posicionamento inicial do centroide, que é feito de forma aleatória e pode não ser a ideal, e o algoritmo *clustering* ficar confinado a executar considerando desde o começo K *clusters* simultaneamente, assim dificultando a escolha ideal de alocação das instâncias envolvidas na execução do algoritmo.

Distância Euclidiana

É a distância calculada através da raiz quadrada da somatória das diferenças quadradas entre os vetores. Data pela Equação 4, x e y representam dois vetores de valores e n representa a quantidade de valores (dimensões) existentes em cada vetor. É utilizada nos algoritmos de *clustering* K-Means e X-Means, inclusive sendo a distância padrão nas principais ferramentas que aplicam estes algoritmos

$$D_E(x, y) = \sqrt{\sum_{i=1}^n (x_i - y_i)^2} \quad (4)$$

Distância Manhattan

É a distância calculada através da somatória das diferenças absolutas entre os dois vetores. Data pela Equation 5, x e y representam dois vetores de valores e n representa a quantidade de valores (dimensões) existentes em cada vetor. É utilizada nos algoritmos de *clustering* K-Means e X-Means, inclusive sendo a distância padrão nas principais ferramentas que aplicam estes algoritmos.

$$D_M(x, y) = \sum_{i=1}^n |x_i - y_i| \quad (5)$$

Expectation Maximization (EM)

Algoritmo de *clustering* que usa um processo iterativo para a estimação de parâmetros em modelos probabilísticos com variáveis incompletas ou que não podem ser observadas diretamente (DEMPSTER; LAIRD; RUBIN, 1977). Consiste de duas etapas: - Expectativa e Maximização - que são executadas até ocorrer a convergência. A etapa Expectativa utiliza variáveis latentes que estimam os dados faltantes para completar a amostra. Já na etapa Maximização, o algoritmo é aplicado fazendo-se uso dos dados estimados, de modo que os parâmetros atuais sejam substituídos pelos novos.

Essas duas etapas continuam em execução até ocorrer a convergência. Neste momento, a execução do algoritmo é então concluída e como resultado identificam-se os *clusters* com os elementos mais similares.

2.5 Considerações Finais

Neste capítulo foi apresentada uma sucinta fundamentação teórica sobre teste de software, na qual foram abordados os principais conceitos envolvidos na atividade de teste de software e a importância de sua utilização para melhorar e garantir a qualidade dos produtos criados. Também foi exposta uma visão geral sobre as principais técnicas. No que diz respeito ao critério *Análise de Mutantes*, o texto detalhou seus conceitos. Também apresentaram-se as abordagens de abstrações e similaridades de programas, citando as técnicas envolvidas neste trabalho.

No próximo capítulo serão apresentados os resultados da RS e uma atualização de estudos relacionados à hipótese abordada por esta pesquisa, necessários à caracterização do estado da arte em estratégias de redução de custos para testes de mutação, focando inclusive sistemas de software de grande porte.

Capítulo 3

Revisão Bibliográfica

Neste capítulo apresenta-se uma revisão bibliográfica da literatura (RSL) que resume e analisa o estado da arte quanto a técnicas de redução de custo para o teste de mutação. Resultados preliminares foram incluídos em um artigo apresentado no workshop Mutation'18 (FERRARI; PIZZOLETO; OFFUTT, 2018), e resultados completos e estendidos foram incluídos em um artigo publicado no periódico JSS - Journal of Systems and Software (PIZZOLETO et al., 2019). Ambos os artigos são incluídos, na íntegra, nos Apêndices A e B, respectivamente. Trabalhos relacionados mais recentes – isto é, divulgados após a publicação do artigo no periódico JSS – também são sumarizados ao longo deste capítulo, na Seção 3.3.

3.1 Objetivo, Planejamento e Execução da RSL

Historicamente, muitas técnicas para redução de custo no teste de mutação tem sido propostas, desenvolvidas e estudadas. Há um grande volume de estudos publicados, tornando difícil sumariá-los, e ainda mais difícil entender as compensações de custo-benefício apresentadas. Tendo conhecimento dessas dificuldades e objetivando ajudar pesquisadores atuais e futuros a entender o que foi feito e direcioná-los a novos rumos, bem como organizar o conhecimento para apoiar a transição tecnológica, realizou-se neste trabalho de doutorado uma revisão sistemática da literatura (RSL) para caracterizar a história e o estado da arte da redução de custos de testes de mutação. Ressalta-se que uma revisão sistemática da literatura (RSL) é um tipo de estudo secundário que segue um processo rigoroso e reproduzível, fornecendo uma visão geral de uma área de pesquisa (KITCHENHAM et al., 2009). A RSL realizada no contexto deste trabalho visou à caracterização de técnicas de redução de custo para o teste de mutação, assim como métricas que são empregadas para aferir resultados de aplicação dessas técnicas.

Para a RSL apresentada neste capítulo, definiram-se as seguintes três Questões de Pesquisa (RQs) a serem respondidas:

- ❑ (RQ1) Quais técnicas apoiam a redução de custo do teste de mutação?
- ❑ (RQ2) Quais métricas foram usadas para medir a redução de custo do teste de mutação?
- ❑ (RQ3) Quais são os benefícios e perda de efetividade (escore de mutação) para as técnicas?

Para esta RSL, aplicaram-se os seguintes critérios de seleção de inclusão (I) e exclusão (E) de estudos:

- ❑ (I1): O estudo foi incluído se propõe uma abordagem de redução de custo para o teste de mutação.
- ❑ (I2): O estudo foi incluído se aplica uma abordagem para redução de custos para o teste de mutação.
- ❑ (I3): O estudo foi incluído se for uma pesquisa original publicada em congresso ou periódico científico.
- ❑ (E1): O estudo foi excluído se menciona ou usasse uma técnica usada para reduzir o custo para teste de mutação, mas a redução de custo não é o objetivo principal desse estudo.
- ❑ (E2): O estudo foi excluído se usa informações de teste de mutação (por exemplo, utilizando-se o escore de mutação) para avaliar a qualidade de conjuntos de teste gerados com base em critérios de teste diferentes do teste de mutação).

Selecionaram-se os estudos que primeiro passaram em I3 e depois em I1 ou I2, ou seja, $(I3 \wedge (I1 \vee I2))$. Descartaram-se os estudos que passaram em E1 ou E2, ou seja, $(E1 \vee E2)$. Todo o processo de pesquisa, análise e seleção dos estudos foi dividido e executado em quatro etapas, como descrito a seguir:

Etapa 1–Busca Automática: Esta etapa contou com strings de busca e mecanismos de pesquisa nas principais bases. Os principais termos na string foram “*mutation testing*” e “*cost reduction*,”. A string de busca completa foi a apresentada a seguir:

(“mutation testing” or “mutation analysis” or “mutant analysis”) **and** (“cost reduction” or “sufficient operator” or “sufficient mutation” or “constrained mutation” or “selective mutation” or “weak mutation” or “random selection” or “random mutation” or “random mutants” or “equivalent mutant” or “equivalent mutation”)

A string de busca, que alguns casos precisou ser adaptada às particularidades de cada mecanismo de busca, foi executada nas seguintes bases: IEEE Xplore,¹ ACM Digital Library,² Elsevier ScienceDirect,³ Springer SpringerLink,⁴ e Wiley Online Library.⁵

Etapa 2–Snowballing: A técnica *backward snowballing* (WOHLIN, 2014) foi aplicada ao conjunto de estudos selecionados na Etapa 1, realizando somente uma interação no processo de análise. Para cada estudo selecionado, analisou-se a lista de referências para identificar outros estudos de interesse, de acordo com os critérios de inclusão e exclusão.

Etapa 3–Consulta a Autores: Foi feito contato com todos os autores dos estudos selecionados nas etapas anteriores e solicitaram-se sugestões de estudos adicionais, as quais foram analisadas de acordo com os critérios de inclusão e exclusão.

Etapa 4–Atualizações pós Fase 1: Todo o conjunto de estudos selecionados foi atualizado após a publicação preliminar (FERRARI; PIZZOLETEO; OFFUTT, 2018). Particularmente, foram analisados estudos publicados no 13th International Workshop on Mutation Analysis (Mutation’18), 11th International Conference on Software Testing, Verification and Validation (ICST’18), and 40th International Conference on Software Engineering (ICSE’18).

Tabela 1 – Número de estudos por rodada de busca. A coluna intitulada *Sel* inclui estudos não-duplicados para a seleção final. As colunas intituladas *Rec* incluem resultados duplicados de busca.

Rodada de Busca	Base / Motor de Busca												Total	
	IEEE		ACM		Elsevier		Springer		Wiley		Experts		..	
	Rec	Sel	Rec	Sel	Rec	Sel	Rec	Sel	Rec	Sel	Rec	Sel	Rec	Sel
1 - Autom (Abr, 2016)	467	27	612	19	606	12	464	0	771	13	19	13	2939	84
	8.88%	17.65%	11.64%	12.42%	11.52%	7.84%	8.82%	0.00%	14.66%	8.50%	0.36%	8.50%	55.89%	54.90%
2 - Autom (Out, 2017)	303	8	642	10	706	1	78	0	376	1	0	0	2105	20
	5.76%	5.23%	12.21%	6.54%	13.42%	0.65%	1.48%	0.00%	7.15%	0.65%	0.00%	0.00%	40.03%	13.07%
3 - Snowb (Out, 2017)	-	-	-	-	-	-	-	-	-	-	-	-	72	17
	-	-	-	-	-	-	-	-	-	-	-	-	1.37%	11.11%
4 - Survey (Out-Dez, 2017)	-	-	-	-	-	-	-	-	-	-	-	-	135	25
	-	-	-	-	-	-	-	-	-	-	-	-	2.57%	16.34%
5 - Atualização (Ago, 2018)													8	7
													0.15%	4.58%
TOTAL	770	35	1254	29	1312	13	542	0	1147	14	19	13	5259	153
	14.64%	22.88%	23.84%	18.95%	24.95%	8.50%	10.31%	0.00%	21.81%	9.15%	0.36%	8.50%	-	-

Para cada etapa de busca, realizou-se uma seleção preliminar por meio da leitura de partes específicas dos estudos (inicialmente, título e resumo e, se necessário, introdução e conclusão) para verificar se deveriam ser selecionados para leitura completa. Na Tabela 1 apresentam-se as quantidades de estudos retornados em cada uma das etapas, mecanismos de busca e quantidade de arquivos selecionados. Na etapa final de seleção, dois pesquisadores leram na íntegra cada estudo e tomaram a decisão final de seleção, de acordo

¹ <<http://ieeexplore.ieee.org>> - acessada em novembro, 2022.

² <<https://dl.acm.org/>>-acessada em novembro, 2022.

³ <<http://www.sciencedirect.com/>>-acessada em novembro, 2022.

⁴ <<http://link.springer.com/>>-acessada em novembro, 2022.

⁵ <<http://onlinelibrary.wiley.com/>>-acessada em novembro, 2022.

com os critérios de inclusão e exclusão. Todos os conflitos na seleção dos estudos foram resolvidos pelos dois pesquisadores a fim de reduzir potenciais ameaças.

Para esta RSL, foram necessárias informações mais específicas para as etapas de síntese e análise. A primeira consiste nas características das técnicas de redução de custos e nas métricas utilizadas para mensuração da redução de custos. A segunda consiste nos valores obtidos em relação à economia de redução de custo e os escores de mutação alcançados.

3.2 Resultados da RSL

Nesta seção apresentam-se os principais resultados gerados com a seleção e análise dos estudos selecionados. Ressalta-se que outros resultados obtidos podem ser encontrados no artigo que é incluído, na íntegra, no Anexo B deste documento.

No restante desta seção, apresenta-se a lista de técnicas de redução de custo (Seção 3.2.1), as quais são acompanhadas de suas breves descrições e dos respectivos estudos nos quais as técnicas foram exploradas. Além disso, apresenta-se uma relação das principais métricas que foram empregadas para medir as reduções de custo obtidas, junto aos estudos que as utilizaram (Seção 3.2.2). Por fim, apresentam-se as conclusões delineadas para a RSL (Seção 3.2.3).

3.2.1 Técnicas de Redução de Custo do Teste de Mutação

No que diz respeito às técnicas de redução de custo, foram identificadas 21 técnicas, as quais são apresentadas na sequência, enumeradas de T-1 a T-21.

T-1 *Mutação aleatória*: Esta técnica seleciona aleatoriamente do conjunto completo de mutantes, de acordo com uma distribuição de probabilidade predefinida. A mutação aleatória foi interpretada como “escolha X% de todos os mutantes”, “para cada mutante, gere-o com X% de probabilidade”, e “escolha X% dos mutantes gerados por cada operador” (WONG; MATHUR, 1995; ACREE et al., 1979; MATHUR; WONG, 1993; DEREZIŃSKA; RUDNIK, 2017; GOPINATH et al., 2017; PARSAI; MURGIA; DEMEYER, 2016; BLUEMKE; KULESZA, 2013; PAPADAKIS; MALEVRIS, 2010a; ZHANG et al., 2010; ZHANG et al., 2013; PETROVIĆ; IVANKOVIĆ, 2018; KURTZ et al., 2016).

T-2 *Mutação de ordem superior*: Esta técnica combina duas ou mais mutações simples para criar um único mutante complexo (ACREE et al., 1979; GOPINATH; MATHIS; ZELLER, 2018; LIMA et al., 2016; REULING et al., 2015; HARMAN et al., 2014; MADEYSKI et al., 2014; REALES; POLO; ALEMÁN, 2013; OMAR; GHOSH, 2012; KAMINSKI et al., 2011; KINTIS; PAPADAKIS; MALEVRIS, 2010;

- POLO; PIATTINI; GARCÍA-RODRÍGUEZ, 2009; ABULJADAYEL; WEDYAN, 2018; DEREZIŃSKA, 2016; KAMINSKI; AMMANN, 2011; PAPADAKIS; MALEVRIS, 2010a; DEVROEY et al., 2016; KINTIS; PAPADAKIS; MALEVRIS, 2015).
- T-3 *Mutação fraca*: Esta técnica de execução verifica se o estado do mutante está infectado logo após a execução do local mutado, em vez de verificar a saída após o término da execução. Se o estado estiver infectado, o mutante é morto imediatamente (HOWDEN, 1982; ZHU; PANICHELLA; ZAIDMAN, 2017; PAPADAKIS; MALEVRIS, 2011b; OFFUTT; LEE, 1994; ZHU; PANICHELLA; ZAIDMAN, 2018; DEVROEY et al., 2017; MA; KIM, 2016; FRASER; ARCURI, 2015; ZHANG et al., 2010; KINTIS; PAPADAKIS; MALEVRIS, 2010; MARSHALL et al., 1990; KIM; MA; KWON, 2012).
- T-4 *Mutação firme*: Esta é uma variante da mutação fraca em que a execução pode continuar por uma duração predefinida após o estado ser infectado (WOODWARD; HALEWOOD, 1988; JACKSON; WOODWARD, 2000; OFFUTT; LEE, 1994).
- T-5 *Execução paralela*: Esta técnica executa mutantes em processadores paralelos, reduzindo o tempo total necessário para realizar a análise de mutação (LI et al., 2015; REALES; POLO, 2013; BYOUNGJU; MATHUR, 1993; OFFUTT et al., 1992; KRAUSER; MATHUR; REGO, 1991; WANG et al., 2017; JACKSON; WOODWARD, 2000; DELAMARO et al., 2021).
- T-6 *Análise de fluxo de dados*: Esta técnica usa informações relacionadas ao fluxo de dados do programa para decidir quais mutantes gerar, e também para analisar os mutantes. A técnica considera se as variáveis mais propensas a falhas durante a execução são alcançadas e referenciadas (AL-HAJJAJI et al., 2017; KINTIS; MALEVRIS, 2015; PAPADAKIS; DELAMARO; Le Traon, 2014; KINTIS; MALEVRIS, 2014; SCHULER; ZELLER, 2013; PAPADAKIS; Le Traon, 2013; JUST; KAPFHAMMER; SCHWEIGGERT, 2012; PATRICK; ORIOL; CLARK, 2012; SCHULER; DALLMEIER; ZELLER, 2009; OFFUTT; MA; KWON, 2006; HIERONS; HARMAN; DANICIC, 1999b; OFFUTT; CRAFT, 1994; MARSHALL et al., 1990; DELGADO-PÉREZ et al., 2017a; KINTIS; PAPADAKIS; MALEVRIS, 2015; HARMAN; HIERONS; DANICIC, 2000).
- T-7 *Minimização e priorização de conjuntos de teste*: Esta técnica analisa o conjunto de testes para pontuar os casos de teste com base em sua eficácia em matar mutantes e, em seguida, elimina os casos de teste que são ineficazes, ou executa os casos de teste mais eficazes antes dos casos de teste menos eficazes (SAHINOGLU; SPAFFORD, 1990; DEREZIŃSKA, 2013; TUYA; SUÁREZ-CABAL; RIVA, 2007).
- T-8 *Otimização do compilador*: Esta técnica usa técnicas relacionadas ao compilador para otimizar a execução e análise de mutantes (por exemplo, para detectar

automaticamente mutantes equivalentes) (DENISOV; PANKEVICH, 2018; KINTIS et al., 2017; DEMILLO; KRAUSER; MATHUR, 1991; DELGADO-PÉREZ; MEDINA-BULO; MERAYO, 2017; JUST; KAPFHAMMER; SCHWEIGGERT, 2011; OFFUTT; CRAFT, 1994).

- T-9 *Mutação restrita*: esta técnica escolhe um subconjunto de operadores de mutação para usar. A escolha depende da intuição dos testadores em relação ao significado de grupos específicos de mutantes (MATHUR, 1991; MATHUR; WONG, 1993; PETROVIĆ; IVANKOVIĆ, 2018; WONG et al., 1994; GOPINATH et al., 2017).
- T-10 *Metamutantes (ou esquemas de mutantes)*: Esta técnica gera e executa mutantes incorporando todos os mutantes em um programa parametrizado chamado *metamutante*. O metamutante é então compilado para execução rápida. Quando executado, o metamutante recebe um parâmetro que informa qual mutante executar (UNTCH, 1992; GOPINATH; JENSEN; GROCE, 2016; REALES; POLO, 2014; WRIGHT; KAPFHAMMER; MCMINN, 2013; MA; OFFUTT; KWON, 2005; UNTCH; HARROLD; OFFUTT, 1997; UNTCH; OFFUTT; HARROLD, 1993; KIM; MA; KWON, 2012; PAPADAKIS; MALEVRIS, 2010b; WEISS; FLEYSHGAKKER, 1993; MA; KIM, 2016; PAPADAKIS; MALEVRIS, 2011b).
- T-11 *Análise de fluxo de controle*: Essa técnica usa informações relacionadas ao fluxo de controle do programa, com foco nas características de execução, para identificar ramificações e comandos que ajudam a determinar quais estruturas são mais relevantes para a geração e execução de mutantes (MARCOZZI et al., 2018; CHEN; ZHANG, 2018; MCMINN et al., 2018; DELGADO-PÉREZ et al., 2017a; JUST; KURTZ; AMMANN, 2017; SUN et al., 2017b; HOLLING et al., 2016; PATEL; HIERONS, 2016; KINTIS; PAPADAKIS; MALEVRIS, 2015; ZHANG et al., 2012; PAPADAKIS; MALEVRIS, 2012; KAMINSKI; AMMANN, 2011; ZHANG et al., 2010; PAPADAKIS; MALEVRIS, 2010b; PAPADAKIS; MALEVRIS, 2009; JI et al., 2009; LIU et al., 2006; HARMAN; HIERONS; DANICIC, 2000; OFFUTT; PAN, 1997; DEMILLO; OFFUTT, 1993; PETROVIĆ; IVANKOVIĆ, 2018; PAPADAKIS; DELAMARO; Le Traon, 2014; SCHULER; ZELLER, 2013; PAPADAKIS; Le Traon, 2013; JUST; KAPFHAMMER; SCHWEIGGERT, 2012; PATRICK; ORIOL; CLARK, 2012; PAPADAKIS; MALEVRIS, 2011b; SCHULER; DALLMEIER; ZELLER, 2009; OFFUTT; MA; KWON, 2006; BASHIR; NADEEM, 2017; FRASER; ARCURI, 2015; DEMILLO; OFFUTT, 1991; OFFUTT; JIN; PAN, 1999).
- T-12 *Mutação seletiva*: Esta técnica tenta evitar a aplicação de operadores de mutação que são responsáveis pela maioria dos mutantes, ou para selecionar operadores de mutação que resultam em mutantes que demandam casos de teste que matam muitos mutantes criados por outros operadores. A ideia é que se um conjunto de teste, T_{op} ,

adequado para um subconjunto de operadores de mutação *op*, também mata uma porcentagem muito alta de todos os mutantes, então pode-se selecionar apenas os operadores em *op* (SUN et al., 2017a; PRAPHAMONTRIPONG; OFFUTT, 2017; DELGADO-PÉREZ; SEGURA; MEDINA-BULO, 2017; BLUEMKE; KULESZA, 2014b; BLUEMKE; KULESZA, 2014a; ZHANG et al., 2013; GLIGORIC et al., 2013; DEREZIŃSKA; RUDNIK, 2012; SRIDHARAN; Siami-Namin, 2010; MRESA; BOTTACI, 1999; OFFUTT et al., 1996; OFFUTT; ROTHERMEL; ZAPF, 1993; KURTZ et al., 2016; ZHANG et al., 2010; TUYA; SUÁREZ-CABAL; RIVA, 2007; DELGADO-PÉREZ; MEDINA-BULO; MERAYO, 2017; GOPINATH et al., 2017).

- T-13 *Execução serial*: Esta técnica determina dinamicamente classes de mutantes que se comportam de forma semelhante, diminuindo assim o número de mutantes a serem executados. A sobrecarga incorrida precisa ser pequena em relação ao tempo economizado (MA; KIM, 2016; KIM; MA; KWON, 2012; FLEYSHGAKKER; WEISS, 1994; WEISS; FLEYSHGAKKER, 1993).
- T-14 *Operadores essenciais*: Esta técnica tenta determinar um conjunto essencial de operadores de mutação por meio da aplicação de procedimentos personalizados (JUST; SCHWEIGGERT, 2015; LACERDA; FERRARI, 2014; DELAMARO et al., 2014b; Siami-Namin; ANDREWS; MURDOCH, 2008; Siami-Namin; ANDREWS, 2006; VINCENZI et al., 2001; BARBOSA; MALDONADO; VINCENZI, 2001; ZHANG et al., 2010).
- T-15 *Otimização de geração, execução e análise de mutantes*: Esta técnica agrupa abordagens que reduzem o custo do teste de mutação explorando estratégias que não se encaixavam em outras categorias desta lista (ZHU; PANICHELLA; ZAIDMAN, 2018; IIDA; TAKADA, 2017; FERNANDES et al., 2017; MCMINN; KAPFHAMMER; WRIGHT, 2016; ZHANG et al., 2016; DEREZIŃSKA; HAŁAS, 2015; BELLI; BEYAZIT, 2015; JUST; ERNST; FRASER, 2014; DEREZIŃSKA, 2013; KAMINSKI; AMMANN; OFFUTT, 2013; FERRARI; RASHID; MALDONADO, 2013; KINTIS; MALEVRIS, 2013a; CACHIA; MICALLEF; COLOMBO, 2013; INOZEMTSEVA; HEMMATI; HOLMES, 2013; ZHANG; MARINOV; KHURSHID, 2013; WEDYAN; GHOSH, 2012; GLIGORIC et al., 2012; DURELLI; OFFUTT; DELAMARO, 2012; HU; LI; OFFUTT, 2011; JUST; KAPFHAMMER; SCHWEIGGERT, 2011; STEIMANN; THIES, 2010; KAMINSKI; AMMANN, 2009; ANBALAGAN; XIE, 2008; BOGACKI; WALTER, 2006; VINCENZI et al., 2002; ALEXANDER et al., 2002; CHEN; ZHANG, 2018; ZHU; PANICHELLA; ZAIDMAN, 2017; DEVROEY et al., 2016; GOPINATH; JENSEN; GROCE, 2016; JUST; SCHWEIGGERT, 2015; REULING et al., 2015; WRIGHT; KAPFHAMMER; MCMINN, 2013; ZHANG et al., 2012).

- T-16 *Algoritmos evolutivos*: Esta técnica usa algoritmos evolutivos para reduzir o número de mutantes, para reduzir o número de casos de teste ou para identificar mutantes equivalentes (ABULJADAYEL; WEDYAN, 2018; DELGADO-PÉREZ; MEDINA-BULO; MERAYO, 2017; BASHIR; NADEEM, 2017; DELGADO-PÉREZ et al., 2017b; QUYEN et al., 2016; Matnei Filho; VERGILIO, 2015; FRASER; ARCURI, 2015; HENARD; PAPADAKIS; Le Traon, 2014; OLIVEIRA; CAMILO-JUNIOR; VINCENZI, 2013; NOBRE; VERGILIO; POZO, 2012; FRASER; ZELLER, 2012; PAPADAKIS; MALEVRIS, 2011a; DOMÍNGUEZ-JIMÉNEZ et al., 2011; DOMÍNGUEZ-JIMÉNEZ; ESTERO-BOTARO; MEDINA-BULO, 2009; AYARI; BOUKTIF; ANTONIOL, 2007; BAUDRY et al., 2005; ADAMOPOULOS; HARMAN; HIERONS, 2004; HARMAN et al., 2014; SRIDHARAN; Siami-Namin, 2010; JI et al., 2009; PAPADAKIS; MALEVRIS, 2011b).
- T-17 *Mutação One-op*: Esta técnica usa apenas um único operador de mutação, que tem a vantagem de produzir poucos mutantes, mas também fornecer uma cobertura abrangente do programa (DEREZIŃSKA, 2016; DELAMARO et al., 2014a; DELAMARO; OFFUTT; AMMANN, 2014; DENG; OFFUTT; LI, 2013; UNTCH, 2009; KURTZ et al., 2016).
- T-18 *Análise de rastro de execução*: Esta técnica usa rastros de execução do programa original e alguns mutantes para decidir qual dos mutantes restantes deve ser executado (REALES; POLO, 2014; FRASER; ZELLER, 2012).
- T-19 *Mutação baseada em modelo*: Esta técnica transforma modelos formais ou informais do programa e, em seguida, usa os mutantes para gerar automaticamente casos de teste que são posteriormente usados para matar os mutantes do programa (DEVROEY et al., 2017; DEVROEY et al., 2016; AICHERNIG et al., 2014; AICHERNIG; JöBSTL; KEGELE, 2013).
- T-20 *Análise baseada em estado*: Esta técnica compara estados de diferentes execuções de mutantes. Quando dois mutantes levam ao mesmo estado de mutação, ou seja, quando o mesmo caminho de execução é observado, apenas um precisa ser executado e o resultado do outro pode ser inferido. Da mesma forma, a técnica cria grupos de classes que definem certas sequências de transição de forma que apenas uma precisa ser verificada (WANG et al., 2017; GLIGORIC; JAGANNATH; MARINOV, 2010; BASHIR; NADEEM, 2017; JUST; ERNST; FRASER, 2014; GLIGORIC et al., 2012).
- T-21 *Mutação mínima*: Esta técnica identifica e elimina mutantes redundantes aplicando os conceitos de inclusão de mutante (do inglês, *mutant subsumption*) e mutantes do-

minadores (KURTZ et al., 2016; AMMANN; DELAMARO; OFFUTT, 2014; JUST; KURTZ; AMMANN, 2017).

3.2.2 Métricas Utilizadas em Estudos de Redução de Custo do Teste de Mutação

No total, foram identificadas 18 métricas, as quais são apresentadas na sequência, enumeradas de M-1 a M-18, juntamente com suas descrições e com os estudos nas quais foram utilizadas. Observa-se que, de acordo com as descrições das métricas, as intenções de algumas métricas se sobrepõem. Por exemplo, as métricas *M-6 (Número de mutantes equivalentes automaticamente detectados)*, *M-11 (Número de mutantes matáveis automaticamente detectados)*, *M-12 (Número de mutantes equivalentes gerados)* e *M-17 (Número de mutantes duplicados automaticamente detectados)* têm intenções semelhantes, apesar de coletarem valores diferentes dos conjuntos de mutantes. Observa-se também que a definição de cada métrica pode ser derivada de seus nomes e intenção. Por exemplo, *M-3 (Aceleração de execução mutante)* pode ser definido como o tempo gasto (em uma determinada unidade de tempo) para execução mutante quando comparado à execução convencional (ou seja, sem aplicar uma técnica de redução de custo).

M-1 *Número de casos de teste requeridos para atingir cobertura de mutantes*: Estima o número de casos de teste que devem ser criados (automaticamente ou manualmente) ou executados. (WONG; MATHUR, 1995; LIU et al., 2006; TUYA; SUÁREZ-CABAL; RIVA, 2007; PAPADAKIS; MALEVRIS, 2010a; KAMINSKI; AMMANN, 2011; PAPADAKIS; MALEVRIS, 2012; DEREZIŃSKA, 2013; HARMAN et al., 2014; DELAMARO; OFFUTT; AMMANN, 2014; DELAMARO et al., 2014b; BLUEMKE; KULESZA, 2014a; LIMA et al., 2016; REALES; POLO; ALEMÁN, 2013; DELGADO-PÉREZ; SEGURA; MEDINA-BULO, 2017; CHEN; ZHANG, 2018; DELAMARO et al., 2014a; ZHU; PANICHELLA; ZAIDMAN, 2017; WONG; MATHUR; MALDONADO, 1995; JI et al., 2009; CACHIA; MICALLEF; COLOMBO, 2013; DEREZIŃSKA; RUDNIK, 2017; DEREZIŃSKA; RUDNIK, 2012; SAHINOGLU; SPAFFORD, 1990; OLIVEIRA; CAMILO-JUNIOR; VINCENZI, 2013; WONG et al., 1994).

M-2 *Número de mutantes a serem executados*: Estima o número de mutantes que precisam ser executados. (OFFUTT; ROTHERMEL; ZAPF, 1993; WONG; MATHUR, 1995; OFFUTT et al., 1996; BARBOSA; MALDONADO; VINCENZI, 2001; VINCENZI et al., 2001; TUYA; SUÁREZ-CABAL; RIVA, 2007; Siami-Namin; ANDREWS; MURDOCH, 2008; KAMINSKI; AMMANN, 2009; POLO; PIATTINI; GARCÍA-RODRÍGUEZ, 2009; UNTCH, 2009; SRIDHARAN; Siami-Namin, 2010; PAPADAKIS; MALEVRIS, 2010a; STEIMANN; THIES, 2010; KAMINSKI et

al., 2011; DOMÍNGUEZ-JIMÉNEZ et al., 2011; OMAR; GHOSH, 2012; JUST; KAPFHAMMER; SCHWEIGGERT, 2012; NOBRE; VERGILIO; POZO, 2012; BLUEMKE; KULESZA, 2014a; DEREZIŃSKA, 2013; DENG; OFFUTT; LI, 2013; REALES; POLO; ALEMÁN, 2013; ZHANG et al., 2013; GLIGORIC et al., 2013; JUST; SCHWEIGGERT, 2015; MADEYSKI et al., 2014; HARMAN et al., 2014; LACERDA; FERRARI, 2014; AMMANN; DELAMARO; OFFUTT, 2014; DELAMARO et al., 2014a; DELAMARO; OFFUTT; AMMANN, 2014; DELAMARO et al., 2014b; BELLI; BEYAZIT, 2015; LIMA et al., 2016; DELGADO-PÉREZ et al., 2017a; DELGADO-PÉREZ; SEGURA; MEDINA-BULO, 2017; WANG et al., 2017; ZHU; PANICHELLA; ZAIDMAN, 2017; GOPINATH et al., 2017; PARSAI; MURGIA; DEMEYER, 2016; PRAPHAMONTRIPONG; OFFUTT, 2017; SUN et al., 2017a; DEREZIŃSKA; HAŁAS, 2015; BLUEMKE; KULESZA, 2014b; CHEN; ZHANG, 2018; ZHU; PANICHELLA; ZAIDMAN, 2018; BLUEMKE; KULESZA, 2013; MARSHALL et al., 1990; WONG; MATHUR; MALDONADO, 1995; Siami-Namin; ANDREWS, 2006; JI et al., 2009; ZHANG et al., 2010; CACHIA; MICALLEF; COLOMBO, 2013; INOZEMTSEVA; HEMMATI; HOLMES, 2013; REULING et al., 2015; AL-HAJJAJI et al., 2017; JUST; KURTZ; AMMANN, 2017; DEREZIŃSKA; RUDNIK, 2017; DEREZIŃSKA, 2016; DEREZIŃSKA; RUDNIK, 2012; OLIVEIRA; CAMILO-JUNIOR; VINCENZI, 2013; HENARD; PAPADAKIS; Le Traon, 2014; MARCOZZI et al., 2018; KAMINSKI; AMMANN, 2009; QUYEN et al., 2016; KAMINSKI; AMMANN; OFFUTT, 2013; IIDA; TAKADA, 2017)

M-3 *Aumento na velocidade da execução de mutantes:* Estima quanto tempo é necessário para executar os mutantes. (OFFUTT et al., 1992; BYOUNGJU; MATHUR, 1993; OFFUTT; LEE, 1994; UNTCH; HARROLD; OFFUTT, 1997; MA; OFFUTT; KWON, 2005; GLIGORIC; JAGANNATH; MARINOV, 2010; PAPADAKIS; MALEVRIS, 2011b; JUST; KAPFHAMMER; SCHWEIGGERT, 2012; GLIGORIC et al., 2012; DURELLI; OFFUTT; DELAMARO, 2012; REALES; POLO, 2014; JUST; SCHWEIGGERT, 2015; JUST; ERNST; FRASER, 2014; DELGADO-PÉREZ et al., 2017a; WANG et al., 2017; ZHU; PANICHELLA; ZAIDMAN, 2017; ZHANG et al., 2016; DEVROEY et al., 2016; LI et al., 2015; DEREZIŃSKA; HAŁAS, 2015; KRAUSER; MATHUR; REGO, 1991; MCMINN; KAPFHAMMER; WRIGHT, 2016; KIM; MA; KWON, 2012; ZHANG et al., 2013; MADEYSKI et al., 2014; CHEN; ZHANG, 2018; ZHU; PANICHELLA; ZAIDMAN, 2018; UNTCH; OFFUTT; HARROLD, 1993; REALES; POLO, 2013; WRIGHT; KAPFHAMMER; MCMINN, 2013; MA; KIM, 2016; GOPINATH; JENSEN; GROCE, 2016; DEREZIŃSKA; RUDNIK, 2017; WEISS; FLEYSHGAKKER, 1993; FLEYSHGAKKER; WEISS, 1994; BOGACKI; WALTER, 2006)

- M-4 *Eficiência na execução de mutantes*: Estima os ganhos ou perdas de desempenho quando duas ou mais configurações de execução (por exemplo, infraestruturas de processadores e rede) são comparadas. (OFFUTT et al., 1992)
- M-5 *Aumento na velocidade da geração de casos de teste*: Estima quanto tempo é necessário para a geração de casos de teste. (DEMILLO; OFFUTT, 1993; LIU et al., 2006)
- M-6 *Número de mutantes equivalentes automaticamente detectados*: Estima o número de mutantes que precisam ser considerados na etapa de análise de mutantes. (OFFUTT; CRAFT, 1994; OFFUTT; PAN, 1997; ANBALAGAN; XIE, 2008; FERRARI; RASHID; MALDONADO, 2013; KINTIS; MALEVRIS, 2013a; WRIGHT; KAPFHAMMER; MCMINN, 2014; PAPADAKIS; DELAMARO; Le Traon, 2014; KINTIS; MALEVRIS, 2015; FERNANDES et al., 2017; KINTIS et al., 2017; DEVROEY et al., 2017; OFFUTT; MA; KWON, 2006; SCHULER; DALLMEIER; ZELLER, 2009; PAPADAKIS; Le Traon, 2013; PATEL; HIERONS, 2016; MARCOZZI et al., 2018; MCMINN et al., 2018)
- M-7 *Número de ciclos (ou gerações) de algoritmos evolucionários*: Estima o esforço necessário para executar um algoritmo evolucionário em particular. (BASHIR; NADEEM, 2017; BAUDRY et al., 2005)
- M-8 *Aumento na velocidade da compilação de mutantes*: Estima quanto tempo é necessário para a compilação de mutantes. (DELGADO-PÉREZ et al., 2017a; FERNANDES et al., 2017; JUST; KAPFHAMMER; SCHWEIGGERT, 2011; MA; OFFUTT; KWON, 2005)
- M-9 *Aumento na velocidade da geração de mutantes*: Estima quanto tempo é necessário para a geração de mutantes. (MA; OFFUTT; KWON, 2005; DOMÍNGUEZ-JIMÉNEZ et al., 2011; KIM; MA; KWON, 2012; JUST; KAPFHAMMER; SCHWEIGGERT, 2011)
- M-10 *Completude do conjunto de testes gerado*: Estima o esforço necessário para evoluir o conjunto de testes atual para alcançar a cobertura pretendida de mutantes. (PAPADAKIS; MALEVRIS, 2009; KAMINSKI; AMMANN, 2009; PAPADAKIS; MALEVRIS, 2012; Matnei Filho; VERGILIO, 2015; ZHANG et al., 2010)
- M-11 *Número de mutantes matáveis automaticamente detectados*: Estima o número de mutantes que precisam ser considerados na etapa de análise de mutantes. (DEVROEY et al., 2017; HOLLING et al., 2016; SCHULER; DALLMEIER; ZELLER, 2009; PATEL; HIERONS, 2016)

- M-12 *Número de mutantes equivalentes gerados*: Estima o número de mutantes que precisam ser considerados na etapa de análise de mutantes. (PAPADAKIS; MALEVRIS, 2010a; KINTIS; PAPADAKIS; MALEVRIS, 2010; HU; LI; OFFUTT, 2011; KAMINSKI et al., 2011; WEDYAN; GHOSH, 2012; DENG; OFFUTT; LI, 2013; KINTIS; MALEVRIS, 2014; DELAMARO et al., 2014b; POLO; PIATTINI; GARCÍA-RODRÍGUEZ, 2009; DELAMARO et al., 2014a; DELAMARO; OFFUTT; AMMANN, 2014; MADEYSKI et al., 2014)
- M-13 *Número de ciclos de algoritmos de geração de testes*: Estima o esforço necessário para executar um algoritmo de geração de casos de teste em particular. (PAPADAKIS; MALEVRIS, 2010b)
- M-14 *Aumento na velocidade da análise de mutantes*: Estima quanto tempo é necessário para analisar os mutantes. (STEIMANN; THIES, 2010; WRIGHT; KAPFHAMMER; MCMINN, 2014; MADEYSKI et al., 2014; DEVROEY et al., 2017; AICHERNIG; JöBSTL; KEGELE, 2013)
- M-15 *Probabilidade de matar mutantes*: Estima uma probabilidade de cobertura de mutantes baseado no conjunto de testes disponível. (PATRICK; ORIOLE; CLARK, 2012)
- M-16 *Número de execuções de casos de teste*: Estima o número de execuções de casos de teste para um dado conjunto de mutantes. (ZHANG et al., 2012; ZHANG; MARINOV; KHURSHID, 2013; BASHIR; NADEEM, 2017; DEREZIŃSKA; HAŁAS, 2015; KIM; MA; KWON, 2012; SAHINOGLU; SPAFFORD, 1990; MA; KIM, 2016; REALES; POLO, 2014; GOPINATH; MATHIS; ZELLER, 2018; CHEN; ZHANG, 2018)
- M-17 *Número de mutantes duplicados automaticamente detectados*: Estima o número de mutantes que precisam ser considerados na etapa de análise de mutantes. (KINTIS et al., 2017; FERNANDES et al., 2017; MCMINN et al., 2018)
- M-18 *Variação no escore de mutação*: Estima ganhos e perdas com respeito ao escore de mutação. (SUN et al., 2017b)

3.2.3 Conclusões da RSL

Com respeito à questão RQ1 (*Quais técnicas apoiam a redução de custo do teste de mutação?*), percebeu-se que custos relacionados ao teste de mutação podem ser reduzidos por meio da aplicação de uma grande variedade de técnicas, algumas individualmente, outras de forma combinada. Algumas das técnicas mais empregadas consistem em métodos de análise de software (por exemplo, análise de fluxo de controle e análise de fluxo de dados). Técnicas como, por exemplo, mutação seletiva e mutação de ordem superior,

são tipicamente exploradas na pesquisa em teste de mutação, ao passo que outras técnicas são amplamente utilizadas nas áreas de Ciência da Computação e Matemática (por exemplo, algoritmos evolucionários, otimização de compiladores, e otimização em geral). Em resumo, a pesquisa em redução de custo do teste de mutação tem se tornado mais interdisciplinar, o que demanda e motiva mais colaboração entre pesquisadores de áreas que vão além do teste de mutação sem si.

Com respeito à questão RQ2 (*Quais métricas foram usadas para medir a redução de custo do teste de mutação?*), notou-se que diversas métricas já foram utilizadas. Dentre elas, destacam-se *Número de mutantes a serem executados* (*M-2*), *Aumento na velocidade da execução de mutantes* (*M-3*), e *Número de casos de teste requeridos para atingir cobertura de mutantes* (*M-1*), todas essas abordando fatores importantes relacionados ao custo do teste de mutação. A despeito disso, os resultados da RSL também indicam que um grande desafio relacionado ao teste de mutação refere-se a como lidar com mutantes equivalentes; isso é refletido pelo número de estudos que mediram fatores relacionados a esse tipo de mutante. Mais precisamente, apenas 11% (17 de 151) dos estudos empregaram a métrica *M-6* (*Número de mutantes equivalentes automaticamente detectados*) e 8% (12 de 151) empregaram *M-12* (*Número de mutantes equivalentes gerados*).

Por fim, com respeito à questão RQ3 (*Quais são os benefícios e perda de efetividade (escore de mutação) para as técnicas?*), foram observadas substanciais variações em termos de efetividade das técnicas em termos de redução do custo de mutação e manutenção da qualidade dos testes relacionados. Dessa forma, torna-se bastante difícil estabelecer uma base de comparação para novos experimentos. A despeito disso, pesquisadores podem se beneficiar dos resultados da RSL de forma a utilizá-la como uma referência para obter detalhes sobre técnicas e estudos em particular, assim como para planejar estudos experimentais que sejam mais comparáveis e reproduzíveis.

3.3 Outros Trabalhos Relacionados

Nesta seção apresentam-se alguns estudos relacionados a este trabalho que não estavam presentes no conjunto de estudos analisado na RSL descrita nas seções anteriores deste capítulo. Buscou-se discutir estudos que se apoiaram em abstrações, na similaridade, ou informações internas, de programas para auxiliar no teste de mutação. Ressalta-se que os estudos apresentados é o resultado da pesquisa na base de dados Scopus⁶, usando a mesma string de pesquisa apresentada na 3.1 no período de janeiro/2019 a janeiro/2022.

Yu e Ma (2019) analisaram e evidenciaram a extensão da clusterização como apoio à redução de custo do teste de mutação, com foco no agrupamento de mutantes. Os autores analisaram a clusterização a níveis de instrução, expressão e bloco de código, com os quais obtiveram uma redução de custos, em média de 24,44%, 10,51% e 1,06%, res-

⁶ <<https://www.scopus.com/home.uri>> - acessado em dezembro, 2022.

pectivamente. Para o experimento foram utilizados nove programas do mundo real e oito operadores aritméticos de mutação com mutação fraca. Os autores analisaram e trabalharam com partes específicas do código e também operadores específicos, comprovando que a similaridade influencia na seleção de operadores e colaborando parcialmente com a hipótese apresentada nesta pesquisa, porém com objetivos distintos, tendo em vista que Yu e Ma (2019) focaram em agrupar mutantes, e não programas que seriam alvos das mutações.

TITCHEU CHEKAM et al. (2020) usaram informações estruturais para apoiar o teste de mutação. Nos testes realizados, utilizaram informações internas do programa e aprendizado supervisionado para selecionar mutantes que tornam o teste de mutação mais eficiente. O foco do método foi analisar o menor número possível de mutantes “matáveis” e priorizar os mutantes que revelam a maioria dos defeitos. A abordagem, chamada FaRM, aplica aprendizado supervisionado aos mutantes gerados de um grupo de programas defeituosos, e constrói um modelo de aprendizado. Esse modelo é então aplicado a outros mutantes e gera uma classificação de mutantes matáveis e reveladores de defeitos. Nos testes realizados, a técnica apresentou um desempenho superior à técnica de mutação seletiva.

O estudo de Wei et al. (2021) apresentou uma técnica que define a similaridade entre mutantes utilizando clusterização espectral. O objetivo é identificar os mutantes similarmente próximos e selecionar os mutantes representativos para execução dos casos de testes. A técnica foi aplicada em 12 programas e mostrou que pode reduzir o número de mutantes a serem executados, e consequentemente o tempo de execução dos mesmos, sem comprometer os resultados. Nota-se que a abordagem explorada neste trabalho de doutorado difere da abordagem de Wei et al. (2021), já que a primeira não requer a geração dos mutantes para análise.

Em seu estudo, Ojdanic et al. (2021) fizeram uso de processamento de linguagem natural (PLN) para gerar mutantes sintaticamente parecidos com os mutantes reais. Neste sentido, levanta-se a questão de saber se a geração de mutantes sintaticamente semelhantes de fato resulta em mutantes semanticamente semelhantes. De maneira geral, busca-se, com o uso da similaridade, identificar se os mutantes sintaticamente gerados estão distantes dos mutantes semanticamente gerados. Usuários utilizaram várias ferramentas para a geração de mutantes para provar que a similaridade sintática não reflete a similaridade semântica. Em seus experimentos, Ojdanic et al. (2021) utilizaram processamento de linguagem natural que objetiva definir a semelhança entre os mutantes gerados. Sendo assim há necessidade de gerar todos os mutantes para tal comparação, não focando-se em determinar os operadores que foram melhores classificados quanto à técnica explorada.

No trabalho de Strug e Strug (2013) foi proposto um método de classificação de mutantes baseada no uso de similaridade estrutural e um classificador K-NN. Os autores acreditam que usando-se o resultado de classificação seja possível prever se o mutante

seria detectado por testes ou não, permitindo assim a diminuição de mutantes e, consequentemente, de custo de execução. A ideia consiste em usar similaridade estrutural nos mutantes gerados, onde cada mutante é convertido em um grafo que representa o fluxo, variáveis e condições do programa. Fazendo uso dos grafos, a similaridade entre os programas é calculada e usada para prever se um determinado teste detecta ou não o mutante. Essa classificação se dá pelo método K-NN, que faz a decomposição de estruturas complexas em estruturas mais simples, dessa forma facilitando a identificação do quão semelhante esses códigos são. Os autores apresentaram uma ideia interessante, mas, uma vez mais, a similaridade está relacionada aos mutantes gerados, com possíveis ganhos em relação ao custo menor de execução dos mutantes. Entretanto, o custo de criação desse montante total de mutantes continuaria a existir, diferenciando-se da proposta apresentada neste trabalho, que consiste em eliminar desde o início a criação de mutantes desnecessários. Strug e Strug (2017), apresentaram uma evolução da proposta apresentada, fazendo uso de redes neurais no trabalho mais recente.

Outro trabalho que busca empregar similaridade de código de mutantes foi apresentado por Kintis e Malevris (2013b). Os autores apresentaram um conceito de mutantes espelhados, ou seja, que afetam fragmentos de código semelhantes. A ideia dos autores foi utilizar similaridade para identificar mutantes equivalentes. Para isso, argumentaram que mutantes espelhados exibem um comportamento análogo em relação à sua equivalência, assim se um deles for equivalente ao programa original, os outros mutantes também podem ser equivalentes. Os autores analisaram a similaridade em fragmentos de códigos fazendo uma análise intra-métodos e inter-métodos. Para a identificação de mutantes similares, os autores utilizaram uma ferramenta para detecção de Clones intitulada CCFinderx, a qual analisa fragmentos de códigos em busca de similaridade.

3.4 Considerações Finais

Neste capítulo apresentou-se, de forma resumida, a revisão sistemática da literatura sobre técnicas de redução de custo para o teste de mutação, que é um tema central deste trabalho de doutorado. Apresentou-se também um conjunto de estudos que exploraram similaridade entre programas como forma de apoio ao teste de mutação, porém sendo todos eles com foco diferente do que é investigado neste trabalho. De maneira geral, pode-se concluir que há muito a ser explorado, mas que essa pesquisa é inovadora, já que tem o foco principal no programa original que é alvo do teste de mutação, ao contrário do que se propõe nos estudos apresentados, os quais focam nos mutantes ou casos de teste. Tendo como base os fatos apresentados, esta pesquisa, incluindo a solução investigada e resultados, que serão apresentados nos próximos capítulos, tem um caráter original e visa a preencher uma lacuna até então pouco explorada pela comunidade de pesquisa em teste de mutação.

Capítulo 4

Framework *SiMut*

O objetivo pretendido para este trabalho de doutorado demandou a definição de um framework conceitual, o qual incorpora uma série de atividades, e o desenvolvimento de ferramental de apoio para viabilizar a condução de experimentos para gerar evidências que poderiam embasar conclusões com respeito à hipótese delineada. Conceitos-chaves do framework definido e uma visão geral dos mecanismos de apoio automatizado desenvolvidos são descritos neste capítulo.

A descrição completa do processo incorporado no framework foi incluída em um artigo apresentado no workshop Mutation'20 (PIZZOLETO et al., 2020), ao passo que uma implementação (incluindo módulos de software desenvolvidos neste trabalho) e algumas evoluções no processo foram descritos em um artigo apresentado no workshop Mutation'22 (GUARNIERI; PIZZOLETE; FERRARI, 2022) Ambos os artigos são incluídos, na íntegra, nos Apêndices C e D, respectivamente.

4.1 *SiMut*: Histórico e Visão Geral

O framework apresentado neste capítulo, intitulado *SiMut* (Program **S**imilarity to support Cost Reduction of **M**utation), foi concebido e desenvolvido no contexto de trabalho do Laboratório de Pesquisa em Engenharia de Software (LaPES) do Departamento de Computação da Universidade Federal de São Carlos (DC/UFSCar). A concepção e desenvolvimento do *SiMut*, incluindo o processo conceitual e aspectos de implementação, foram liderados pelo autor desta tese e por seu orientador.

Conceitualmente, a aplicação do *SiMut* se baseia em um conjunto de programas P já testados com o teste de mutação; isto é, P inclui apenas programas para os quais os artefatos do teste de mutação (por exemplo, mutantes, casos de teste, e resultados

de execução dos mutantes com os testes) já estejam disponíveis. Dados P , um programa ainda não testado u , e um tipo de abstração de programas (por exemplo, métricas internas ou código-fonte simplificado), a abordagem encapsulada no *SiMut* consiste em identificar um grupo de programas G , chamado de *grupo de referência*, que contenha os programas de P que são mais similares a u . Com base em G , o framework calcula parâmetros de redução de custo do teste de mutação que serão sugeridos para serem aplicados a u . Exemplos de parâmetros de redução de custo são melhores operadores (por exemplo, com base em no score de mutação dos mesmos) (UNTCH, 2009), operadores essenciais (BARBOSA; MALDONADO; VINCENZI, 2001), e parâmetros ajustados de classificadores de mutantes construídos com técnicas de aprendizado de máquina (Titcheu Chekam et al., 2019).

Na Figura 2 apresenta-se o processo geral encapsulado no *SiMut*.¹ A linguagem BPMN (do inglês, *Business Process Model and Notation*) é utilizada para representar o processo, a qual foca em subprocessos e tarefas, ambos representados com retângulos com cantos arredondados. Detalhes do subprocesso 1 (isto é, a composição do grupo de referência G) são apresentados a seguir (Seção 4.2), ao passo que detalhes do subprocesso 3 (isto é, aplicação da técnica de redução de custo do teste de mutação no grupo de referência) são apresentados na Seção 4.3. Em ambas as seções 4.2 e 4.3 também são apresentadas informações sobre a implementação do framework, cuja arquitetura é ilustrada na Figura 3. A descrição completa de todos os subprocessos, incluindo entradas em saídas de cada etapa, e também sobre a implementação, estão disponíveis nos artigos incluídos nos apêndices C e D.

Com respeito à implementação (Figura 3), os módulos *SiMut.Abstraction*, *SiMut.Similarity*, *SiMut.Core*, e *SiMut.DB*, que se baseiam na tecnologia Java, foram implementados no escopo de um trabalho de mestrado (GUARNIERI, 2022) realizado no mesmo grupo de pesquisa do qual o autor desta tese faz parte, sendo que o módulo *SiMut.DB* contou com a colaboração do autor desta tese. O módulo *CostReduction*, por sua vez, foi implementado com tecnologia .NET no escopo do trabalho relatado nesta tese.

4.2 Criação de Abstrações e Cálculo de Similaridade entre Programas

A composição do grupo de referência G envolve, inicialmente, a criação de abstrações tanto para o programa não testado u (subprocesso 1.1 na Figura 4) quanto para os programas já testados que compõem o conjunto P (caso estas últimas ainda não tenham sido criadas e armazenadas para uso posterior). O módulo *SiMut.Abstraction* se encarrega desta etapa.

¹ Para manter a consistência com as figuras originalmente apresentadas nos artigos publicados, optou-se por manter os elementos textuais das mesmas na língua inglesa.

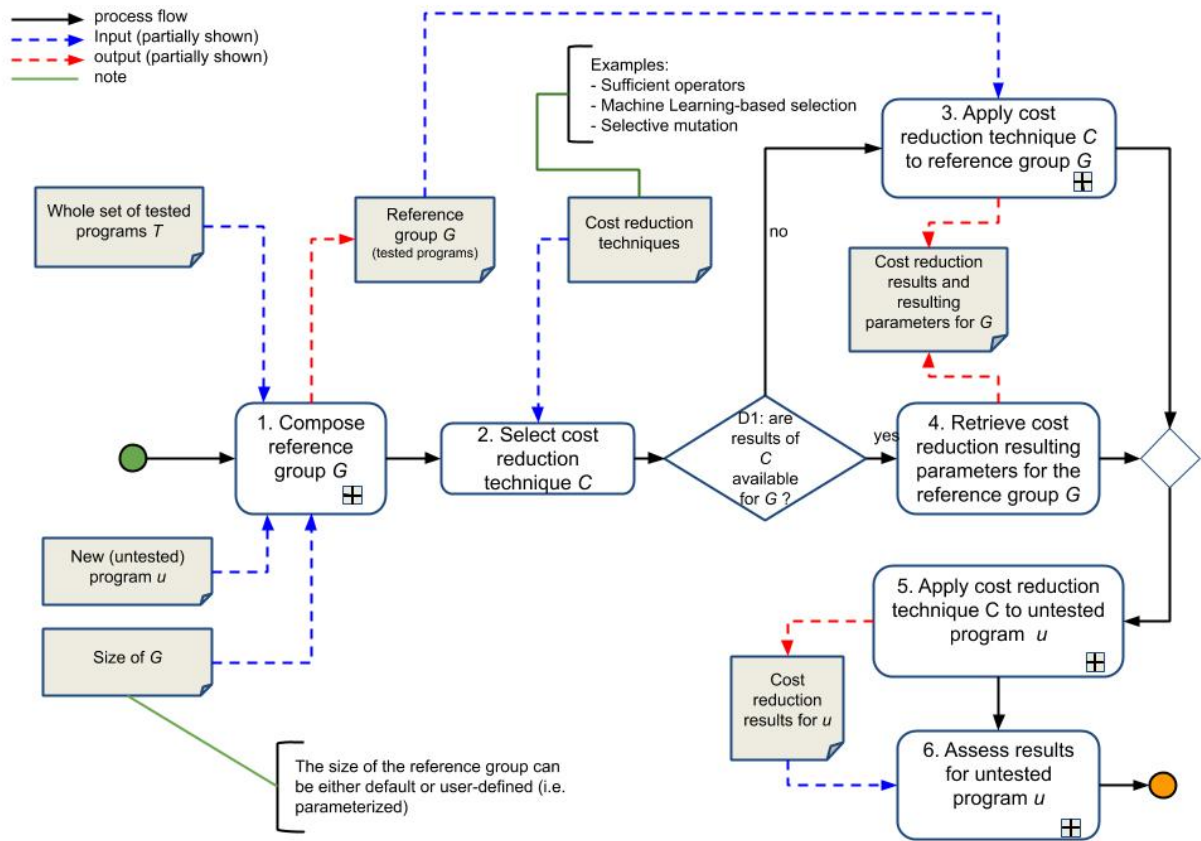


Figura 2 – Processo geral do SiMut (PIZZOLETO et al., 2020).

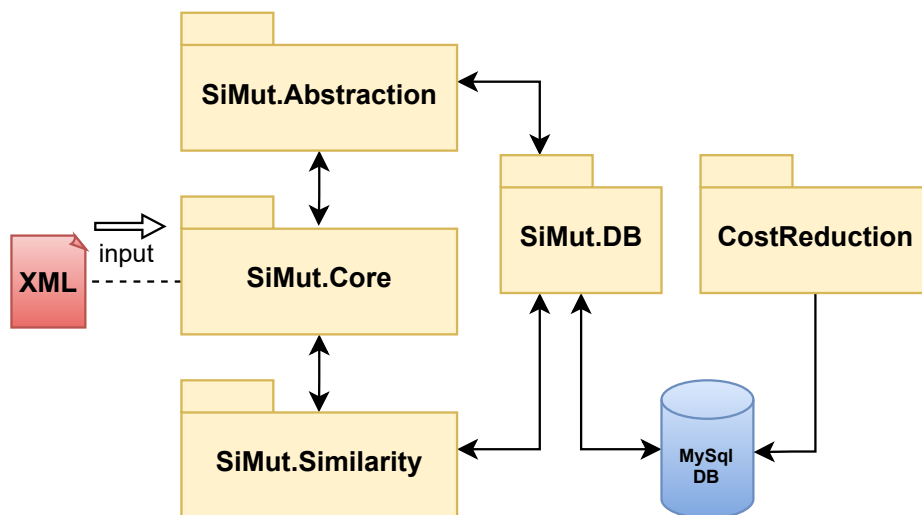


Figura 3 – Arquitetura da *SiMut* (GUARNIERI; PIZZOLETEO; FERRARI, 2022).

Um arquivo de entrada (Figura 3), em formato XML, define o conjunto de programas já testados P que será considerado no processamento, e também o programa não testado u . As abstrações implementadas e manipuladas pelo *SiMut* são: métricas internas de complexidade de código, código-fonte ofuscado, e código-fonte original. Detalhes das

ferramentas externas incorporadas ao *SiMut* são apresentados no artigo original (Apêndice D).

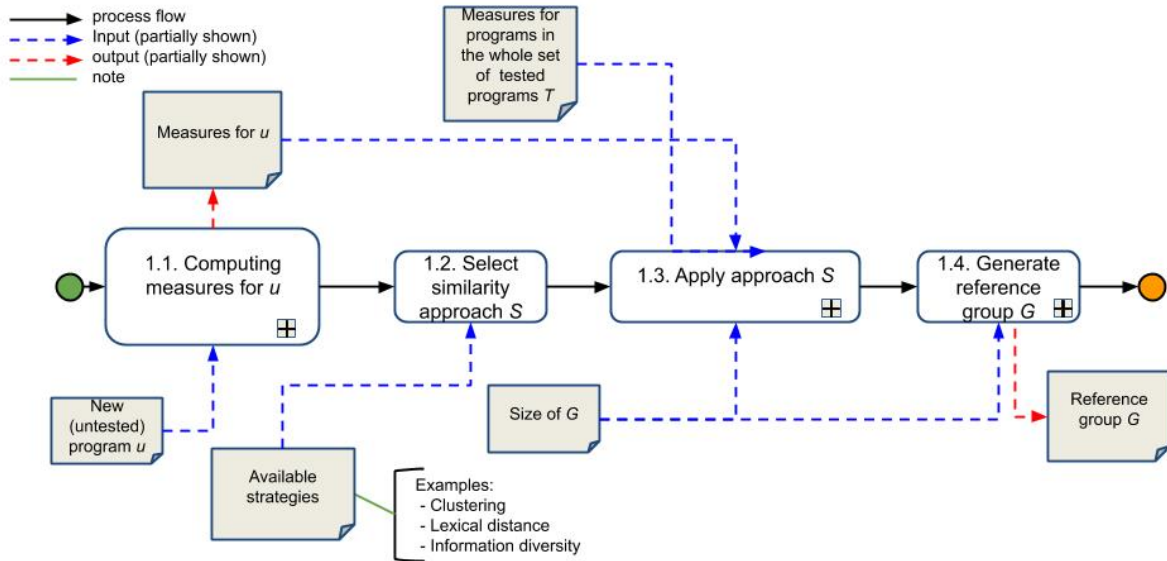


Figura 4 – Composição do grupo de referência G (PIZZOLETO et al., 2020).

Uma vez que as abstrações já tenham sido criadas, o módulo *SiMut.Similarity* cria grupos de programas (subprocessos 1.3 e 1.4 na Figura 4), sendo que um desses grupos – o grupo de referência G – conterá os programas de P que são mais similares a u . As técnicas de similaridade implementadas pelo *SiMut* são: agrupamento (do inglês, *clustering*), plágio, e diversidade de informação (FELDT et al., 2016). O tamanho dos grupos de programas são definidos no arquivo XML de entrada (Figura 3). Detalhes das ferramentas externas incorporadas ao *SiMut* para computar a similaridade entre programas são apresentados no artigo original (Apêndice D).

4.3 Cálculo de Parâmetros de Redução de Custo do Teste de Mutação

Os grupos de programas formados, Grupo G , pelo módulo *SiMut.Similarity* são elementos de entrada para o módulo *CostReduction*, que automatiza o subprocesso 3 e a tarefa 4 da Figura 2. Duas técnicas de redução de custo do teste de mutação que foram implementadas no módulo *CostReduction* são: seleção de melhores operadores de mutação² com base em escores de mutação gerados pelos operadores, e o Procedimento Essencial (BARBOSA; MALDONADO; VINCENZI, 2001). Ao passo que a primeira técnica (isto é, a

² Essa técnica é inspirada na técnica One-op (UNTCH, 2009), a qual preconiza a seleção de um único operador de mutação cujos mutantes demandam testes que são capazes de cobrir um conjunto amplo de mutantes produzidos por outros operadores.

seleção de melhores operadores de mutação) é simples de ser implementada (bastando ordenar os operadores de forma decrescente em relação ao escore de mutação que produzem), o Procedimento Essencial envolve um conjunto de heurísticas que é detalhado no Capítulo 6.

O escore de mutação de cada operador de mutação é um elemento central para as duas técnicas de redução de custo implementadas no *SiMut*. Para obter os escores de mutação, o módulo **CostReduction** dinamicamente cria uma matriz de mutantes mortos (do inglês, *killing matrix*), a qual inclui mutantes e casos de teste para todos os programas considerados na execução em curso do *SiMut*. Os programas, no caso, consistem em arquivos de código-fonte Java que podem incluir uma ou mais classes cada. Dentre as informações que podem ser extraídas da matriz, destacam-se: o escore de mutação de um programa em particular; o escore de mutação para um grupo de programas (por exemplo, um *cluster*); o escore de mutação para todos os programas considerados na execução; e o escore de mutação para um operador de mutação em particular, seja para programas individuais ou para grupos de programas. Todos os resultados produzidos pelo módulo **CostReduction** são armazenados em arquivos em formato CSV como forma de facilitar o processamento por outras ferramentas (por exemplo, *scripts* de processamento em lote).

4.4 Resumo das Contribuições e Limitações

As contribuições apresentadas neste capítulo incluem a definição de um framework conceitual, intitulado *SiMut*, que tem como objetivo incorporar uma abordagem de apoio ao teste de mutação a custo reduzido com base em similaridades de programas já testados. Ademais, este capítulo também trouxe informações sobre a automatização do *SiMut*, sendo ela realizada parcialmente pelo autor desta tese. Dois artigos que relatam tais contribuições estão incluídos como apêndices deste documento (em particular, Apêndice C e Apêndice D), sendo que resultados experimentais obtidos com um conjunto pequeno de programas são revisitados no Capítulo 6 desta tese.

As principais limitações do *SiMut*, em particular quanto à sua automatização, referem-se à quantidade de variantes implementadas (1) para criar abstrações de programas, (2) para calcular a similaridade entre programas, e (3) para computar parâmetros de redução de custo do teste de mutação. Com relação ao item (1), três abstrações foram consideradas: métricas internas de complexidade de código, código-fonte ofuscado, e o código-fonte original. Para o item (2), *clustering*, plágio, e diversidade de informação foram incorporadas ao framework. Por fim, com relação ao item (3), as técnicas de seleção de operadores baseada em maiores escores de mutação, e o Procedimento Essencial (BARBOSA; MALDONADO; VINCENZI, 2001) foram implementadas. Dado esse contexto, conclusões derivadas dos resultados obtidos com os experimentos que são des-

critos no Capítulo 6 se restringem à aplicação dessas variantes na abordagem incorporada ao *SiMut*.

Capítulo 5

Criação de uma Base de Conhecimento

Desde os primeiros anos de pesquisa a respeito do teste de mutação, desenvolveram-se dezenas de formas de criar, gerar e analisar mutantes, bem como métodos para reduzir os custos de sua aplicação. No entanto, a pesquisa fica, neste âmbito, aquém do esperado quando se trata de produzir um conjunto representativo e reutilizável de artefatos que permita a execução de experimentos fazendo uso da infinidade de abordagens de mutação existentes. Diante desta necessidade, optou-se por definir e implementar uma base de conhecimento de artefatos relacionados ao teste de mutação para apoiar experimentos com esse critério de teste de software.

Neste capítulo apresenta-se resumidamente o processo de criação da base e o resultado obtido. A descrição completa foi incluída em um artigo apresentado na Trilha de Idéias Inovadoras e Resultados Emergentes do SBES'21 (PIZZOLETO; GUARNIERI; FERRARI, 2021). O artigo está incluído, na íntegra, no Apêndice E.

5.1 Motivação e Requisitos para a Criação da Base

Em sua revisão sistemática da literatura, Pizzoleto et al. (2019) descobriram que o material experimental utilizado nos estudos primários envolvendo redução do custo do teste de mutação não são padronizados. Tais materiais necessitam de organização para permitir a reutilização, e raramente são disponibilizados para reutilização em outros experimentos. Essas revelações comprovam a dificuldade em reproduzir pesquisas anteriores e consequentemente evoluí-las. Esse problema de falta de padronização foi comprovado

em um trabalho subsequente, no qual Pizzoleto et al. (2020) tentaram formar um conjunto de artefatos produzidos e utilizados por outros pesquisadores. Foi dispensado um grande esforço na obtenção, entendimento da organização, e definição das características dos artefatos para importá-los para uma base de dados relacional. Por fim, Pizzoleto et al. (2020) concluíram que a fusão de artefatos de fontes experimentais variadas gerou um banco de dados limitado quanto à homogeneidade de elementos como, por exemplo, as ferramentas utilizadas, operadores de mutação aplicados, representatividade dos programas originais, qualidade dos dados do teste, e a precisão das informações sobre a cobertura dos mutantes.

Em linhas gerais, analisando-se todo o cenário que compõe o processo de desenvolvimento da investigação científica, particularmente o enfoque na investigação relacionada ao teste de mutação, comprovar a eficiência e eficácia de qualquer abordagem é um fator chave para o seu sucesso. Sendo assim, tal comprovação só é obtida através da utilização de conjuntos de artefatos, dados e informações que permitem a comparação dos resultados obtidos da abordagem atual com os resultados de outras abordagens semelhantes. Seguindo esse raciocínio, definiu-se que para se ter uma base de conhecimento coerente e concisa, composta por um conjunto de projetos de software que auxilie pesquisadores em teste de mutação, essa base de conhecimento deve seguir as principais características listadas a seguir:

- ❑ Para todos os projetos de software utilizados, compilação, geração dos casos de testes e execução de mutantes foram baseados em projetos Maven;¹
- ❑ Os casos de testes para cada classe são gerados automaticamente pela ferramenta EvoSuite,² e pelos desenvolvedores;
- ❑ O conjunto de mutantes para cada classe foi gerado pela ferramenta PIT;³
- ❑ O status de cada mutante é estabelecido, incluindo sua relação com casos de testes quando o mesmo foi morto; e
- ❑ A possibilidade de gerar escores de mutação de acordo com as informações selecionadas pelo usuário.

Dentre outras tantas consultas que o banco de dados relacional permite, algumas informações importantes que devem estar disponíveis para os usuários são: código-fonte original das classes, mutantes criados por operadores de mutação específicos ou por um grupo de operadores, bytecode de todos os mutantes gerados, código-fonte dos casos de testes definidos para cada classe, e conjunto de mutantes mortos e os casos de testes que os mataram.

¹ <<https://maven.apache.org/>> - acessado em novembro, 2022.

² <<https://www.evosuite.org/>> - acessado em novembro, 2022.

³ <<http://pitest.org/>> - acessado em novembro, 2022.

5.2 Processo de Criação da Base de Conhecimento

Considerando todos os fatores apresentados na seção anterior, os quais influenciam negativamente a replicação e evolução das pesquisas realizadas com o teste de mutação, decidiu-se definir, construir e manter uma base de conhecimento de artefatos, dados e informações para a reutilização em experimentos no domínio de teste de mutação. Ressalta-se que essa base de conhecimento pode ser utilizada em experimentos que utilizam outros critérios de teste, desde que contenha informações suficientes para este fim.

No processo de definição da base de conhecimento considerou-se como prioritário facilitar o acesso a todas as informações nela contida, desde o código-fonte das classes testadas até os escores de mutação obtidos. Assim, optou-se por utilizar o Apache Maven como ferramenta padrão de gerenciamento de projetos, bem como um banco de dados relacional com o intuito de concentrar em um único local o código-fonte das classes, casos de teste, mutantes, operadores de mutação, dentre outras informações.

O modelo de banco de dados é apresentado na Figura 5. O modelo foi definido para atender as características citadas e, principalmente, para permitir a rastreabilidade das informações. Por exemplo, partindo-se do programa ou da classe, é possível filtrar os mutantes gerados e os operadores utilizados, os casos de testes executados em cada mutante, e seu status para cada caso de teste. O modelo também permite que o usuário faça a rastreabilidade reversa, ou seja, a partir do caso de teste ou do operador de mutação, pode-se identificar o programa ou classe ao qual estão vinculados.

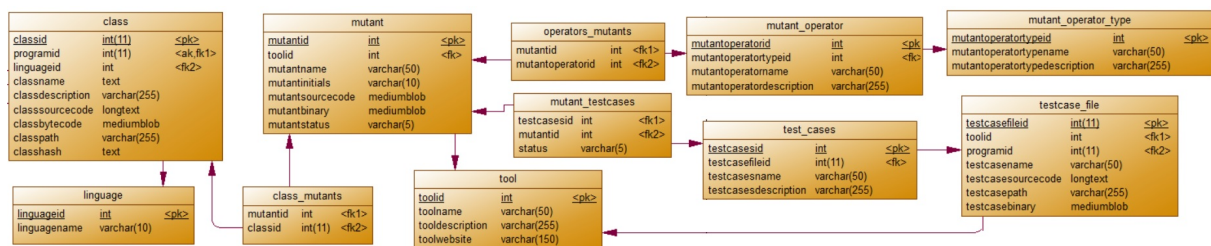


Figura 5 – Modelo da Base de Dados.

5.2.1 Projetos que Popularam a Base

Para compor a base de dados, foi realizada uma seleção criteriosa dos artefatos iniciais para serem inseridos na mesma. Incluíram-se apenas projetos compatíveis com a ferramenta Maven, já que seriam utilizadas ferramentas automáticas para geração de casos de teste e mutantes (Evosuite e PIT, respectivamente) que também são executadas via Maven. Uma consulta informal junto a alguns investigadores que trabalham no domínio de teste de software foi realizada e concluiu-se que um bom conjunto inicial deve apresentar

artefato de tamanhos variáveis. Sendo assim, optou-se por incluir projetos pequenos com apenas uma classe, e também projetos compostos por várias classes.

Em relação aos pequenos projetos, foram selecionados 28 dos 38 projetos utilizados por Dallilo, Pizzoleto e Ferrari (2019) de modo a esses projetos serem compiláveis. Por questões de conveniência e praticidade, decidiu-se juntar esses 28 projetos em um único projeto, daqui em diante denominados SimplePrograms, listado na última linha da Tabela 2, facilitando assim sua operação. Posteriormente foram adicionadas mais 7 classes, totalizando 35 (de um total de 38 utilizadas no trabalho anterior (DALLILO; PIZZOLETEO; FERRARI, 2019)).

Para grandes projetos, optou-se por incluir quatro projetos extraídos de repositórios do GitHub. Eles estão listados nas primeiras quatro linhas da Tabela 2. O projeto Apache Commons Lang⁴ (version 3-3.10) oferece vários utilitários auxiliares, como métodos de manipulação de strings e métodos numéricos básicos. Apache Commons Math⁵ (version 3-3.6.1) é uma biblioteca de componentes matemáticos e estatísticos leves e independentes. Javassist⁶ (version 3.27.0-GA) é uma biblioteca que facilita a manipulação do bytecode da aplicação. Finalmente, Joda-Time⁷ (version 2.10.7) oferece uma solução de alto nível para lidar com data/hora.

Na Tabela 2 mostram-se algumas métricas descritivas das aplicações e casos de teste. A tabela está dividida em três partes: Código da Aplicação, que diz respeito ao código fonte da aplicação, Código de Teste dos Desenvolvedores, que diz respeito ao código de teste original (quando disponível) e Código de Teste Evosuite, que diz respeito ao código de teste gerado pela ferramenta Evosuite. Na tabela, TLOC representa o total de linhas de código, TCs representa os casos de teste, AVG CC representa a complexidade ciclomática média das classes e AVG StdDev CC informa o desvio padrão médio da complexidade ciclomática entre as classes. Os números de arquivos e classes Java também são mostrados, a partir do qual podem-se notar que alguns arquivos de código-fonte Java definem mais de uma classe.

Como já mencionado, utilizou-se a ferramenta Evosuite⁸ na versão 1.0.6, que gera automaticamente caso de teste seguindo o padrão JUnit. Quanto à geração dos mutantes e consequentemente sua execução, optou-se pela ferramenta PIT⁹ na versão 1.6.2. Ambas as ferramentas são amplamente utilizados para a experimentação em projetos de teste de software. Adicionalmente, foi implementada uma ferramenta auxiliar para apoiar a preparação das informações que são processadas em cada uma das etapas de criação da base de conhecimento.

⁴ <<http://commons.apache.org/proper/commons-lang/>> - accessed in May, 2021.

⁵ <<http://commons.apache.org/proper/commons-math/>> - accessed in May, 2021.

⁶ <<https://www.javassist.org/>> - accessed in May, 2021.

⁷ <<https://www.joda.org/joda-time/>> - accessed in May, 2021.

⁸ <<https://www.evosuite.org/>> - accessed in May, 2021.

⁹ <<http://pitest.org/>> - accessed in May, 2021.

Tabela 2 – Métricas das aplicações incluídas na base.

Aplicação	Código da Aplicação						Código de Teste dos Desenvolvedores						Código de Teste Evosuite					
	# arquivos Java	# classes	TLOC	AVG CC	AVG StdDev	AVG CC	# arquivos Java	# classes	TLOC	AVG CC	AVG StdDev	AVG CC	# arquivos Java	# classes	TLOC	AVG CC	AVG StdDev	# TCs
commons-lang	161	258	28,497	2.433	3.297	3.297	179	442	49,864	1.139	0.673	0.673	222	222	176,466	1.398	0.491	6,655
commons-math	715	1,130	99,453	2.266	4.293	4.293	611	863	105,330	1.822	2.187	2.187	940	940	264,179	1.399	0.490	10,023
javassist	234	398	37,765	2.362	4.696	4.696	298	570	44,107	1.338	0.870	0.870	460	460	570,670	1.449	0.514	7,385
joda-time	204	245	28,790	1.770	2.422	2.422	159	226	55,608	1.292	1.025	1.025	282	282	280,305	1.374	0.486	7,761
simplePrograms	28	28	1,337	2.885	2.505	2.505	0	0	0	0	0	0	56	56	5,122	1.154	0.361	318
Total	1,342	2,059	195,842				1,247	2,101	254,909				1,960	1,960	1,296,742			32,142

A ferramenta Evosuite utiliza uma abordagem híbrida para gerar e otimizar os casos de testes para que o critério de cobertura seja atendido, adiciona um conjunto de oráculos possíveis. Com pequenas asserções que resumem o comportamento atual da classe atestado, permite identificar possíveis desvios de comportamento. A Evosuite possui vários critérios de cobertura; para esse trabalho optou-se por utilizar todos. Na Figura 6 apresenta-se a linha de comando utilizada para gerar os casos de teste. Um parâmetro

chave para a execução é o `-Dsearch_budget`, o que estabelece o tempo, em milissegundos, utilizado para gerar os casos de teste. Quanto maior este tempo, maior a quantidade e melhora a qualidade dos casos de teste gerados.

```
java -Dtools_jar_location=... -jar evosuite-1.0.6.jar -target <project path .jar>
-Duse_separate_classloader=false -Dsearch_budget=6000 -Dstopping_condition=MaxTime
```

Figura 6 – Linha de comando - Evosuite.

A ferramenta PIT destaca-se como ferramenta de teste de mutação voltada para a indústria, mas sendo utilizada em diversas pesquisas acadêmicas. Por realizar a mutação no nível de bytecode, a PIT torna o processo de criação e execução de mutantes mais rápido e menos custoso. A ferramenta implementa 28 operadores de mutação, os quais foram classificados em quatro categorias, assim como Barbosa, Maldonado e Vincenzi (2001) classificaram os operadores de mutação Proteum utilizados na linguagem C. As categorias de elementos sobre os quais atuam os operadores de mutação são: Variáveis (5 operadores, no total); Constantes (1 operador, no total); Operadores (9 operadores, no total); e Comandos (13 operadores, no total). Sua execução foi realizada por linhas de comando, conforme exemplificado na Figura 7. Entre os parâmetros, `mutators` indica o conjunto de operadores de mutação a serem aplicados, `timeoutFactor` estabelece o tempo limite para parar a execução de um teste (por exemplo, para evitar laços infinito), e os recursos permitem o acesso a informações de mutação relevantes, como o código mutante.

```
java -cp <dependency of project> org.pitest.mutationtest.commandline.
MutationCoverageReport --timestampedReports true --mutators ALL --sourceDirs
[<source code, test code>],packages=[program.*] --reportDir <result path>
--targetClasses program.* --targetTests <test case file> --verbose true
--outputFormats CSV --timeoutFactor 3 --features=+EXPORT
```

Figura 7 – Linha de comando - PIT.

Além das ferramentas Evosuite e PIT, foi implementada uma ferramenta auxiliar que tem a finalidade de estruturar os dados brutos gerados pelas demais ferramentas, para se ter um maior conjunto de informações relacionadas a cada um dos artefatos contidos na base. Além disso, busca otimizar sua preparação para execução em suas respectivas ferramentas e também para inclusão na base de dados. A ferramenta apresenta quatro funções básicas e necessárias para execução do processo de criação da base de conhecimento: elaboração de casos de testes; geração da linha de comando para PIT; criação da killing matrix; e inclusão da killing matrix na base de conhecimento relacional.

5.3 Procedimentos e Resultados

Resultados obtidos com o uso da Evosuite e PIT, assim como as informações serão extraídas da base de conhecimento, são apresentadas nesta seção. Para gerar os artefatos, seguiu-se um processo que é dividido em seis fases, possibilitando a evolução da base de conhecimento. Um exemplo de evolução refere-se à inserção de novas classes, casos de teste e mutantes. Cada uma dessas fases representa ações que foram analisadas, discutidas e devem ser realizadas para que as informações geradas ao final sejam completas e compatíveis com as informações já armazenadas na base de conhecimento. Ressalta-se que a definição e apresentação das fases se fazem necessárias para manter a integridade da base de conhecimento, para que a mesma seja capaz de armazenar artefatos de diferentes linguagens de programação.

Fase 1 – Execução da ferramenta Evosuite: A execução da Evosuite nos cinco projetos apresentados na Tabela 2 gerou um número substancial de casos de teste. A ferramenta foi configurada para usar todos os seus critérios de geração de casos de teste. O objetivo foi ter o conjunto mais completo de casos de teste possível para todas as classes de todos os projetos. Na Tabela 2 mostram-se os resultados quantitativos dessa execução. Nesta etapa, não houve preocupação em separar os casos de teste de acordo com os critérios disponíveis na ferramenta; isso ficou para trabalhos futuros.

Fase 2 – Preparação dos conjuntos de teste: classes de teste são geradas pela Evosuite para cada classe do código-fonte. Para facilitar o processo de identificação dos mutantes que são mortos por cada caso de teste, a ferramenta auxiliar mencionada na seção anterior separa cada caso de teste em uma classe específica. Este mesmo procedimento foi aplicado às classes de teste geradas pelos desenvolvedores, quando disponíveis.

Fase 3 – Definições das linhas de comando para executar a ferramenta PIT: Nesta etapa do processo, decidiu-se criar uma linha de comando de execução para cada caso de teste, apesar da possibilidade de executá-los todos com uma única linha de comando. Essa decisão está diretamente relacionada ao propósito de identificar quais casos de teste são capazes de matar cada um dos mutantes, pois esta é uma informação crucial para a obtenção do escore de mutação. Outro objetivo está ligado ao processo operacional de permitir a execução da próxima fase em várias estações de trabalho, agilizando a finalização do processamento.

Fase 4 – Execução da PIT: Cada execução da PIT gerou como resultado um arquivo CSV com informações sobre os mutantes gerados, a linha de código-fonte mutada, o operador de mutação utilizado, o caso de teste executado e as informações sobre o status do mutante, dentre as seguintes possibilidades: mortos, vivos, ou não cobertos. Um exemplo com conteúdo parcial deste arquivo é mostrado na Figura 8. No total, para cada caso de teste (seja um teste Evosuite ou um teste de desenvolvedor) foi produzido um arquivo CSV.

```

Bisect.java,program.Bisect,'..'.MathMutator,sqrt,41,KILLED,program.Bisect_ESTest.program.Bisect_ESTest
Bisect.java,program.Bisect,'..'.MathMutator,sqrt,49,TIMED_OUT,none
Bisect.java,program.Bisect,'..'.RemoveConditionalMutator_ORDER_IF,sqrt,42,SURVIVED,none
Bisect.java,program.Bisect,'..'.ReturnValsMutator,sqrt,56,NO_COVERAGE,none
...

```

Figura 8 – Exemplo de um arquivo CSV que representa os resultados de execução da PIT.

Na Tabela 3 apresentam-se o número de mutantes gerados para cada um dos projetos e os respectivos escores de mutação. Vale ressaltar que todos os operadores de mutação foram usados para gerar mutantes.

Tabela 3 – Resultados da PIT.

Aplicação	# Mutantes	Escore de Mutação
commons-lang3	40,301	64.52%
commons-math3	60,697	69.79%
javassist	44,461	64.54%
joda-time	46,999	68.54%
simplePrograms	1,888	68.70%
Total	194,346	—

Fase 5 – Geração da killing matrix: Nesta fase, a ferramenta auxiliar mescla todos os arquivos CSV gerados em um único arquivo, obtendo assim a killing matrix para as classes consideradas. A ferramenta também melhora a apresentação das informações da seguinte forma: o ponto-e-vírgula é usado para separar as informações; remoção de redundâncias (por exemplo, informações de mutação que aparecem em todos os arquivos), e uma nova representação para o status dos mutantes, a saber: Y para KILLED, OUT para TIMED_OUT, N para SURVIVED e N_C para NO_COVERED.

Na Figura 9 mostra-se um exemplo parcial desse arquivo. A primeira linha representa o cabeçalho. As linhas restantes representam informações de mutantes individuais, incluindo os resultados da execução do caso de teste (por exemplo, a figura mostra os resultados do caso de teste Bisect_ESTest).

```

Mutation Operator      ;Function;line;TC_1;TC_2;TC_3;'..';Bisect_ESTest; .....
InlineConstantMutator  ;toString;77  ;N_C ;N_C ;N_C ;'..';Y          ; .....
MathMutator            ;toString;76  ;N_C ;N_C ;N_C ;'..';N_C        ; .....
NegateConditionalsMutator;toString;74  ;N_C ;N_C ;N_C ;'..';Y          ; .....
NonVoidMethodCallMutator ;toString;76  ;N_C ;N_C ;N_C ;'..';OUT        ; .....
...

```

Figura 9 – Exemplo de um arquivo CSV que representa a killing matrix.

Fase 6 – Inclusão das informações na base de conhecimento: Nesta última fase do processo, todos os artefatos gerados foram incluídos no banco de dados relacional definido.

A inclusão também é tratada pela ferramenta auxiliar, a qual executa todo o processo de inclusão de dados na seguinte ordem: (1) Inclusão de todas as classes pertencentes ao projeto visado; (2) Inclusão de todos os operadores de mutação utilizados; (3) Inclusão da ferramenta que gerou os mutantes (por exemplo: PIT); (4) Inclusão de todos os mutantes gerados pela ferramenta. Neste momento, o bytecode mutante e o relacionamento com a classe que gerou o mutante também são incluídos no banco de dados; (5) Inclusão de todos os casos de teste gerados pela Evosuite, bem como casos de teste dos desenvolvedores. Isso também inclui-se o código-fonte da classe que contém o caso de teste e o relacionamento com a classe que gerou o caso de teste; e (6) Inclusão da relação entre o caso de teste e o mutante quando o caso de teste executa o mutante. Morto e vivo são status de mutantes considerados neste momento.

No que diz respeito à extração das informações da base de conhecimento, a mesma fornece acesso total ao código-fonte original das classes ou um conjunto de classes de escolha do usuário, uma vez que o código de todas as classes são armazenadas no SGBD. O usuário poderá realizar vários cálculos relacionados a: escore de mutação de classes ou conjunto de classes, escore de mutação para operadores de mutação, e assim por diante. Também é possível, sem a reexecução de todos os casos de teste e mutantes, fazer comparações entre as diversas técnicas de redução de custo de mutação.

5.4 Resumo das Contribuições e Limitações

Introduziu-se neste capítulo um novo conceito de base de conhecimento para padronização geral de artefatos que podem ser utilizados para verificação e validação de propostas de pesquisa no domínio da qualidade e teste de software, principalmente no domínio do teste de mutação. A base de conhecimento gerada pode ser facilmente compartilhada, permitindo assim uma fácil reprodução das pesquisas que a utilizaram, além da comparação de resultados, e conferindo maior confiabilidade aos mesmos, assim como permitindo inúmeras configurações e o uso de parâmetros de seleção de artefatos.

A versão atual da base de conhecimento possui algumas limitações relacionadas à sua estrutura e a alguns artefatos incluídos. Por exemplo, a base permite livre acesso a todas as classes, seus mutantes e casos de teste; entretanto, não provê acesso ao projeto completo que inclui os arquivos Java que contêm as classes inseridas na base. Com relação aos mutantes, a base possui apenas o arquivo de bytecode do mutante. Ainda em relação aos mutantes, os equivalentes não foram identificados, mas sua estrutura está preparada para esta identificação por meio das tabelas `mutants` e `mutants_testcases`.

Como trabalhos futuros, pretende-se incluir novos artefatos, assim fornecendo aos pesquisadores um conjunto maior de material experimental. Também se planeja avaliar se os artefatos listados atendem às necessidades de sub domínio de mutação específicos

como, por exemplo, redução de conjuntos de testes usando mutação, determinação de operadores essenciais, e técnicas de otimização para detectar mutantes equivalentes.

Capítulo 6

Experimentos

Neste capítulo apresentam-se e analisam-se os resultados obtidos com experimentos realizados com o framework SiMut. Deste modo, o objetivo é detalhar cada etapa utilizada na experimentação, iniciando-se pelo seu planejamento, definindo os objetivos do estudo, e descrevendo a execução e os processos para a realização da análise dos resultados. Ao final, derivam-se conclusões sobre a relevância de se empregar conceitos e técnicas de similaridade entre programas no contexto de redução do custo do teste de mutação.

Especificamente, este capítulo está organizado da seguinte forma: na seção 6.1 apresenta-se como foi realizado o planejamento dos experimentos, incluindo o objetivo dos mesmos, as questões de pesquisa que os experimentos pretendem responder, os artefatos que foram utilizados, e o modelo utilizado para a interpretação dos resultados obtidos; na seção 6.2 descreve-se o processo utilizado para aplicação da técnica de redução de custo do teste de mutação (em particular, o Procedimento Essencial (BARBOSA; MALDONADO; VINCENZI, 2001)); na seção 6.3 são apresentados os resultados obtidos e suas análises; na seção 6.4 são apresentadas as conclusões preliminares; na seção 6.5 é realizada uma análise estatística mais aprofundada dos resultados obtidos como forma de dar mais credibilidade aos resultados alcançados; e, por fim, na seção 6.6 são apresentadas as considerações finais, que são revisitadas no Capítulo 7.

6.1 Planejamento dos Experimentos

Para a geração dos resultados, foram definidas quatro configurações experimentais. A primeira configuração experimental, que foi definida no escopo de trabalho de Guarnieri, Pizzoleto e Ferrari (2022), considera, para cada configuração do *SiMut*,¹ um grupo

¹ Detalhes sobre as variantes do *SiMut* são apresentados no Capítulo 4 deste documento.

com os quatro melhores operadores de mutação (com base nos escores de mutação de cada operador para os programas considerados no experimento). Na segunda configuração experimental, trabalhou-se com a mesma base de artefatos utilizada por Guarnieri (2022), mas fazendo uso da técnica de Conjunto Essenciais de operadores de mutação (BARBOSA; MALDONADO; VINCENZI, 2001). Nas duas configurações experimentais remanescentes, realizou-se a mudança do conjunto de artefatos, na qual foram utilizados 221 arquivos de código fonte Java, cada um contendo uma ou mais classes, para os quais usou-se a análise dos quatro melhores operadores e a técnica de Conjunto Essenciais. Ressalta-se que, embora a primeira configuração experimental tenha sido definida e executada no contexto do trabalho de Guarnieri, Pizzoleto e Ferrari (2022), a mesma foi revisitada neste trabalho para permitir a realização de comparações e análises associadas aos resultados obtidos em todos os experimentos.

6.1.1 Objetivos

De forma geral, os experimentos descritos neste capítulo visaram a analisar possíveis benefícios que diferentes técnicas de abstrações e técnicas de similaridade entre programas podem trazer para a redução do custo do teste de mutação, particularmente quando se visa a aplicar o teste de mutação com custo reduzido em um programa ainda não testado. Variações do tamanho do grupo de referência foram consideradas para permitir a avaliação do impacto que os mesmos poderiam ter sobre os resultados obtidos, considerando tanto os operadores essenciais identificados quanto os melhores operadores em termos de escore de mutação.

Ressalta-se que diversas configurações do *SiMut* foram exploradas e executadas (20, no total), com o intuito de responder as questões de pesquisas definidas e apresentadas a seguir. A lista das configurações é apresentado na Tabela 4. A execução de cada configuração foi seguida pela aplicação das duas técnicas de redução de custo consideradas neste trabalho: melhores operadores, e Operadores Essenciais.

6.1.2 Questões de Pesquisa

Buscando sempre atender aos critérios de pesquisa, para este experimento definiram-se algumas questões de pesquisa que foram respondidas, objetivando validar o uso de similaridade de programas como forma de apoio à redução dos custos de execução do teste de mutação. Ressalta-se que algumas questões de pesquisa pertinentes já foram tratadas e respondidas por (GUARNIERI, 2022). Para os experimentos descritos neste capítulo, as seguintes questões de pesquisa foram definidas:

- **RQ1** - *A similaridade entre programas, calculada com base em abstrações dos mesmos, pode auxiliar na identificação de parâmetros de redução de custo do teste de mutação que sejam consistentes entre programas já testados e programas ainda não testados?*

Tabela 4 – Lista de configurações selecionadas para os experimentos.

Abstração	Similaridade
Todas as métricas	EM
Todas as métricas	X-Means - Euclidean
Todas as métricas	X-Means - Manhattan
Métricas selecionadas	EM
Métricas selecionadas	X-Means - Euclidean
Métricas selecionadas	X-Means - Manhattan
Código processado	mdist - Jaccard
Código processado	mdist - Levenshtein
Código processado	mdist - NCD
Código processado	Weka - X-Means - Euclidean
Código processado	Weka - X-Means - Manhattan
Código processado	Weka - EM
Código processado	JPlag
Código original	mdist - Jaccard
Código original	mdist - Levenshtein
Código original	mdist - NCD
Código original	Weka - X-Means - Euclidean
Código original	Weka - X-Means - Manhattan
Código original	Weka - EM
Código original	JPlag

Fonte: Guarnieri, Pizzoleto e Ferrari (2022)

- **RQ1.1** - *A similaridade entre programas, calculada com base em abstrações dos mesmos, pode auxiliar na identificação melhores operadores de mutação (com base nos escore de mutação dos mesmos) para novos programas (ainda não testados)?*
- **RQ1.2** - *A similaridade entre programas, calculada com base em abstrações dos mesmos, pode auxiliar na identificação de operadores essenciais de mutação para novos programas (ainda não testados)?*
- **RQ2** - *A diminuição do tamanho, quanto à quantidade de classes, do grupo de referência impacta na classificação e definição de Melhores Operadores ou Operadores Essenciais?*

6.1.3 Artefatos

Para este trabalho utilizou-se a base de dados (caracterizada como uma base de conhecimento) desenvolvida por Pizzoleto, Guarnieri e Ferrari (2021) e apresentada nesta tese no Capítulo 5. A base tem como proposta servir de insumo para estudos envolvendo teste de mutação. A mesma contém cinco projetos cadastrados e possui aproximadamente 2.000 classes, 50.000 casos de teste e 195.000 mutantes para esses projetos. Para os experimentos deste trabalho, utilizaram-se dois projetos, com os seus respectivos casos de teste e mutantes, denominados *simplePrograms* e *Javassist*, respectivamente.

Na base de dados, os casos de teste foram gerados pela ferramenta EvoSuite² na versão 1.0.6. Já os mutantes foram gerados e executados pela ferramenta PIT³ na versão 1.6.2. As duas ferramentas são amplamente utilizadas para experimentação em pesquisa de teste de software. A base inclui a relação completa de todos os mutantes de cada projeto, indicando o estado de cada um deles como: mutante morto, mutante vivo e mutante não coberto pelo caso de teste. Deste modo, a base também fornece informações de escore de mutação por operador de mutação para cada um dos projetos.

Na Tabela 2 (Capítulo 5) mostram-se as métricas descritivas dos programas e casos de teste. A tabela é dividida em três partes: (1) Código da Aplicação, que diz respeito ao código-fonte da aplicação, (2) Código de Teste dos Desenvolvedores, e (3) Código de Teste EvoSuite, que diz respeito aos testes gerados pela EvoSuite. Na tabela, TLOC é o número total de linhas do código, CC é a média da complexidade ciclomática (McCabe, 1976) dos métodos das respectivas classes, e TCs representa o total de casos de teste gerados para a classe. As métricas do código da aplicação foram calculadas usando-se o *plugin* Eclipse Metrics⁴ na versão 1.3.8.

Na Tabela 3 (também apresentada no Capítulo 5) apresentam-se os totais de mutantes gerados para cada projeto, e os respectivos escores de mutação produzidos pelos casos de teste (somando-se os testes dos desenvolvedores e os testes gerados pela Evosuite.)

Conforme relatado por Pizzoleto, Guarnieri e Ferrari (2021), a EvoSuite foi configurada para aplicar o conjunto completo de critérios de geração de cobertura de teste (ROJAS et al., 2015), enquanto a PIT foi configurada para aplicar o conjunto completo de 28 operadores de mutação. Para este trabalho, o projeto *Javassist* utilizado dispõe de casos de teste desenvolvidos pelos desenvolvedores, que também foram utilizados no experimento, mas em nenhum momento foram executados procedimentos para identificar mutantes equivalentes. Ao adotar os resultados produzidos pelas ferramentas de geração de testes e de geração e execução de mutantes, em conjunto com os casos de teste dos desenvolvedores, buscou-se dar uma conotação mais realista aos experimentos e resultados obtidos, tendo em vista um cenário de completa automatização do processo como um todo, sem envolver tarefas que são manualmente executadas (as quais são naturalmente propensas a erros).

Nota-se que, do conjunto completo de operadores de mutação da PIT, apenas 17 operadores de mutação produziram mutantes para os programas selecionados para o experimento. Na Tabela 6, mostrada mais adiante neste capítulo, listam-se todos os operadores da PIT, apresentando-se o nome e a descrição de cada um deles.

² <<https://www.evosuite.org/>> - acessado em dezembro de 2022.

³ <<http://pitest.org/>> - acessado em dezembro de 2022.

⁴ <<http://metrics2.sourceforge.net/>> acessado em dezembro de 2022

6.1.4 Modelo de Interpretação dos Resultados

Para fins experimentais, cada programa da base de dados foi considerado um programa não testado u . Com isso, foi possível simular um cenário de caso real em que uma nova classe é fornecida ao *SiMut*, e o framework ajuda o testador a aplicar testes de mutação a um custo reduzido. Como no banco de dados existem os artefatos de mutação relacionados a u , pode-se avaliar os resultados caso os mesmos parâmetros de redução de custo sejam aplicados a u .

Para analisar os resultados obtidos e identificar as configurações que podem trazer mais benefícios, foi empregado o modelo de interpretação usado por Guarnieri (2022), com poucas adaptações para contemplar as duas técnicas de redução de custo investigadas neste trabalho. Para cada programa e para cada configuração do *SiMut*, uma tabela de resultado foi criada para armazenar e listar os melhores operadores de mutação, ou os operadores essenciais. Na Tabela 5 exemplificam-se os resultados de uma das iterações do experimento. Os elementos mostrados são definidos da seguinte forma:

Tabela 5 – Trecho dos resultados detalhados, adaptada do trabalho de Guarnieri (2022).

Configuração - Código original / mdist - Levenshtein	
u	numZero
O_u	Constructor_Call, Remove_Conditionals, Return_Vals, Increments, Remove_Increments
OC_u	Constructor_Call, Return_Vals, Increments, Remove_Increments, Conditionals_Boundary
OP_u	Constructor_Call, Increments, Remove_Increments, Non_Void_Method_Call
OC_1	Math, Constructor_Call, Increments, Return_Vals
OC_2	Constructor_Call, Increments, Remove_Increments, Non_Void_Method_Call
R1	V: $IC_u(4) > m(IC_1(3), IC_2(3))$
R2	V: $IC_u(4) > IP_u(3)$
$ORan_1$	Constructor_Call, Increments, Remove_Increments, Math
$ORan_2$	Constructor_Call, Non_Void_Method_Call, Negate_Conditionals, Increments
$ORan_3$	Constructor_Call, Experimental_Member_Variable, Return_Vals, Non_Void_Method_Call
$ORan_4$	Constructor_Call, Increments, Remove_Increments, Experimental_Member_Variable
$ORan_5$	Constructor_Call, Remove_Increments, Increments, Math
$ORan_6$	Constructor_Call, Increments, Math, Return_Vals
$ORan_7$	Constructor_Call, Increments, Return_Vals, Remove_Increments
$ORan_8$	Constructor_Call, Increments, Remove_Increments, Return_Vals
$ORan_9$	Constructor_Call, Remove_Increments, Increments, Remove_Conditionals
$ORan_{10}$	Constructor_Call, Remove_Conditionals, Increments, Math
$IRanAvg_u$	3
R3	V: $IC_u(4) > IRanAvg_u(3)$

Fonte: Guarnieri, Pizzoleto e Ferrari (2022)

- u : representa o programa considerado não testado em uma iteração.
- C : representa um *cluster*. Em particular,
 - C_u é o *cluster* a qual u está associado (exceto u); e
 - C_i ($i=1..4$) é um dos *clusters* restantes.
- P_u : representa o conjunto completo de programas considerados para o experimento, exceto u .

- Ran_i : representa os *clusters* formados aleatoriamente (10 *clusters* aleatórios, no total), exceto u .
- O : representa uma lista de melhores operadores, ou operadores essenciais. Em particular,
 - O_u é a lista para u .
 - OC_u é a lista para C_u .
 - OC_i é a lista para C_i .
 - $ORan_i$ ($i=1..10$) é a lista para Ran_i .
 - e OP_u é a lista para P_u
- I : representa o tamanho da interseção entre uma lista de melhores operadores, ou operadores essenciais, e a lista de melhores operadores, ou operadores essenciais para u . Em particular,
 - IC_u é o tamanho da interseção entre O_u e OC_u
 - IP_u é o tamanho da interseção entre O_u e OP_u
 - $IRanAvg_u$ é a média do tamanho da interseção de O_u com todos os $ORan_i$ ($i = 1..10$)
 - IC_i ($i=1..4$) é o tamanho da interseção entre O_u e um OC_i particular.
- m : representa uma função que retorna o maior valor entre um grupo de valores.

Os valores numéricos entre parênteses que aparecem na Tabela 5 representam o tamanho da interseção. $R1$, $R2$ e $R3$, definidos na sequência, podem ter um resultado positivo (V), neutro (N), ou negativo (X).

- $R1$: compara os resultados obtidos para C_u com os outros *clusters* formados. O resultado é positivo se IC_u for maior que todos os outros resultados para IC_i ; negativo se IC_u for menor que qualquer outro resultado para IC_i ; ou neutro se o resultado for o mesmo em todos os casos.
- $R2$: similarmente a $R1$, $R2$ compara os resultados obtidos para C_u com o conjunto completo de programas, exceto u . Resultado positivo, neutro e negativo são igualmente computados como feito para $R1$.
- $R3$: similarmente a $R1$ e $R2$, $R3$ compara os resultados obtidos para C_u com a média dos resultados obtidos para os *clusters* aleatórios.

Na Tabela 5 listam-se os resultados para uma iteração de uma configuração do *SiMut* na qual consideraram-se os quatro melhores operadores, ou mais operadores em caso de empate de escore entre eles, para cada grupo de programas. Para o programa `numZero` (apresentado na primeira linha da tabela), os resultados são:

- O_u tem 5 operadores: *Constructor Call Mutator*, *Remove Conditional Mutator*, *_ORDER_ELSE*, *Return Vals Mutator*, *Increments Mutator* e *Remove Increments Mutator*
- OC_u tem 5 operadores: *Constructor Call Mutator*, *Return Vals Mutator*, *Increments Mutator*, *Remove Increments Mutator* e *Conditionals Boundary Mutator*
- OP_u tem 4 operadores: *Constructor Call Mutator*, *Increments Mutator*, *Remove Increments Mutator* e *Non Void Method Call Mutator*
- $R1$: tem resultado positivo, pois IC_u é maior que $m(IC_1, IC_2)$.
- $R2$: tem resultado positivo, pois IC_u é maior que IP_u .
- $IRanAvg_u$: é igual 3 pois esse é o resultado da média de todas as interseções entre O_u com todos os $ORan_i$ ($i=1..10$).
- $R3$: tem um resultado positivo, pois IC_u é maior que $IRanAvg_u$.

Este processo foi executado para todas as classes em todas as configurações mencionadas na seção anterior. Nas próximas seções descrevem-se a aplicação da técnica de redução de custo Operadores Essenciais (BARBOSA; MALDONADO; VINCENZI, 2001), e os resultados obtidos para todos os experimentos realizados.

6.2 Aplicação da Técnica de Redução de Custo

Após a formação dos grupos de programas mais similares, realizado pelo *SiMut*, a próxima etapa é a aplicação da técnica de redução de custo de mutação. Para esta pesquisa, duas técnicas foram exploradas. A primeira, aqui referenciada com *Melhores Operadores*, faz uso dos quatro melhores operadores (primeiros classificados quanto aos escores de mutação), e a segunda determina o *Conjunto Essencial* de operadores, conforme proposto por (BARBOSA; MALDONADO; VINCENZI, 2001).

Para a técnica *Melhores Operadores*, que foi originalmente apresentada por (GUARNIERI, 2022), foi desenvolvido um módulo implementado em linguagem C# (detalhes são apresentados no Capítulo 4) que é responsável por computar os processos relacionados à redução de custo do teste de mutação. Em particular, para a obtenção dos escores de mutação, os mesmos foram calculados extraindo-se as informações da base de conhecimento, sendo que o módulo *CostReduction* dinamicamente cria a *killing matrix* que abrange mutantes e casos de teste para todas as classes utilizadas na execução do

SiMut. Todos os resultados gerados pelo módulo são armazenados em arquivos CSV. Deste modo, para cada execução realizada através do *SiMut*, um arquivo CSV é gerado. Esse arquivo pode então ser interpretado e processado por outras ferramentas. Para este trabalho, foi desenvolvido um *script* em NodeJS que interpreta os resultados gerados pelo módulo *CostReduction*, onde é possível filtrar os N melhores operadores de cada grupo de programas.

Já para a técnica Operadores Essenciais, o módulo *CostReduction* foi estendido para incorporar a série de passos definida para o Procedimento Essencial (BARBOSA; MALDONADO; VINCENZI, 2001). A descrição dos passos, que foi revisitada neste trabalho, é apresentada a seguir.

6.2.1 Implementação da Técnica de Conjunto Essenciais

Este módulo, implementado em C# e baseado no procedimento proposto por (BARBOSA; MALDONADO; VINCENZI, 2001), faz uso das informações processadas pelo *SiMut* e também da base de conhecimento, já que todas as informações se encontram lá, e segue na íntegra os seis passos de definição do Conjunto Essencial, adicionada de outras atividades necessárias para o módulo executar precisamente as análises e processo. Vale ressaltar que a execução de todas as fases descritas a seguir foi totalmente automatizada; isso só se tornou possível devido às informações estarem contidas na base. Na sequência descrevem-se cada uma das fases implementadas.

Fase 1 – Definição do Projeto que será analisado: Nesta fase é informado para o módulo *CostReduction* qual o projeto a ser analisado. Neste momento, as informações de *clusters* e classes são preparadas e armazenadas para a execução. Os próximos passos se repetirão até que todas as composições de abstrações e similaridades sejam executadas.

Fase 2 – Definição do Conjunto Essencial de Operadores: Neste momento, passa-se a analisar e definir o conjunto de operadores. Ressalta-se que são calculados os conjuntos essenciais de operadores para cada uma das classe, para o conjunto total das classes, e dos *clusters* gerados pelo *SiMut*. Também é calculado o Conjunto Essencial para os *clusters* aleatoriamente formados. O cálculo dos operadores essenciais segue os seis passos detalhados a seguir.

Passo 01: Selecionar operadores que determinam um alto escore de mutação: Aqui são selecionados os operadores com melhores escores de mutação, para se garantir que o conjunto essencial de operadores possua um alto escore de mutação. A seleção dos operadores é feita a partir de um índice médio de score de mutação (*IMES*), que deve ser definido de acordo com as características do conjunto de programas analisados. Para esta execução, considerou-se um *IMES* igual a 56,35%, dadas as características do conjunto de programas e seus escores de mutação. Este valor foi computado dinamicamente por um método, desenvolvido pelo autor, que analisa as característica dos artefatos e os

operadores de mutação para cada categoria definindo um valor médio aceitável de escore de mutação para ser utilizado. Todos os operadores selecionados apresentam escores de mutação igual ou superior ao *IMES* definido. Por ser um valor definido pelo usuário, esta informação é atribuída ao método como parâmetro, o qual pode ser facilmente modificado se necessário. Ressalta-se que o *IMES* utilizado é um valor inferior ao usado no trabalho de Barbosa, Maldonado e Vincenzi (2001) devido as características dos artefatos com compõem a base.

Passo 02: Selecionar operadores de cada classe de mutação: Considerando que cada classe de operadores de mutação simula grupos de defeitos específicos, é desejável que o conjunto de operadores contenha pelo menos um operador de cada classe. Para isso, os operadores de mutação incluídos da PIT foram classificados, como apresentados na Tabela 6. Esta classificação é aplicada automaticamente pelo módulo e gravado em memória.

Passo 03: Avaliar a inclusão empírica entre os operadores: Neste momento do processo é feita uma análise que verifica se há operadores que foram incluídos empiricamente por outros operadores; os operadores incluídos empiricamente são removidos do conjunto essencial. Esta operação se repete até que todos os operadores incluídos empiricamente por outros operadores sejam removidos.

Passo 04: Estabelecer uma estratégia incremental de aplicação: Neste passo, para análise do conjunto essencial, classificam-se os operadores tendo como base o custo dos mesmos. Os operadores são classificados usando uma ordem crescente de menor custo de execução para o de maior custo.

Passo 05: Selecionar operadores que proporcionem incrementos no escore de mutação: Neste passo a ideia é analisar o conjunto de operadores já incluídos com os ainda não incluídos e que não foram excluídos em passos anteriores, verificando se sua adição melhora incrementalmente o escore de mutação do conjunto essencial. Se o critério for atendido, o operador é incluído.

Passo 06: Selecionar operadores com alto *strength*: Neste passo, faz-se uma nova análise com os demais operadores, mas agora considerando seu *strength* (operadores com maior dificuldade de satisfação, já que possuem um menor escore de mutação médio quando executados com testes adequados para mutantes gerados por outros operadores) ao conjunto total de operadores. Estes operadores são pouco incluídos empiricamente pelos demais operadores. A inclusão destes operadores é feita a partir de um índice médio de *strength* (*IMS*), que também é definido seguindo as características do conjunto de programas utilizados. Para este estudo, utilizou-se o valor de **0,300 com variação para mais ou menos de 0,005**, que calcula a média dos *strengths* em relação aos operadores selecionados, determinando se algum novo operador será inserido ao conjunto. A cada novo operador incluído, o cálculo se repete até que o processo seja concluído.

Tabela 6 – Categorização dos operadores da PIT.

Categorias	Operadores	Descrição
Comandos	CONSTRUCTOR_CALLS	Realiza a substituição nas chamadas dos constructors criando um objeto do tipo Null. Obs.: Só afeta o método Constructor
Comandos	EMPTY_RETURNS	Realiza a substituição de o retorno por um valor vazio. Obs.: A Pit automaticamente realiza um filtro para evitar a criação de mutantes equivalente em métodos que já retornam um valor vazio.
Comandos	EXPERIMENTAL_ARGUMENT_PROPAGATION	Realiza a substituição da chamada de um método por um de seus parâmetros correspondentes.
Comandos	EXPERIMENTAL_BIG_INTEGER	Realiza a substituição de métodos de numeros inteiro grandes (Big_Integer)
Comandos	EXPERIMENTAL_NAKED_RECEIVER	Realiza a substituição do método por uma simples chamada.
Comandos	EXPERIMENTAL_SWITCH	Realiza a substituição do primeiro rótulo de um switch que seja diferente do padrão.
Comandos	FALSE_RETURNS	Realiza a substituição dos retorno lógicos para FALSE. Obs.: Métodos que já retornam FALSE não serão afetados, evitando assim a criação de mutantes equivalentes.
Comandos	NON_VOID_METHOD_CALLS	Realiza a substituição das chamadas de métodos que retornam valores não nulos, atribuindo aos mesmos os valores default do Java, de acordo com o tipo de retorno.
Comandos	NULL_RETURNS	Realiza a substituição dos valores de retorno para NULLs. Métodos que são anotados diretamente com NotNull não sofrerão mutação e que podem sobre mutação pelo EMPTY_RETURNS não sofrem mutação.
Comandos	PRIMITIVE_RETURNS	Realiza a substituição de todos valores de retorno restantes por '0'. Métodos que já retornarem '0' não sofrem mutação.
Comandos	REMOVE_CONDITIONALS	Realiza a substituição de todas as instruções condicionais das estruturas de decisões para que sempre sejam executados. Faz a substituição para valores lógicos.
Comandos	TRUE_RETURNS	Realiza a substituição dos retorno lógicos para TRUE. Obs.: Métodos que já retornam TRUE não serão afetados, evitando assim a criação de mutantes equivalentes.
Comandos	VOID_METHOD_CALLS	Realiza a remoção das chamadas de métodos que não possuem retorno.
Constantes	INLINE_CONSTS	Realiza a substituição dos valores das constantes. Obs.: Pela Pit trabalhar com o bytecode, pode ser que o compilador otimize o uso das constantes e o operador de mutação na gere nenhuma mutação. Há algumas regras para definir as mutações e como o valor da contante será alterado.
Operadores	AOD	Realiza a substituição e/ou exclusão de membros contidos em uma operação aritmética.
Operadores	AOR	Realiza a substituição de operadores aritméticos por um outro operador.
Operadores	OBBN	Realiza a substituição de operadores lógicos "e" e "ou" invertendo estes operadores.
Operadores	CONDITIONALS_BOUNDARY	Realiza a substituição de Operadores Relacionais/Condicionais
Operadores	MATH	Realiza a Substituição de Operadores Aritméticos. Mas também influência em concatenação de Strings que utilizam o Operador "+". Obs.: Realiza mutações em variáveis com os Operadores de Incrementos e Decrementos
Operadores	NEGATE_CONDITIONALS	Realiza a negação em Operadores Condicionais
Operadores	ROR	Realiza a substituição de Operadores Relacionais/Condicionais
Operadores	REMOVE_INCREMENTS	Realiza a remoção dos incrementos de variáveis locais.
Operadores	UOI	Realiza a inclusão de operadores unários (incrementos e decrementos) na chamada das variáveis.
Variáveis	CRCR	Realiza mudanças nas constantes inline, atualizando seus valores.
Variáveis	EXPERIMENTAL_MEMBER_VARIABLE	Realiza a substituição de valores de variáveis por seus valores iniciais padrão do Java, isso para cada tipo específico.
Variáveis	RETURN_VALS	Realiza alteração de valores nos retornos de métodos
Variáveis	INCREMENTS	Realiza a inversão da atribuição de incrementos e decrementos. Se a variável recebe a opção de incremento o mesmo será substituído para um decremento e vice-versa. Obs.: Só gera a mutação para variáveis locais.
Variáveis	INVERT_NEGS	Inverte a negação numérica (Inteiros e Flutuantes). A negação da variável será removida, tornando-a positiva.
Variáveis	ABS	Realiza a substituição de todas as variáveis numéricas por uma negação a seu valor atribuído.

Fonte: Elaborada pelo autor.

Fase 3 – Geração dos arquivos CSVs: Nesta fase, ocorre a geração dos arquivos CSV com todas as informações computadas do conjunto essencial e seus respectivos escores de mutação. Deste modo, para cada execução realizada através do *SiMut*, um arquivo CSV é gerado. Esse arquivo pode então ser interpretado e processado por outras ferramentas. Para este trabalho foi desenvolvido um *script* em NodeJS que interpreta os resultados gerados pelo módulo *CostReduction*, o qual recupera os operadores essenciais identificados para cada grupo de programas. Como pode ser observado, nesta última fase tem-se o mesmo padrão de informações e o mesmo processo de execução é gerado.

Na seção que segue os resultados obtidos com a execução de todos os experimentos são apresentados e discutidos.

6.3 Resultados e Análise

Nesta seção apresentam-se os resultados obtidos da execução dos experimentos e uma análise desses resultados. O objetivo inicial foi considerar a comparação entre os operadores com melhores escores de mutação e o conjunto essencial de operadores identificados seguindo a execução normal do *SiMut*. Após realizada uma primeira análise, constatou-se que, devido ao aumento substancial da quantidade de classe do projeto, o grupo de referência ficaria com uma grande quantidade de classes, podendo chegar a 1/3 da quantidade total das classes consideradas na execução do experimento. Dessa forma, decidiu-se por gerar outros dois conjuntos de execução, nos quais o tamanho do grupo de referência teria o seu tamanho (isto é, quantidade de classes contidas) gerenciado e definido para aproximadamente 5% e 10% do total de classes consideradas no experimento.

Na sequência, apresentam-se os resultados obtidos, com as execuções, e suas análises.

6.3.1 Resultados e Análise: Melhores Operadores para SimplePrograms

Para que se tenha uma base inicial de comparação dos resultados dos experimentos realizados, inicialmente revisitam-se os resultados apresentados por Guarnieri, Pizzoleto e Ferrari (2022), especificamente para o experimento que identificou os quatro melhores operadores para o grupo de programas considerado (isto é, o projeto SimplePrograms).

6.3.1.1 Análise dos Resultados

Segundo Guarnieri, Pizzoleto e Ferrari (2022), na execução de seus experimentos, cujo foco seria considerar apenas o melhor operador para cada grupo de programas, constatou-se que os dados foram inconclusivos e consequentemente se optou por expandir o grupo de melhores operadores para a quantidade de quatro operadores, em média, permitindo assim um melhor equilíbrio nos resultados.

Com intuito de apresentar os resultados obtidos por Guarnieri, Pizzoleto e Ferrari (2022), na Tabela 7 ilustram-se os resultados sumarizados considerando os quatro melhores operadores de mutação para a configuração executada do *SiMut*. Para complementar as informações, na Tabela 8 é apresentado um ranqueamento onde os valores de R1, R2 e R3 são sumarizados e podendo variar de 1 a 35. Destaca-se que os valores nas colunas R1 e R2 são a soma dos resultados positivos e neutros; já na coluna R3 somente os resultados positivos são apresentados, e na coluna T apresenta-se a soma dos resultados de R1, R2 e R3. Cada percentual entre parenteses representa a efetividade obtida dentro do intervalo.

Tabela 7 – Resumo dos resultados considerando os quatro melhores operadores (projeto SimplePrograms).

Métricas selecionadas / X-Means - Euclidean		Código processado / X-Means - Manhattan	
R1	V(19), N(3), X(13)	R1	V(17), N(1), X(17)
R2	V(16), N(13), X(6)	R2	V(12), N(13), X(10)
R3	V(21), N(3), X(11)	R3	V(15), N(2), X(18)
Todas as métricas / EM		Todas as métricas / X-Means - Euclidean	
R1	V(21), N(2), X(12)	R1	V(21), N(1), X(13)
R2	V(19), N(10), X(6)	R2	V(17), N(12), X(6)
R3	V(22), N(3), X(10)	R3	V(22), N(1), X(12)
Todas as métricas / X-Means - Manhattan		Métricas selecionadas / EM	
R1	V(21), N(0), X(14)	R1	V(21), N(4), X(10)
R2	V(17), N(12), X(6)	R2	V(17), N(14), X(4)
R3	V(21), N(1), X(13)	R3	V(23), N(3), X(9)
Métricas selecionadas / X-Means - Manhattan		Código original / JPlag	
R1	V(20), N(4), X(11)	R1	V(19), N(5), X(11)
R2	V(16), N(15), X(4)	R2	V(17), N(10), X(8)
R3	V(22), N(3), X(10)	R3	V(19), N(1), X(15)
Código original / mdist - Jaccard		Código original / mdist - Levenshtein	
R1	V(19), N(7), X(9)	R1	V(21), N(6), X(8)
R2	V(17), N(15), X(3)	R2	V(19), N(12), X(4)
R3	V(21), N(2), X(12)	R3	V(22), N(3), X(10)
Código original / EM		Código original / X-Means - Euclidean	
R1	V(21), N(2), X(12)	R1	V(18), N(2), X(15)
R2	V(16), N(11), X(8)	R2	V(11), N(22), X(2)
R3	V(19), N(2), X(14)	R3	V(20), N(4), X(11)
Código original / X-Means - Manhattan		Código processado / JPlag	
R1	V(19), N(1), X(15)	R1	V(17), N(5), X(13)
R2	V(13), N(20), X(2)	R2	V(16), N(11), X(8)
R3	V(22), N(3), X(10)	R3	V(17), N(3), X(15)
Código processado / mdist - Jaccard		Código processado / mdist - Levenshtein	
R1	V(23), N(6), X(6)	R1	V(22), N(4), X(9)
R2	V(20), N(12), X(3)	R2	V(22), N(7), X(6)
R3	V(27), N(0), X(8)	R3	V(24), N(0), X(11)
Código processado / EM		Código processado / X-Means - Euclidean	
R1	V(19), N(0), X(16)	R1	V(17), N(0), X(18)
R2	V(14), N(11), X(10)	R2	V(13), N(12), X(10)
R3	V(15), N(2), X(18)	R3	V(15), N(2), X(18)
Código processado / mdist - NCD		Código original / mdist - NCD	
R1	V(19), N(7), X(9)	R1	V(19), N(10), X(6)
R2	V(19), N(11), X(5)	R2	V(19), N(13), X(3)
R3	V(23), N(1), X(11)	R3	V(24), N(3), X(8)

Fonte: Versão adaptada de Guarnieri, Pizzoleto e Ferrari (2022)

Tabela 8 – Classificação dos resultados (R_i podendo variar de 1 a 35, Melhores Operadores, projeto SimplePrograms).

#	Configuração	R1	R2	R3	T
1	Código processado / mdist - Jaccard	29 (82,86%)	32 (91,43%)	27 (77,14%)	88 (83,81%)
2	Código original / mdist - NCD	29 (82,86%)	32 (91,43%)	24 (68,57%)	85 (80,95%)
3	Código original / mdist - Levenshtein	27 (77,14%)	31 (88,57%)	22 (62,86%)	80 (76,19%)
4	Métricas selecionadas / EM	25 (71,43%)	31 (88,57%)	23 (65,71%)	79 (75,24%)
5	Código original / mdist - Jaccard	26 (74,29%)	32 (91,43%)	21 (60,00%)	79 (75,24%)
6	Código processado / mdist - Levenshtein	26 (74,29%)	29 (82,86%)	24 (68,57%)	79 (75,24%)
7	Código processado / mdist - NCD	26 (74,29%)	30 (85,71%)	23 (65,71%)	79 (75,24%)
8	Métricas selecionadas / X-Means - Manhattan	24 (68,57%)	31 (88,57%)	22 (62,86%)	77 (73,33%)
9	Código original / X-Means - Manhattan	20 (57,14%)	33 (94,29%)	22 (62,86%)	75 (71,43%)
10	Todas as métricas / EM	23 (65,71%)	29 (82,86%)	22 (62,86%)	74 (70,48%)
11	Todas as métricas / X-Means - Euclidean	22 (62,86%)	29 (82,86%)	22 (62,86%)	73 (69,52%)
12	Código original / X-Means - Euclidean	20 (57,14%)	33 (94,29%)	20 (57,14%)	73 (69,52%)
13	Métricas selecionadas / X-Means - Euclidean	22 (62,86%)	29 (82,86%)	21 (60,00%)	72 (68,57%)
14	Todas as métricas / X-Means - Manhattan	21 (60,00%)	29 (82,86%)	21 (60,00%)	71 (67,62%)
15	Código original / jPlag	24 (68,57%)	27 (77,14%)	19 (54,29%)	70 (66,67%)
16	Código original / EM	23 (65,71%)	27 (77,14%)	19 (54,29%)	69 (65,71%)
17	Código processado / jPlag	22 (62,86%)	27 (77,14%)	17 (48,57%)	66 (62,86%)
18	Código processado / EM	19 (54,29%)	25 (71,43%)	15 (42,86%)	59 (56,19%)
19	Código processado / X-Means - Manhattan	18 (51,43%)	25 (71,43%)	15 (42,86%)	58 (55,24%)
20	Código processado / X-Means - Euclidean	17 (48,57%)	25 (71,43%)	15 (42,86%)	57 (54,29%)

Fonte: Versão adaptada de Guarnieri, Pizzoleto e Ferrari (2022)

Com base nesses resultados, Guarnieri, Pizzoleto e Ferrari (2022) realizaram dois tipos de análise. A primeira análise baseou-se nos resultados de R1, R2 e R3 conjuntamente, na qual se constatou que dentre as configurações executadas no *SiMut*, as que apresentam resultados promissores foram *Código processado / mdist - Jaccard* e *Código original / mdist - NCD*, respectivamente, somando 88 e 85 dos resultados positivos de um máximo de 105 possíveis. Destaque pode ser dado à configuração que combinou código processado pelo ProGuard e similaridade computada com a medida de distância Jaccard implementada pela ferramenta mdist gerando uma efetividade de aproximadamente 84% indicada pelos melhores operadores de mutação.

Já a segunda análise baseou-se em realizar a análise dos resultados de R1, R2 e R3 separadamente. Observou-se que para os resultados R1, algumas configurações tiveram efetividade acima de 75%, das quais todas usaram a ferramenta de similaridade mdist e com abstrações e medidas de distância variando: (i) *Código processado / mdist - Jaccard*, (ii) *Código original / mdist - NCD*, e (iii) *Código original / mdist - Levenshtein*. Considerando R2, exceto pelas últimas três configurações apresentadas na Tabela 7, todas as demais configurações apresentaram efetividade acima de 75%. Por fim, considerando R3, destaca-se que as melhores configurações quanto à efetividade variaram de 65% a 68%. Destaca-se que, para todas as melhores configurações, a ferramenta de similaridade mdist se sobressai.

6.3.2 Resultados e Análise: Conjuntos Essenciais para SimplePrograms

Com base no experimento executado por Guarnieri, Pizzoleto e Ferrari (2022), optou-se neste trabalho por explorar a técnica Operadores Essenciais como uma técnica alternativa de redução de custo, como descrito na subseção 6.2.1, que foi aplicada inicialmente sobre o mesmo conjunto de programas englobados no projeto SimplePrograms. Na Tabela 9 listam-se os resultados sumarizados referentes à definição dos Conjuntos Essenciais para cada configuração executada do *SiMut*. Os resultados são discutidos na sequência.

Tabela 9 – Resumo dos resultados considerando os operadores essenciais (projeto SimplePrograms).

Métricas selecionadas / X-Means - Euclidean		Código processado / X-Means - Manhattan	
R1	V(28), N(7), X(0)	R1	V(5), N(28), X(2)
R2	V(0), N(35), X(0)	R2	V(0), N(33), X(2)
R3	V(0), N(35), X(0)	R3	V(3), N(30), X(2)
Todas as métricas / EM		Todas as métricas / X-Means - Euclidean	
R1	V(0), N(35), X(0)	R1	V(7), N(28), X(0)
R2	V(0), N(35), X(0)	R2	V(0), N(35), X(0)
R3	V(4), N(31), X(0)	R3	V(6), N(29), X(0)
Todas as métricas / X-Means - Manhattan		Métricas selecionadas / EM	
R1	V(7), N(28), X(0)	R1	V(7), N(28), X(0)
R2	V(0), N(35), X(0)	R2	V(0), N(35), X(0)
R3	V(6), N(29), X(0)	R3	V(3), N(32), X(0)
Métricas selecionadas / X-Means - Manhattan		Código original / jPlag	
R1	V(7), N(28), X(0)	R1	V(0), N(35), X(0)
R2	V(0), N(35), X(0)	R2	V(0), N(35), X(0)
R3	V(2), N(33), X(0)	R3	V(0), N(35), X(0)
Código original / mdist - Jaccard		Código original / mdist - Levenshtein	
R1	V(0), N(35), X(0)	R1	V(0), N(35), X(0)
R2	V(0), N(35), X(0)	R2	V(0), N(35), X(0)
R3	V(0), N(35), X(0)	R3	V(0), N(35), X(0)
Código original / EM		Código original / X-Means - Euclidean	
R1	V(7), N(28), X(0)	R1	V(0), N(35), X(0)
R2	V(0), N(35), X(0)	R2	V(0), N(35), X(0)
R3	V(5), N(30), X(0)	R3	V(5), N(30), X(0)
Código original / X-Means - Manhattan		Código processado / jPlag	
R1	V(0), N(35), X(0)	R1	V(0), N(35), X(0)
R2	V(0), N(35), X(0)	R2	V(0), N(35), X(0)
R3	V(4), N(31), X(0)	R3	V(0), N(35), X(0)
Código processado / mdist - Jaccard		Código processado / mdist - Levenshtein	
R1	V(0), N(35), X(0)	R1	V(0), N(35), X(0)
R2	V(0), N(35), X(0)	R2	V(0), N(35), X(0)
R3	V(0), N(35), X(0)	R3	V(0), N(35), X(0)
Código processado / EM		Código processado / X-Means - Euclidean	
R1	V(5), N(28), X(2)	R1	V(5), N(28), X(2)
R2	V(0), N(33), X(2)	R2	V(0), N(33), X(2)
R3	V(3), N(30), X(2)	R3	V(3), N(30), X(2)
Código processado / mdist - NCD		Código original / mdist - NCD	
R1	V(0), N(35), X(0)	R1	V(0), N(35), X(0)
R2	V(0), N(35), X(0)	R2	V(0), N(35), X(0)
R3	V(0), N(35), X(0)	R3	V(0), N(35), X(0)

Fonte: Elaborada pelo autor.

6.3.2.1 Análise dos Resultados

Seguindo o mesmo processo de análise apresentado por Guarnieri, Pizzoleto e Ferrari (2022), para permitir a comparação de resultados. Como definido, ressalta-se que R1 avalia se C_u (ou seja, o *cluster* ao qual a classe u está associada) é um melhor (ou pelo menos tão bom quanto) preditor do escore de mutação do que os demais clusters C_i computados pelo *SiMut*.

Para tanto, considera-se que um resultado neutro pode ser satisfatório tanto para R1 quanto para R2, tendo em vista que C_u é um preditor de escore de mutação do operador de mutação tão bom quanto os demais clusters formados, e tão bom quanto o conjunto completo de programas, tendo em vista que computar o conjunto essencial de operadores para um único *cluster* é menos custoso do que para o conjunto de programas como um todo. Mas, para R3, os resultados classificados como neutro são claramente insatisfatórios, já que há um baixo custo para a formação dos *clusters* aleatoriamente, sem a necessidade de computar abstrações, medidas e similaridades.

Seguindo a lógica de interpretação e considerando os resultados gerados pelo Conjunto Essencial, na Tabela 10 é apresentado um ranqueamento em que os valores de R1, R2 e R3 podem variar de 1 a 35. Os valores das colunas R1 e R2 são as somas dos resultados positivos e neutros; já para coluna R3 somente totalizou-se os resultados positivos. Na coluna T apresenta-se a soma de R1, R2 e R3. Vale ressaltar que cada percentual entre parênteses representa a efetividade obtida dentro do intervalo apresentado.

Tabela 10 – Classificação dos resultados (Ri podendo variar de 1 a 35, Operadores Essenciais, projeto SimplePrograms).

#	Configuração	R1	R2	R3	T
1	Todas as métricas / X-Means - Euclidean	35 (100,00%)	35 (100,00%)	6 (17,14%)	76 (72,38%)
2	Todas as métricas / X-Means - Manhattan	35 (100,00%)	35 (100,00%)	6 (17,14%)	76 (72,38%)
3	Código original / EM	35 (100,00%)	35 (100,00%)	5 (14,29%)	75 (71,43%)
4	Código original / X-Means - Euclidean	35 (100,00%)	35 (100,00%)	5 (14,29%)	75 (71,43%)
5	Todas as métricas / EM	35 (100,00%)	35 (100,00%)	4 (11,43%)	74 (70,48%)
6	Código original / X-Means - Manhattan	35 (100,00%)	35 (100,00%)	4 (11,43%)	74 (70,48%)
7	Métricas Seleccionadas / EM	35 (100,00%)	35 (100,00%)	3 (8,57%)	73 (69,52%)
8	Métricas Seleccionadas / X-Means - Manhattan	35 (100,00%)	35 (100,00%)	2 (5,71%)	72 (68,57%)
9	Métricas Seleccionadas / X-Means - Euclidean	35 (100,00%)	35 (100,00%)	0 (0,00%)	70 (66,67%)
10	Código original / jPlag	35 (100,00%)	35 (100,00%)	0 (0,00%)	70 (66,67%)
11	Código original / mdist - Jaccard	35 (100,00%)	35 (100,00%)	0 (0,00%)	70 (66,67%)
12	Código original / mdist - Levenshtein	35 (100,00%)	35 (100,00%)	0 (0,00%)	70 (66,67%)
13	Código processado / jPlag	35 (100,00%)	35 (100,00%)	0 (0,00%)	70 (66,67%)
14	Código processado / mdist - Jaccard	35 (100,00%)	35 (100,00%)	0 (0,00%)	70 (66,67%)
15	Código processado / mdist - Levenshtein	35 (100,00%)	35 (100,00%)	0 (0,00%)	70 (66,67%)
16	Código processado / mdist - NCD	35 (100,00%)	35 (100,00%)	0 (0,00%)	70 (66,67%)
17	Código original / mdist - NCD	35 (100,00%)	35 (100,00%)	0 (0,00%)	70 (66,67%)
18	Código processado / X-Means - Manhattan	33 (94,29%)	33 (94,29%)	3 (8,57%)	69 (65,71%)
19	Código processado / EM	33 (94,29%)	33 (94,29%)	3 (8,57%)	69 (65,71%)
20	Código processado / X-Means - Euclidean	33 (94,29%)	33 (94,29%)	3 (8,57%)	69 (65,71%)

Fonte: Elaborada pelo autor.

Análise Baseada em R1, R2 e R3, em Conjunto

Realizando a análise das informações apresentadas na Tabela 10, constatarem-se resultados aproximados aos obtidos por Guarnieri, Pizzoleto e Ferrari (2022), mas destacando as configurações *Todas as métricas / X-Means - Euclidean*, *Todas as métricas / X-Means - Manhattan*, *Código original / EM* e *Código original / X-Means - Euclidean*, respectivamente, somaram 76, 76, 75 e 75 dos resultados positivos de um máximo de 105 possíveis.

De modo geral, todas as demais configurações apresentaram resultados próximos, quanto a efetividade, ficando todos com valores superior a 65,71% e tendo máximo em 72,38%.

Análise Baseada em R1, R2 e R3, Individualmente

Analisando os resultados de R1 observou-se uma efetividade na maioria das configurações chegando a 100%, mesmo variando suas Abstrações e medidas de distância. Vale ressaltar que somente três configurações tiveram efetividade de 94.29%. Neste caso R2, obteve os mesmos resultados que R1.

Considerando R3 os resultados não foram promissores, já que os melhores resultados de efetividade são de 17,14% para as configurações *Todas as métricas / X-Means - Euclidean* e *Todas as métricas / X-Means - Manhattan*. Outras configurações como *Código original / jPlag*, *Código original / mdist - Jaccard*, dentre outras apresentaram 0% de efetividade.

6.3.3 Resultados e Análises: Melhores Operadores para Javassist

Como apresentado na subseção anterior, realizou-se uma comparação usando as 35 classes contidas no projeto SimplePrograms. A comparação dos resultados gerados foi tanto para melhores operadores quanto para os operadores definidos pelo conjunto essencial. Entretanto, surge a seguinte questão: pode-se considerar que os resultados obtidos e analisados na subseção anterior podem se repetir ao fazer uso de um conjunto maior de classes?

Sendo assim, para responder a essa questão, definiu-se uma nova experimentação de modo a fazer uso de um conjunto maior de classes; neste caso, o projeto Javassist foi utilizado, o qual contém 221 arquivos de código fonte Java⁵ que englobam acima de 350 classes (detalhes na Tabela 2). As mesmas configurações do *SiMut* que foram empregadas no experimento descrito na seção anterior foram utilizadas.

Na Tabela 11 sumarizam-se os resultados obtidos fazendo uso da técnica Melhores Operadores de mutação, novamente considerando um conjunto de quatro melhores operadores (ou mais, quando ocorre empate entre operadores). A análise dos resultados é apresentada a seguir.

⁵ Na Tabela 2, para o projeto Javassist, constam 234 arquivos de código fonte Java. Entretanto, apenas para 221 deles a ferramenta PIT gerou mutantes.

6.3.3.1 Análise dos Resultados

Tabela 11 – Resumo dos resultados considerando os quatro melhores operadores (projeto Javassist, clusters tamanho padrão).

Métricas selecionadas / X-Means - Euclidean		Código processado / X-Means - Manhattan	
R1	V(78), N(55), X(88)	R1	V(71), N(46), X(104)
R2	V(56), N(128), X(37)	R2	V(55), N(159), X(7)
R3	V(78), N(50), X(93)	R3	V(91), N(54), X(76)
Todas as métricas / EM		Todas as métricas / X-Means - Euclidean	
R1	V(98), N(52), X(71)	R1	V(88), N(51), X(82)
R2	V(88), N(109), X(24)	R2	V(73), N(123), X(25)
R3	V(108), N(49), X(64)	R3	V(93), N(54), X(74)
Todas as métricas / X-Means - Manhattan		Métricas selecionadas / EM	
R1	V(78), N(53), X(90)	R1	V(101), N(51), X(69)
R2	V(64), N(130), X(27)	R2	V(87), N(111), X(23)
R3	V(83), N(52), X(86)	R3	V(111), N(52), X(58)
Métricas selecionadas / X-Means - Manhattan		Código original / jPlag	
R1	V(79), N(57), X(85)	R1	V(110), N(49), X(62)
R2	V(55), N(128), X(38)	R2	V(77), N(123), X(21)
R3	V(75), N(48), X(98)	R3	V(100), N(52), X(69)
Código original / mdist - Jaccard		Código original / mdist - Levenshtein	
R1	V(101), N(52), X(68)	R1	V(87), N(56), X(78)
R2	V(74), N(121), X(26)	R2	V(63), N(125), X(33)
R3	V(100), N(48), X(73)	R3	V(80), N(48), X(93)
Código original / EM		Código original / X-Means - Euclidean	
R1	V(60), N(48), X(113)	R1	V(82), N(47), X(92)
R2	V(69), N(121), X(31)	R2	V(71), N(127), X(23)
R3	V(89), N(52), X(80)	R3	V(97), N(52), X(72)
Código original / X-Means - Manhattan		Código processado / jPlag	
R1	V(63), N(44), X(114)	R1	V(105), N(51), X(65)
R2	V(67), N(138), X(16)	R2	V(76), N(122), X(23)
R3	V(92), N(54), X(75)	R3	V(99), N(53), X(69)
Código processado / mdist - Jaccard		Código processado / mdist - Levenshtein	
R1	V(97), N(52), X(72)	R1	V(78), N(58), X(85)
R2	V(69), N(127), X(25)	R2	V(58), N(131), X(32)
R3	V(99), N(51), X(71)	R3	V(82), N(49), X(90)
Código processado / EM		Código processado / X-Means - Euclidean	
R1	V(42), N(45), X(134)	R1	V(45), N(44), X(132)
R2	V(25), N(164), X(32)	R2	V(62), N(130), X(29)
R3	V(55), N(50), X(116)	R3	V(85), N(52), X(84)
Código processado / mdist - NCD		Código original / mdist - NCD	
R1	V(79), N(53), X(89)	R1	V(76), N(52), X(93)
R2	V(62), N(124), X(35)	R2	V(62), N(126), X(33)
R3	V(81), N(52), X(88)	R3	V(81), N(48), X(92)

Fonte: Elaborada pelo autor.

Seguindo a lógica de interpretação e considerando os resultados gerados fazendo uso dos Melhores Operadores de mutação, na Tabela 12 é apresentado um ranqueamento em que os valores de R1, R2 e R3 podem variar de 1 a 221. Analogamente aos experimentos anteriores (que empregaram os 35 programas do projeto SimplePrograms), os valores das colunas R1 e R2 são as somas dos resultados positivos e neutros; já para coluna R3,

Tabela 12 – Classificação dos resultados (R_i podendo variar de 1 a 221, Melhores Operadores, projeto Javassist, clusters tamanho padrão).

#	Configuração	R1	R2	R3	T
1	Selected metrics / EM	157 (71,04%)	186 (84,16%)	107 (48,42%)	450 (67,87%)
2	Original code / jPlag	157 (71,04%)	173 (78,28%)	108 (48,87%)	438 (66,06%)
3	Original code / X-Means - Manhattan	107 (48,42%)	220 (99,55%)	98 (44,34%)	425 (64,10%)
4	ProGuard / jPlag	153 (69,23%)	170 (76,92%)	101 (45,70%)	424 (63,95%)
5	Original code / mdist - Jaccard	151 (68,33%)	169 (76,47%)	100 (45,25%)	420 (63,35%)
6	All metrics / X-Means - Manhattan	136 (61,54%)	186 (84,16%)	91 (41,18%)	413 (62,29%)
7	All metrics / X-Means - Euclidean	129 (58,37%)	182 (82,35%)	94 (42,53%)	405 (61,09%)
8	Selected metrics / X-Means - Manhattan	147 (66,52%)	178 (80,54%)	76 (34,39%)	401 (60,48%)
9	Selected metrics / X-Means - Euclidean	145 (65,61%)	161 (72,85%)	90 (40,72%)	396 (59,73%)
10	ProGuard / mdist - Jaccard	138 (62,44%)	160 (72,40%)	93 (42,08%)	391 (58,97%)
11	ProGuard / mdist - NCD	139 (62,90%)	157 (71,04%)	89 (40,27%)	385 (58,07%)
12	ProGuard / mdist - Levenshtein	142 (64,25%)	153 (69,23%)	89 (40,27%)	384 (57,92%)
13	All metrics / EM	122 (55,20%)	174 (78,73%)	85 (38,46%)	381 (57,47%)
14	Original code / mdist - Levenshtein	135 (61,09%)	154 (69,68%)	80 (36,20%)	369 (55,66%)
15	Original code / mdist - NCD	136 (61,54%)	147 (66,52%)	86 (38,91%)	369 (55,66%)
16	ProGuard / X-Means - Euclidean	90 (40,72%)	186 (84,16%)	82 (37,10%)	358 (54,00%)
17	ProGuard / EM	100 (45,25%)	167 (75,57%)	80 (36,20%)	347 (52,34%)
18	Original code / X-Means - Euclidean	93 (42,08%)	146 (66,06%)	77 (34,84%)	316 (47,66%)
19	ProGuard / X-Means - Manhattan	95 (42,99%)	139 (62,90%)	57 (25,79%)	291 (43,89%)
20	Original code / EM	80 (36,20%)	133 (60,18%)	53 (23,98%)	266 (40,12%)

Fonte: Elaborada pelo autor.

somente totalizaram-se os resultados positivos. Na coluna T apresenta-se a soma de R1, R2 e R3.

Análise Baseada em R1, R2 e R3, em Conjunto

Realizando a análise das informações apresentadas na Tabela 12, as configurações *Métricas selecionadas / EM* e *Código original / jPlag* destacam-se. Respectivamente, essas configurações levaram a um total de 450 e 438 de resultados positivos, de um máximo de 663 possíveis.

Análise Baseada em R1, R2 e R3, Individualmente

Analisando os resultados de R1 observou-se uma efetividade similar em 65% das configurações com resultados acima de 58,37%, mesmo variando suas abstrações e medidas de distância. Dentre as abstrações, citam-se código processado e código original, e dentre as medidas de distância, Jaccard, NCD, e Levenshtein.

Já para R2, tem-se uma efetividade acima de 60,18% para todas as configurações, inclusive chegando em quatro configurações acima de 84,16% de efetividade.

Quando considerado R3, quando usa-se a formulação de clusters aleatórios, os resultados se apresentaram insatisfatórios, já que se obtém uma baixa efetividade quando se variam as configurações definidas no *SiMut*. Em específico, para todas as configurações, a efetividade foi inferior a 50%; ou seja, um cluster aleatoriamente formado é tão efetivo quanto (ou mais efetivo que) um cluster computado com base em similaridade entre programas com respeito à capacidade de prever escores de operadores de mutação.

Ressalta-se que os resultados analisados nesta seção baseiam-se em “execuções padronizadas” do *SiMut*, nas quais os tamanhos dos clusters de referência de programas, e também os cluster aleatórios, têm tamanho predefinido (com aproximadamente 20% dos programas considerados), ou têm tamanho definidos pelo próprio algoritmo de agrupamento. Decidiu-se, então, aprimorar a experimentação. Neste ponto, surge a seguinte questão: “Tendo clusters, principalmente o de referência, menores, o distanciamento entre as classes, com relação à similaridade, será menor e a probabilidade de acertos poderá aumentar?”. Para responder a esta questão, realizou-se a execução de duas novas configurações experimentais: uma usando 10% das classes para geração do cluster de referência, e outra usando 5%. Ressalta-se que tal processo foi empregado somente para as configurações do *SiMut* que incluíam as abstrações *mdist* e *jPlag*, já que para a ferramenta *Weka* tal confirmação não pode ser realizada, pois a ferramenta é quem define a quantidade de elementos dos clusters de acordo com seu algoritmo de aproximação. Na Seção 6.3.3.2 apresentam-se os resultados obtidos na execução e as análises.

6.3.3.2 Análise dos Resultados - Cluster Referência com 10% e 5% dos Programas

Nesta seção são apresentados os resultados experimentais seguindo as mesmas configurações anteriores do *SiMut*, e com o mesmo conjunto de programas que compõem projeto *Javassist*. Como já mencionado, nesta seção são abordados os experimentos para os quais os clusters de referência contiveram 10% e 5% dos programas. Em particular, do total de 221 programas do projeto *Javassist*, os clusters de referência continham 22 e 11 programas, respectivamente. Os mesmos tamanhos foram considerados para os clusters formados aleatoriamente.

Nas Tabelas 13 e 15 resumiram-se os resultados obtidos para as variadas configurações do *SiMut*, considerando as configurações experimentais que definiram a formação de clusters de referência com 10% e 5% dos programas, respectivamente. Similarmente, nas Tabelas 14 e 16 mostram-se os ranqueamentos dos resultados obtidos, listando-se os valores individuais para R1, R2 e R3, e a somatória de todos eles na coluna T.

Os resultados nos experimentos nos quais o cluster de referência foi reduzido para 10% e 5% são similares aos resultados observados para o experimento no qual o cluster de referência foi gerado de forma padrão (ou seja, com aproximadamente 20% dos programas considerados), conforme discutido na Seção 6.3.3.1. Em linhas gerais, embora algumas configurações do *SiMut* tenham se destacado em relação aos resultados R1 e R2, nenhuma delas apresentou resultados superiores aos obtidos para os clusters aleatoriamente formados. Essa constatação se baseia nos dados apresentados na coluna R3 das Tabelas 14 e 16.

Tabela 13 – Resumo dos resultados considerando os quatro melhores operadores (projeto Javassist, clusters de referência com 10%).

Métricas selecionadas / X-Means - Euclidean		Código processado / X-Means - Manhattan	
R1	V(78), N(55), X(88)	R1	V(71), N(46), X(104)
R2	V(56), N(128), X(37)	R2	V(55), N(159), X(7)
R3	V(69), N(54), X(98)	R3	V(80), N(58), X(83)
Todas as métricas / EM		Todas as métricas / X-Means - Euclidean	
R1	V(98), N(52), X(71)	R1	V(88), N(51), X(82)
R2	V(88), N(109), X(24)	R2	V(73), N(123), X(25)
R3	V(97), N(53), X(71)	R3	V(87), N(55), X(79)
Todas as métricas / X-Means - Manhattan		Métricas selecionadas / EM	
R1	V(78), N(53), X(90)	R1	V(101), N(51), X(69)
R2	V(64), N(130), X(27)	R2	V(87), N(111), X(23)
R3	V(76), N(52), X(93)	R3	V(102), N(56), X(63)
Métricas selecionadas / X-Means - Manhattan		Código original / jPlag	
R1	V(79), N(57), X(85)	R1	V(105), N(54), X(62)
R2	V(55), N(128), X(38)	R2	V(72), N(127), X(22)
R3	V(66), N(51), X(104)	R3	V(90), N(54), X(77)
Código original / mdist - Jaccard		Código original / mdist - Levenshtein	
R1	V(93), N(54), X(74)	R1	V(91), N(55), X(75)
R2	V(75), N(120), X(26)	R2	V(71), N(116), X(34)
R3	V(87), N(53), X(81)	R3	V(83), N(53), X(85)
Código original / EM		Código original / X-Means - Euclidean	
R1	V(60), N(48), X(113)	R1	V(82), N(47), X(92)
R2	V(69), N(121), X(31)	R2	V(71), N(127), X(23)
R3	V(79), N(55), X(87)	R3	V(85), N(56), X(80)
Código original / X-Means - Manhattan		Código processado / jPlag	
R1	V(63), N(44), X(114)	R1	V(101), N(52), X(68)
R2	V(67), N(138), X(16)	R2	V(71), N(122), X(28)
R3	V(87), N(56), X(78)	R3	V(85), N(54), X(82)
Código processado / mdist - Jaccard		Código processado / mdist - Levenshtein	
R1	V(92), N(51), X(78)	R1	V(94), N(57), X(70)
R2	V(67), N(123), X(31)	R2	V(70), N(119), X(32)
R3	V(80), N(54), X(87)	R3	V(78), N(53), X(90)
Código processado / EM		Código processado / X-Means - Euclidean	
R1	V(42), N(45), X(134)	R1	V(45), N(44), X(132)
R2	V(25), N(164), X(32)	R2	V(62), N(130), X(29)
R3	V(52), N(53), X(116)	R3	V(75), N(54), X(92)
Código processado / mdist - NCD		Código original / mdist - NCD	
R1	V(91), N(59), X(71)	R1	V(95), N(51), X(75)
R2	V(69), N(124), X(28)	R2	V(77), N(108), X(36)
R3	V(80), N(58), X(83)	R3	V(89), N(51), X(81)

Fonte: Elaborada pelo autor.

6.3.4 Resultados e Análises: Operadores Essenciais para Javassist

Nesta seção, apresentam-se os resultados dos experimentos que fizeram uso da técnica Operadores Essenciais, como descrito na 6.2.1. Ressalta-se que se utilizou o mesmo grupo de configurações de abstrações e métricas de similaridade para a definição dos clusters processados pelo *SiMut*. Os resultados são analisados em bloco, na seção a seguir.

Tabela 14 – Classificação dos resultados (Ri podendo variar de 1 a 221, Melhores Operadores, projeto Javassist, clusters de referência com 10%).

#	Configuração	R1	R2	R3	T
1	Selected metrics / EM	157 (71,04%)	186 (84,16%)	102 (46,15%)	445 (67,12%)
2	Original code / jPlag	157 (71,04%)	168 (76,02%)	95 (42,99%)	420 (63,35%)
3	ProGuard / jPlag	159 (71,95%)	167 (75,57%)	94 (42,53%)	420 (63,35%)
4	Original code / X-Means - Manhattan	107 (48,42%)	220 (99,55%)	91 (41,18%)	418 (63,05%)
5	Original code / mdist - Jaccard	152 (68,78%)	166 (75,11%)	93 (42,08%)	411 (61,99%)
6	All metrics / X-Means - Manhattan	136 (61,54%)	186 (84,16%)	85 (38,46%)	407 (61,39%)
7	Original code / mdist - Levenshtein	143 (64,71%)	171 (77,38%)	93 (42,08%)	407 (61,39%)
8	Selected metrics / X-Means - Manhattan	147 (66,52%)	178 (80,54%)	73 (33,03%)	398 (60,03%)
9	ProGuard / mdist - Jaccard	145 (65,61%)	157 (71,04%)	95 (42,99%)	397 (59,88%)
10	All metrics / X-Means - Euclidean	129 (58,37%)	182 (82,35%)	84 (38,01%)	395 (59,58%)
11	ProGuard / mdist - Levenshtein	138 (62,44%)	160 (72,40%)	94 (42,53%)	392 (59,13%)
12	Selected metrics / X-Means - Euclidean	145 (65,61%)	161 (72,85%)	82 (37,10%)	388 (58,52%)
13	Original code / mdist - NCD	142 (64,25%)	159 (71,95%)	79 (35,75%)	380 (57,32%)
14	All metrics / EM	122 (55,20%)	174 (78,73%)	79 (35,75%)	375 (56,56%)
15	ProGuard / mdist - NCD	138 (62,44%)	155 (70,14%)	78 (35,29%)	371 (55,96%)
16	ProGuard / X-Means - Euclidean	90 (40,72%)	186 (84,16%)	73 (33,03%)	349 (52,64%)
17	ProGuard / EM	100 (45,25%)	167 (75,57%)	74 (33,48%)	341 (51,43%)
18	Original code / X-Means - Euclidean	93 (42,08%)	146 (66,06%)	69 (31,22%)	308 (46,46%)
19	ProGuard / X-Means - Manhattan	95 (42,99%)	139 (62,90%)	54 (24,43%)	288 (43,44%)
20	Original code / EM	80 (36,20%)	133 (60,18%)	46 (20,81%)	259 (39,06%)

Fonte: Elaborada pelo autor.

6.3.4.1 Análise dos Resultados

Nas Tabelas 17, 19 e 21 sumarizam-se os resultados obtidos para as variadas configurações do *SiMut*, considerando as configurações experimentais que definiram a formação de clusters de referência com tamanho padrão, com 10% dos programas, e com 5% dos programas, respectivamente. Nas Tabelas 18, 20 e 22, por suas vezes, mostram-se os ranqueamentos dos resultados obtidos, listando-se os valores individuais para R1, R2 e R3, e a somatória de todos eles na coluna T.

Quando a técnica de redução de custo Operadores Essenciais foi empregada, independentemente do tamanho do cluster de referência formado, os resultados nos experimentos são similares aos resultados observados para os experimentos que empregaram a técnica Melhores Operadores (descritos na Seção 6.3.3).

Em linhas gerais, embora algumas configurações do *SiMut* tenham se destacado em relação aos resultados R1 e R2, nenhuma delas apresentou resultados superiores aos obtidos para os clusters aleatoriamente formados. Essa constatação se baseia nos dados apresentados na coluna R3 das Tabelas 18, 20 e 22. Uma única exceção pode ser observada na Tabela 22, na qual o valor de R3 para a configuração *Código Original / mdist / Levenshtein* do *SiMut* é 51,13%.

6.4 Conclusões Preliminares

Com base no que foi apresentado na Seção 6.3 e suas subseções, já é possível inferir algumas conclusões preliminares e responder as questões de pesquisa apresentadas na 6.1.2. Para isso, inicialmente foram considerados três parâmetros de análise: configurações em

Tabela 15 – Resumo dos resultados considerando os quatro melhores operadores (projeto Javassist, clusters de referência com 5%).

Métricas selecionadas / X-Means - Euclidean		Código processado / X-Means - Manhattan	
R1	V(78), N(55), X(88)	R1	V(71), N(46), X(104)
R2	V(56), N(128), X(37)	R2	V(55), N(159), X(7)
R3	V(75), N(52), X(94)	R3	V(84), N(57), X(80)
Todas as métricas / EM		Todas as métricas / X-Means - Euclidean	
R1	V(98), N(52), X(71)	R1	V(88), N(51), X(82)
R2	V(88), N(109), X(24)	R2	V(73), N(123), X(25)
R3	V(100), N(54), X(67)	R3	V(85), N(53), X(83)
Todas as métricas / X-Means - Manhattan		Métricas selecionadas / EM	
R1	V(78), N(53), X(90)	R1	V(101), N(51), X(69)
R2	V(64), N(130), X(27)	R2	V(87), N(111), X(23)
R3	V(76), N(52), X(93)	R3	V(103), N(55), X(63)
Métricas selecionadas / X-Means - Manhattan		Código Original / jPlag	
R1	V(79), N(57), X(85)	R1	V(111), N(51), X(59)
R2	V(55), N(128), X(38)	R2	V(82), N(114), X(25)
R3	V(67), N(55), X(99)	R3	V(94), N(56), X(71)
Código Original / mdist - Jaccard		Código Original / mdist - Levenshtein	
R1	V(108), N(52), X(61)	R1	V(95), N(53), X(73)
R2	V(86), N(108), X(27)	R2	V(77), N(113), X(31)
R3	V(108), N(52), X(61)	R3	V(94), N(53), X(74)
Código Original / EM		Código Original / X-Means - Euclidean	
R1	V(60), N(48), X(113)	R1	V(82), N(47), X(92)
R2	V(69), N(121), X(31)	R2	V(71), N(127), X(23)
R3	V(87), N(51), X(83)	R3	V(95), N(53), X(73)
Código Original / X-Means - Manhattan		Código processado / jPlag	
R1	V(63), N(44), X(114)	R1	V(110), N(50), X(61)
R2	V(67), N(138), X(16)	R2	V(83), N(109), X(29)
R3	V(85), N(54), X(82)	R3	V(100), N(51), X(70)
Código processado / mdist - Jaccard		Código processado / mdist - Levenshtein	
R1	V(103), N(51), X(67)	R1	V(102), N(54), X(65)
R2	V(80), N(110), X(31)	R2	V(79), N(115), X(27)
R3	V(95), N(53), X(73)	R3	V(96), N(53), X(72)
Código processado / EM		Código processado / X-Means - Euclidean	
R1	V(42), N(45), X(134)	R1	V(45), N(44), X(132)
R2	V(25), N(164), X(32)	R2	V(62), N(130), X(29)
R3	V(54), N(55), X(112)	R3	V(83), N(54), X(84)
Código processado / mdist - NCD		Código Original / mdist - NCD	
R1	V(99), N(55), X(67)	R1	V(104), N(54), X(63)
R2	V(79), N(113), X(29)	R2	V(82), N(103), X(36)
R3	V(92), N(55), X(74)	R3	V(96), N(52), X(73)

Fonte: Elaborada pelo autor.

destaque, efetividade para os resultados R1, R2 e R3 conjuntamente, e resultado individual de R3. Como apresentado, fez-se uso de dois conjuntos de programas, organizados nos projetos intitulados SimplePrograms e Javassist. Discutindo-se os resultados apresentados na análise do SimplePrograms, nota-se que quando se trabalhou com as técnicas Melhores Operadores ou Operadores Essenciais, os resultados mostraram-se promissores com alta efetividade em um subgrupo de configurações do *SiMut*. Porém, ressalta-se que o Simpleprograms é um projeto composto por programas menores, que para facili-

Tabela 16 – Classificação dos resultados (Ri podendo variar de 1 a 221, Melhores Operadores, projeto Javassist, clusters de referência com 5%).

#	Configuração	R1	R2	R3	T
1	Selected metrics / EM	157 (71,04%)	186 (84,16%)	106 (47,96%)	449 (67,72%)
2	Original code / mdist - Jaccard	157 (71,04%)	172 (77,83%)	106 (47,96%)	435 (65,61%)
3	Original code / jPlag	155 (70,14%)	172 (77,83%)	103 (46,61%)	430 (64,86%)
4	ProGuard / mdist - Jaccard	154 (69,68%)	169 (76,47%)	104 (47,06%)	427 (64,40%)
5	Original code / X-Means - Manhattan	107 (48,42%)	220 (99,55%)	95 (42,99%)	422 (63,65%)
6	ProGuard / mdist - NCD	150 (67,87%)	169 (76,47%)	99 (44,80%)	418 (63,05%)
7	ProGuard / jPlag	149 (67,42%)	167 (75,57%)	96 (43,44%)	412 (62,14%)
8	ProGuard / mdist - Levenshtein	141 (63,80%)	167 (75,57%)	101 (45,70%)	409 (61,69%)
9	All metrics / X-Means - Manhattan	136 (61,54%)	186 (84,16%)	85 (38,46%)	407 (61,39%)
10	Original code / mdist - NCD	151 (68,33%)	160 (72,40%)	96 (43,44%)	407 (61,39%)
11	Original code / mdist - Levenshtein	144 (65,16%)	167 (75,57%)	93 (42,08%)	404 (60,94%)
12	Selected metrics / X-Means - Manhattan	147 (66,52%)	178 (80,54%)	73 (33,03%)	398 (60,03%)
13	All metrics / X-Means - Euclidean	129 (58,37%)	182 (82,35%)	86 (38,91%)	397 (59,88%)
14	Selected metrics / X-Means - Euclidean	145 (65,61%)	161 (72,85%)	86 (38,91%)	392 (59,13%)
15	All metrics / EM	122 (55,20%)	174 (78,73%)	86 (38,91%)	382 (57,62%)
16	ProGuard / X-Means - Euclidean	90 (40,72%)	186 (84,16%)	86 (38,91%)	362 (54,60%)
17	ProGuard / EM	100 (45,25%)	167 (75,57%)	74 (33,48%)	341 (51,43%)
18	Original code / X-Means - Euclidean	93 (42,08%)	146 (66,06%)	68 (30,77%)	307 (46,30%)
19	ProGuard / X-Means - Manhattan	95 (42,99%)	139 (62,90%)	56 (25,34%)	290 (43,74%)
20	Original code / EM	80 (36,20%)	133 (60,18%)	54 (24,43%)	267 (40,27%)

Fonte: Elaborada pelo autor.

tar a execução dos experimentos, foram agrupados em um único projeto. Frente a isso, neste primeiro momento considera-se que os resultados do Simpleprograms são de pouca relevância, ou seja, conclusões neles embasadas poderão gerar um viés nesta análise preliminar, já que 95% das configurações se mostraram efetivas para ambas as técnicas de redução de custo empregadas.

Realizando-se um comparativo entre os resultados analisados para o Javassist, o qual representa um estudo de maior escala (e, por consequência, permite a derivação de respostas mais relevantes às questões de pesquisa), em todas as suas execuções e configurações, derivam-se as seguintes respostas às questões de pesquisa:

- Para **RQ1.1** (*A similaridade entre programas, calculada com base em abstrações dos mesmos, pode auxiliar na identificação dos melhores operadores de mutação (com base nos escore de mutação dos mesmos) para novos programas (ainda não testados)*), pode-se concluir que computar similaridade para definição dos melhores operadores não é interessante, pois os *clusters* formados aleatoriamente foram tão bons quanto ou melhores que os *clusters* formados utilizando métricas internas.
- Para **RQ1.2** (*A similaridade entre programas, calculada com base em abstrações dos mesmos, pode auxiliar na identificação de operadores essenciais de mutação para novos programas (ainda não testados)?*), pode-se concluir que computar similaridade para definição dos conjuntos essenciais não é interessante, pois os *clusters* formados aleatoriamente foram tão bons quanto ou melhores que os *clusters* formados utilizando métricas.

Tabela 17 – Resumo dos resultados considerando os operadores essenciais (projeto Javasist, clusters tamanho padrão).

Métricas selecionadas / X-Means - Euclidean		Código processado / X-Means - Manhattan	
R1	V(66), N(76), X(79)	R1	V(94), N(50), X(77)
R2	V(26), N(158), X(37)	R2	V(1), N(216), X(4)
R3	V(65), N(79), X(77)	R3	V(72), N(84), X(65)
Todas as métricas / EM		Todas as métricas / X-Means - Euclidean	
R1	V(36), N(154), X(31)	R1	V(79), N(119), X(23)
R2	V(5), N(203), X(13)	R2	V(8), N(212), X(1)
R3	V(67), N(84), X(70)	R3	V(68), N(81), X(72)
Todas as métricas / X-Means - Manhattan		Métricas selecionadas / EM	
R1	V(68), N(91), X(62)	R1	V(77), N(86), X(58)
R2	V(26), N(173), X(22)	R2	V(17), N(182), X(22)
R3	V(75), N(78), X(68)	R3	V(71), N(78), X(72)
Métricas selecionadas / X-Means - Manhattan		Código Original / jPlag	
R1	V(68), N(121), X(32)	R1	V(23), N(39), X(159)
R2	V(1), N(212), X(8)	R2	V(1), N(62), X(158)
R3	V(70), N(75), X(76)	R3	V(56), N(5), X(160)
Código Original / mdist - Jaccard		Código Original / mdist - Levenshtein	
R1	V(106), N(15), X(100)	R1	V(100), N(15), X(106)
R2	V(1), N(119), X(101)	R2	V(10), N(142), X(69)
R3	V(117), N(5), X(99)	R3	V(149), N(4), X(68)
Código Original / EM		Código Original / X-Means - Euclidean	
R1	V(77), N(50), X(94)	R1	V(73), N(51), X(97)
R2	V(11), N(192), X(18)	R2	V(12), N(183), X(26)
R3	V(74), N(82), X(65)	R3	V(68), N(77), X(76)
Código Original / X-Means - Manhattan		Código processado / jPlag	
R1	V(84), N(54), X(83)	R1	V(16), N(41), X(164)
R2	V(2), N(212), X(7)	R2	V(1), N(59), X(161)
R3	V(66), N(84), X(71)	R3	V(52), N(6), X(163)
Código processado / mdist - Jaccard		Código processado / mdist - Levenshtein	
R1	V(65), N(31), X(125)	R1	V(112), N(15), X(94)
R2	V(1), N(96), X(124)	R2	V(10), N(116), X(95)
R3	V(97), N(0), X(124)	R3	V(124), N(4), X(93)
Código processado / EM		Código processado / X-Means - Euclidean	
R1	V(60), N(58), X(103)	R1	V(102), N(47), X(72)
R2	V(7), N(187), X(27)	R2	V(2), N(206), X(13)
R3	V(59), N(77), X(85)	R3	V(67), N(79), X(75)
Código processado / mdist - NCD		Código Original / mdist - NCD	
R1	V(88), N(25), X(108)	R1	V(103), N(19), X(99)
R2	V(3), N(112), X(106)	R2	V(6), N(121), X(94)
R3	V(112), N(3), X(106)	R3	V(125), N(3), X(93)

Fonte: Elaborada pelo autor.

□ Para **RQ2** (*A diminuição do tamanho, quanto à quantidade de classes, do grupo de referência impacta na classificação e definição de Melhores Operadores ou Operadores Essenciais?*), pode-se concluir que os resultados são consistentes, aumentando ou diminuindo o tamanho dos *clusters* de referência, pois as tabelas que mostram os resultados para R3 mostram resultados sempre abaixo de 50%, comparando os *clusters* gerados aleatoriamente com os gerados usando métricas internas.

Tabela 18 – Classificação dos resultados (R_i podendo variar de 1 a 221, Operadores Essenciais, projeto Javassist, clusters tamanho padrão).

#	Configuração	R1	R2	R3	T
1	All metrics / X-Means - Euclidean	198 (89,59%)	220 (99,55%)	68 (30,77%)	486 (73,30%)
2	Selected metrics / X-Means - Manhattan	189 (85,52%)	213 (96,38%)	70 (31,67%)	472 (71,19%)
3	All metrics / EM	190 (85,97%)	208 (94,12%)	67 (30,32%)	465 (70,14%)
4	All metrics / X-Means - Manhattan	159 (71,95%)	199 (90,05%)	75 (33,94%)	433 (65,31%)
5	ProGuard / X-Means - Manhattan	144 (65,16%)	217 (98,19%)	72 (32,58%)	433 (65,31%)
6	Selected metrics / EM	163 (73,76%)	199 (90,05%)	71 (32,13%)	433 (65,31%)
7	ProGuard / X-Means - Euclidean	149 (67,42%)	208 (94,12%)	67 (30,32%)	424 (63,95%)
8	Original code / X-Means - Manhattan	138 (62,44%)	214 (96,83%)	66 (29,86%)	418 (63,05%)
9	Original code / mdist - Levenshtein	115 (52,04%)	152 (68,78%)	149 (67,42%)	416 (62,75%)
10	Original code / EM	127 (57,47%)	203 (91,86%)	74 (33,48%)	404 (60,94%)
11	Selected metrics / X-Means - Euclidean	142 (64,25%)	184 (83,26%)	65 (29,41%)	391 (58,97%)
12	Original code / X-Means - Euclidean	124 (56,11%)	195 (88,24%)	68 (30,77%)	387 (58,37%)
13	ProGuard / mdist - Levenshtein	127 (57,47%)	126 (57,01%)	124 (56,11%)	377 (56,86%)
14	Original code / mdist - NCD	122 (55,20%)	127 (57,47%)	125 (56,56%)	374 (56,41%)
15	ProGuard / EM	118 (53,39%)	194 (87,78%)	59 (26,70%)	371 (55,96%)
16	Original code / mdist - Jaccard	121 (54,75%)	120 (54,30%)	117 (52,94%)	358 (54,00%)
17	ProGuard / mdist - NCD	113 (51,13%)	115 (52,04%)	112 (50,68%)	340 (51,28%)
18	ProGuard / mdist - Jaccard	96 (43,44%)	97 (43,89%)	97 (43,89%)	290 (43,74%)
19	Original code / jPlag	62 (28,05%)	63 (28,51%)	56 (25,34%)	181 (27,30%)
20	ProGuard / jPlag	57 (25,79%)	60 (27,15%)	52 (23,53%)	169 (25,49%)

Fonte: Elaborada pelo autor.

6.5 Análise Estatística

Nas Seções 6.3 e 6.4, foram apresentadas algumas análises referentes aos resultados apresentados, incluindo a derivação de respostas preliminares às questões de pesquisas definidas para este estudo. Entretanto, há pontos que precisam ser melhor esclarecido como, por exemplo, comprovar estatisticamente os resultados obtidos.

Com o intuito de comparar estatisticamente os dados de resultados apresentados, fez-se uso de duas técnicas: *Análise de Variância* (também conhecido como *ANOVA*), e *Teste t*. Os objetivos e as aplicações de ambas as técnicas de análise são descritas na sequência.

Análise de variância

Análise de variância é a técnica estatística que permite avaliar afirmações sobre as médias de populações. A análise visa, fundamentalmente, a verificar se existe uma diferença significativa entre as médias e se os fatores exercem influência em alguma variável dependente (MARTINS, 2005).

O objetivo da análise de variância é comparar médias de diferentes resultados para verificar se esses resultados possuem médias iguais ou não. Assim, essa técnica permite que vários grupos sejam comparados a um só tempo. Em outras palavras, a análise de variância é utilizada quando se quer decidir se as diferenças amostrais observadas são reais (causadas por diferenças significativas nos resultados observados) ou casuais (decorrentes da mera variabilidade amostral). Portanto, essa análise parte do pressuposto que o acaso só produz pequenos desvios, sendo as grandes diferenças geradas por causas reais (MARTINS, 2005).

Segundo MARTINS (2005), os pressupostos básicos da análise de variância são:

Tabela 19 – Resumo dos resultados considerando os operadores essenciais (projeto Javassist, clusters de referência com 10%).

Métricas selecionadas / X-Means - Euclidean		Código processado / X-Means - Manhattan	
R1	V(67), N(67), X(87)	R1	V(95), N(50), X(76)
R2	V(26), N(158), X(37)	R2	V(1), N(217), X(3)
R3	V(75), N(63), X(83)	R3	V(78), N(72), X(71)
Todas as métricas / EM		Todas as métricas / X-Means - Euclidean	
R1	V(61), N(107), X(53)	R1	V(81), N(103), X(37)
R2	V(4), N(204), X(13)	R2	V(7), N(212), X(2)
R3	V(68), N(68), X(85)	R3	V(90), N(65), X(66)
Todas as métricas / X-Means - Manhattan		Métricas selecionadas / EM	
R1	V(70), N(91), X(60)	R1	V(79), N(76), X(66)
R2	V(27), N(173), X(21)	R2	V(18), N(181), X(22)
R3	V(68), N(59), X(94)	R3	V(78), N(64), X(79)
Métricas selecionadas / X-Means - Manhattan		Código original / jPlag	
R1	V(74), N(97), X(50)	R1	V(131), N(9), X(81)
R2	V(1), N(212), X(8)	R2	V(0), N(141), X(80)
R3	V(79), N(64), X(78)	R3	V(133), N(5), X(83)
Código original / mdist - Jaccard		Código original / mdist - Levenshtein	
R1	V(73), N(98), X(50)	R1	V(71), N(91), X(59)
R2	V(73), N(129), X(19)	R2	V(71), N(125), X(25)
R3	V(73), N(7), X(141)	R3	V(71), N(8), X(142)
Código original / EM		Código original / X-Means - Euclidean	
R1	V(77), N(50), X(94)	R1	V(76), N(51), X(94)
R2	V(13), N(190), X(18)	R2	V(13), N(184), X(24)
R3	V(81), N(61), X(79)	R3	V(77), N(60), X(84)
Código original / X-Means - Manhattan		Código processado / jPlag	
R1	V(85), N(54), X(82)	R1	V(117), N(10), X(94)
R2	V(2), N(213), X(6)	R2	V(1), N(126), X(94)
R3	V(79), N(68), X(74)	R3	V(120), N(5), X(96)
Código processado / mdist - Jaccard		Código processado / mdist - Levenshtein	
R1	V(12), N(59), X(150)	R1	V(9), N(67), X(145)
R2	V(1), N(85), X(135)	R2	V(2), N(75), X(144)
R3	V(89), N(4), X(128)	R3	V(82), N(5), X(134)
Código processado / EM		Código processado / X-Means - Euclidean	
R1	V(60), N(58), X(103)	R1	V(102), N(47), X(72)
R2	V(6), N(188), X(27)	R2	V(2), N(206), X(13)
R3	V(68), N(69), X(84)	R3	V(75), N(68), X(78)
Código processado / mdist - NCD		Código original / mdist - NCD	
R1	V(28), N(61), X(132)	R1	V(39), N(58), X(124)
R2	V(2), N(89), X(130)	R2	V(0), N(97), X(124)
R3	V(97), N(3), X(121)	R3	V(101), N(5), X(115)

Fonte: Elaborada pelo autor.

- ☐ As amostras são aleatórias e independentes.
- ☐ As populações têm distribuição normal.
- ☐ As variâncias dos resultados são iguais.

Na prática, esses pressupostos não precisam ser todos rigorosamente satisfeitos. Os resultados são empiricamente verdadeiros sempre que os resultados são aproximadamente normais e têm variâncias próximas.

Tabela 20 – Classificação dos resultados (R_i podendo variar de 1 a 221, Operadores Essenciais, projeto Javassist, cluster de referência com 10%).

#	Configuração	R1	R2	R3	T
1	All metrics / X-Means - Euclidean	184 (83,26%)	219 (99,10%)	90 (40,72%)	493 (74,36%)
2	Selected metrics / X-Means - Manhattan	171 (77,38%)	213 (96,38%)	79 (35,75%)	463 (69,83%)
3	Original code / mdist - Jaccard	171 (77,38%)	202 (91,40%)	73 (33,03%)	446 (67,27%)
4	All metrics / EM	168 (76,02%)	208 (94,12%)	68 (30,77%)	444 (66,97%)
5	ProGuard / X-Means - Manhattan	145 (65,61%)	218 (98,64%)	78 (35,29%)	441 (66,52%)
6	Original code / X-Means - Manhattan	139 (62,90%)	215 (97,29%)	79 (35,75%)	433 (65,31%)
7	ProGuard / X-Means - Euclidean	149 (67,42%)	208 (94,12%)	75 (33,94%)	432 (65,16%)
8	Selected metrics / EM	155 (70,14%)	199 (90,05%)	78 (35,29%)	432 (65,16%)
9	All metrics / X-Means - Manhattan	161 (72,85%)	200 (90,50%)	68 (30,77%)	429 (64,71%)
10	Original code / mdist - Levenshtein	162 (73,30%)	196 (88,69%)	71 (32,13%)	429 (64,71%)
11	Original code / jPlag	140 (63,35%)	141 (63,80%)	133 (60,18%)	414 (62,44%)
12	Original code / EM	127 (57,47%)	203 (91,86%)	81 (36,65%)	411 (61,99%)
13	Original code / X-Means - Euclidean	127 (57,47%)	197 (89,14%)	77 (34,84%)	401 (60,48%)
14	Selected metrics / X-Means - Euclidean	134 (60,63%)	184 (83,26%)	75 (33,94%)	393 (59,28%)
15	ProGuard / EM	118 (53,39%)	194 (87,78%)	68 (30,77%)	380 (57,32%)
16	ProGuard / jPlag	127 (57,47%)	127 (57,47%)	120 (54,30%)	374 (56,41%)
17	Original code / mdist - NCD	97 (43,89%)	97 (43,89%)	101 (45,70%)	295 (44,49%)
18	ProGuard / mdist - NCD	89 (40,27%)	91 (41,18%)	97 (43,89%)	277 (41,78%)
19	ProGuard / mdist - Jaccard	71 (32,13%)	86 (38,91%)	89 (40,27%)	246 (37,10%)
20	ProGuard / mdist - Levenshtein	76 (34,39%)	77 (34,84%)	82 (37,10%)	235 (35,44%)

Fonte: Elaborada pelo autor.

Para MARTINS (2005), as hipóteses nula e alternativa da análise a serem testadas na análise de variância são:

- Hipótese nula (H_0): as médias dos resultados são iguais.
- Hipótese alternativa (H_1): as médias dos resultados são diferentes, ou seja, pelo menos uma das médias é diferente das demais.

Teste t de Student

O teste t de Student, ou somente teste t, é um teste de hipótese que usa conceitos estatísticos para rejeitar ou não uma hipótese nula quando a estatística de teste (t) segue uma distribuição t de Student (MARTINS, 2005).

Os Testes t são testes de hipótese úteis na estatística quando é necessário comparar médias. Pode-se comparar uma média amostral com um valor hipotético, ou com um valor alvo, usando-se um Teste t para uma amostra. Pode-se também comparar as médias de dois grupos com um Teste t para duas amostras (MARTINS, 2005).

Para esta pesquisa, usou-se o tipo de Teste t para 2 amostras, o qual testa se a diferença entre as médias de duas populações independentes é igual a um valor alvo. Quando o resultado da comparação for inferior a 5% significa que as amostras são diferentes.

6.5.1 Análise Estatística dos Resultados

Nesta seção apresenta-se um conjunto de análises estatísticas para os experimentos realizados. O objetivo principal é definir, através da variância, a semelhança entre os

Tabela 21 – Resumo dos resultados considerando os operadores essenciais (projeto Javassist, clusters de referência com 5%).

Métricas Seleccionadas / X-Means - Euclidean		Código processado / X-Means - Manhattan	
R1	V(67), N(67), X(87)	R1	V(95), N(50), X(76)
R2	V(26), N(158), X(37)	R2	V(1), N(217), X(3)
R3	V(78), N(61), X(82)	R3	V(88), N(63), X(70)
Todas as métricas / EM		Todas as métricas / X-Means - Euclidean	
R1	V(61), N(107), X(53)	R1	V(81), N(103), X(37)
R2	V(4), N(204), X(13)	R2	V(7), N(212), X(2)
R3	V(86), N(59), X(76)	R3	V(101), N(58), X(62)
Todas as métricas / X-Means - Manhattan		Métricas Seleccionadas / EM	
R1	V(70), N(91), X(60)	R1	V(79), N(76), X(66)
R2	V(27), N(173), X(21)	R2	V(18), N(181), X(22)
R3	V(62), N(50), X(109)	R3	V(90), N(57), X(74)
Métricas Seleccionadas / X-Means - Manhattan		Código Original / jPlag	
R1	V(74), N(97), X(50)	R1	V(48), N(5), X(168)
R2	V(1), N(212), X(8)	R2	V(6), N(49), X(166)
R3	V(84), N(57), X(80)	R3	V(62), N(14), X(145)
Código Original / mdist - Jaccard		Código Original / mdist - Levenshtein	
R1	V(68), N(0), X(153)	R1	V(98), N(0), X(123)
R2	V(3), N(65), X(153)	R2	V(9), N(90), X(122)
R3	V(80), N(14), X(127)	R3	V(113), N(13), X(95)
Código Original / EM		Código Original / X-Means - Euclidean	
R1	V(77), N(50), X(94)	R1	V(76), N(51), X(94)
R2	V(13), N(190), X(18)	R2	V(13), N(184), X(24)
R3	V(81), N(61), X(79)	R3	V(82), N(58), X(81)
Código Original / X-Means - Manhattan		Código processado / jPlag	
R1	V(85), N(54), X(82)	R1	V(58), N(1), X(162)
R2	V(2), N(213), X(6)	R2	V(10), N(52), X(159)
R3	V(89), N(58), X(74)	R3	V(66), N(13), X(142)
Código processado / mdist - Jaccard		Código processado / mdist - Levenshtein	
R1	V(64), N(0), X(157)	R1	V(90), N(0), X(131)
R2	V(3), N(56), X(162)	R2	V(7), N(83), X(131)
R3	V(68), N(14), X(139)	R3	V(101), N(12), X(108)
Código processado / EM		Código processado / X-Means - Euclidean	
R1	V(60), N(58), X(103)	R1	V(102), N(47), X(72)
R2	V(6), N(188), X(27)	R2	V(2), N(206), X(13)
R3	V(76), N(58), X(87)	R3	V(83), N(60), X(78)
Código processado / mdist - NCD		Código Original / mdist - NCD	
R1	V(60), N(0), X(161)	R1	V(72), N(0), X(149)
R2	V(5), N(53), X(163)	R2	V(4), N(64), X(153)
R3	V(72), N(13), X(136)	R3	V(79), N(11), X(131)

Fonte: Elaborada pelo autor.

resultados e se algumas das execuções pode ser descartada já que apresenta resultado igual à outra. Para esta análise, selecionaram-se os resultados de *Totais* e de *R3* obtidos das execuções tanto para Melhores Operadores quanto para Operadores Essenciais.

Analizando os resultados “T” para as execuções de cluster de referência padrão (20%), 10% e 5% considerando a técnica Melhores Operadores

Na Tabela 23 apresentam-se os resultados obtidos para as execuções quanto ao cluster de referência padrão (20%), com 10%, e com 5% da quantidade total de programas do

Tabela 22 – Classificação dos resultados (R_i podendo variar de 1 a 221, Operadores Essenciais, projeto Javassist, cluster de referência com 5%).

#	Configuração	R1	R2	R3	T
1	All metrics / X-Means - Euclidean	184 (83,26%)	219 (99,10%)	101 (45,70%)	504 (76,02%)
2	Selected metrics / X-Means - Manhattan	171 (77,38%)	213 (96,38%)	84 (38,01%)	468 (70,59%)
3	All metrics / EM	168 (76,02%)	208 (94,12%)	86 (38,91%)	462 (69,68%)
4	ProGuard / X-Means - Manhattan	145 (65,61%)	218 (98,64%)	88 (39,82%)	451 (68,02%)
5	Selected metrics / EM	155 (70,14%)	199 (90,05%)	90 (40,72%)	444 (66,97%)
6	Original code / X-Means - Manhattan	139 (62,90%)	215 (97,29%)	89 (40,27%)	443 (66,82%)
7	ProGuard / X-Means - Euclidean	149 (67,42%)	208 (94,12%)	83 (37,56%)	440 (66,37%)
8	All metrics / X-Means - Manhattan	161 (72,85%)	200 (90,50%)	62 (28,05%)	423 (63,80%)
9	Original code / EM	127 (57,47%)	203 (91,86%)	81 (36,65%)	411 (61,99%)
10	Original code / X-Means - Euclidean	127 (57,47%)	197 (89,14%)	82 (37,10%)	406 (61,24%)
11	Selected metrics / X-Means - Euclidean	134 (60,63%)	184 (83,26%)	78 (35,29%)	396 (59,73%)
12	ProGuard / EM	118 (53,39%)	194 (87,78%)	76 (34,39%)	388 (58,52%)
13	Original code / mdist - Levenshtein	98 (44,34%)	99 (44,80%)	113 (51,13%)	310 (46,76%)
14	ProGuard / mdist - Levenshtein	90 (40,72%)	90 (40,72%)	101 (45,70%)	281 (42,38%)
15	Original code / mdist - NCD	72 (32,58%)	68 (30,77%)	79 (35,75%)	219 (33,03%)
16	Original code / mdist - Jaccard	68 (30,77%)	68 (30,77%)	80 (36,20%)	216 (32,58%)
17	ProGuard / mdist - Jaccard	64 (28,96%)	59 (26,70%)	68 (30,77%)	191 (28,81%)
18	ProGuard / mdist - NCD	60 (27,15%)	58 (26,24%)	72 (32,58%)	190 (28,66%)
19	ProGuard / jPlag	59 (26,70%)	62 (28,05%)	66 (29,86%)	187 (28,21%)
20	Original code / jPlag	53 (23,98%)	55 (24,89%)	62 (28,05%)	170 (25,64%)

Fonte: Elaborada pelo autor.

Javassist. Estas execuções foram computadas usando-se a técnica de redução de custo Melhores Operadores. A ideia é analisar estatisticamente a variância entre os três resultados, buscando-se definir se há execuções semelhantes que possam ser suprimidas.

Tabela 23 – Resultados de T das execuções com Melhores Operadores.

Configurações	20%	10%	5%
All metrics / EM	381	375	382
All metrics / X-Means - Euclidean	405	395	397
All metrics / X-Means - Manhattan	413	407	407
Original code / EM	266	259	267
Original code / jPlag	438	420	430
Original code / mdist - Jaccard	420	411	435
Original code / mdist - Levenshtein	369	407	404
Original code / mdist - NCD	369	380	407
Original code / X-Means - Euclidean	316	308	307
Original code / X-Means - Manhattan	425	418	422
ProGuard / EM	347	341	341
ProGuard / jPlag	424	420	412
ProGuard / mdist - Jaccard	391	397	427
ProGuard / mdist - Levenshtein	384	392	409
ProGuard / mdist - NCD	385	371	418
ProGuard / X-Means - Euclidean	358	349	362
ProGuard / X-Means - Manhattan	291	288	290
Selected metrics / EM	450	445	449
Selected metrics / X-Means - Euclidean	396	388	392
Selected metrics / X-Means - Manhattan	401	398	398

Fonte: Elaborada pelo autor.

Empregando-se a técnica de Análise de Variância, obtiveram-se os resultados apresentados na Figura 10. Nota-se que a informação *valor-p* obteve valor 0.82373767 , maior que

0.05, comprovando estatisticamente que os resultados das três execuções são iguais, ou seja, considerando este resultado, não há a necessidade de alterar o tamanho do cluster de referência. Sendo assim, pode-se realizar a execução do cluster de referência em 5%, reduzindo assim o custo de processamento.

Outro fator que comprova esta igualdade é a informação da média; nota-se, nesse caso, que os valores estão próximos.

RESUMO						
Grupo	Contagem	Soma	Média	Variância		
20%	20	7629	381,45	2272,892105		
10%	20	7569	378,45	2271,734211		
5%	20	7756	387,8	2483,642105		

ANOVA						
Fonte da variação	SQ	gl	MQ	F	valor-P	F crítico
Entre grupos	911,6333333	2	455,8166667	0,194564282	0,82373767	3,158842719
Dentro dos grupos	133537,1	57	2342,75614			
Total	134448,7333	59				

Figura 10 – Resultado da Análise de Variância analisando T - Melhores Operadores.

Fonte: Elaborada pelo autor.

Analizando os resultados “R3” para as execuções de cluster de referência padrão (20%), 10% e 5% considerando a técnica Melhores Operadores

Na Tabela 24, apresentam-se os resultados obtidos para as execuções quanto ao cluster de referência padrão (20%), 10% e 5% da quantidade total de programas do Javassist. Estas execuções foram computadas usando-se a técnica de redução de custo Melhores Operadores. A ideia é analisar estatisticamente a variância entre os três resultados, buscando-se definir se há execuções semelhantes que possam ser suprimidas.

Empregando a técnica de Análise de Variância, obtiveram-se os resultados apresentados na Figura 11. Nota-se que a informação *valor-p* obteve valor 0.38449551, maior que 0.05, comprovando estatisticamente que os resultados das três execuções são iguais; ou seja, considerando-se este resultado, não há a necessidade de alterar o tamanho do cluster de referência. Sendo assim, pode-se realizar a execução do cluster de referência em 5%, reduzindo assim o custo de processamento.

Outro fator que comprova esta igualdade é a informação da média, novamente, nota-se que os valores estão próximos.

Tabela 24 – Resultados de R3 das execuções com Melhores Operadores.

Configurações	20%	10%	5%
All metrics / EM	85	79	86
All metrics / X-Means - Euclidean	94	84	86
All metrics / X-Means - Manhattan	91	85	85
Original code / EM	53	46	54
Original code / jPlag	108	95	103
Original code / mdist - Jaccard	100	93	106
Original code / mdist - Levenshtein	80	93	93
Original code / mdist - NCD	86	79	96
Original code / X-Means - Euclidean	77	69	68
Original code / X-Means - Manhattan	98	91	95
ProGuard / EM	80	74	74
ProGuard / jPlag	101	94	96
ProGuard / mdist - Jaccard	93	95	104
ProGuard / mdist - Levenshtein	89	94	101
ProGuard / mdist - NCD	89	78	99
ProGuard / X-Means - Euclidean	82	73	86
ProGuard / X-Means - Manhattan	57	54	56
Selected metrics / EM	107	102	106
Selected metrics / X-Means - Euclidean	90	82	86
Selected metrics / X-Means - Manhattan	76	73	73

Fonte: Elaborada pelo autor.

RESUMO

Grupo	Contagem	Soma	Média	Variância
20%	20	1736	86,8	202,5894737
10%	20	1633	81,65	203,6078947
5%	20	1753	87,65	244,45

ANOVA

Fonte da variação	SQ	gl	MQ	F	valor-P	F crítico
Entre grupos	421,6333333	2	210,8166667	0,972031903	0,38449551	3,158842719
Dentro dos grupos	12362,3	57	216,8824561			
Total	12783,93333	59				

Figura 11 – Resultado da Análise de Variância analisando R3 - Melhores Operadores.

Fonte: Elaborada pelo autor.

Analizando os resultados “T” para as execuções de cluster de referência padrão (20%), 10% e 5% considerando a técnica Operadores Essenciais

Na Tabela 25, apresentam-se os resultados obtidos para as execuções quanto ao cluster de referência padrão (20%), 10% e 5% da quantidade total de programas do Javassist. Estas execuções foram computadas usando-se a técnica de redução de custo Operadores

Essenciais. Analogamente às análises anteriores, a ideia é analisar estatisticamente a variância entre os três resultados, buscando-se definir se há execuções semelhantes que possam ser suprimidas.

Tabela 25 – Resultados de R3 das execuções com Operadores Essenciais.

Configurações	20%	10%	5%
All metrics / EM	465	444	462
All metrics / X-Means - Euclidean	486	493	504
All metrics / X-Means - Manhattan	433	429	423
Original code / EM	404	411	411
Original code / jPlag	181	414	170
Original code / mdist - Jaccard	358	446	216
Original code / mdist - Levenshtein	416	429	310
Original code / mdist - NCD	374	295	219
Original code / X-Means - Euclidean	387	401	406
Original code / X-Means - Manhattan	418	433	443
ProGuard / EM	371	380	388
ProGuard / jPlag	169	374	187
ProGuard / mdist - Jaccard	290	246	191
ProGuard / mdist - Levenshtein	377	235	281
ProGuard / mdist - NCD	340	277	190
ProGuard / X-Means - Euclidean	424	432	440
ProGuard / X-Means - Manhattan	433	441	451
Selected metrics / EM	433	432	444
Selected metrics / X-Means - Euclidean	391	393	396
Selected metrics / X-Means - Manhattan	472	463	468

Fonte: Elaborada pelo autor.

Os resultados obtidos com o emprego da técnica de Análise de Variância são apresentados na Figura 12. Nota-se que a informação *valor-p* obteve valor 0.3188409 , maior que 0.05 , comprovando estatisticamente que os resultados das três execuções são iguais; ou seja, considerando-se este resultado, não há a necessidade de alterar o tamanho do cluster de referência. Sendo assim, pode-se realizar a execução do cluster de referência em 5%, reduzindo assim o custo de processamento.

Novamente, outro fator que comprova esta igualdade é a informação da média; nota-se que os valores estão próximos.

Analisando os resultados “R3” para as execuções de cluster de referência 20%, 10% e 5% considerando a técnica Operadores Essenciais

Na Tabela 26, apresentam-se os resultados obtidos para as execuções quanto ao cluster de referência padrão (20%), 10% e 5% da quantidade total de programas do Javassist. Estas execuções foram computadas usando-se a técnica de redução de custo Operadores Essenciais. Dessa forma, analisa-se estatisticamente a variância entre os três resultados, buscando-se definir se há execuções semelhantes que possam ser suprimidas.

Empregando-se a técnica de Análise de Variância, obtiveram-se os resultados apresentados na Figura 13. Obteve-se 0.94777313 para *valor-p*, sendo esse valor maior que

RESUMO

Grupo	Contagem	Soma	Média	Variância
0,2	20	7622	381,1	7114,831579
0,1	20	7868	393,4	5307,2
0,05	20	7000	350	13318,10526

ANOVA

Fonte da variação	SQ	gl	MQ	F	valor-P	F crítico
Entre grupos	20013,73333	2	10006,86667	1,166295276	0,3188409	3,158842719
Dentro dos grupos	489062,6	57	8580,045614			
Total	509076,3333	59				

Figura 12 – Resultado da Análise de Variância analisando T - Operadores Essenciais.

Fonte: Elaborada pelo autor.

Tabela 26 – Resultados de R3 das execuções com Operadores Essenciais.

Configurações	20%	10%	5%
All metrics / EM	67	68	86
All metrics / X-Means - Euclidean	68	90	101
All metrics / X-Means - Manhattan	75	68	62
Original code / EM	74	81	81
Original code / jPlag	56	133	62
Original code / mdist - Jaccard	117	73	80
Original code / mdist - Levenshtein	149	71	113
Original code / mdist - NCD	125	101	79
Original code / X-Means - Euclidean	68	77	82
Original code / X-Means - Manhattan	66	79	89
ProGuard / EM	59	68	76
ProGuard / jPlag	52	120	66
ProGuard / mdist - Jaccard	97	89	68
ProGuard / mdist - Levenshtein	124	82	101
ProGuard / mdist - NCD	112	97	72
ProGuard / X-Means - Euclidean	67	75	83
ProGuard / X-Means - Manhattan	72	78	88
Selected metrics / EM	71	78	90
Selected metrics / X-Means - Euclidean	65	75	78
Selected metrics / X-Means - Manhattan	70	79	84

Fonte: Elaborada pelo autor.

0.05. Estatisticamente, comprova-se que os resultados das três execuções são iguais; ou seja, considerando-se este resultado, não há a necessidade de alterar o tamanho do cluster de referência. Sendo assim, pode-se realizar a execução do cluster de referência em 5%, reduzindo assim o custo de processamento.

Outro fator que comprova esta igualdade é a informação da média, para a qual nota-se que os valores estão próximos.

RESUMO						
Grupo	Contagem	Soma	Média	Variância		
0,2	20	1654	82,7	758,5368421		
0,1	20	1682	84,1	295,7789474		
0,05	20	1641	82,05	172,1552632		

ANOVA						
Fonte da variação	SQ	gl	MQ	F	valor-P	F crítico
Entre grupos	43,9	2	21,95	0,053690627	0,94777313	3,158842719
Dentro dos grupos	23302,95	57	408,8236842			
Total	23346,85	59				

Figura 13 – Resultado da Análise de Variância analisando R3 - Operadores Essenciais.

Fonte: Elaborada pelo autor.

Analizando os resultados de “R1” e “R3” para as execuções de cluster de referência 5% considerando a técnica Melhores Operadores

Na Tabela 27, apresentam-se os resultados obtidos para as execuções quanto ao cluster de referência composto por 5% da quantidade total de programas do Javassist. Esta execução foi computada usando-se a técnica de redução de custo Melhores Operadores. A ideia é analisar estatisticamente a variância entre os dois resultados, buscando definir se há execuções semelhantes que possam ser suprimidas. Como esta análise trabalha com dois conjuntos de valores, usa-se o Test t para realizar o cálculo de variância.

Os resultados obtidos para o Test t são apresentados na Figura 14. Nota-se que a informação *valor-p* obteve valor *0.0000006130*, menor que *0.05*, comprovando estatisticamente que o resultado dos dois conjuntos não são semelhantes. Tal resultado vem para confirmar que a criação de clusters com base em similaridade de programas vem apresentado resultados insatisfatórios.

Analizando os resultados de “R1” e “R3” para as execuções de cluster de referência 5% considerando a técnica Operadores Essenciais

Na Tabela 28, apresentam-se os resultados obtidos para as execuções quanto ao cluster de referência composto por 5% da quantidade total de programas do Javassist. Esta execução foi computada usando-se a técnica de redução de custo Operadores Essenciais. A ideia é analisar estatisticamente a variância entre os dois resultados, buscando definir se há execuções semelhantes que possam ser suprimidas. Como esta análise considera dois conjunto de valores, usa-se o Test t para realizar o cálculo de variância.

Tabela 27 – Resultados de R1 e R3 das execuções com Melhores Operadores

Configurações	R1	R3
All metrics / EM	122	86
All metrics / X-Means - Euclidean	129	86
All metrics / X-Means - Manhattan	136	85
Original code / EM	80	54
Original code / jPlag	155	103
Original code / mdist - Jaccard	157	106
Original code / mdist - Levenshtein	144	93
Original code / mdist - NCD	151	96
Original code / X-Means - Euclidean	93	68
Original code / X-Means - Manhattan	107	95
ProGuard / EM	100	74
ProGuard / jPlag	149	96
ProGuard / mdist - Jaccard	154	104
ProGuard / mdist - Levenshtein	141	101
ProGuard / mdist - NCD	150	99
ProGuard / X-Means - Euclidean	90	86
ProGuard / X-Means - Manhattan	95	56
Selected metrics / EM	157	106
Selected metrics / X-Means - Euclidean	145	86
Selected metrics / X-Means - Manhattan	147	73

Fonte: Elaborada pelo autor.

	R1	R3
Média	130,1	87,65
Variância	679,7789474	244,45
Observações	20	20
Hipótese da diferença	0	
gl	31	
Stat t	6,244575331	
P(T<=t) uni-caudal	0,00000030649	
t crítico uni-caudal	1,695518783	
P(T<=t) bi-caudal	0,0000006130	
t crítico bi-caudal	2,039513446	

Figura 14 – Resultado do Test t considerando-se R1 e R3 - Melhores Operadores.

Fonte: Elaborada pelo autor.

Os resultados obtidos para o Test t são apresentados na Figura 15. Note-se que a informação *valor-p* obteve valor *0.004256359*, menor que *0.05*, comprovando estatisticamente que o resultado dos dois conjuntos não são semelhantes. Tal resultado vem para confirmar que a criação de clusters com base em similaridade de programas vem apresentado resultados insatisfatórios.

6.6 Considerações Finais

Tabela 28 – Resultados de R1 e R3 das execuções com Conjunto Essencial

Configurações	R1	R3
All metrics / EM	168	86
All metrics / X-Means - Euclidean	184	101
All metrics / X-Means - Manhattan	161	62
Original code / EM	127	81
Original code / jPlag	53	62
Original code / mdist - Jaccard	68	80
Original code / mdist - Levenshtein	98	113
Original code / mdist - NCD	72	79
Original code / X-Means - Euclidean	127	82
Original code / X-Means - Manhattan	139	89
ProGuard / EM	118	76
ProGuard / jPlag	59	66
ProGuard / mdist - Jaccard	64	68
ProGuard / mdist - Levenshtein	90	101
ProGuard / mdist - NCD	60	72
ProGuard / X-Means - Euclidean	149	83
ProGuard / X-Means - Manhattan	145	88
Selected metrics / EM	155	90
Selected metrics / X-Means - Euclidean	134	78
Selected metrics / X-Means - Manhattan	171	84

Fonte: Elaborada pelo autor.

	168	86
Média	114,4210526	81,8421053
Variância	1804,146199	180,807018
Observações	20	20
Hipótese da diferen	0	
gl	22	
Stat t	3,187415725	
P(T<=t) uni-caudal	0,002128179	
t crítico uni-caudal	1,717144374	
P(T<=t) bi-caudal	0,004256359	
t crítico bi-caudal	2,073873068	

Figura 15 – Resultado do Test t considerando-se R1 e R3 - Operadores Essenciais.

Fonte: Elaborada pelo autor.

Neste capítulo exploraram-se as técnicas de redução de custo Melhores Operadores e Operadores Essenciais, no contexto da abordagem de redução de custo do teste de mutação com base em similaridade de programas embutida no framework *SiMut*. Ressalta-se que a técnica Operadores Essenciais foi revisitada e automatizada no contexto deste trabalho para viabilizar a realização de experimentos com o framework *SiMut*. Resultados foram gerados, apresentados analiticamente, e avaliados estatisticamente.

Na Seção 6.4, as questões de pesquisas propostas foram respondidas, e após a conclusão das análises estatísticas, as mesmas são retomadas aqui para concluir as respectivas respos-

tas. Mais especificamente, mantém-se as conclusões já apresentadas para **RQ1.1**, **RQ1.2** e **RQ2**, agora apoiando-se nos resultados da análise estatística realizada. Independentemente do tamanho do cluster de referência e da técnica de redução de custo empregada, os resultados não são satisfatórios quando comparados a resultados decorrentes do emprego de clusters de referência aleatoriamente formados.

Capítulo 7

Conclusão

Nesta tese, apresenta-se um conceito de framework denominado *SiMut*, posteriormente implementado. Esta implementação objetivou a redução de custo do teste de mutação fazendo uso de grupos de programas já testados. A implementação permitiu a inclusão de diversas formas de computar abstrações e similaridade entre estes programas. Utilizaram-se e implementaram-se as técnicas de Melhores Operadores e Operadores Essenciais, apoiados pela base de dados também concebida e implementada, permitindo a definição dos escores de mutação para as classes dos programas considerados nos experimentos realizados.

Para as experimentações foram utilizados dois conjuntos de programas, Simpleprograms e Javassist com 35 e 221 arquivos de código fonte Java (cada um contendo uma ou mais classes), respectivamente. Os resultados de experimentos geraram várias combinações efetivas para prever os Melhores Operadores e Conjunto Essencial de operadores para classes não testadas. Foram utilizadas 20 combinações variadas de abstrações de programas e métricas de similaridade entre programas, inclusive com variação para o tamanho do *cluster* de referência da classe não testada. Apresentaram-se um conjunto de análise e comparações para os resultados, objetivando comprovar uma melhor efetividade das abstrações e similaridades utilizadas nas configurações, mesmo em alguns casos constando a pouca efetividade de algumas abstrações.

Diante de todas as análises realizadas nos resultados gerados, pode-se concluir que o uso de similaridade na geração dos *clusters* de referências não é vantajoso para a definição tanto dos melhores operadores quanto para o conjunto essencial de operadores, tendo em vista que os *clusters* gerados aleatoriamente apresentam resultados iguais ou superiores aos *clusters* gerados usando-se a abordagem de cálculo de similaridade entre programas explorada neste trabalho.

Dentre as motivações deste trabalho, há a crença de que os resultados em pesquisas de redução de custo em teste de mutação estão relacionados aos artefatos utilizados nos experimentos. Tal fato apoiou o uso de similaridades juntamente para desmistificar esta crença, mas os experimentos realizados e seus resultados obtidos concluem que ainda não conseguimos derrubar tal afirmação mantendo esta crença ativa, mesmo com toda a evolução e adequação das pesquisas anteriores (DALLILO; PIZZOLETEO; FERRARI, 2019; PIZZOLETEO et al., 2020; GUARNIERI; PIZZOLETEO; FERRARI, 2022).

7.1 Contribuições

Comparado com pesquisas anteriores (DALLILO; PIZZOLETEO; FERRARI, 2019; PIZZOLETEO et al., 2020; GUARNIERI; PIZZOLETEO; FERRARI, 2022) e com o estado da arte em teste de mutação, este trabalho traz as seguintes contribuições:

- ❑ Modelo conceitual para o framework *SiMut*, posteriormente desenvolvido pelo grupo de pesquisa e utilizado em outros trabalhos de pós-graduação.
- ❑ Definição, projeto, criação, população e validação de uma base de dados composta de programas de pequeno e grande porte, e de complexidades variadas, contendo: mutantes, casos de teste, relação entre mutantes e casos de teste, operadores de mutação que geraram cada um dos mutantes, dentre outras informações.
- ❑ Método para população da base de dados, ilustrando as fases para definição dos mutantes, casos de teste, geração das killingmatrix, e importação dos dados para a base de dados.
- ❑ Revisão sistemática da literatura que apresenta o estado da arte com relação a redução do custo do teste de mutação, na qual fora criado um mapa da evolução e utilização de técnicas que buscam a redução dos custos na aplicação deste teste, com métricas associadas.
- ❑ Implementação da técnica de Operadores Essenciais de operadores de mutação 100% automatizada, sem a necessidade de interferência humana no processo de geração dos conjuntos essenciais.
- ❑ Experimentação com 20 configurações para o framework *SiMut*, para os programas contidos em dois projetos de portes diferentes (SimplePrograms e Javassist), com a predição dos melhores operadores quanto aos escores de mutação, e dos operadores essenciais para programas não testados.

7.2 Ameaças à validade

A condução de estudos empíricos na área de engenharia de software, especialmente aqueles que envolvem análise de similaridade e testes automatizados, demanda uma atenção rigorosa à mitigação de riscos, a fim de garantir a validade e a confiabilidade dos resultados. Este texto apresenta uma discussão aprofundada sobre os riscos inerentes a tais estudos e as estratégias adotadas para minimizá-los, abordando as dimensões de construção, validade interna e externa, e confiabilidade.

Construção

Superando as Limitações das Métricas de Similaridade e da Geração de Casos de Teste

A utilização de métricas de similaridade, embora fundamental para a análise de dados, pode introduzir vieses e distorções nos resultados. Para mitigar esse risco, adotamos métricas amplamente reconhecidas e validadas na literatura, como EM e X-Means, implementadas na plataforma Weka. A escolha dessas métricas baseia-se em sua robustez e em sua aplicação bem-sucedida em diversos estudos empíricos.

A geração de um conjunto de casos de teste abrangente e representativo é crucial para a avaliação da eficácia de técnicas de teste automatizadas. Para garantir a qualidade e a eficiência desse conjunto, empregamos a ferramenta Evosuite, reconhecida por sua capacidade de gerar testes de alta cobertura e por sua ampla utilização na comunidade de pesquisa.

Validade Interna

Controlando a Influência de Fatores Aleatórios

A precisão dos resultados de um estudo empírico pode ser comprometida por fatores aleatórios, como variações no ambiente de execução ou na ordem de aplicação das técnicas avaliadas. Para mitigar esse risco, adotamos a estratégia de repetir a execução do estudo diversas vezes, permitindo a análise estatística dos resultados e a identificação de possíveis outliers.

Validade Externa

Buscando a Generalização dos Resultados A generalização dos resultados para programas de software diversos é um desafio inerente a estudos empíricos. Para abordar essa questão, realizamos o estudo em um conjunto de programas de tamanhos e complexidades variadas, incluindo tanto programas pequenos quanto grandes. No entanto, reconhecemos a necessidade de expandir o escopo do estudo para incluir um conjunto ainda mais diversificado de programas, a fim de fortalecer a validade externa dos resultados.

Confiabilidade

Facilitando a Replicação e a Validação por Outros Pesquisadores

A confiabilidade de um estudo empírico depende da sua capacidade de ser replicado por outros pesquisadores. Para garantir a transparência e a reprodutibilidade do nosso estudo, detalhamos minuciosamente todo o processo utilizado em cada etapa, desde a seleção das métricas e ferramentas até a análise dos resultados. Além disso, disponibilizamos publicamente a base de conhecimento, os protótipos e os resultados do estudo, permitindo que outros pesquisadores possam validar e expandir nossos achados.

Pode-se mencionar que a adoção de uma abordagem metodológica rigorosa, que inclua a seleção criteriosa de métricas e ferramentas, a repetição de execuções, a diversificação do conjunto de programas analisados e a disponibilização dos materiais do estudo, é fundamental para mitigar os riscos inerentes a estudos empíricos na área de engenharia de software e para garantir a validade e a confiabilidade dos resultados.

7.3 Limitações

Como toda pesquisa e resultados possuem algumas limitações, também é necessário apresentar as que podem influenciar nos resultados. Na sequência, listam-se tais limitações.

- ❑ O grau de confiabilidade dos resultados coletados estão limitados ao funcionamento correto das ferramentas acopladas (ProGuard, mdist, Weka, JPlag e CKJM extended) ao *framework* implementado.
- ❑ A efetividade dos resultados de teste de mutação coletados nos experimentos também se limita à qualidade da base de dados de artefatos (PIZZOLETO; GUARNIERI; FERRARI, 2021) utilizada neste trabalho.
- ❑ O tratamento de todos os mutantes, desconsiderando a existência de mutantes equivalentes, os quais podem influenciar nos escores de mutação gerados e, consequentemente, em sua classificação e seleção nos experimentos.
- ❑ Até a data da escrita deste trabalho, o módulo responsável pela redução de custo e do Conjunto Essencial não funcionam em outros sistemas operacionais além do Microsoft Windows.

7.4 Trabalhos Futuros

Esta tese de doutorado apresentou diversos conceitos, ferramentas, base de conhecimento e análises que auxiliam outros pesquisadores a continuarem os trabalhos aqui

descritos, fazendo uso de outras técnicas de redução de custo ou uso de novos artefatos. Mas todo este conhecimento listado não deve ser esquecido e sim atualizado e evoluído. Para isso destacam-se algumas possibilidades de trabalhos futuros substanciais:

- ❑ Aplicação da base de dados/conhecimento incluindo novos programas de complexidade elevada e de outras linguagens; inclusão de novos mutantes gerados por outras ferramentas e novos casos de teste, automatizados ou não.
- ❑ Exploração de outras técnicas de redução de custo, incluindo métricas não atreladas a orientação à objetos, tendo em vista que nesta pesquisa empregaram-se as técnicas Melhores Operadores e Operadores Essenciais. Ressalta-se que há outras que se destacam e são listadas no capítulo da Revisão Sistemática. As ferramentas são fortemente acopláveis.
- ❑ Evolução do *SiMut* integrando todos as ferramentas auxiliares que foram desenvolvidas para auxílio, deixando assim todo o processo integrado.
- ❑ Refinamento dos mutantes incluídos na base de dados, identificando os mutantes equivalentes e os mutantes vivos, deixando assim a base mais completa e permitindo outros tipos de pesquisas.

Referências

ABULJADAYEL, A.; WEDYAN, F. An Approach for the Generation of Higher Order Mutants Using Genetic Algorithms. **International Journal of Intelligent Systems and Applications**, Modern Education and Computer Science Press, Hong Kong, China, v. 10, n. 1, p. 34–45, 2018.

ACREE, A. T. et al. **Mutation Analysis**. Atlanta/GA - USA, 1979.

ADAMOPOULOS, K.; HARMAN, M.; HIERONS, R. M. How to Overcome the Equivalent Mutant Problem and Achieve Tailored Selective Mutation Using Co-evolution. In: **Proceedings of the 3rd Genetic and Evolutionary Computation Conference (GECCO)**. Seattle, WA, USA: Springer, 2004. p. II-1338–1349 (LNCS v.3103).

AGRAWAL, H. et al. **Design of Mutant Operators for the C Programming Language**. West Lafayette/IN - USA, 1989.

AICHERNIG, B. K. et al. Model-Based Mutation Testing of an Industrial Measurement Device. In: **Proceedings of the 8th International Conference Tests and Proofs (TAP)**. York, UK: Springer, 2014. p. 1–19. ISBN 978-3-319-09099-3.

AICHERNIG, B. K.; JöBSTL, E.; KEGELE, M. Incremental refinement checking for test case generation. In: **Proceedings of the 7th International Conference Tests and Proofs (TAP)**. Budapest, Hungary: Springer, 2013. p. 1–19. ISBN 978-3-642-38916-0.

AL-HAJJAJI, M. et al. Efficient Mutation Testing in Configurable Systems. In: **Proceedings of the IEEE/ACM 2nd International Workshop on Variability and Complexity in Software Design (VACE)**. Buenos Aires, Argentina: IEEE, 2017. p. 2–8.

ALEXANDER, R. T. et al. Mutation of Java Objects. In: **Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE)**. Annapolis, MD, USA: IEEE, 2002. p. 341–351. ISSN 1071-9458.

AMMANN, P.; DELAMARO, M. E.; OFFUTT, A. J. Establishing Theoretical Minimal Sets of Mutants. In: **Proceedings of the 7th Conference on Software Testing, Verification and Validation (ICST)**. Cleveland, OH, USA: IEEE, 2014. p. 21–30.

ANBALAGAN, P.; XIE, T. Automated Generation of Pointcut Mutants for Testing Pointcuts in AspectJ Programs. In: **Proceeding of the 19th International**

Symposium on Software Reliability Engineering (ISSRE). [S.l.]: IEEE, 2008. p. 239–248.

ANDREWS, J. H.; BRIAND, L. C.; LABICHE, Y. Is Mutation an Appropriate Tool for Testing Experiments? In: **Proceedings of the 27th International Conference on Software Engineering (ICSE)**. St. Louis, MO, USA: ACM, 2005. p. 402–411. ISBN 1-59593-963-2.

AYARI, K.; BOUKTIF, S.; ANTONIOL, G. Automatic Mutation Test Input Data Generation via Ant Colony. In: **Proceedings of the 9th Genetic and Evolutionary Computation Conference (GECCO)**. London, UK: ACM, 2007. p. 1074–1081.

BANSIYA, J. Class cohesion metric for object oriented designs. **Journal of Object-Oriented Programming**, v. 11, n. 8, p. 47–52, 1999.

BANSIYA, J.; DAVIS, C. A hierarchical model for object-oriented design quality assessment. **IEEE Transactions on Software Engineering**, v. 28, n. 1, p. 4–17, 2002.

BARBOSA, E. F.; MALDONADO, J. C.; VINCENZI, A. M. R. Toward the Determination of Sufficient Mutant Operators for C. **Software Testing, Verification and Reliability**, John Wiley & Sons, Malden, MA, USA, v. 11, n. 2, p. 113–136, 2001.

BARBOSA, E. F. et al. Anais da 4^o escola regional de informática de minas gerais. In: _____. Belo Horizonte/MG - Brasil: Sociedade Brasileira de Computação. (Org.), 2005. cap. Teste Estrutural e de Mutação no Contexto de Programas OO, p. 313–362.

BASHIR, M. B.; NADEEM, A. Improved Genetic Algorithm to Reduce Mutation Testing Cost. **IEEE Access**, IEEE, Piscataway, NJ, USA, v. 5, p. 3657–3674, 2017. ISSN 2169-3536.

BAUDRY, B. et al. From Genetic to Bacteriological Algorithms for Mutation-Based Testing. **Software Testing, Verification and Reliability**, John Wiley & Sons, Malden, MA, USA, v. 15, n. 2, p. 73–96, 2005.

BELLI, F.; BEYAZIT, M. Exploiting Model Morphology for Event-Based Testing. **IEEE Transactions on Software Engineering**, IEEE, v. 41, n. 2, p. 113–134, 2015.

BLUEMKE, I.; KULESZA, K. Reduction of Computational Cost in Mutation Testing by Sampling Mutants. In: **Proceedings of the 8th International Conference on Dependability and Complex Systems (DepCoS-RELCOMEX)**. Brunów, Poland: Springer, 2013. p. 41–51.

_____. Reduction in Mutation Testing of Java Classes. In: **Proceedings of the 9th International Conference on Software Engineering and Applications (ICSOFT-EA)**. Vienna, Austria: IEEE, 2014. p. 297–304.

_____. Reductions of Operators in Java Mutation Testing. In: **Proceedings of the 9th International Conference on Dependability and Complex Systems (DepCoS-RELCOMEX)**. Brunów, Poland: Springer, 2014. p. 93–102. ISBN 978-3-319-07013-1.

- BOGACKI, B.; WALTER, B. Aspect-oriented response injection: an alternative to classical mutation testing. In: **Proceedings of the IFIP Working Conference on Software Engineering Techniques: Design for Quality (SET)**. Warsaw, Poland: Springer, 2006. p. 273–282. ISBN 978-0-387-39388-9.
- BRADBURY, J.; CORDY, J.; DINGEL, J. Mutation operators for concurrent Java (J2SE 5.0). In: **Proceedings of the 2nd Workshop on Mutation Analysis (Mutation) - held in conjunction with ISSRE**. Raleigh/NC -USA: Kluwer Academic Publishers, 2006.
- BYOUNGJU, C.; MATHUR, A. P. High-Performance Mutation Testing. **Journal of Systems and Software**, Elsevier, v. 20, n. 2, p. 135 – 152, 1993.
- CACHIA, M. A.; MICALLEF, M.; COLOMBO, C. Towards Incremental Mutation Testing. In: **Proceedings of the 2013 Validation Strategies for Software Evolution Workshop (VSSE)**. Rome, Italy: Elsevier, 2013. p. 2–11.
- Capgemini Group; Sogeti; HP. **World Quality Report 2015-16**. [S.l.], 2016. <<https://www.capgemini.com/thought-leadership/world-quality-report-2015-16>> (accessado em 29/11/2016).
- CHEN, L.; ZHANG, L. Speeding up Mutation Testing via Regression Test Selection: An Extensive Study. In: **Proceedings of the 11th International Conference on Software Testing, Verification and Validation (ICST)**. Västerås, Sweden: IEEE, 2018. p. 58–69.
- CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object oriented design. **IEEE Transactions on Software Engineering**, IEEE, v. 20, n. 6, 1994. ISSN 0098-5589.
- CILIBRASI, R.; VITÁNYI, P. M. Clustering by compression. **IEEE Transactions on Information theory**, IEEE, v. 51, n. 4, p. 1523–1545, 2005.
- COUSOT, P.; COUSOT, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: **Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages**. New York, NY, USA: Association for Computing Machinery, 1977. (POPL '77), p. 238–252. ISBN 9781450373500. Disponível em: <<https://doi.org/10.1145/512950.512973>>.
- DALLILO, L. D.; PIZZOLETO, A. V.; FERRARI, F. C. An Evaluation of Internal Program Metrics as Predictors of Mutation Operator Score. In: **SAST'19**. [S.l.: s.n.], 2019. ISBN 978-1-4503-6555-0.
- DELAMARO, M. E. et al. Parallel execution of programs as a support for mutation testing: A replication study. **jseke**, IEEE, v. 31, n. 03, p. 337–380, 2021.
- _____. Experimental Evaluation of SDL and One-Op Mutation for C. In: **Proceedings of the 7th International Conference on Software Testing, Verification and Validation (ICST)**. Cleveland, OH, USA: IEEE, 2014. p. 203–212.
- _____. Growing a Reduced Set of Mutation Operators. In: **Proceedings of the 28th Brazilian Symposium on Software Engineering (SBES)**. Maceió, AL, Brazil: IEEE, 2014. p. 81–90.

- DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. **Introdução ao teste de software**. [S.l.]: Elsevier, 2007.
- DELAMARO, M. E.; OFFUTT, A. J.; AMMANN, P. Designing Deletion Mutation Operators. In: **Proceedings of the 7th International Conference on Software Testing, Verification and Validation (ICST)**. Cleveland, OH, USA: IEEE, 2014. p. 11–20.
- DELGADO-PÉREZ, P.; MEDINA-BULO, I.; MERAYO, M. G. Using Evolutionary Computation to Improve Mutation Testing. In: **Proceedings of the 14th International Work-Conference on Artificial Neural Networks (IWANN)**. Cadiz, Spain: Springer, 2017. p. 381–391. ISBN 978-3-319-59147-6.
- DELGADO-PÉREZ, P. et al. Assessment of Class Mutation Operators for C++ with the MuCPP Mutation System. **Information and Software Technology**, Elsevier, Butterworth-Heinemann Newton, MA, USA, p. 169–184, 2017. ISSN 0950-5849.
- _____. GiGAn: Evolutionary Mutation Testing for C++ Object-oriented Systems. In: **Proceedings of the 32nd ACM Symposium on Applied Computing (SAC)**. Marrakech, Morocco: ACM, 2017. p. 1387–1392. ISBN 978-1-4503-4486-9.
- DELGADO-PÉREZ, P.; SEGURA, S.; MEDINA-BULO, I. Assessment of C++ Object-Oriented Mutation Operators: A Selective Mutation Approach. **Software Testing, Verification and Reliability**, John Wiley & Sons, Malden, MA, USA, v. 27, n. 4-5, p. 1630 – 1649, 2017. ISSN 1099-1689.
- DEMILLO, R. A.; KRAUSER, E. W.; MATHUR, A. P. Compiler-integrated Program Mutation. In: **Proceedings of the 15th IEEE Annual Computer Software and Applications Conference (COMPSAC)**. Tokyo, Japan: IEEE, 1991. p. 351–356. ISBN 0-8186-2152-4.
- DEMILLO, R. A.; LIPTON, R. J.; SAYWARD, F. G. Hints on Test Data Selection: Help for the Practicing Programmer. **IEEE Computer**, IEEE, v. 11, n. 4, p. 34–43, 1978.
- DEMILLO, R. A.; OFFUTT, A. J. Constraint-based Automatic Test Data Generation. **IEEE Transactions on Software Engineering**, IEEE, v. 17, n. 9, p. 900–910, September 1991.
- _____. Experimental Results from an Automatic Test Case Generator. **ACM Transactions on Software Engineering and Methodology**, ACM, New York, NY, USA, v. 2, n. 2, p. 109–127, 1993.
- DEMPSTER, A. P.; LAIRD, N. M.; RUBIN, D. B. Maximum likelihood from incomplete data via the EM algorithm. **Journal of the Royal Statistical Society: Series B (Methodological)**, Wiley Online Library, v. 39, n. 1, p. 1–22, 1977.
- DENG, L. et al. Towards mutation analysis of android apps. In: **Proceedings of the 8th International Conference on Software Testing, Verification and Validation Workshops (ICSTW)**. Graz - Austria: IEEE Press, 2015. p. 1–10.

- DENG, L.; OFFUTT, A. J.; LI, N. Empirical Evaluation of the Statement Deletion Mutation Operator. In: **Proceedings of the 6th International Conference on Software Testing, Verification and Validation (ICST)**. Luxembourg City, Luxembourg: IEEE, 2013. p. 84–93.
- DENISOV, A.; PANKEVICH, S. Mull It Over: Mutation Testing Based on LLVM. In: **Proceedings of the 13th International Workshop on Mutation Analysis (Mutation)**. Västerås, Sweden: IEEE, 2018. p. 25–31.
- DEREZIŃSKA, A. A Quality Estimation of Mutation Clustering in C# Programs. In: **Proceedings of the 8th International Conference on Dependability and Complex Systems (DepCoS-RELCOMEX)**. Brunów, Poland: Springer, 2013. p. 119–129 (AISC v.224).
- DEREZIŃSKA, A. Evaluation of Deletion Mutation Operators in Mutation Testing of C# Programs. In: **Proceedings of the 11th International Conference on Dependability and Complex Systems (DepCoS-RELCOMEX)**. Brunów, Poland: Springer, 2016. p. 97–108. ISBN 978-3-319-39639-2.
- DEREZIŃSKA, A.; HAŁAS, K. Improving Mutation Testing Process of Python Programs. In: **Proceedings of the 4th Computer Science On-line Conference Software Engineering in Intelligent Systems (CSOC)**. [S.l.]: Springer, 2015. p. 233–242. ISBN 978-3-319-18473-9.
- DEREZIŃSKA, A.; RUDNIK, M. Quality Evaluation of Object-Oriented and Standard Mutation Operators Applied to C# Programs. In: **Proceedings of the 50th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS)**. Prague, Czech Republic: Springer, 2012. p. 42–57. ISBN 978-3-642-30561-0.
- _____. Evaluation of Mutant Sampling Criteria in Object-Oriented Mutation Testing. In: **Proceedings of the Federated Conference on Computer Science and Information Systems (ACISIS)**. Prague, Czech Republic: Polskie Towarzystwo Informatyczne, 2017. p. 1315–1324. ISBN 978-83-946253-7-5.
- DEVROEY, X. et al. Featured Model-based Mutation Analysis. In: **Proceedings of the 38th International Conference on Software Engineering (ICSE)**. Austin, TX, USA: ACM, 2016. p. 655–666. ISBN 978-1-4503-3900-1.
- _____. Automata Language Equivalence vs. Simulations for Model-Based Mutant Equivalence: An Empirical Evaluation. In: **Proceedings of the 10th IEEE International Conference on Software Testing, Verification and Validation (ICST)**. Tokyo, Japan: IEEE, 2017. p. 424–429.
- DOMÍNGUEZ-JIMÉNEZ, J. J. et al. Evolutionary Mutation Testing. **Information and Software Technology**, Elsevier, Butterworth-Heinemann Newton/MA - USA, v. 53, p. 1108–1123, 2011.
- DOMÍNGUEZ-JIMÉNEZ, J. J.; ESTERO-BOTARO, A.; MEDINA-BULO, A. G.-D. and I. Gamera: An Automatic Mutant Generation System for WS-BPEL Compositions. In: **Proceedings of the 7th IEEE European Conference on Web Services (ECOWS)**. Eindhoven, The Netherlands: IEEE, 2009. p. 97–106.

DURELLI, V. H. S.; OFFUTT, A. J.; DELAMARO, M. E. Toward Harnessing High-Level Language Virtual Machines for Further Speeding Up Weak Mutation Testing. In: **Proceedings of the 5th International Conference on Software Testing, Verification and Validation (ICST)**. Montreal, QC, Canada: IEEE, 2012. p. 681–690.

FELDT, R. et al. Test set diameter: Quantifying the diversity of sets of test cases. In: IEEE. **2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)**. [S.l.], 2016. p. 223–233.

FERNANDES, L. et al. Avoiding Useless Mutants. In: **Proceedings of the 16th ACM International Conference on Generative Programming: Concepts and Experiences (GPCE)**. Vancouver, BC, Canada: ACM, 2017. p. 187–198. ISBN 978-1-4503-5524-7.

FERRARI, F. C.; MALDONADO, J. C.; RASHID, A. Mutation testing for aspect-oriented programs. In: **Proceedings of the 1st International Conference on Software Testing, Verification and Validation (ICST)**. Lillehammer - Norway: IEEE Press, 2008. p. 52–61. ISBN 978-0-7695-3127-4.

FERRARI, F. C. et al. Automating the mutation testing of aspect-oriented Java programs. In: **Proceedings of the 5th ICSE International Workshop on Automation of Software Test (AST)**. Cape Town - South Africa: ACM Press, 2010. p. 51–58. ISBN 978-1-60558-970-1.

FERRARI, F. C.; PIZZOLETE, A. V.; OFFUTT, A. J. A Systematic Review of Cost Reduction Techniques for Mutation Testing: Preliminary Results. In: **Proceedings of the 13th International Workshop on Mutation Analysis (Mutation)**. Västerås, Sweden: IEEE, 2018. p. 1–10.

FERRARI, F. C.; RASHID, A.; MALDONADO, J. C. Towards the Practical Mutation Testing of AspectJ Programs. **Science of Computer Programming**, Elsevier, Amsterdam, The Netherlands, v. 78, n. 9, p. 1639–1662, 2013.

FLEYSHGAKKER, V. N.; WEISS, S. N. Efficient Mutation Analysis: A New Approach. In: **Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)**. Seattle, WA, USA: ACM, 1994. p. 185–195.

FRASER, G.; ARCURI, A. Achieving Scalable Mutation-based Generation of Whole Test Suites. **Empirical Software Engineering**, Springer, v. 20, n. 3, p. 783–812, 2015.

FRASER, G.; ZELLER, A. Mutation-Driven Generation of Oracles and Unit Tests. **IEEE Transactions on Software Engineering**, v. 38, n. 2, p. 278–292, 2012.

GARVIN, B.; COHEN, M.; DWYER, M. Using feature locality: Can we leverage history to avoid failures during reconfiguration? In: **Proceedings of the 8th Workshop on Assurances for Self-adaptive Systems (ASAS)**. Szeged, Hungary: ACM Press, 2011. p. 24–33.

GLIGORIC, M. et al. Efficient Mutation Testing of Multithreaded Code. **Software Testing, Verification and Reliability**, John Wiley & Sons, Malden, MA, USA, v. 23, n. 5, p. 375–403, 2012. ISSN 1099-1689.

- GLIGORIC, M.; JAGANNATH, V.; MARINOV, D. MuTMuT: Efficient Exploration for Mutation Testing of Multithreaded Code. In: **Proceedings of the 3th International Conference on Software Testing, Verification and Validation (ICST)**. Paris, France: IEEE, 2010. p. 55–64.
- GLIGORIC, M. et al. Selective Mutation Testing for Concurrent Code. In: **Proceedings of the 22nd International Symposium on Software Testing and Analysis (ISSTA)**. Lugano, Switzerland: ACM, 2013. p. 224–234.
- GOPINATH, R. et al. Mutation Reduction Strategies Considered Harmful. **IEEE Transactions on Software Reliability**, IEEE, Piscataway, NJ, USA, v. 66, n. 3, p. 854–874, 2017. ISSN 0018-9529.
- GOPINATH, R.; JENSEN, C.; GROCE, A. Topsy-Turvy: A Smarter and Faster Parallelization of Mutation Analysis. In: **Proceedings of the 38th International Conference on Software Engineering Companion (ICSE)**. Austin, TX, USA: ACM, 2016. p. 740–743. ISBN 978-1-4503-4205-6.
- GOPINATH, R.; MATHIS, B.; ZELLER, A. If You Can't Kill a Supermutant, You Have a Problem. In: **Proceedings of the 13th International Workshop on Mutation Analysis (Mutation)**. Västerås, Sweden: IEEE, 2018. p. 18–24.
- GUARNIERI, G. F. **SiMut: Um Framework Automatizado para Apoiar a Redução do Custo do Teste de Mutação com Base em Similaridade de Programas**. Dissertação (Mestrado) — Departamento de Computação, Universidade Federal de São Carlos (UFSCar), Sorocaba, SP, Brasil, 2022.
- GUARNIERI, G. F.; PIZZOLETEO, A. V.; FERRARI, F. C. An Automated Framework for Cost Reduction of Mutation Testing Based on Program Similarity. In: **17th International Workshop on Mutation Analysis (Mutation)**. Online Conference: IEEE, 2022. p. 179–188. ISSN 2159-4848.
- HARMAN, M.; HIERONS, R. M.; DANICIC, S. The Relationship Between Program Dependence and Mutation Analysis. In: **Proceedings of the Mutation 2000 Symposium**. San Jose, CA, USA: Kluwer Academic Publishers, 2000. p. 5–13. ISSN 1386-2944.
- HARMAN, M. et al. Angels and Monsters: An Empirical Investigation of Potential Test Effectiveness and Efficiency Improvement from Strongly Subsuming Higher Order Mutation. In: **Proceedings of the 29th International Conference on Automated Software Engineering (ASE)**. Vasteras, Sweden: ACM, 2014. p. 397–408.
- HENARD, C.; PAPADAKIS, M.; Le Traon, Y. Mutation-Based Generation of Software Product Line Test Configurations. In: **Proceedings of the 6th International Symposium Search-Based Software Engineering (SSBSE)**. Fortaleza, CE, Brazil: Springer, 2014. p. 92–106. ISBN 978-3-319-09940-8.
- HENDERSON-SELLERS, B. **Object-oriented metrics: measures of complexity**. [S.l.]: Prentice-Hall, Inc., 1995.
- HIERONS, R.; HARMAN, M.; DANICIC, S. Using program slicing to assist in the detection of equivalent mutants. **Software Testing, Verification and Reliability**, John Wiley & Sons, v. 9, n. 4, p. 233–262, 1999.

- HIERONS, R. M.; HARMAN, M.; DANICIC, S. Using Program Slicing to Assist in the Detection of Equivalent Mutants. **Software Testing, Verification and Reliability**, John Wiley & Sons, Malden, MA, USA, v. 9, n. 4, p. 233–262, 1999.
- HOLLING, D. et al. Nequivack: Assessing Mutation Score Confidence. In: **Proceedings of the 11th International Workshop on Mutation Analysis (Mutation)**. Chicago, IL, USA: IEEE, 2016. p. 152–161.
- HOWDEN, W. E. Weak Mutation Testing and Completeness of Test Sets. **IEEE Transactions on Software Engineering**, IEEE, Piscataway/NJ - USA, v. 8, n. 4, p. 371–379, 1982. ISSN 0098-5589.
- HU, J.; LI, N.; OFFUTT, A. J. An Analysis of OO Mutation Operators. In: **Proceedings of the 6th International Workshop on Mutation Analysis (Mutation)**. Berlin, Germany: IEEE, 2011. p. 334–341.
- IEEE. **IEEE 829 Standard for Software and System Test Documentation**. New York/NY - USA, 1990.
- IIDA, C.; TAKADA, S. Reducing Mutants with Mutant Killable Precondition. In: **Proceedings of the 12th International Workshop on Mutation Analysis (Mutation)**. Tokyo, Japan: IEEE, 2017. p. 128–133.
- INOZEMTSEVA, L.; HEMMATI, H.; HOLMES, R. Using Fault History to Improve Mutation Reduction. In: **Proceedings of the 9th Joint Meeting on Foundations of Software Engineering: New Ideas Track (ESEC/FSE)**. Saint Petersburg, Russia: ACM, 2013. p. 639–642.
- JACCARD, P. Étude comparative de la distribution florale dans une portion des alpes et des jura. **Bull Soc Vaudoise Sci Nat**, v. 37, p. 547–579, 1901.
- JACKSON, D.; WOODWARD, M. R. Parallel Firm Mutation of Java Programs. In: **Proceedings of the Mutation 2000 Symposium**. San Jose, CA, USA: Kluwer Academic Publishers, 2000. p. 55–61.
- JI, C. et al. A Novel Method of Mutation Clustering Based on Domain Analysis. In: **Proceedings of the 21th International Conference on Software Engineering and Knowledge Engineering (SEKE)**. Boston, MA, USA: Knowledge Systems Institute Graduate School, 2009. p. 1–6.
- JIA, Y.; HARMAN, M. An Analysis and Survey of the Development of Mutation Testing. **IEEE Transactions on Software Engineering**, IEEE, v. 37, n. 5, p. 649–678, 2011. ISSN 0098-5589.
- JUST, R.; ERNST, M. D.; FRASER, G. Efficient Mutation Analysis by Propagating and Partitioning Infected Execution States. In: **Proceedings of the 23rd International Symposium on Software Testing and Analysis (ISSTA)**. San Jose, CA, USA: ACM, 2014. p. 315–326.
- JUST, R. et al. Are Mutants a Valid Substitute for Real Faults in Software Testing? In: **Proceedings of the 22th International Symposium on Foundations of Software Engineering (FSE)**. Hong Kong, China: ACM, 2014. p. 654–665. ISBN 978-1-4503-3056-5.

- JUST, R.; KAPFHAMMER, G. M.; SCHWEIGGERT, F. Using Conditional Mutation to Increase the Efficiency of Mutation Analysis. In: **Proceedings of the 6th International Workshop on Automation of Software Test (AST)**. Waikiki, Honolulu, HI, USA: ACM, 2011. p. 50–56.
- _____. Using Non-Redundant Mutation Operators and Test Suite Prioritization to Achieve Efficient and Scalable Mutation Analysis. In: **Proceedings of the IEEE 23th International Symposium on Software Reliability Engineering (ISSRE)**. Dallas, TX, USA: IEEE, 2012. p. 11–20.
- JUST, R.; KURTZ, B.; AMMANN, P. Inferring Mutant Utility from Program Context. In: **Proceedings of the 26th International Symposium on Software Testing and Analysis (ISSTA)**. Santa Barbara, CA, USA: ACM, 2017. p. 284–294. ISBN 978-1-4503-5076-1.
- JUST, R.; SCHWEIGGERT, F. Higher Accuracy and Lower Run Time: Efficient Mutation Analysis Using Non-redundant Mutation Operators. **Software Testing, Verification and Reliability**, John Wiley & Sons, Malden, MA, USA, v. 25, n. 5-7, p. 490–507, 2015. ISSN 1099-1689.
- KAMINSKI, G.; AMMANN, P. Using a Fault Hierarchy to Improve the Efficiency of DNF Logic Mutation Testing. In: **Proceedings of the 2nd International Conference on Software Testing, Verification and Validation (ICST)**. Denver, CO, USA: IEEE, 2009. p. 386–395.
- _____. Reducing Logic Test Set Size while Preserving Fault Detection. **Software Testing, Verification and Reliability**, John Wiley & Sons, Malden, MA, USA, v. 21, n. 3, p. 155–193, 2011.
- KAMINSKI, G.; AMMANN, P.; OFFUTT, A. J. Improving logic-based testing. **Journal of Systems and Software**, Elsevier, v. 86, n. 8, p. 2002 – 2012, 2013.
- KAMINSKI, G. et al. A Logic Mutation Approach to Selective Mutation for Programs and Queries. **Information and Software Technology**, Elsevier, Butterworth-Heinemann Newton, MA, USA, v. 53, n. 10, p. 1137–1152, 2011.
- KIM, S.-W.; MA, Y.-S.; KWON, Y.-R. Combining Weak and Strong Mutation for a Noninterpretive Java Mutation System. **Software Testing, Verification and Reliability**, John Wiley & Sons, Malden, MA, USA, v. 23, n. 8, p. 647–668, 2012.
- KINTIS, M.; MALEVRIS, N. Identifying More Equivalent Mutants via Code Similarity. In: **Proceedings of the 20th Asia-Pacific Software Engineering Conference (APSEC)**. Bangkok, Thailand: IEEE, 2013. p. 180–188.
- _____. Identifying more equivalent mutants via code similarity. In: **2013 20th Asia-Pacific Software Engineering Conference (APSEC)**. [S.l.: s.n.], 2013. v. 1, p. 180–188.
- _____. Using Data Flow Patterns for Equivalent Mutant Detection. In: **Proceedings of the 9th International Workshop on Mutation Analysis (Mutation)**. Cleveland, OH, USA: IEEE, 2014. p. 196–205.

- _____. MEDIC: A Static Analysis Framework for Equivalent Mutant Identification. **Information and Software Technology**, Elsevier, Butterworth-Heinemann Newton, MA, USA, v. 68, p. 1–17, 2015.
- KINTIS, M. et al. Detecting Trivial Mutant Equivalences via Compiler Optimisations. **IEEE Transactions on Software Engineering**, IEEE, Piscataway, NJ, USA, v. 44, n. 4, p. 1–25, 2017. ISSN 0098-5589.
- KINTIS, M.; PAPADAKIS, M.; MALEVRIS, N. Evaluating Mutation Testing Alternatives: A Collateral Experiment. In: **Proceedings of the 17th Asia-Pacific Software Engineering Conference (APSEC)**. Sydney, Australia: IEEE, 2010. p. 300–309.
- _____. Employing Second-order Mutation for Isolating First-order Equivalent Mutants. **Software Testing, Verification and Reliability**, John Wiley & Sons, Malden, MA, USA, v. 25, n. 5-7, p. 508–535, 2015.
- KITCHENHAM, B. et al. Systematic literature reviews in software engineering – a systematic literature review. **Information and Software Technology**, Elsevier, v. 51, p. 7–15, 2009.
- KRAUSER, E. W.; MATHUR, A. P.; REGO, V. J. High Performance Software Testing on SIMD Machines. **IEEE Transactions on Software Engineering**, IEEE, Piscataway, NJ, USA, v. 17, n. 5, p. 403–423, May 1991. ISSN 0098-5589.
- KURTZ, B. et al. Analyzing the Validity of Selective Mutation with Dominator Mutants. In: **Proceedings of the 24th International Symposium on Foundations of Software Engineering (FSE)**. Seattle, WA, USA: ACM, 2016. p. 571–582. ISBN 978-1-4503-4218-6.
- LACERDA, J. T. S.; FERRARI, F. C. Towards the Establishment of a Sufficient Set of Mutation Operators for AspectJ Programs. In: **Proceedings of the 8th Brazilian Workshop on Systematic and Automated Software Testing (SAST)**. Maceio, AL, Brazil: Brazilian Computer Society, 2014. p. 21–30.
- LEME, F. G. et al. Multi-level mutation testing of Java and AspectJ programs supported by the ProteumAJv2 tool. In: **Proceedings of the 6th Brazilian Conference on Software: Theory and Practice (CBSOFT) – Tools Session**. Belo Horizonte/MG - Brazil: Brazilian Computer Society, 2015.
- LEVENSHTEIN, V. I. et al. Binary codes capable of correcting deletions, insertions, and reversals. In: SOVIET UNION. **Soviet physics doklady**. [S.l.], 1966. v. 10, n. 8, p. 707–710.
- LI, N. et al. Mutation Testing in Practice Using Ruby. In: **Proceedings of the 10th International Workshop on Mutation Analysis (Mutation)**. Graz, Austria: IEEE, 2015. p. 1–6. ISBN 978-1-4799-1885-0.
- LI, W. Software product metrics. **IEEE Potentials**, v. 18, n. 5, p. 24–27, 2000.
- LIMA, J. A. P. et al. Evaluating Different Strategies for Reduction of Mutation Testing Costs. In: **Proceedings of the 1st Brazilian Symposium on Systematic and Automated Software Testing (SAST)**. Maringá, PR, Brazil: ACM, 2016. p. 4:1–4:10.

- LINN, C.; DEBRAY, S. Obfuscation of executable code to improve resistance to static disassembly. In: **Proceedings of the 10th ACM conference on Computer and communications security**. [S.l.: s.n.], 2003. p. 290–299.
- LIU, M.-H. et al. An Approach to Test Data Generation for Killing Multiple Mutants. In: **Proceedings of the 22th International Conference on Software Maintenance (ICSM)**. Philadelphia, PA, USA: IEEE, 2006. p. 113–122.
- MA, Y.-S.; KIM, S.-W. Mutation Testing Cost Reduction by Clustering Overlapped Mutants. **Journal of Systems and Software**, Elsevier, v. 115, n. C, p. 18–30, 2016.
- MA, Y. S.; KWON, Y. R.; OFFUTT, A. J. Inter-class mutation operators for Java. In: **Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE)**. Annapolis/MD - USA: IEEE Computer Society Press, 2002. p. 352–366.
- MA, Y.-S.; OFFUTT, A. J.; KWON, Y. R. MuJava: An Automated Class Mutation System. **Software Testing, Verification and Reliability**, John Wiley & Sons, Malden, MA, USA, v. 15, n. 2, p. 97–133, 2005.
- MADEYSKI, L. et al. Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation. **IEEE Transactions on Software Engineering**, IEEE, Piscataway, NJ, USA, v. 40, n. 1, p. 23–42, 2014. ISSN 0098-5589.
- MALDONADO, J. et al. Introdução ao teste de software. **XIV Simpósio Brasileiro de Engenharia de Software**, Piscataway/NJ - USA, 2000.
- MALDONADO, J. C. **Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software**. Tese (Doutorado) — DCA/FEE, Universidade Estadual de Campinas (UNICAMP), Campinas, SP - Brasil, 1991.
- MARCOZZI, M. et al. Time to Clean Your Test Objectives. In: **Proceedings of the 40th International Conference on Software Engineering (ICSE) (to appear)**. Gothenburg, Sweden: IEEE, 2018. p. 456–467.
- MARSHALL, A. C. et al. Static Dataflow-aided Weak Mutation Analysis (SDAWM). **Information and Software Technology**, Elsevier, Butterworth-Heinemann Newton, MA, USA, v. 32, n. 1, p. 99–104, 1990.
- MARTIN, R. Oo design quality metrics. **An analysis of dependencies**, v. 12, n. 1, p. 151–170, 1994.
- MARTINS, G. **Estatística geral e aplicada**. [S.l.]: Atlas, 2005.
- MATEO, P. R.; USAOLA, M. P. Reducing mutation costs through uncovered mutants. **Journal of Software Testing, Verification and Reliability**, John Wiley & Sons, v. 25, n. 5-7, p. 464–489, 2014. ISSN 1099-1689.
- MATHUR, A. P. Performance, Effectiveness, and Reliability Issues in Software Testing. In: **Proceedings of the 15th IEEE Annual Computer Software and Applications Conference (COMPSAC)**. Tokyo, Japan: IEEE, 1991. p. 604–605. ISBN 0-8186-2152-4.

_____. **Foundations of Software Testing**. 1st. ed. Toronto, ON, Canada: Addison-Wesley Professional, 2007.

MATHUR, A. P.; WONG, W. E. Evaluation of the Cost of Alternative Mutation Testing Strategies. In: **Proceedings of the 7th Brazilian Symposium on Software Engineering (SBES)**. João Pessoa, PB , Brazil: Brazilian Computer Society, 1993. p. 320–335.

Matnei Filho, R. A.; VERGILIO, S. R. A Mutation and Multi-objective Test Data Generation Approach for Feature Testing of Software Product Lines. In: **Proceedings of the 29th Brazilian Symposium on Software Engineering (SBES)**. Belo Horizonte, MG, Brazil: IEEE, 2015. p. 21–30. ISBN 978-1-4673-9272-3.

McCabe, T. J. A Complexity Measure. **IEEE Transactions on Software Engineering**, SE-2, n. 4, 1976. ISSN 2326-3881.

MCMINN, P.; KAPFHAMMER, G. M.; WRIGHT, C. J. Virtual Mutation Analysis of Relational Database Schemas. In: **Proceedings of the 11th International Workshop on Automation of Software Test (AST)**. Austin, TX, USA: ACM, 2016. p. 36–42. ISBN 978-1-4503-4151-6.

MCMINN, P. et al. Automatic Detection and Removal of Ineffective Mutants for the Mutation Analysis of Relational Database Schemas (in press). **IEEE Transactions on Software Engineering**, IEEE, p. 1–1, 2018. ISSN 0098-5589.

MERDES, M. et al. Ubiquitous rats: How resource-aware run-time tests can improve ubiquitous software system. In: **Proceedings of the 6th International Workshop on Software Engineering and Middleware (SEM)**. New York/NY - USA: ACM Press, 2006. p. 55–62. ISBN 1-59593-585-1.

MORELL, L. J. A theory of fault-based testing. **IEEE Transactions on Software Engineering**, IEEE, v. 16, n. 8, p. 844–857, 1990. ISSN 0098-5589.

MRESA, E. S.; BOTTACI, L. Efficiency of Mutation Operators and Selective Mutation Strategies: an Empirical Study. **Software Testing, Verification and Reliability**, John Wiley & Sons, Malden, MA, USA, v. 9, n. 4, p. 205–232, 1999.

MYERS, G. J.; SANDLER, C.; BADGETT, T. **The Art of Software Testing**. Hoboken, New Jersey: John Wiley & Sons, 2011.

NOBRE, T.; VERGILIO, S. R.; POZO, A. Reducing Interface Mutation Costs with Multiobjective Optimization Algorithms. **International Journal of Natural Computing Research**, IGI Global, p. 21–40, 2012.

OFFUTT, A. J.; CRAFT, W. M. Using Compiler Optimization Techniques to Detect Equivalent Mutants. **Software Testing, Verification and Reliability**, John Wiley & Sons, Malden, MA, USA, v. 4, n. 3, p. 131–154, 1994.

OFFUTT, A. J.; JIN, Z.; PAN, J. The Dynamic Domain Reduction Approach to Test Data Generation. **Software Practice and Experience**, John Wiley & Sons, Malden, MA, USA, v. 29, n. 2, p. 167–193, 1999.

- OFFUTT, A. J. et al. An Experimental Determination of Sufficient Mutant Operators. **ACM Transactions on Software Engineering and Methodology**, ACM, New York, NY, USA, v. 5, n. 2, p. 99–118, 1996.
- OFFUTT, A. J.; LEE, S. D. An Empirical Evaluation of Weak Mutation. **IEEE Transactions on Software Engineering**, IEEE, Piscataway, NJ, USA, v. 20, n. 5, p. 337–344, 1994.
- OFFUTT, A. J.; MA, Y.-S.; KWON, Y.-R. The Class-Level Mutants of MuJava. In: **Proceedings of the International Workshop on Automation of Software Test (AST)**. Shanghai, China: ACM, 2006. p. 78–84.
- OFFUTT, A. J.; PAN, J. Detecting Equivalent Mutants and the Feasible Path Problem. In: **Proceedings of the 11th Annual Conference on Computer Assurance (COMPASS)**. Gaithersburg, MD, USA: ACM, 1996. p. 224–236.
- _____. Automatically Detecting Equivalent Mutants and Infeasible Paths. **Software Testing, Verification and Reliability**, John Wiley & Sons, Malden, MA, USA, v. 7, n. 3, p. 165–192, 1997.
- OFFUTT, A. J. et al. Mutation Testing of Software Using a MIMD Computer. In: **Proceedings of the International Conference on Parallel Processing (ICPP)**. Chicago, Illinois, USA: John Wiley & Sons, 1992. p. II–257–266.
- OFFUTT, A. J.; ROTHERMEL, G.; ZAPF, C. An Experimental Evaluation of Selective Mutation. In: **Proceedings of the 15th International Conference on Software Engineering (ICSE)**. Baltimore, MD, USA: IEEE, 1993. p. 100–107.
- OFFUTT, A. J.; UNTCH, R. H. Mutation 2000: Uniting the orthogonal. In: _____. **Mutation Testing for the New Century**. Boston/MA - USA: Springer US, 2001. p. 34–44.
- OFFUTT, A. J.; VOAS, J.; PAYNE, J. **Mutation Operators for Ada**. Fairfax/VA - USA, 1996.
- OJDANIC, M. et al. **Syntactic Vs. Semantic similarity of Artificial and Real Faults in Mutation Testing Studies**. arXiv, 2021. Disponível em: <<https://arxiv.org/abs/2112.14508>>.
- OLIVEIRA, A. A. L.; CAMILO-JUNIOR, C. G.; VINCENZI, A. M. R. A Coevolutionary Algorithm to Automatic Test Case Selection and Mutant in Mutation Testing. In: **Proceedings of the 2013 IEEE Congress on Evolutionary Computation (CEC)**. Cancun, Mexico: IEEE, 2013. p. 829–836.
- OMAR, E.; GHOSH, S. An Exploratory Study of Higher Order Mutation Testing in Aspect-Oriented Programming. In: **Proceedings of the 23th International Symposium on Software Reliability Engineering (ISSRE)**. Dallas, TX, USA: IEEE, 2012. p. 1–10.
- PAPADAKIS, M.; DELAMARO, M.; Le Traon, Y. Mitigating the Effects of Equivalent Mutants with Mutant Classification Strategies. **Science of Computer Programming**, Elsevier, p. 298–319, 2014.

PAPADAKIS, M. et al. Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique. In: **Proceedings of the 37th International Conference on Software Engineering (ICSE)**. Florence, Italy: ACM, 2015. p. 936–946.

PAPADAKIS, M.; Le Traon, Y. Mutation Testing Strategies using Mutant Classification. In: **Proceedings of the 28th ACM Symposium on Applied Computing (SAC)**. Coimbra, Portugal: ACM, 2013. p. 1223–1229.

PAPADAKIS, M.; MALEVRIS, N. An Effective Path Selection Strategy for Mutation Testing. In: **Proceedings of the 16th Asia-Pacific Software Engineering Conference (APSEC)**. Batu Ferringhi, Penang, Malaysia: IEEE, 2009. p. 422–429.

_____. An Empirical Evaluation of the First and Second Order Mutation Testing Strategies. In: **Proceedings of the 5th International Workshop on Mutation Analysis (Mutation)**. Paris, France: IEEE, 2010. p. 90–99.

_____. Automatic Mutation Test Case Generation via Dynamic Symbolic Execution. In: **Proceedings of the IEEE 21th International Symposium on Software Reliability Engineering (ISSRE)**. San Jose, CA, USA: IEEE, 2010. p. 121–130.

_____. Automatic Mutation based Test Data Generation. In: **Proceedings of the 13th Genetic and Evolutionary Computation Conference (GECCO): Search-Based Software Engineering Track (Poster Session)**. Dublin, Ireland: ACM, 2011. p. 247–248.

_____. Automatically Performing Weak Mutation with the Aid of Symbolic Execution, Concolic Testing and Search-based Testing. **Software Quality Journal**, Springer, v. 19, n. 4, p. 691–723, 2011.

_____. Mutation Based Test Case Generation Via a Path Selection Strategy. **Information and Software Technology**, Elsevier, Butterworth-Heinemann Newton, MA, USA, v. 54, n. 9, p. 915–932, 2012.

PAPADAKIS, M.; MALEVRIS, N.; KALLIA, M. Towards Automating the Generation of Mutation Tests. In: **Proceedings of the 5th Workshop on Automation of Software Test (AST)**. Cape Town, South Africa: ACM, 2010. p. 111–118.

PARSAI, A.; MURGIA, A.; DEMEYER, S. Evaluating Random Mutant Selection at Class-level in Projects with Non-adequate Test Suites. In: **Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering (EASE)**. Limerick, Ireland: ACM, 2016. p. 1–10. ISBN 978-1-4503-3691-8.

PATEL, K.; HIERONS, R. M. Resolving the Equivalent Mutant Problem in the Presence of Non-determinism and Coincidental Correctness. In: **Proceedings of the 28th International Conference on Testing Software and Systems (ICTSS)**. Graz, Austria: Springer, 2016. p. 123–138 (LNCS v.9976).

PATRICK, M.; ORIOL, M.; CLARK, J. A. MESSI: Mutant Evaluation by Static Semantic Interpretation. In: **Proceedings of the 5th International Conference on Software Testing, Verification and Validation (ICST)**. Montreal, QC, Canada: IEEE, 2012. p. 711–719.

- PELLEG, D.; MOORE, A. W. X-means: Extending K-means with Efficient Estimation of the Number of Clusters. In: **Proc. the 17th Internl. Conference on Machine Learning**. [S.l.]: Morgan Kaufmann Publishers, 2000. ISBN 1-55860-707-2.
- PETROVIĆ, G.; IVANKOVIĆ, M. State of Mutation Testing at Google. In: **Proceedings of the 40th International Conference on Software Engineering - Software Engineering in Practice Track (ICSE-SEIP) (to appear)**. Gothenburg, Sweden: IEEE, 2018.
- PIZZOLETO, A. V. et al. SiMut: Exploring Program Similarity to Support the Cost Reduction of Mutation Testing. In: **Mutation'20**. [S.l.: s.n.], 2020. ISBN 978-1-5386-6352-3.
- _____. A Systematic Literature Review of Techniques and Metrics to Reduce the Cost of Mutation Testing. **Journal of Systems and Software**, v. 157, 2019. ISSN 0164-1212.
- PIZZOLETO, A. V.; GUARNIERI, G. F.; FERRARI, F. C. Definition of a Knowledge Base Towards a Benchmark for Experiments with Mutation Testing. In: **Proceedings of the 35th Brazilian Symposium on Software Engineering – Insightful Ideas and Emerging Results Track (SBES-IIER)**. Joinville, SC, Brazil: ACM, 2021. p. 215–220.
- POLO, M.; PIATTINI, M.; GARCÍA-RODRÍGUEZ, I. Decreasing the Cost of Mutation Testing with Second-Order Mutants. **Software Testing, Verification and Reliability**, John Wiley & Sons, Malden, MA, USA, v. 19, n. 2, p. 111–131, 2009.
- PRAPHAMONTRIPONG, U.; OFFUTT, A. J. Finding Redundancy in Web Mutation Operators. In: **Proceedings of the 12th International Workshop on Mutation Analysis (Mutation)**. Tokyo, Japan: IEEE, 2017. p. 134–142. ISBN 978-1-5090-6676-6.
- PRECHELT, L. et al. Finding plagiarisms among a set of programs with jplag. **J. UCS**, v. 8, n. 11, p. 1016, 2002.
- PRESSMAN, R. S. **Engenharia de Software - Uma Abordagem Profissional**. 8. ed. [S.l.]: Amgh Editoras, 2016.
- QUYEN, N. T. H. et al. Improving Mutant Generation for Simulink Models Using Genetic Algorithm. In: **Proceedings of the International Conference on Electronics, Information, and Communications (ICEIC)**. Da Nang, Vietnam: IEEE, 2016. p. 1–4.
- RAPPS, S.; WEYUKER, E. J. Data flow analysis techniques for program test data selection. In: **Proceedings of the 6th International Conference on Software Engineering (ICSE)**. Tokio - Japan: IEEE Computer Society, 1982. p. 272–278.
- _____. Selecting software test data using data flow information. **Transactions on Software Engineering**, IEEE Press, Piscataway/NJ - USA, v. 11, n. 4, p. 367–375, 1985.
- REALES, P.; POLO, M. Mutant Execution Cost Reduction: Through MUSIC (Mutant Schema Improved with Extra Code). In: **Proceedings of the 5th International Conference on Software Testing, Verification and Validation (ICST)**. Montreal, QC, Canada: IEEE, 2012. p. 664–672.

_____. Parallel Mutation Testing. **Software Testing, Verification and Reliability**, John Wiley & Sons, Malden, MA, USA, v. 23, n. 4, p. 315–350, 2013. ISSN 1099-1689.

_____. Reducing Mutation Costs Through Uncovered Mutants. **Software Testing, Verification and Reliability**, John Wiley & Sons, Malden, MA, USA, v. 25, n. 5-7, p. 464–489, 2014. ISSN 1099-1689.

REALES, P.; POLO, M.; ALEMÁN, J. L. F. Validating Second-Order Mutation at System Level. **IEEE Transactions on Software Engineering**, IEEE, v. 39, n. 4, p. 570–587, 2013.

REULING, D. et al. Fault-based Product-line Testing: Effective Sample Generation Based on Feature-Diagram Mutation. In: **Proceedings of the 19th International Conference on Software Product Line (SPLC)**. Nashville, TN, USA: ACM, 2015. p. 131–140.

ROJAS, J. M. et al. Combining Multiple Coverage Criteria in Search-Based Unit Test Generation. In: **Search-Based Software Engineering**. Cham: [s.n.], 2015.

SAHINOGLU, M.; SPAFFORD, E. H. A Bayes Sequential Statistical Procedure for Approving Software Products. In: **Proceedings of the IFIP Conference on Approving Software Products (ASP)**. Banff, AB, Canada: Elsevier, 1990. p. 43–56.

SCHLEIMER, S.; WILKERSON, D. S.; AIKEN, A. Winnowing: local algorithms for document fingerprinting. In: **Proceedings of the 2003 ACM SIGMOD international conference on Management of data**. [S.l.: s.n.], 2003. p. 76–85.

SCHULER, D.; DALLMEIER, V.; ZELLER, A. Efficient Mutation Testing by Checking Invariant Violations. In: **Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA)**. Chicago, IL, USA: ACM, 2009. p. 69–80.

SCHULER, D.; ZELLER, A. Covering and Uncovering Equivalent Mutants. **Software Testing, Verification and Reliability**, John Wiley & Sons, Malden, MA, USA, v. 23, n. 5, p. 353–374, 2013.

Siامي-Namin, A.; ANDREWS, J. H. Finding Sufficient Mutation Operators via Variable Reduction. In: **Proceedings of the 2nd Workshop on Mutation Analysis (Mutation)**. Raleigh, NC, USA: IEEE, 2006. p. 1–10. ISBN 0-7695-2897-X.

Siامي-Namin, A.; ANDREWS, J. H.; MURDOCH, D. J. Sufficient Mutation Operators for Measuring Test Effectiveness. In: **Proceedings of the 30th International Conference on Software Engineering (ICSE)**. Leipzig, Germany: ACM, 2008. p. 351–360.

SRIDHARAN, M.; Siامي-Namin, A. Prioritizing Mutation Operators based on Importance Sampling. In: **Proceedings of the IEEE 21th International Symposium on Software Reliability Engineering (ISSRE)**. San Jose, CA, USA: IEEE, 2010. p. 378–387.

STEIMANN, F.; THIES, A. From Behaviour Preservation to Behaviour Modification: Constraint-Based Mutant Generation. In: **Proceedings of the 32th International Conference on Software Engineering (ICSE)**. Cape Town, South Africa: ACM, 2010. p. 425–434.

STRUG, J.; STRUG, B. Using structural similarity to classify tests in mutation testing. In: **Applied Mechanics, Materials and Manufacturing**. [S.l.]: Trans Tech Publications Ltd, 2013. (Applied Mechanics and Materials, v. 378), p. 546–551.

_____. Using classification for cost reduction of applying mutation testing. In: **2017 Federated Conference on Computer Science and Information Systems (FedCSIS)**. [S.l.: s.n.], 2017. p. 99–108.

SUN, C. et al. An Empirical Study on Mutation Testing of WS-BPEL Programs. **Computer Journal**, Oxford University Press, v. 60, n. 1, p. 143–158, 2017. ISSN 0010-4620.

_____. A Path-aware Approach to Mutant Reduction in Mutation Testing. **Information and Software Technology**, Elsevier, Butterworth-Heinemann Newton, MA, USA, v. 81, p. 65–81, 2017.

SUN, C. ai et al. A path-aware approach to mutant reduction in mutation testing. **Information and Software Technology**, Elsevier Science Inc., v. 81, p. 65–81, 2016.

TANG, M.-H.; KAO, M.-H.; CHEN, M.-H. An empirical study on object-oriented metrics. In: IEEE. **Proceedings sixth international software metrics symposium (Cat. No. PR00403)**. [S.l.], 1999. p. 242–249.

Titcheu Chekam, T. et al. Selecting Fault Revealing Mutants. **Empirical Software Engineering**, Elsevier, v. 25, n. 1, p. 434–487, 2019.

TITCHEU CHEKAM, T. et al. Selecting Fault Revealing Mutants. **Empirical Software Engineering**, Springer, v. 25, p. 434–487, 2020.

TUYA, J.; SUÁREZ-CABAL, M. J.; RIVA, C. de la. Mutating database queries. **Information and Software Technology**, Elsevier, Butterworth-Heinemann Newton, MA, USA, v. 49, n. 4, p. 398–417, 2007.

UNTCH, R. H. Mutation-based Software Testing Using Program Schemata. In: **Proceedings of the 30th Annual Southeast Regional Conference (ACM-SE)**. Raleigh, NC, USA: ACM, 1992. p. 285–291.

_____. On Reduced Neighborhood Mutation Analysis Using a Single Mutagenic Operator. In: **Proceedings of the 47th Annual Southeast Regional Conference (ACM-SE)**. Clemson, SC, USA: ACM, 2009. p. 71–75.

UNTCH, R. H.; HARROLD, M. J.; OFFUTT, A. J. TUMS: Testing Using Mutant Schemata. In: **Proceedings of the 35th Annual Southeast Regional Conference (ACM-SE)**. Murfreesboro, TN, USA: ACM, 1997. p. 174–181.

UNTCH, R. H.; OFFUTT, A. J.; HARROLD, M. J. Mutation Analysis Using Mutant Schemata. In: **Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)**. Cambridge, MA, USA: ACM, 1993. p. 139–148.

VINCENZI, A. M. R. et al. Unit and Integration Testing Strategies for C Programs Using Mutation. **Software Testing, Verification and Reliability**, John Wiley & Sons, Malden, MA, USA, v. 11, n. 4, p. 249–268, 2001.

- _____. Bayesian-Learning Based Guidelines to Determine Equivalent Mutants. **International Journal of Software Engineering and Knowledge Engineering**, John Wiley & Sons, Hackensack, NJ, USA, v. 12, n. 6, p. 675–689, 2002. ISSN 1793-6403.
- WALENSTEIN, A. et al. Similarity in Programs. In: **Duplication, Redundancy, and Similarity in Software**. [S.l.: s.n.], 2007. (Dagstuhl Seminar Proceedings). ISSN 1862-4405.
- WANG, B. et al. Faster Mutation Analysis via Equivalence Modulo States. In: **Proceedings of the 26th International Symposium on Software Testing and Analysis (ISSTA)**. Santa Barbara, CA, USA: ACM, 2017. p. 295–306. ISBN 978-1-4503-5076-1.
- WEDYAN, F.; GHOSH, S. On Generating Mutants for AspectJ Programs. **Information and Software Technology**, Elsevier, Butterworth-Heinemann Newton, MA, USA, p. 900–914, 2012.
- WEI, C. et al. Spectral clustering based mutant reduction for mutation testing. **Information and Software Technology**, v. 132, p. 106502, 2021. ISSN 0950-5849. Disponível em: <<https://www.sciencedirect.com/science/article/pii/S0950584920302445>>.
- WEISS, S. N.; FLEYSHGAKKER, V. N. Improved Serial Algorithms for Mutation Analysis. In: **Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)**. Cambridge, MA, USA: ACM, 1993. p. 149–158.
- WOHLIN, C. Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering. In: **Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE)**. London, UK: ACM, 2014. p. 1–10.
- WONG, W. E. et al. Constrained Mutation in C Programs. In: **Proceedings of the 8th Brazilian Symposium on Software Engineering (SBES)**. [S.l.]: Brazilian Computer Society, 1994. p. 439–452.
- WONG, W. E.; MATHUR, A. P. Reducing the Cost of Mutation Testing: An Empirical Study. **Journal of Systems and Software**, Elsevier, v. 31, n. 3, p. 185–196, 1995.
- WONG, W. E.; MATHUR, A. P.; MALDONADO, J. C. Mutation Versus All-uses: An Empirical Evaluation of Cost, Strength and Effectiveness. **Software Quality and Productivity: Theory, Practice and Training**, Springer, Boston, MA, USA, p. 258–265, 1995.
- WOODWARD, M. R.; HALEWOOD, K. From Weak to Strong, Dead or Alive? An Analysis of some Mutation Testing Issues. In: **Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis**. Banff, AB, Canada: IEEE, 1988. p. 152–158.
- WRIGHT, C. J.; KAPFHAMMER, G. M.; MCMINN, P. Efficient Mutation Analysis of Relational Database Structure Using Mutant Schemata and Parallelisation. In: **Proceedings of the 8th International Workshop on Mutation Analysis (Mutation)**. Luxembourg City, Luxembourg: IEEE, 2013. p. 63–72.

_____. The Impact of Equivalent, Redundant and Quasi Mutants on Database Schema Mutation Analysis. In: **Proceedings of the 14th International Conference on Quality Software (QSIC)**. Dallas, TX, USA: IEEE, 2014. p. 57–66.

YU, M.; MA, Y.-S. Possibility of cost reduction by mutant clustering according to the clustering scope. **Software Testing, Verification and Reliability**, v. 29, n. 1-2, p. e1692, 2019. E1692 stvr.1692. Disponível em: <<https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1692>>.

ZHANG, J. et al. Predictive Mutation Testing. In: **Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)**. Saarbrücken, Germany: ACM, 2016. p. 342–353. ISBN 978-1-4503-4390-9.

ZHANG, L. et al. Operator-Based and Random Mutant Selection: Better Together. In: **Proceedings of the 28th International Conference on Automated Software Engineering (ASE)**. Palo Alto, CA, USA: IEEE, 2013. p. 92–102.

_____. Is Operator-Based Mutant Selection Superior to Random Mutant Selection? In: **Proceedings of the 32th International Conference on Software Engineering (ICSE)**. Cape Town, South Africa: ACM, 2010. p. 435–444.

ZHANG, L.; MARINOV, D.; KHURSHID, S. Faster Mutation Testing Inspired by Test Prioritization and Reduction. In: **Proceedings of the 22nd International Symposium on Software Testing and Analysis (ISSTA)**. Lugano, Switzerland: ACM, 2013. p. 235–245.

ZHANG, L. et al. Regression Mutation Testing. In: **Proceedings of the 21st International Symposium on Software Testing and Analysis (ISSTA)**. Minneapolis, MN, USA: ACM, 2012. p. 331–341.

_____. Test Generation via Dynamic Symbolic Execution for Mutation Testing. In: **Proceedings of the 26th International Conference on Software Maintenance (ICSM)**. Timisoara, Romania: IEEE, 2010. p. 1–10.

ZHU, Q.; PANICHELLA, A.; ZAIDMAN, A. Speeding-Up Mutation Testing via Data Compression and State Infection. In: **Proceedings of the 12th International Workshop on Mutation Analysis (Mutation)**. Tokyo, Japan: IEEE, 2017. p. 103–109.

_____. An Investigation of Compression Techniques to Speed up Mutation Testing. In: **Proceedings of the 11th International Conference on Software Testing, Verification and Validation (ICST)**. Västerås, Sweden: IEEE, 2018. p. 274–284.

APÊNDICE A

A Systematic Review of Cost Reduction Techniques for Mutation Testing: Preliminary Results

Neste apêndice apresenta-se o conteúdo completo do artigo publicado no *13th International Workshop on Mutation Analysis (Mutation'18)* (FERRARI; PIZZOLETO; OFFUTT, 2018). Um resumo deste artigo é apresentado no Capítulo 3 desta tese.

IEEE COPYRIGHT NOTICE:¹ “©IEEE, 2018. Reprinted, with permission, from the *Proceedings of the 13th International Workshop on Mutation Analysis (Mutation'18)*, <<https://doi.org/10.1109/ICSTW.2018.00021>>”

¹ <<https://journals.ieeeauthorcenter.ieee.org/choose-a-publishing-agreement/avoid-infringement-upon-ieee-copyright/>> - acessado em novembro/2022.

A Systematic Review of Cost Reduction Techniques for Mutation Testing: Preliminary Results

Fabiano Cutigi Ferrari*, Alessandro Viola Pizzoleto*, Jeff Offutt†

*Computing Department, Federal University of São Carlos, São Carlos, SP, Brazil

†Department of Computer Science, George Mason University, Fairfax, VA, USA

Email: fcferrari@ufscar.br; alessandro.pizzoleto@ufscar.br; offutt@gmu.edu

Abstract—This paper reports on results from a systematic review that characterizes the state-of-the-art on cost reduction for mutation testing. It analyzes the evolution of research on this topic and its underlying goals and techniques, and identifies metrics used to measure cost reduction. The mixed search strategy used automatic search, snowballing, and a survey of authors of primary studies. The analysis is based on a set of 165 peer-reviewed studies, from which 146 present either original or updated approaches and results for cost reduction of mutation testing. A list of 6 main goals for cost reduction is presented, and 22 techniques were identified. Historically, 18 metrics have been used to measure the gains and losses observed in experimental studies. In the last decade, substantial growth in the number of published studies was observed, particularly among techniques such as selective mutation, evolutionary algorithms, control-flow analysis, and higher order mutation.

Keywords—Mutation testing, cost reduction, systematic review.

I. INTRODUCTION

Mutation testing [31] has been extensively investigated over the last four decades. On the one hand, it has been empirically shown to be one of the strongest test criteria when compared to others such as control flow- and data flow-based testing [48, 92, 103, 105]. Moreover, it has been used as a test assessment mechanism for software engineering research [11, 73]. On the other hand, mutation testing is expensive due to the typically large number of mutants that need to be executed and analyzed [68, 85, 100].

Mutation testing requires the creation of a set M of mutants of a program P (or any executable software artifact), resulting from the application of mutation operators (*i.e.* code changing rules) to P . Then, for each mutant m , ($m \in M$), the tester runs a test set T originally designed for P . If $\exists t, (t \in T) \mid m(t) \neq P(t)$, mutant m is considered to have been *killed*. If not, the tester should enhance T with at least one test case that reveals the difference between m and P . If m and P are equivalent, then $P(t) = m(t)$ for all possible test cases.

Despite the effectiveness of mutation testing, M is usually large even for small programs, making mutation testing very expensive. Several techniques have been created to reduce the cost of mutation testing. We initially divide them into three groups: partial mutant execution (weak mutation [62] and firm mutation [161]); grouping (mutant schemata [146] and mutant clustering [67]); and mutant reduction (selective mutation [115] and random mutation [104]).

So many cost reduction techniques have been proposed that it has become difficult to track them all, and even more difficult to understand the cost-benefit tradeoffs among

them. This paper describes initial results from a systematic literature review (SLR) that characterizes the state-of-the-art of mutation testing cost reduction. Our characterization consists of: (i) an analysis of the evolution of research on this topic based on its underlying techniques; (ii) the identification and characterization of metrics used to measure cost reduction; and (iii) an analysis of the achieved cost savings as reported by the original authors.

This paper contributes to goals (i) and (ii). The results are based on an analysis of 165 peer-reviewed primary studies¹, published between 1988 and 2017. From the selected set of studies (146 in total), we provide a historical perspective. Our analysis yielded a list of 22 techniques that have been investigated to reduce mutation testing cost. Moreover, it produced a list of 6 primary goals for cost reduction and a list of 18 metrics that have been used to measure cost reduction. Our findings show that the techniques for cost reduction focus on technical and technological issues, without taking into account human costs. The identified metrics corroborate these findings.

The remainder of this paper is organized as follows: Section II provides details of the SLR protocol, including goals, and criteria and procedures for study selection and analysis. Section III presents overall results of search for studies and an initial classification based on Offutt and Untch's [113] categories (*do fewer*, *do smarter*, and *do faster*). Section IV analyzes the results with a focus on the main goals of the studies and the cost reduction techniques. Section V lists metrics used to measure cost reduction. Section VI contains additional discussions and threats to the validity of this SLR. Related research is summarized in Section VII and conclusions and our next steps are presented in Section VIII.

II. REVIEW SETUP

A systematic literature review (SLR) is a rigorous approach to identify, evaluate, and interpret all available evidence about a particular topic of interest [86]. Its process encompasses the definition of a protocol that must be followed and whose execution must be documented to yield consistent, reproducible, and auditable results. The next sections summarize the protocol defined for the SLR reported in this paper. The complete protocol can be found on the companion website [46].

¹*Primary study* is a term used in evidence-based software engineering to describe research results from well-founded experimental procedures or from incipient research approaches. SLRs, on the other hand, are *secondary studies*.

A. Goals and Research Questions

The general goal of this SLR is to characterize the state-of-the-art on cost reduction of mutation testing. The following research questions are defined:

- (RQ1) Which techniques support cost reduction of mutation testing?
- (RQ2) Which metrics are used to measure the cost reduction of mutation testing?
- (RQ3) What are the savings (benefit) and loss of effectiveness (regarding mutation score) for the techniques?

This paper addresses RQ1 (Section IV) and RQ2 (Section V). RQ3 will be addressed in future work.

B. Control Group and Study Selection Criteria

For this SLR, the control group, the studies referenced, includes cost reduction-related studies described in Jia and Harman's survey [68]. We applied the following inclusion (I) and exclusion (E) study selection criteria to choose our papers:

- (I1): The study proposes an approach to reduce the cost of mutation testing.
- (I2): The study applies an approach to reduce the cost of mutation testing.
- (I3): The study is peer-reviewed and published either in a conference or workshop proceedings, or in a scientific journal.
- (E1): The study addresses a technique that can be used to reduce the cost of mutation testing, but cost reduction is not on focus for that study.
- (E2): The study uses mutation testing information for test quality assessment (e.g. the use of mutation score to assess the quality of automatically generated test sets).

We selected studies that passed either I1 or I2, and I3 ($I3 \wedge (I1 \vee I2)$). We discarded studies that passed either E1 or E2 ($E1 \vee E2$). Moreover, we use the term *approach* to refer to research that applies classical cost reduction techniques (selective mutation and random mutation) or more contemporaneous techniques such as evolutionary algorithms and minimal mutation. As explained in Section IV-B, several selected studies applied varied techniques in a combined manner, so that the term *approach* is a simple way to refer to such combinations.

C. Search String, Repositories, and Selection Procedures

Step 1–Automatic search: This phase relied on search strings and search engines. The main terms in the string were “*mutation testing*” and “*cost reduction*,” which were used to compose the full search string as follows:

“mutation testing” or “mutation analysis” or “mutant analysis”) and (“cost reduction” or “sufficient operator” or “sufficient mutation” or “constrained mutation” or “selective mutation” or “weak mutation” or “random selection” or “random mutation” or “random mutants” or “equivalent mutant” or “equivalent mutation”)

The search string was submitted to the search engines of the following databases²: IEEE Xplore³, ACM Digital Library⁴

²The original repositories of the selected studies can be found in the references at the end of this paper as well in the companion website [46]. Note that IEEE, Elsevier, and Springer are “pure” study repositories since they only store studies into their own database. ACM and Wiley, on the other hand, have “hybrid” characteristics, since their search engines retrieve studies from a variety of databases, thus overlapping results from other searches.

³<http://ieeexplore.ieee.org> - last accessed on 05/jan/2018

⁴<https://dl.acm.org/> - last accessed on 05/jan/2018

Elsevier ScienceDirect⁵, Springer SpringerLink⁶, and Wiley Online Library⁷. Additional databases are also sources of primary studies in the snowballing and author survey steps.

Step 2 – Snowballing: the snowballing forward technique [157] was applied to the set of studies selected in Step 1. For each selected study, we analyzed the list of references to identify other studies of interest.

Step 3 – Author survey: we contacted every author of a selected study from the previous steps and asked for suggestions of additional studies. We analyzed their suggestions according to the inclusion and exclusion criteria.

General study selection procedures: for each search step, one researcher performed the preliminary selection by reading specific parts of the studies (title and abstract, and, if necessary, introduction and conclusion) to check if they should be selected for full reading. Next, two researchers fully read each study and made the final selection decision.

D. Data Extraction, Synthesis, and Analysis

Data of interest was extracted and stored in customized forms. General data such as authors, study title, and year of publication were used to compose references. For this SLR, more specific pieces of information are necessary for the synthesis and analysis steps. The first consists of characteristics of the cost reduction techniques and the metrics used for cost reduction measurement. The second consists of the values obtained with respect to cost reduction savings and the achieved mutation scores. This paper presents synthesis and analysis only for the first group, in Sections IV and V.

III. RESULTS

This section summarizes the results in terms of the study selection steps (Section III-A) and in terms of the *do fewer*, *do smarter*, and *do faster* categories [113] (Section III-B).

A. Overview of Search Results

Table I presents the numbers of studies retrieved and selected from each repository and search round. The term *search round* refers to either an automatic search round, the snowballing step, or the author survey. Table I only displays total numbers for the snowballing and author survey rounds since the papers were not from a specific database.

All rounds were performed with support of the StArt tool [59]. The first round was performed in April 2016, and resulted in 85 studies, and the second round was run in October 2017, and resulted in 20 studies. Snowballing (third round) was performed over the set of studies selected in the first search round and resulted in 19 additional studies. The author survey (fourth round) targeted authors of the three previous rounds. It started on 24 October 2017, and responses arrived until 18 December 2017 (54 days). In total, 41 authors replied and provided 131 suggestions of studies that were analyzed according to the inclusion goals of the SLR. From these, 22 additional, non-duplicate, studies were selected.

⁵<http://www.sciencedirect.com/> - accessed on last accessed on 05/jan/2018

⁶<http://link.springer.com/> - last accessed on 05/jan/2018

⁷<http://onlinelibrary.wiley.com/> - last accessed on 05/jan/2018

TABLE I. NUMBER OF STUDIES PER SEARCH ROUND (ONLY CONSIDERS NON-DUPLICATED RESULTS).

Search Round	IEEE		ACM		Database / Search Engine Elsevier		Springer		Wiley		Experts		Total	
	Retr.	Sel.	Retr.	Sel.	Retr.	Sel.	Retr.	Sel.	Retr.	Sel.	Retr.	Sel.	Retr.	Sel.
1st - Autom	468	29	611	18	607	12	464	0	771	13	19	13	2940	85
	8.92%	19.86%	11.65%	12.33%	11.57%	8.22%	8.84%	0.00%	14.70%	8.90%	0.36%	8.90%	56.04%	58.22%
2nd - Autom	303	8	640	10	706	1	78	0	376	1	0	0	2103	20
	5.78%	5.48%	12.20%	6.85%	13.46%	0.68%	1.49%	0.00%	7.17%	0.68%	0.00%	0.00%	40.09%	13.70%
3rd - Snowb	-	-	-	-	-	-	-	-	-	-	-	-	72	19
	-	-	-	-	-	-	-	-	-	-	-	-	1.37%	13.01%
4th - Survey	-	-	-	-	-	-	-	-	-	-	-	-	131	22
	-	-	-	-	-	-	-	-	-	-	-	-	2.50%	15.07%
TOTAL	771	37	1251	28	1313	13	542	0	1147	14	19	13	5246	146
	14.70%	25.34%	23.85%	19.18%	25.03%	8.90%	10.33%	0.00%	21.86%	9.59%	0.36%	8.90%	-	-

TABLE II. NUMBER OF STUDIES PER DATABASE.

Database	# of Studies	Database	# of Studies
IEEE	51	Elsevier	17
ACM	33	Wiley	19
Springer	16	Others	10
Total	146		100.00%

Table I only includes total numbers of non-subsumed studies. It does not account for 19 studies that were updated or extended by more recent studies. References to those studies can be found in the companion website [46]. Furthermore, as explained in Section II-C, due to their “hybrid” nature, ACM’s and Wiley’s search engines retrieved results that overlap results from other queried databases. Therefore, Table II provides more precise information about studies per database.

Figure 1 shows the distribution of selected studies per year. Starting from 2009, it is clear the number of studies is growing.

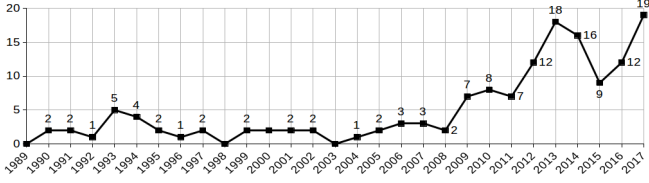


Fig. 1. Distribution of studies per year.

B. Initial Classification: do fewer, do smarter, do faster

Offutt and Untch [113] defined three categories to classify cost reduction techniques as follows:

The “do fewer” approaches seek ways of running fewer mutant programs without incurring intolerable information loss. The “do smarter” approaches seek to distribute the computational expense over several machines or factor the expense over several executions by retaining state information between runs or seek to avoid complete execution. The “do faster” approaches focus on ways of generating and running each mutant program as quickly as possible.

The classification of the 146 selected studies based on these categories is presented in Figure 2. The largest bar in the chart represents the total number of studies split into three groups. Some studies have more than one goal, so they explored combined techniques that can be classified in more than one category. If one or more cost reduction techniques were applied to support (or combined with) other techniques, for that study those “supportive” techniques were classified as a second, or third category, as shown in Figure 2. As an example, to automatically generate test cases to cover mutants,

Papadakis and Malevris [123] applied control-flow analysis supported by the mutant schemata technique. We categorized this paper first as do smarter, then as do faster.

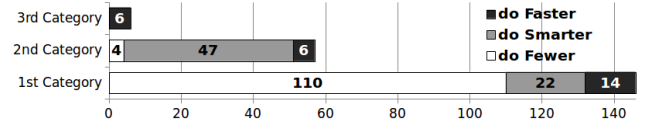


Fig. 2. Number of studies per Offutt and Untch’s [113] categories.

Figure 3 shows the distribution of studies by year and category. We found more *do fewer* and *do smarter* approaches. For example, 17 out of 18 studies published in 2013 tried to run fewer mutants without significant effectiveness loss. In same year, 10 out of 18 studies tried to reduce costs by distributing computational expense, processing intermediate mutant and original program execution information, or by analyzing mutant execution statistics.

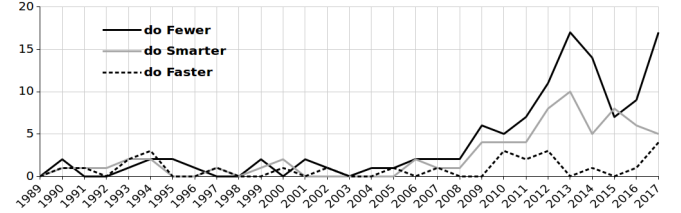


Fig. 3. Distribution of studies per year for Offutt and Untch’s [113] categories.

IV. GOALS AND TECHNIQUES FOR COST REDUCTION

This section presents the analysis of the selected studies based on the primary goals regarding cost reduction (Section IV-A) and on the techniques investigated (Section IV-B).

A. Primary Goals Regarding Cost Reduction

As mentioned in Section I, one contribution of this paper is a list of primary goals for mutation cost reduction research. We characterize a goal as *primary* when it is the main focus regarding cost reduction of mutation testing, either based on the description or based on the experiments and measurements. As an example, Harman et al. [57] tried to reduce both the number of mutants and the number of required test cases. So we classified that study into two categories: *reduction of the number of mutants* and *reduction of either the number of test cases or the number of test case executions*.

The 6 primary goals are given next. Each is labeled with a symbol that is used as a marker in figures later in the paper.

A complete list of references per primary goal is provided in the companion website [46].

- PG-1: (∇) *Reducing the number of mutants*: The main objective is to reduce the number of mutants that will be executed, with a secondary goal of reducing effectiveness as little as possible.
- PG-2: ($\equiv$) *Detecting equivalent mutants*: The main objective is to identify which mutants are equivalent to the original program, contributing to reduce the time of analysis of the remaining mutants.
- PG-3: ($\gg$) *Executing faster*: The main objective is to reduce the time to execute the mutants by using novel algorithms, tool improvements, or special-purpose hardware. Each generated mutant is analyzed to decide whether it can be partially executed or even discarded without being executed.
- PG-4: (ϵ) *Reducing the number of test cases or the number of test case executions*: The objective is either to identify reduced sets of test cases that are still as effective at killing mutants or to identify groups of similar mutants to reduce the number of test runs.
- PG-5: ($\odot$) *Avoiding the creation of certain mutants*: The main objective is to define mutation operators or mutation generation procedures that generate fewer mutants. Hopefully, only non-trivial (and non-equivalent) mutants will be generated to achieve high efficiency and effectiveness.
- PG-6: (τ) *Automatically generating test cases*: The main objective is to generate test cases automatically that kill as many mutants as possible.

Figure 4 shows the number of studies that targeted each of the primary goals. The chart reveals that *reducing the number of mutants* was far the most common goal of cost reduction research in mutation testing, followed by *executing faster* and *detecting equivalent mutants*.

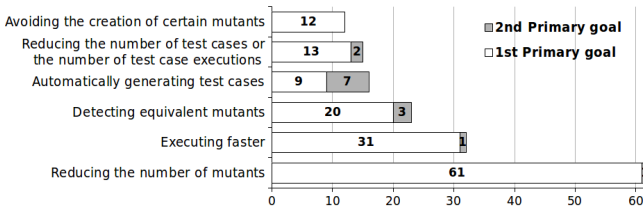


Fig. 4. Number of studies per primary goal

B. Cost Reduction Techniques

Another contribution of this paper is a classification of cost reduction-related research according to a broad set of categories. The categories rely either on mutation-specific research (e.g. *weak mutation* and *higher order mutation*) or on “classical” techniques used for software execution and analysis (e.g. *parallel execution* and *control-flow analysis*). In total, 22 categories (hereafter called *techniques*) were identified and are described next. The timeline in Figure 5 shows the year each technique was first explored.

Five techniques appear twice in Figure 5: *random mutation* [2, 104], *higher order mutation* [2, 151], *weak mutation* [62, 98], *firm mutation* [66, 161], and *constrained mutation* [102, 104]. They were first described either without a focus on cost reduction [62, 161], superficially [102], or in

a non-peer-reviewed publication [2], and were later published in peer-reviewed studies [66, 98, 104, 151]. The original references appear with different notation (dotted lines and smaller font size) in the timeline. The categories also have some overlap (e.g. *selective mutation*, *sufficient operators*, *one-op mutation*, and *minimal mutation*), as discussed in Section VI.

The techniques in Figure 5 are annotated with the symbols from the list of primary goals. For instance, *data-flow analysis* and *weak mutation* were first explored to support the cost reduction of mutation testing with the aim of avoiding the creation of certain mutants [98], whereas *metamutants* was first explored for faster execution of mutants [146]. The following list enumerates the techniques from T-1 to T-22 in the order they appear in Figure 5. The description of each category is brief and does not consider specifics, such as customizations, performed on the technique over the years.

- T-1: *Random mutation*: This technique randomly selects from the complete set of mutants according to a predefined percentage. Random mutation can be interpreted as “choose X% of all mutants” or as “choose X% of mutants generated by each operator.” [2, 17, 38, 55, 89, 104, 122, 129, 158, 160, 165, 168].
- T-2: *Higher order mutation*: This technique combines two or more mutations into the same mutated program. *Higher order mutants* are complex mutants that can include the composition of simple defects [1, 2, 35, 39, 57, 76, 77, 83, 84, 93, 97, 101, 119, 122, 134, 151].
- T-3: *Weak mutation*: This execution technique checks whether the state of the mutant is infected shortly after the mutated location has been executed, rather than checking the mutant output after execution ends. If the state is infected, the mutant is killed immediately [40, 49, 62, 79, 83, 95, 98, 111, 125, 166, 170].
- T-4: *Firm mutation*: This is a variant of weak mutation where execution is allowed to proceed for some pre-defined duration after the state is infected [66, 161].
- T-5: *Parallel execution*: This technique executes mutants in parallel processors, reducing the total time needed to perform mutation analysis [21, 66, 88, 91, 99, 114, 154].
- T-6: *Data-flow analysis*: This technique uses program data flow-related information to decide which type of mutant will be generated and to analyze mutants. It considers whether variables that are more prone to failure during execution are reached and referenced [7, 27, 56, 60, 71, 81, 82, 84, 98, 110, 117, 131, 136].
- T-7: *Minimization and prioritization of test sets*: This technique analyzes the test suite to identify and keep test cases that will be efficient and effective at killing mutants [34, 135].
- T-8: *Compiler optimization*: This technique uses compiler-related techniques to optimize mutant execution and analysis (for instance, to automatically detect equivalent mutants) [26, 32, 70, 85, 110].
- T-9: *Constrained mutation*: This technique chooses a subset of mutation operators to use. The goal is to reduce the number of mutants without reducing effectiveness [55, 102, 104, 158–160].
- T-10: *Metamutants (or mutant schemata)*: This technique generates and executes mutants by embedding all mutants in one parameterized program called a *metamutant*. The metamutant is then compiled for fast execution. When run, the metamutant takes a parameter that tells it which

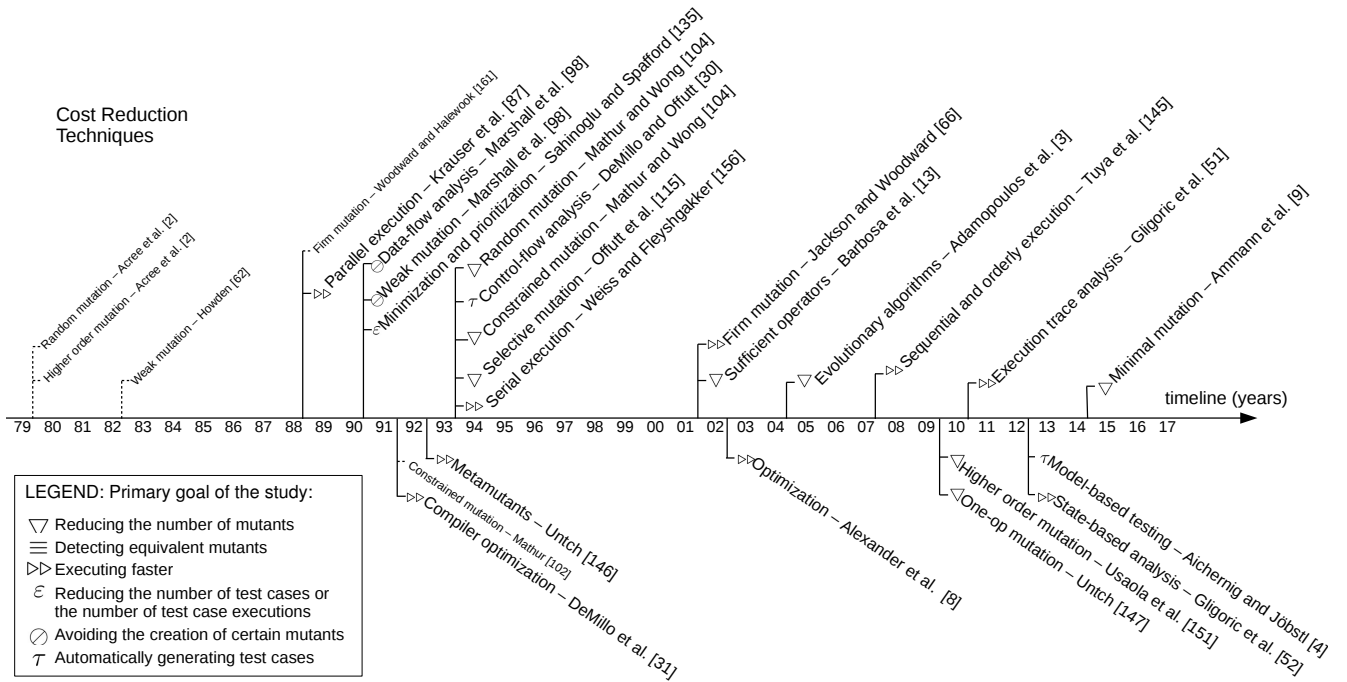


Fig. 5. Timeline for cost reduction techniques according to their application in peer-reviewed studies.

mutant to run. [54, 79, 95, 96, 100, 123, 125, 146, 148, 149, 156, 162].

- T-11: *Control-flow analysis*: This technique uses program control flow-related information, focusing on execution characteristics to identify branches and commands that help determine which structures are most relevant to the generation and execution of mutants [14, 27, 30, 49, 56, 61, 67, 71, 74, 76, 84, 94, 112, 117, 121, 123, 125–127, 130, 131, 136, 144, 163, 166, 167].
- T-12: *Selective mutation*: This technique tries to avoid the application of mutation operators that are responsible for the most mutants or that are the hardest to kill [18, 19, 26, 29, 37, 53, 55, 89, 108, 116, 132, 141, 143, 145, 168].
- T-13: *Serial execution*: This technique dynamically determines classes of mutants that behave similarly thus decreasing the number of mutants to be executed. The overhead incurred should be small in proportion to the time saved [47, 79, 95, 156].
- T-14: *Sufficient operators*: This technique tries to determine an essential set of mutation operators by applying customized procedures (e.g. based on heuristics or statistics) [13, 24, 69, 90, 138, 139, 152].
- T-15: *Optimization of generation, execution, and analysis of mutants*: this technique groups approaches that reduce the cost of mutation testing by exploring a variety of strategies that did not fit into any other category of this list⁸. More details are given later [8, 10, 16, 20, 22, 34, 36, 39, 43–45, 52, 54, 63–65, 69, 70, 72, 75, 78, 80, 107, 120, 134, 137, 142, 153, 155, 162, 164, 167, 169, 170].
- T-16: *Evolutionary algorithms*: This technique uses evolutionary algorithms to reduce the number of mutants, to reduce the number of test cases, or to identify equivalent mutants [1, 3, 12, 14, 15, 26, 28, 41, 42, 49, 50, 57, 58, 67, 106, 109, 118, 124, 125, 133, 141].

T-17: *Sequential and orderly execution*: This technique selects an ordering to execute mutants and test cases to kill more mutants early [145].

T-18: *One-op mutation*: This technique uses only the most relevant mutation operator regarding one or more specific cost reduction goals [23, 25, 33, 35, 89, 147].

T-19: *Execution trace analysis*: This technique uses traces of execution of the original program and some mutants to make decisions with respect to the execution of the remaining mutants [50, 51].

T-20: *Model-based testing (MBT)*: It consists in mutating formal or informal models of the program. The mutants are used to automatically generate test cases, which are then used to check if the faults simulated by the mutant models are also present in the program under test [5, 6, 39, 40].

T-21: *State-based analysis*: This technique compares states of different mutant executions. When two mutants lead to the same state (e.g. the same execution path is observed), only one needs to be executed and the result of the other can be inferred. Similarly, the technique creates groups of classes to define certain transition sequences so that only one of the mutants needs to be verified [14, 52, 72, 154].

T-22: *Minimal mutation*: This technique identifies and eliminates redundant mutants by applying the concepts of mutant subsumption and dominator mutant [9, 74, 89].

Category T-15 (*optimization of generation, execution, and analysis of mutants*) is used to classify studies that do not fit into any other category. For instance, Alexander et al. [8] implemented a mutation tool that uses Java reflection to apply mutations to Java objects at run-time, thus avoiding mutant compilation. Derezińska [34] proposed a clustering algorithm that takes into account results of mutant execution on subsets of test cases. Another, more recent, example is the study of Zhang et al. [164], which applied machine learning to devise predictive models to avoid executing some mutants. The models are based on previous versions of the software.

⁸Notice that this category appears as “Optimization” in Figure 5 for brevity.

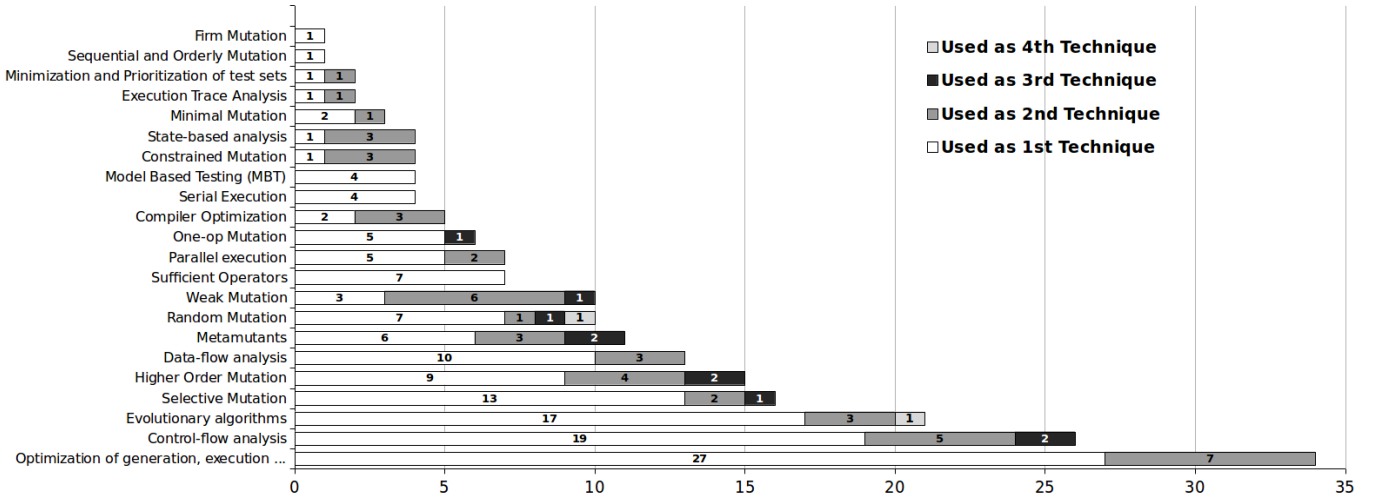


Fig. 6. Number of studies per cost reduction technique.

Figure 6 shows the distribution of studies per technique. Some studies used several techniques. These are listed in Figure 6 with counts. For example, *minimal mutation* has two units for “Used as 1st Technique” and two units for “Used as 2nd Technique.” To avoid subjectivity, for each study, the criterion for assigning a level of *participation* to an associated technique was based on how the authors described the focus of the technique. As an example, Marshall et al. [98] reduced mutation costs by applying data-flow analysis to support weak mutation. Therefore, *data-flow analysis* was set as the main technique while *weak mutation* was set as secondary. In another example, Wong and Mathur [158] reported on a comparison of *random mutation* with *constrained mutation*, without any “preference” for either. In this cases, assigning higher participation for a particular technique was arbitrary. These decisions introduced some noise into the categorizations.

Figure 7 shows the distribution of the six most investigated techniques per year, split into two charts for readability. Even without statistical tests for time series, it is clear that researchers are studying some techniques, such as *evolutionary algorithms*, *control-flow analysis*, *higher order mutation*, and *selective mutation*, more in recent years.

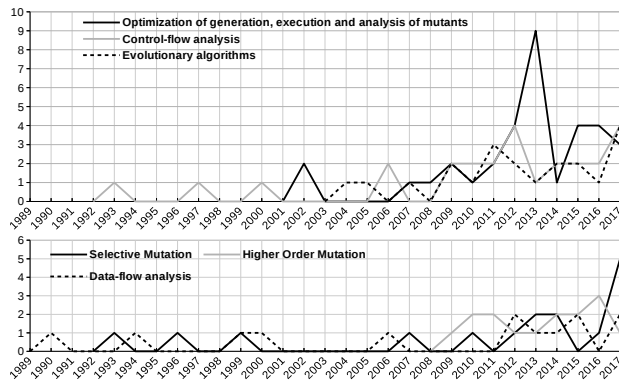


Fig. 7. Distribution of studies for the top six techniques per year.

V. METRICS FOR COST REDUCTION MEASUREMENT

An additional contribution of this paper is an initial characterization of metrics to measure the cost reduction of mutation testing. This characterization can help researchers design new experiments and set baselines for comparison. It can also help practitioners and researchers better measure the effectiveness of cost reduction techniques.

Figure 8 lists all metrics with a timeline. The first uses of these metrics were published in 1990 [98, 135] and 1991 [88]. Marshall et al. [98] estimated how many mutants would be created after using static data-flow analysis to avoid creating certain mutants. Sahinoğlu and Spafford [135] applied statistics to estimate the size of test sets required to reach a threshold mutation score. Both studies reflected their times by using programs that were small by today’s standards. Krauser et al. [88] ran simulations to measure mutant execution speedup on parallel machines. The timeline also shows that (i) there was a concentration of metrics (M-1 to M-6) at the early stages of cost reduction investigation (1990 to 1994), and (ii) lots of different metrics were used in the last 12 years (M-7 to M-18), corresponding with the large number of studies since 2005.

As additional information, the numbers in parentheses after the metric names in Figure 8 represent the number of studies that used each metric. *Number of mutants to be executed* used the most (63 studies), followed by *Mutant execution speedup* (31 studies). A complete list of references per metric is provided in the companion website [46].

VI. FURTHER DISCUSSIONS AND THREATS TO VALIDITY

As mentioned in Section IV-B, the cost reduction techniques have some overlap. For instance, *constrained mutation* [102], *selective mutation* [115], *sufficient operators* [13], *one-op mutation* [147], and the recent *minimal mutation* technique [9] could all be grouped into a “super” category of cost reduction based on reducing the number of mutants. We divided them on fairly specific characteristics, for example, *constrained mutation* relies on the testers’ intuition, and *one-op mutation* relies on individual mutation operators. We could also combine *weak mutation* [62] with *firm mutation* [161], by viewing weak mutation as a special case of firm mutation.

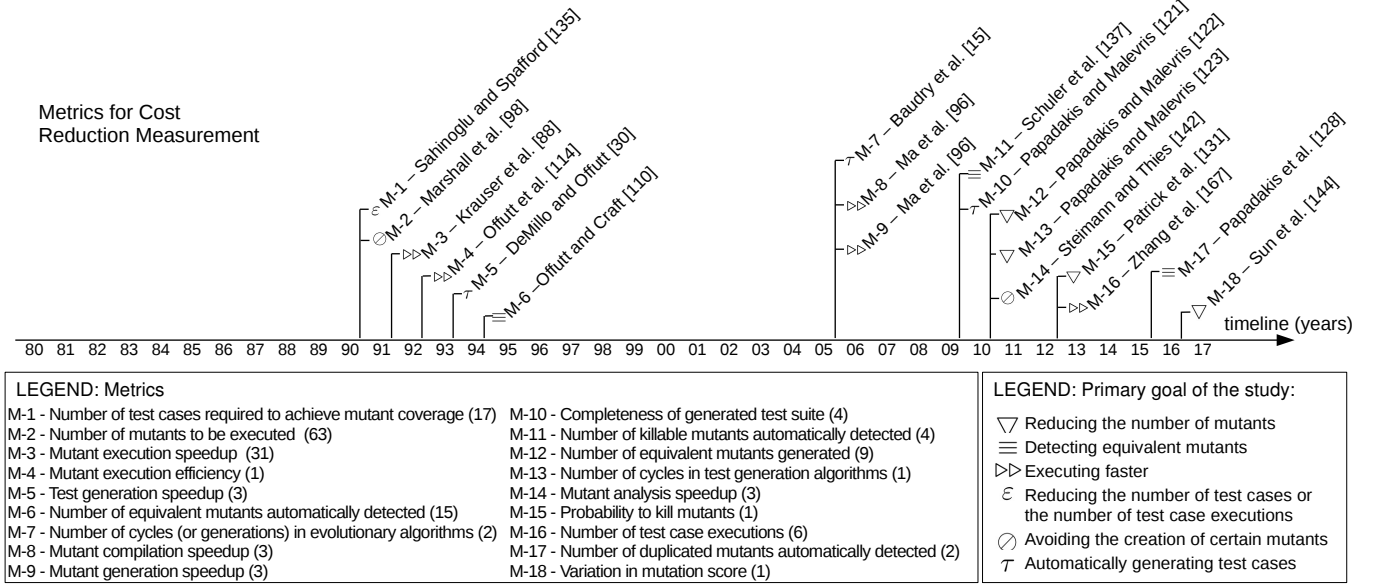


Fig. 8. Timeline for metrics used to measure cost reduction of mutation testing according to their application in peer-reviewed studies.

There is also some overlap among the primary goals described in Section IV-A. Some goals may be “side-effects” of others. For example, reducing the number of mutants (PG-1) implicitly reduces the number of test case executions (PG-4) and (probably) the number of automatically generated test cases (PG-6). Another example is automatically detecting equivalent mutants (PG-2), which also reduces the number of mutants. In-depth analysis of characteristics of cost reduction techniques and primary goals will be addressed in future work.

Another interesting observation regards the application of combined techniques. As shown in Figure 9, techniques have been applied together since the early stages of research on this topic. However, the number of studies that combined techniques has increased, particularly since 2010. For example, in 2010, six of nine studies applied two techniques to achieve cost reduction. In 2017, six of 18 studies applied two techniques and three studies applied three. In total, of the 101 studies selected since 2010, 38 combined at least two techniques.

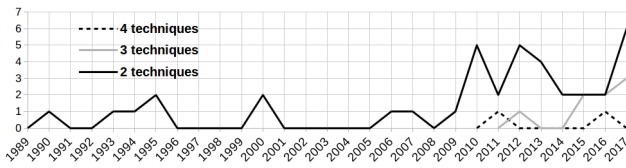


Fig. 9. Distribution of studies that applied combined techniques per year.

The two main threats to the validity of this study are: (i) the completeness of the results; and (ii) the classification of studies based on varied lists of categories. Regarding (i), the results come from an analysis of a set of peer-reviewed studies published in journals, conferences, and workshops. This set does not include Masters and PhD theses or gray literature such as technical reports and specifications, and other unrefereed documents. Even though other surveys on mutation testing (summarized in Section VII) include such types of studies, they show similar trends, particularly with respect to the growing number of published studies in the last 15 years [68, 97, 140].

The inclusion of additional, non-peer-reviewed, studies would not invalidate our results. Instead, it would probably increase the numbers presented in this paper.

Snowballing was used after the first search round with only one level of depth. Since Wohlin [157]’s suggested snowballing should be an iterative process that ends only when no new studies are found, this may create another slight threat to completeness. To mitigate this threat, we used Wohlin’s alternative suggestion and contacted authors of primary studies to ask for additional papers. Every author of selected primary studies (either through automatic search or via snowballing) were contacted by electronic mail in order suggest additional papers to be analyzed. Jia and Harman [68] did the same, calling it a ‘transitive closure’ on the literature.

For threat (ii), the initial classification of studies based on the *do fewer*, *do smarter*, and *do faster* categories [113] may not be precise enough. When these categories were first defined, fewer techniques were known, and more recent techniques were sometimes hard to categorize. In addition, since 2000 it has become more common to combine multiple techniques, as can be seen in Figures 5 and 9. As far as possible, the classification was performed in a discerning and unbiased manner, even though a precise classification of more recent techniques might require a revision of those categories. A final threat is the list of cost reduction techniques. As discussed in Section IV-B, several studies were assigned category T-15 as an “others” category. It might be possible to further analyze those techniques to identify subcategories, increasing the list of cost reduction techniques. It might also be reasonable to combine some categories, as mentioned at the beginning of Section VI.

VII. RELATED WORK

Usaola and Mateo [150] surveyed the main steps of mutation: mutant generation; test case generation and execution; and result analysis. Their goal was to identify, for each step, the largest costs and the techniques to reduce them. As opposed to

this SLR, their paper did not use systematic search criteria. The survey also did not classify studies per technique. All the 10 cost reduction-related studies analyzed in Usaola and Mateo's paper are included here.

Jia and Harman [68] published an extensive survey of mutation testing. They identified several cost reduction techniques, including equivalent mutant detection techniques. Jia and Harman identified the first peer-reviewed cost reduction paper as the 1982 paper that introduced *weak mutation* [62]. This paper is included in Figure 5, but not as a selected paper, as it did not focus on cost reduction and included no empirical validation. Our SLR search found the first study that used *weak mutation* to reduce cost as by Marshall et al. [98] in 1990. From a timeframe 1982 through 2009, Jia and Harman summarized 66 publications, including 52 peer-reviewed papers and 14 non-indexed (mostly, gray literature) items such as PhD and Masters Theses, technical reports, and Doctoral Symposium papers. In the same timeframe, this paper includes 44 peer-reviewed studies focused on cost reduction.

Madeyski et al. [97] published an SLR of the equivalent mutant problem. Based on 24 studies, the authors identified 17 methods that contribute to reducing costs related to equivalent mutants. This paper includes 15 of Madeyski et al.'s studies; the others either did not focus on cost reduction, were not peer-reviewed, or are included here as related work.

The most recent related work, by Silva et al. [140], was an SLR that characterized the application of Search-based Software Testing (SBST) techniques to mutation testing. The authors identified 69 papers published between 1998 and 2014, focusing on the meta-heuristics applied. As opposed to this study, Silva et al. provided an overall characterization of their topic without focusing on cost reduction-related studies, and included non-peer-reviewed studies.

VIII. FINAL REMARKS AND FUTURE WORK

Mutation testing is a very effective way to generate tests and assess test quality, but is also very costly. This has led to many approaches to reduce the cost of mutation. This paper summarized partial results of an ongoing systematic literature review that tries to characterize the state-of-the-art on cost reduction for mutation testing. This paper analyzed the evolution of research on this topic and summarized its underlying goals, techniques, and metrics. A total of 165 peer-reviewed studies were analyzed, of which 146 present either original or updated approaches and results for mutation cost reduction. The 22 identified techniques have 6 main goals for cost reduction, and 18 metrics were used to measure experimental results. The most investigated techniques in the last decade are selective mutation, evolutionary algorithms, control-flow analysis, and higher order mutation.

As with any SLR, it is unlikely that we found all primary studies, as acknowledged as a validity threat. Despite the used search strategy, the lack of standardized terminology and the frequent updates in indexed repositories mean no search can be perfect. Still, we hope this paper will help the community understand the state-of-the-art in this research topic.

The next step in this study will be to analyze results of experiments in the selected studies. Special attention will be given to the cost-benefit tradeoffs of the various cost reduction approaches. Further analysis will also group comparable

results, that is, results that come from different studies but use the same set of metrics. Future work will also include regular updates on the studies analyzed.

ACKNOWLEDGEMENTS

Fabiano Cutigi Ferrari is supported by FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo, grant #2016/21251-0, Brazil) and CNPq (grant #306310/2016-3, Brazil). Alessandro Viola Pizzolo is supported by CAPES (CAPES/DS scholarship, Brazil). Jeff Offutt is partly supported by KKS (The Knowledge Foundation), by project 20130085, Testing of Critical System Characteristics (TOCSYC).

REFERENCES

- [1] A. Abuljadayel and F. Wedyan, "An Approach for the Generation of Higher Order Mutants Using Genetic Algorithms (to appear)," *Internl. Journal of Intell. Systems and Applic.*, 2018.
- [2] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Mutation Analysis," School of Information and Computer Science, Georgia Tech, Tech. Rep. GIT-ICS-79/08, 1979.
- [3] K. Adamopoulos, M. Harman, and R. M. Hierons, "How to Overcome the Equivalent Mutant Problem and Achieve Tailored Selective Mutation Using Co-evolution," in *GECCO'04*, 2004.
- [4] B. K. Aichernig and E. Jöbstl, "Efficient Refinement Checking for Model-Based Mutation Testing," in *QISIC'12*, 2012.
- [5] B. K. Aichernig, E. Jöbstl, and M. Kegele, "Incremental refinement checking for test case generation," in *TAP'13*, 2013.
- [6] B. K. Aichernig, J. Auer, E. Jöbstl, R. K. adn W. Krenn, and R. S. adn B. V. Schmidt, "Model-Based Mutation Testing of an Industrial Measurement Device," in *TAP'14*, 2014.
- [7] M. Al-Hajjaji, J. Krüger, F. Benduhn, T. Leich, and G. Saake, "Efficient Mutation Testing in Configurable Systems," in *VACE'17*, 2017.
- [8] R. T. Alexander, J. M. Bieman, S. Ghosh, and B. Ji, "Mutation of Java Objects," in *ISSRE'02*, 2002.
- [9] P. Ammann, M. E. Delamaro, and A. J. Offutt, "Establishing Theoretical Minimal Sets of Mutants," in *ICST'14*, 2014.
- [10] P. Anbalagan and T. Xie, "Automated Generation of Pointcut Mutants for Testing Pointcuts in AspectJ Programs," in *ISSRE'08*, 2008.
- [11] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is Mutation an Appropriate Tool for Testing Experiments?" in *ICSE'05*, 2005.
- [12] K. Ayari, S. Bouktif, and G. Antoniol, "Automatic Mutation Test Input Data Generation via Ant Colony," in *GECCO'07*, 2007.
- [13] E. F. Barbosa, J. C. Maldonado, and A. Vincenzi, "Toward the Determination of Sufficient Mutant Operators for C," *STVR*, vol. 11, no. 2, 2001.
- [14] M. B. Bashir and A. Nadeem, "Improved Genetic Algorithm to Reduce Mutation Testing Cost," *IEEE Access*, vol. 5, 2017.
- [15] B. Baudry, F. Fleurey, J. Jézéquel, and Y. Le Traon, "From Genetic to Bacteriological Algorithms for Mutation-Based Testing," *STVR*, vol. 15, no. 2, 2005.
- [16] F. Belli and M. Beyazit, "Exploiting Model Morphology for Event-Based Testing," *IEEE TSE*, vol. 41, no. 2, 2015.
- [17] I. Blumke and K. Kulesza, "Reduction of Computational Cost in Mutation Testing by Sampling Mutants," in *DepCoS-RELCOMEX'13*, 2013.
- [18] —, "Reductions of Operators in Java Mutation Testing," in *DepCoS-RELCOMEX'14*, 2014.
- [19] —, "Reduction in Mutation Testing of Java Classes," in *ICSOFTEA'14*.
- [20] B. Bogacki and B. Walter, "Aspect-oriented Response Injection: An Alternative to Classical Mutation Testing," in *SET'06*, 2006.
- [21] C. Byoungju and A. P. Mathur, "High-Performance Mutation Testing," *JSS*, vol. 20, no. 2, 1993.
- [22] M. A. Cachia, M. Micallef, and C. Colombo, "Towards Incremental Mutation Testing," in *VSSSE'13*, 2013.
- [23] M. E. Delamaro, L. Deng, V. Durelli, N. Li, and A. J. Offutt, "Experimental Evaluation of SDL and One-Op Mutation for C," in *ICST'14*, 2014.
- [24] M. E. Delamaro, L. Deng, N. Li, V. Durelli, and A. J. Offutt, "Growing a Reduced Set of Mutation Operators," in *SBES'14*, 2014.
- [25] M. E. Delamaro, A. J. Offutt, and P. Ammann, "Designing Deletion Mutation Operators," in *ICST'14*, 2014.
- [26] P. Delgado-Pérez, I. Medina-Bulo, and M. G. Merayo, "Using Evolutionary Computation to Improve Mutation Testing," in *IWANN'17*, 2017.
- [27] P. Delgado-Pérez, I. Medina-Bulo, F. Palomo-Lozano, A. García-Domínguez, and J. J. Domínguez-Jiménez, "Assessment of Class Mutation Operators for C++ with the MuCPP Mutation System," *Inf. and Soft. Tech.*, 2017.
- [28] P. Delgado-Pérez, I. Medina-Bulo, S. Segura, A. García-Domínguez, and J. Juan, "GiGAN: Evolutionary Mutation Testing for C++ Object-oriented Systems," in *SAC'17*, 2017.

- [29] P. Delgado-Pérez, S. Segura, and I. Medina-Bulo, "Assessment of C++ Object-Oriented Mutation Operators: A Selective Mutation Approach," *STVR*, 2017.
- [30] R. A. DeMillo and A. J. Offutt, "Experimental Results from an Automatic Test Case Generator," *ACM TOSEM*, vol. 2, no. 2, 1993.
- [31] R. A. DeMillo, R. J. Lipton, and F. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *IEEE Comp.*, vol. 11, no. 4, 1978.
- [32] R. A. DeMillo, E. W. Krauser, and A. P. Mathur, "Compiler-integrated Program Mutation," in *COMPASAC'91*, 1991.
- [33] L. Deng, A. J. Offutt, and N. Li, "Empirical Evaluation of the Statement Deletion Mutation Operator," in *ICST'13*, 2013.
- [34] A. Derezińska, "A Quality Estimation of Mutation Clustering in C# Programs," in *DepCoS-RELCOMEX'13*, 2013.
- [35] A. Derezińska, "Evaluation of Deletion Mutation Operators in Mutation Testing of C# Programs," in *DepCoS-RELCOMEX'16*, 2016.
- [36] A. Derezińska and K. Hałas, "Improving Mutation Testing Process of Python Programs," in *CSOC'15*, 2015.
- [37] A. Derezińska and M. Rudnik, "Quality Evaluation of Object-Oriented and Standard Mutation Operators Applied to C# Programs," in *TOOLS*, 2012.
- [38] —, "Evaluation of Mutant Sampling Criteria in Object-Oriented Mutation Testing," in *ACSIS'17*, 2017.
- [39] X. Devroey, G. Perrouin, M. Papadakis, A. Legay, P. Schobbens, and P. Heymans, "Featured Model-based Mutation Analysis," in *ICSE*, 2016.
- [40] —, "Automata Language Equivalence vs. Simulations for Model-Based Mutant Equivalence: An Empirical Evaluation," in *ICST'17*, 2017.
- [41] J. J. Domínguez-Jiménez, A. Estero-Botaro, and A. G.-D. and I. Medina-Bulo, "GAMera: An Automatic Mutant Generation System for WS-BPEL Compositions," in *ECOWS'09*, 2009.
- [42] J. J. Domínguez-Jiménez, A. Estero-Botaro, A. García-Domínguez, and I. Medina-Bulo, "Evolutionary Mutation Testing," *Inf. and Soft. Tech.*, vol. 53, 2011.
- [43] V. H. S. Durelli, A. J. Offutt, and M. E. Delamaro, "Toward Harnessing High-Level Language Virtual Machines for Further Speeding Up Weak Mutation Testing," in *ICST'12*, 2012.
- [44] L. Fernandes, M. Ribeiro, L. Carvalho, R. Gheyi, M. Mongiovi, A. Santos, A. Cavalcanti, F. C. Ferrari, and J. C. Maldonado, "Avoiding Useless Mutants," in *GPCE'17*, 2017.
- [45] F. C. Ferrari, A. Rashid, and J. Maldonado, "Towards the Practical Mutation Testing of AspectJ Programs," *Sci. Comp. Prog.*, vol. 78, no. 9, 2013.
- [46] F. C. Ferrari, A. V. Pizzoleto, and J. Offutt, "SLR on Cost Reduction for Mutation Testing – Companion Website." Available Online - last accessed on 28/Feb/2018., 2018, <http://goo.gl/edyF1n>.
- [47] V. N. Fleyshgaker and S. N. Weiss, "Efficient Mutation Analysis: A New Approach," in *ISSTA'94*, 1994.
- [48] P. G. Frankl, S. N. Weiss, and C. Hu, "All-uses vs Mutation Testing: An Experimental Comparison of Effectiveness," *JSS*, vol. 38, no. 3, 1997.
- [49] G. Fraser and A. Arcuri, "Achieving Scalable Mutation-based Generation of Whole Test Suites," *Emp. Soft. Engin.*, vol. 20, no. 3, 2015.
- [50] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *IEEE TSE*, vol. 38, no. 2, pp. 278–292, 2012.
- [51] M. Gligoric, V. Jagannath, and D. Marinov, "MuT: Efficient Exploration for Mutation Testing of Multithreaded Code," in *ICST'10*, 2010.
- [52] M. Gligoric, V. Jagannath, Q. Luo, and D. Marinov, "Efficient Mutation Testing of Multithreaded Code," *STVR*, vol. 23, no. 5, 2012.
- [53] M. Gligoric, L. Zhang, C. Pereira, and G. Pokam, "Selective Mutation Testing for Concurrent Code," in *ISSTA'13*, 2013.
- [54] R. Gopinath, C. Jensen, and A. Groce, "Topsy-Turvy: A Smarter and Faster Parallelization of Mutation Analysis," in *ICSE'16*, 2016.
- [55] R. Gopinath, I. Ahmed, M. A. Alipour, C. Jensen, and A. Groce, "Mutation Reduction Strategies Considered Harmful," *IEEE Trans. on Soft. Reliability*, vol. 66, no. 3, 2017.
- [56] M. Harman, R. M. Hierons, and S. Danicic, "The Relationship Between Program Dependence and Mutation Analysis," in *Mutation'00*, 2000.
- [57] M. Harman, Y. Jia, P. R. Mateo, and M. P. Usaola, "Angels and Monsters: An Empirical Investigation of Potential Test Effectiveness and Efficiency Improvement from Strongly Subsuming Higher Order Mutation," in *ASE'14*, 2014.
- [58] C. Henard, M. Papadakis, and Y. Le Traon, "Mutation-Based Generation of Software Product Line Test Configurations," in *SSBSE'14*, 2014.
- [59] E. C. M. Hernandez, A. B. Zamboni, S. C. P. F. Fabbri, and A. Di Thomaz, "Using GQM and TAM to Evaluate StArt - A Tool that Supports Systematic Review," *CLEI Elect. Journal*, vol. 15, no. 1, 2012.
- [60] R. Hierons, M. Harman, and S. Danicic, "Using Program Slicing to Assist in the Detection of Equivalent Mutants," *STVR*, vol. 9, no. 4, 1999.
- [61] D. Holling, S. Banescu, M. Probst, A. Petrovska, and A. Pretschner, "Nequivack: Assessing Mutation Score Confidence," in *Mutation'16*, 2016.
- [62] W. E. Howden, "Weak Mutation Testing and Completeness of Test Sets," *IEEE TSE*, vol. 8, no. 4, 1982.
- [63] J. Hu, N. Li, and A. J. Offutt, "An Analysis of OO Mutation Operators," in *Mutation'11*, 2011.
- [64] C. Iida and S. Takada, "Reducing Mutants with Mutant Killable Precondition," in *Mutation'17*, 2017.
- [65] L. Inozemtseva, H. Hemmati, and R. Holmes, "Using Fault History to Improve Mutation Reduction," in *ESEC/FSE'13 (New Ideas Track)*, 2013.
- [66] D. Jackson and M. R. Woodward, "Parallel Firm Mutation of Java Programs," in *Mutation'00*, 2000.
- [67] C. Ji, Z. Chen, B. Xu, and Z. Zhao, "A Novel Method of Mutation Clustering Based on Domain Analysis," in *SEKE'09*, 2009.
- [68] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *IEEE TSE*, vol. 37, no. 5, 2011.
- [69] R. Just and F. Schweiggert, "Higher Accuracy and Lower Run Time: Efficient Mutation Analysis Using Non-redundant Mutation Operators," *STVR*, vol. 25, no. 5-7, 2015.
- [70] R. Just, G. M. Kapfhammer, and F. Schweiggert, "Using Conditional Mutation to Increase the Efficiency of Mutation Analysis," in *AST'11*, 2011.
- [71] —, "Using Non-Redundant Mutation Operators and Test Suite Prioritization to Achieve Efficient and Scalable Mutation Analysis," in *ISSRE'12*, 2012.
- [72] R. Just, M. D. Ernst, and G. Fraser, "Efficient Mutation Analysis by Propagating and Partitioning Infected Execution States," in *ISSSTA'14*, 2014.
- [73] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are Mutants a Valid Substitute for Real Faults in Software Testing?" in *FSE'14*, 2014.
- [74] R. Just, B. Kurtz, and P. Ammann, "Inferring Mutant Utility from Program Context," in *ISSTA'17*, 2017.
- [75] G. Kaminski and P. Ammann, "Using a Fault Hierarchy to Improve the Efficiency of DNF Logic Mutation Testing," in *ICST'09*, 2009.
- [76] —, "Reducing Logic Test Set Size while Preserving Fault Detection," *STVR*, vol. 21, no. 3, 2011.
- [77] G. Kaminski, U. Praphamontipong, P. Ammann, and A. J. Offutt, "A Logic Mutation Approach to Selective Mutation for Programs and Queries," *Inf. and Soft. Tech.*, vol. 53, no. 10, 2011.
- [78] G. Kaminski, P. Ammann, and A. J. Offutt, "Improving Logic-based Testing," *JSS*, vol. 86, no. 8, 2013.
- [79] S.-W. Kim, Y.-S. Ma, and Y.-R. Kwon, "Combining Weak and Strong Mutation for a Noninterpretive Java Mutation System," *STVR*, vol. 23, no. 8, 2012.
- [80] M. Kintis and N. Malevris, "Identifying More Equivalent Mutants via Code Similarity," in *APSEC'13*, 2013.
- [81] —, "Using Data Flow Patterns for Equivalent Mutant Detection," in *Mutation'14*, 2014.
- [82] —, "MEDIC: A Static Analysis Framework for Equivalent Mutant Identification," *Inf. and Soft. Tech.*, vol. 68, 2015.
- [83] M. Kintis, M. Papadakis, and N. Malevris, "Evaluating Mutation Testing Alternatives: A Collateral Experiment," in *APSEC'10*, 2010.
- [84] —, "Employing Second-order Mutation for Isolating First-order Equivalent Mutants," *STVR*, vol. 25, no. 5-7, 2015.
- [85] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. Le Traon, and M. Harman, "Detecting Trivial Mutant Equivalences via Compiler Optimisations," *IEEE TSE*, vol. PP, no. 99, 2017.
- [86] B. Kitchenham, "Procedures for Performing Systematic Reviews," Keele University, and National ICT Australia Ltd, Joint Tech. Rep. TR/SE-0401 (Keele) - 0400011T.1 (NICTA), 2004.
- [87] E. W. Krauser, A. P. Mathur, and V. J. Rego, "High Performance Testing on SIMD Machines," in *2nd Work. on Soft. Test., Verif. and Analysis*, 1988.
- [88] —, "High Performance Software Testing on SIMD Machines," *IEEE TSE*, vol. 17, no. 5, May 1991.
- [89] B. Kurtz, P. Ammann, A. J. Offutt, M. E. Delamaro, M. Kurtz, and N. Gökçe, "Analyzing the Validity of Selective Mutation with Dominator Mutants," in *FSE'16*, 2016.
- [90] J. T. S. Lacerda and F. C. Ferrari, "Towards the Establishment of a Sufficient Set of Mutation Operators for AspectJ Programs," in *SAST'14*, 2014.
- [91] N. Li, M. West, A. Escalona, and V. H. S. Durelli, "Mutation Testing in Practice Using Ruby," in *Mutation'15*, 2015.
- [92] N. Li, U. Praphamontipong, and A. J. Offutt, "An Experimental Comparison of Four Unit Test Criteria: Mutation, Edge-Pair, All-Uses and Prime Path Coverage," in *Mutation'09*, 2009, pp. 220–229.
- [93] J. A. P. Lima, G. Guizzo, S. R. Vergilio, A. P. C. Silva, H. L. J. Filho, and H. V. Ehrenfried, "Evaluating Different Strategies for Reduction of Mutation Testing Costs," in *SAST'16*, 2016.
- [94] M.-H. Liu, Y.-F. Gao, J.-H. Shan, J.-H. Liu, L. Zhang, and J.-S. Sun, "An Approach to Test Data Generation for Killing Multiple Mutants," in *ICSM'06*, 2006.
- [95] Y.-S. Ma and S.-W. Kim, "Mutation Testing Cost Reduction by Clustering Overlapped Mutants," *JSS*, vol. 115, no. C, 2016.
- [96] Y.-S. Ma, A. J. Offutt, and Y. R. Kwon, "MuJava: An Automated Class Mutation System," *STVR*, vol. 15, no. 2, 2005.
- [97] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Józala, "Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation," *IEEE TSE*, vol. 40, no. 1, 2014.
- [98] A. C. Marshall, D. Hedley, I. J. Riddell, and M. A. Hennell, "Static Dataflow-aided Weak Mutation Analysis (SDAWM)," *Inf. and Soft. Tech.*, vol. 32, no. 1, 1990.

- [99] P. R. Mateo and M. P. Usaola, "Parallel mutation testing," *STVR*, vol. 23, no. 4, 2013.
- [100] —, "Reducing Mutation Costs through Uncovered Mutants," *STVR*, vol. 25, no. 5-7, 2014.
- [101] P. R. Mateo, M. P. Usaola, and J. L. F. Alemán, "Validating Second-Order Mutation at System Level," *IEEE TSE*, vol. 39, no. 4, 2013.
- [102] A. P. Mathur, "Performance, Effectiveness, and Reliability Issues in Software Testing," in *COMPSAC'01*, 1991.
- [103] —, *Foundations of Software Testing*, 1st ed., 2007.
- [104] A. P. Mathur and W. E. Wong, "Evaluation of the Cost of Alternative Mutation Testing Strategies," in *SBES'93*, 1993.
- [105] —, "An Empirical Comparison of Data Flow and Mutation Based Test Adequacy Criteria," *STVR*, vol. 4, no. 1, 1994.
- [106] R. A. Matnei Filho and S. R. Vergilio, "A Mutation and Multi-objective Test Data Generation Approach for Feature Testing of Software Product Lines," in *SBES'15*, 2015.
- [107] P. McMinn, G. M. Kapfhammer, and C. J. Wright, "Virtual Mutation Analysis of Relational Database Schemas," in *AST'16*, 2016.
- [108] E. S. Mresa and L. Bottaci, "Efficiency of Mutation Operators and Selective Mutation Strategies: an Empirical Study," *STVR*, vol. 9, no. 4, 1999.
- [109] T. Nobre, S. R. Vergilio, and A. Pozo, "Reducing Interface Mutation Costs with Multiobjective Optimization Algorithms," *International Journal of Natural Computing Research*, 2012.
- [110] A. J. Offutt and W. M. Craft, "Using Compiler Optimization Techniques to Detect Equivalent Mutants," *STVR*, vol. 4, no. 3, 1994.
- [111] A. J. Offutt and S. D. Lee, "An Empirical Evaluation of Weak Mutation," *IEEE TSE*, vol. 20, no. 5, 1994.
- [112] A. J. Offutt and J. Pan, "Automatically Detecting Equivalent Mutants and Infeasible Paths," *STVR*, vol. 7, no. 3, 1997.
- [113] A. J. Offutt and R. H. Untch, "Mutation 2000: Uniting the Orthogonal," in *Mutation'00*, 2000.
- [114] A. J. Offutt, R. P. Pargas, S. V. Fichter, and P. K. Khambekar, "Mutation Testing of Software Using a MIMD Computer," in *Internl. Conf. on Parallel Processing*, 1992.
- [115] A. J. Offutt, G. Rothermel, and C. Zapf, "An Experimental Evaluation of Selective Mutation," in *ICSE'93*, 1993.
- [116] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An Experimental Determination of Sufficient Mutant Operators," *ACM TOSEM*, vol. 5, no. 2, 1996.
- [117] A. J. Offutt, Y.-S. Ma, and Y.-R. Kwon, "The Class-Level Mutants of MuJava," in *AST'06*, 2006.
- [118] A. A. L. Oliveira, C. G. Camilo-Junior, and A. M. R. Vincenzi, "A Coevolutionary Algorithm to Automatic Test Case Selection and Mutant in Mutation Testing," in *IEEE CEC'13*, 2013.
- [119] E. Omar and S. Ghosh, "An Exploratory Study of Higher Order Mutation Testing in Aspect-Oriented Programming," in *ISSRE'12*, 2012.
- [120] M. Papadakis and Y. Le Traon, "Mutation Testing Strategies using Mutant Classification," in *SAC'13*, 2013.
- [121] M. Papadakis and N. Maleveris, "An Effective Path Selection Strategy for Mutation Testing," in *APSEC'09*, 2009.
- [122] —, "An Empirical Evaluation of the First and Second Order Mutation Testing Strategies," in *Mutation'10*, 2010.
- [123] —, "Automatic Mutation Test Case Generation via Dynamic Symbolic Execution," in *ISSRE'10*, San Jose, CA, USA, 2010.
- [124] —, "Automatic Mutation based Test Data Generation," in *GECCO*, 2011.
- [125] —, "Automatically Performing Weak Mutation with the Aid of Symbolic Execution, Concolic Testing and Search-based Testing," *Soft. Quality Journal*, vol. 19, no. 4, 2011.
- [126] —, "Mutation Based Test Case Generation Via a Path Selection Strategy," *Inf. and Soft. Tech.*, vol. 54, no. 9, 2012.
- [127] M. Papadakis, M. Delamaro, and Y. Le Traon, "Mitigating the Effects of Equivalent Mutants with Mutant Classification Strategies," *Sci. Comp. Prog.*, 2014.
- [128] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon, "Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique," in *ICSE'15*, 2015.
- [129] A. Parsai, A. Murgia, and S. Demeyer, "Evaluating Random Mutant Selection at Class-level in Projects with Non-adequate Test Suites," in *EASE*.
- [130] K. Patel and R. M. Hierons, "Resolving the Equivalent Mutant Problem in the Presence of Non-determinism and Coincidental Correctness," in *ICTSS'16*, 2016.
- [131] M. Patrick, M. Oriol, and J. A. Clark, "MESSI: Mutant Evaluation by Static Semantic Interpretation," in *ICST'12*, 2012.
- [132] U. Praphamontripong and A. J. Offutt, "Finding Redundancy in Web Mutation Operators," in *Mutation'17*, 2017.
- [133] N. T. H. Quyen, K. T. Tung, L. T. M. Hanh, and N. T. Binh, "Improving Mutant Generation for Simulink Models Using Genetic Algorithm," in *ICEIC'16*, 2016.
- [134] D. Reuling, J. Bürdek, S. Rotärmel, M. Lochau, and U. Kelter, "Fault-based Product-line Testing: Effective Sample Generation Based on Feature-Diagram Mutation," in *SPLC'15*, 2015.
- [135] M. Sahinoğlu and E. H. Spafford, "A Bayes Sequential Statistical Procedure for Approving Software Products," in *ASP'90*, 1990.
- [136] D. Schuler and A. Zeller, "Covering and Uncovering Equivalent Mutants," *STVR*, vol. 23, no. 5, 2013.
- [137] D. Schuler, V. Dallmeier, and A. Zeller, "Efficient Mutation Testing by Checking Invariant Violations," in *ISSTA'09*, 2009.
- [138] A. Siami-Namin and J. H. Andrews, "Finding Sufficient Mutation Operators via Variable Reduction," in *Mutation'06*, 2006.
- [139] A. Siami-Namin, J. H. Andrews, and D. J. Murdoch, "Sufficient Mutation Operators for Measuring Test Effectiveness," in *ICSE'08*, 2008.
- [140] R. A. Silva, S. R. S. Souza, and P. S. L. Souza, "A Systematic Review on Search Based Mutation Testing," *Inf. and Soft. Tech.*, vol. 81, 2017.
- [141] M. Sridharan and A. Siami-Namin, "Prioritizing Mutation Operators based on Importance Sampling," in *ISSRE'10*, San Jose, CA, USA, 2010.
- [142] F. Steimann and A. Thies, "From Behaviour Preservation to Behaviour Modification: Constraint-Based Mutant Generation," in *ICSE'10*, 2010.
- [143] C. Sun, L. Pan, Q. Wang, H. Liu, and X. Zhang, "An Empirical Study on Mutation Testing of WS-BPEL Programs," *The Computer Journal*, vol. 60, no. 1, 2017.
- [144] C. Sun, F. Xue, H. Liu, and X. Zhang, "A Path-aware Approach to Mutant Reduction in Mutation Testing," *Inf. and Soft. Tech.*, vol. 81, 2017.
- [145] J. Tuyá, M. J. Suárez-Cabal, and C. de la Riva, "Mutating Database Queries," *Inf. and Soft. Tech.*, vol. 49, no. 4, 2007.
- [146] R. H. Untch, "Mutation-based Software Testing Using Program Schemata," in *ACM SE'92*, 1992.
- [147] —, "On Reduced Neighborhood Mutation Analysis Using a Single Mutagenic Operator," in *ACM SE'09*, 2009.
- [148] R. H. Untch, A. J. Offutt, and M. J. Harrold, "Mutation Analysis Using Mutant Schemata," in *ISSTA'93*, 1993.
- [149] R. H. Untch, M. J. Harrold, and A. J. Offutt, "TUMS: Testing Using Mutant Schemata," in *Annual Southeast Regional Conf.*, 1997.
- [150] M. P. Usaola and P. R. Mateo, "Mutation Testing Cost Reduction Techniques: A Survey," *IEEE Soft.*, vol. 27, no. 3, 2010.
- [151] M. P. Usaola, M. Piattini, and I. García-Rodríguez, "Decreasing the Cost of Mutation Testing with Second-Order Mutants," *STVR*, vol. 19, no. 2, 2009.
- [152] A. M. R. Vincenzi, J. C. Maldonado, E. F. Barbosa, and M. E. Delamaro, "Unit and Integration Testing Strategies for C Programs Using Mutation," *STVR*, vol. 11, no. 4, 2001.
- [153] A. M. R. Vincenzi, E. Y. Nakagawa, J. C. Maldonado, M. E. Delamaro, and R. A. F. Romero, "Bayesian-Learning Based Guidelines to Determine Equivalent Mutants," *Internl. Journal of Soft. Eng. and Knowl. Eng.*, vol. 12, no. 6, 2002.
- [154] B. Wang, Y. Xiong, Y. Shi, L. Zhang, and D. Hao, "Faster Mutation Analysis via Equivalence Modulo States," in *ISSTA'17*, 2017.
- [155] F. Wedyan and S. Ghosh, "On Generating Mutants for AspectJ Programs," *Inf. and Soft. Tech.*, 2012.
- [156] S. N. Weiss and V. N. Fleishgakkker, "Improved Serial Algorithms for Mutation Analysis," in *ISSTA'93*, 1993.
- [157] C. Wohlin, "Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering," in *EASE'14*. ACM, 2014.
- [158] W. E. Wong and A. P. Mathur, "Reducing the Cost of Mutation Testing: An Empirical Study," *JSS*, vol. 31, no. 3, 1995.
- [159] W. E. Wong, J. C. Maldonado, M. E. Delamaro, and A. P. Mathur, "Constrained Mutation in C Programs," in *SBES'94*, 1994.
- [160] W. E. Wong, A. P. Mathur, and J. C. Maldonado, "Mutation Versus All-uses: An Empirical Evaluation of Cost, Strength and Effectiveness," *Soft. Quality and Productivity: Theory, Practice and Training*, 1995.
- [161] M. R. Woodward and K. Halewood, "From Weak to Strong, Dead or Alive? An Analysis of some Mutation Testing Issues," in *2nd Workshop on Soft. Testing, Verif., and Analysis*, 1988.
- [162] C. J. Wright, G. M. Kapfhammer, and P. McMinn, "Efficient Mutation Analysis of Relational Database Structure Using Mutant Schemata and Parallelisation," in *Mutation'13*, 2013.
- [163] —, "The Impact of Equivalent, Redundant and Quasi Mutants on Database Schema Mutation Analysis," in *QJIC'14*, 2014.
- [164] J. Zhang, Z. Wang, L. Zhang, D. Hao, L. Zang, S. Cheng, and L. Zhang, "Predictive Mutation Testing," in *ISSTA'16*, 2016.
- [165] L. Zhang, S. Hou, J. Hu, T. Xie, and H. Mei, "Is Operator-Based Mutant Selection Superior to Random Mutant Selection?" in *ICSE'10*, 2010.
- [166] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. de Halleux, and H. Mei, "Test Generation via Dynamic Symbolic Execution for Mutation Testing," in *ICSM'10*, 2010.
- [167] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid, "Regression Mutation Testing," in *ISSTA'12*, 2012.
- [168] L. Zhang, M. Gligoric, D. Marinov, and S. Khurshid, "Operator-Based and Random Mutant Selection: Better Together," in *ASE'13*, Palo Alto, CA, USA, 2013.
- [169] L. Zhang, D. Marinov, and S. Khurshid, "Faster Mutation Testing Inspired by Test Prioritization and Reduction," in *ISSTA'13*, 2013.
- [170] Q. Zhu, A. Panichella, and A. Zaidman, "Speeding-Up Mutation Testing via Data Compression and State Infection," in *Mutation'17*, 2017.

APÊNDICE B

A systematic literature review of techniques and metrics to reduce the cost of mutation testing

Neste apêndice apresenta-se o conteúdo completo do artigo publicado no periódico *Journal of Systems and Software* (PIZZOLETO et al., 2019). Um resumo deste artigo é apresentado no Capítulo 3 desta tese.

ELSEVIER COPYTIGHT NOTICE:¹ “*Authors can include their articles in full or in part in a thesis or dissertation for non-commercial purposes.*”. Publicação original disponível em <<https://doi.org/10.1016/j.jss.2019.07.100>>.

¹ <<https://www.elsevier.com/about/policies/copyright/permissions>> - acessado em novembro/2022.



Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

A systematic literature review of techniques and metrics to reduce the cost of mutation testing

Alessandro Viola Pizzoleto^a, Fabiano Cutigi Ferrari^{a,b,1,*}, Jeff Offutt^b, Leo Fernandes^c, Márcio Ribeiro^d^a Computing Department, Federal University of São Carlos—São Carlos, SP, Brazil^b Department of Computer Science, George Mason University—Fairfax, VA, USA^c Informatics Coordination, Federal Institute of Alagoas—Maceió, AL, Brazil^d Computing Institute, Federal University of Alagoas—Maceió, AL, Brazil

ARTICLE INFO

Article history:

Received 5 January 2019

Revised 24 July 2019

Accepted 25 July 2019

Available online 26 July 2019

Keywords:

Mutation analysis

Mutation testing

Cost reduction

Systematic review

ABSTRACT

Historically, researchers have proposed and applied many techniques to reduce the cost of mutation testing. It has become difficult to find all techniques and to understand the cost-benefit tradeoffs among them, which is critical to transitioning this technology to practice. This paper extends a prior workshop paper to summarize and analyze the current knowledge about reducing the cost of mutation testing through a systematic literature review. We selected 175 peer-reviewed studies, from which 153 present either original or updated contributions. Our analysis resulted in six main goals for cost reduction and 21 techniques. In the last decade, a growing number of studies explored techniques such as selective mutation, evolutionary algorithms, control-flow analysis, and higher-order mutation. Furthermore, we characterized 18 metrics, with particular interest in the number of mutants to be executed, test cases required, equivalent mutants generated and detected, and mutant execution speedup. We found that cost reduction for mutation is increasingly becoming interdisciplinary, often combining multiple techniques. Additionally, measurements vary even for studies that use the same techniques. Researchers can use our results to find more detailed information about particular techniques, and to design comparable and reproducible experiments.

© 2019 Elsevier Inc. All rights reserved.

1. Introduction

Mutation testing (Hamlet, 1977; DeMillo et al., 1978; Acree et al., 1979) (*program mutation*, *mutation analysis*, or simply *mutation*) is a testing criterion that has been extensively studied yet lightly used in the last four decades (Offutt and Untch, 2000; Jia and Harman, 2011; Papadakis et al., 2019; Ferrari et al., 2018a). Mutation creates modified versions of a program, called *mutants*, by applying *mutation operators* that make small changes to the software to either simulate faults or guide the tester to edge cases (DeMillo et al., 1978). Testers are expected to find or design tests that cause these mutants to fail, that is, behave differently from the original un-mutated program. If a test case causes a mutant to

fail, then that mutant is said to be *killed*; otherwise, the mutant remains alive. Mutation can be used to help testers design high quality tests or to evaluate and improve existing test sets.

Mutation testing has been empirically shown to be stronger than test criteria such as control-flow based testing and data-flow based testing (Frankl et al., 1997; Mathur and Wong, 1994; Mathur, 2007; Li et al., 2009). Moreover, it has been used to measure the quality of tests and other test criteria in numerous studies (Andrews et al., 2005; Do and Rothermel, 2006; Just et al., 2014b). Despite its effectiveness, several factors make it expensive and therefore difficult to use in practice: large sets of mutants that must be executed, sometimes many times; generation of test cases to kill the mutants; the number of tests needed; and equivalent mutants (Offutt and Untch, 2000; Jia and Harman, 2011; Reales and Polo, 2014; Kintis et al., 2017). For even small methods with a few dozen lines of code, hundreds of mutants may be generated. Some mutants are trivially killed (that is, killed by all tests) and some are easily killed (that is, killed by many tests), yet some are more difficult and require detailed analysis by the tester. It is also very difficult (in fact undecidable in general (Budd and Angluin, 1982))

* Corresponding author at: Computing Department, Federal University of São Carlos—São Carlos, SP, Brazil.

E-mail addresses: alessandro.pizzoleto@ufscar.br (A.V. Pizzoleto), fcferrari@ufscar.br (F.C. Ferrari), offutt@gmu.edu (J. Offutt), leonardo.oliveira@ifal.edu.br (L. Fernandes), marcio@ic.ufal.br (M. Ribeiro).

¹ Was a visiting researcher at George Mason University, Fairfax, VA, USA, in 2017 and 2018.

to determine whether a mutant is equivalent or is simply difficult to kill. And not surprisingly, the most difficult-to-kill mutants often lead to valuable test cases.

These costs have been a major obstacle to practical adoption of mutation testing. For instance, some say the difficulty of identifying equivalent mutants is the primary reason mutation is not used more widely (Jia and Harman, 2011; Madeyski et al., 2014). Researchers have therefore invented many ways to reduce the cost, focusing on several goals. Specific goals include reducing the number of mutants (Acree et al., 1979; Mathur and Wong, 1993; Offutt et al., 1993), not creating certain mutants (Marshall et al., 1990), speeding up mutant execution (Krauser et al., 1988; Untch, 1992; Weiss and Fleishgaker, 1993), automating test set generation (DeMillo and Offutt, 1991; 1993), minimizing or prioritizing test sets (Sahinoğlu and Spafford, 1990), and automatically identifying equivalent mutants (Offutt and Craft, 1994).

Historically, many techniques to reduce mutation testing costs have been proposed, developed, and studied. The sheer volume of papers and the fact that they are published in dozens of different conferences and journals makes it hard to find them all, and even more difficult to understand the cost-benefit tradeoffs among them (Kurtz et al., 2016; Just et al., 2017; Gopinath et al., 2017). To help current and future researchers understand what has been done and discover new directions, as well as to organize the knowledge to support technology transition, we have performed a systematic literature review (SLR) to characterize the history and the state-of-the-art of mutation testing cost reduction. We previously presented preliminary results from our SLR in a workshop paper (Ferrari et al., 2018a), with general classifications of primary studies. This paper greatly extends and updates the prior short paper, and makes four key contributions:

- It presents comprehensive classifications for cost reduction goals and cost reduction techniques, including lists of references to selected studies.
- It provides an overview of all selected studies grouped by cost reduction goal, which is complemented by extensive online material (Ferrari et al., 2018b).
- It characterizes metrics used to measure cost reduction, including lists of references to selected studies.
- It summarizes, analyzes, and discusses the cost savings and test effectiveness as reported in the selected studies.

We analyzed 175 peer-reviewed, published, primary studies² We down-selected these to 153 studies by removing studies that were either extended or updated by later studies (*subsumed*). We then characterized 6 main goals related to cost reduction, and then identified 21 cost reduction techniques. We also identified and characterized 18 metrics that have been used to measure cost reduction. In addition to the fact that cost reduction research has been increasing, we found that cost reduction techniques are becoming interdisciplinary and are more frequently combined. Moreover, we found out that cost reduction techniques have been measured in many ways, and the results vary even for studies that apply the same cost reduction technique.

The remainder of this paper is organized as follows. Section 2 provides an overview of mutation testing and the need to reduce its costs. Section 3 provides details of our SLR protocol, including goals, research questions (RQ1, RQ2 and RQ3), and the criteria and procedures we used to select and analyze the studies. Section 4 summarizes results from our search. Section 5 answers research question RQ1, by analyzing the results with a focus on the goals of the studies and the cost reduction

techniques. Section 6 provides answers to research questions RQ2 and RQ3. It first lists metrics used to measure cost reduction, then analyzes the results of the selected studies regarding cost reduction measurements and achieved mutation scores. We summarize threats to the validity of this SLR in Section 7, and describe related research in Section 8. Finally, conclusions, implications, and future work are presented in Section 9. Appendix A brings tables with selected studies and subsumed studies.

2. Background

Recent literature reviews (Jia and Harman, 2011; Silva et al., 2017; Ferrari et al., 2018a; Papadakis et al., 2019) have shown that research in mutation testing has been steadily increasing through the last four decades. Mutation is a fault-based testing criterion that creates modified versions of the program, called *mutants* (DeMillo et al., 1978). Testers use mutation to design tests and to evaluate externally created tests (Andrews et al., 2005; Do and Rothermel, 2006; Just et al., 2014b). Mutation has also been used to assess other testing criteria. Papadakis et al. recently summarized 217 papers published from 2008 and 2017 that used mutation testing to perform test assessment (Papadakis et al., 2019).

Mutation is based on two fundamental hypotheses: the *Competent Programmer Hypothesis* and the *Coupling Effect Hypothesis* (DeMillo et al., 1978). The *Competent Programmer Hypothesis* says that although programmers do not write correct programs, they write programs that are *close* to being correct. The coupling effect hypothesizes that test data that find faults that are very close to the original program (that is, mutants) can also find more complex faults. That is, complex faults are *coupled* to simple faults (DeMillo et al., 1978). Both hypotheses, taken together, mean that tests that kill all or most simple mutants will probably kill more complex mutants.

Mutations operators modify the program under test to create mutants. For example, an arithmetic operator would change the expression $(a + b)$ to $(a * b)$, $(a - b)$, and (a / b) . Mutation operators use fault taxonomies that are usually based on studies of faults in real programs. Mutation operators are applied to a program P to create a set of mutants M . Each test t in a test set T is run against each mutant m , $m \in M$. If $m(t) \neq P(t)$ for some t , then we say that t has *killed* m . If not, the tester should find or design a test that kills m . If m and P are equivalent, then $P(t) = m(t)$ for all possible test cases.

As originally conceived (DeMillo et al., 1978), mutation testing was applied in four steps: (1) execution of the original program; (2) generation of mutants; (3) execution of the mutants; and (4) analysis of the mutants. After each cycle, the *mutation score* is calculated with Eq. 1. In the mutation score, K is the number of mutants of P killed by T ; M is the total number of mutants created from P ; and E is the number of mutants of P that are equivalent to P . The mutation score is a value in the interval $[0,1]$ that reflects the quality of the test set with respect to the mutants. The closer to 1 the mutation score is, the better the test set is.

$$MS = \frac{K}{M - E} \quad (1)$$

Steps 3 (execution) and 4 (analysis) are very costly, primarily due to two factors: the large number of executions and the need to hand-analyze live mutants. This paper characterizes research into reducing the cost of mutation testing. Step 3 has been highly automated, whereas step 4 is usually done by hand. The cost of step 4 is considered by some to be the main impediment for the practical adoption of mutation testing. Part of this analysis involves detecting equivalent mutants, which is a generally undecidable problem (Budd and Angluin, 1982). According to Madeyski et al. (2014), several partial automated solutions to the *equivalent mutant*

² Primary study is a term used in evidence-based research (Kitchenham et al., 2004) to describe research results from well-founded experimental procedures or from incipient research approaches. SLRs, on the other hand, are *secondary studies*.

problem have been developed. However, these solutions are limited and thus equivalent mutants are usually found by hand.

3. Study setup

A systematic literature review (SLR) is a rigorous approach to identify, evaluate, and interpret all available evidence about a particular topic of interest (Kitchenham, 2004). A basic requirement for SLRs is to document the process to ensure auditability and reproducibility (Kitchenham, 2004).

The following subsections summarize the protocol we used. The complete protocol is provided on a companion website (Ferrari et al., 2018b).

3.1. Goals and research questions

The general goal of this SLR is to summarize and analyze the history and state-of-the-art of efforts to reduce the cost of mutation testing. We define three research questions:

- (RQ1) Which techniques support cost reduction of mutation testing?
- (RQ2) Which metrics have been used to measure the cost reduction of mutation testing?
- (RQ3) What are the savings (benefit) and loss of effectiveness (as proxied by mutation score) for the techniques?

These research questions are addressed in Sections 5 (RQ1) and 6 (RQ2 and RQ3).

3.2. Control group and study selection criteria

For this SLR, the baseline dataset to assess the study selection process (the *control group* (Biolchini et al., 2005)) includes cost reduction-related studies described in Jia and Harman's survey (Jia and Harman, 2011). We applied the following inclusion (I) and exclusion (E) selection criteria to the studies:

- (I1): The study was included if it proposes an approach to reduce the cost of mutation testing.
- (I2): The study was included if it applies an approach to reduce the cost of mutation testing.
- (I3): The study was included if it is a primary study, peer-reviewed, and published either in a conference or a scientific journal.
- (E1): The study was excluded if it mentions or uses a technique that can be used to reduce the cost of mutation testing, but cost reduction is not a main goal of that study.
- (E2): The study was excluded if it uses mutation testing information to assess test quality (e.g. the use of mutation score to assess the quality of test sets generated based on testing criteria other than mutation testing).

We selected studies that first passed I3, and then either I1 or I2 ($I3 \wedge (I1 \vee I2)$). We discarded studies that passed E1 or E2 ($E1 \vee E2$). Notice that we use the term *approach* to refer to research that applies either classical cost reduction techniques (e.g. selective mutation and random mutation) or more contemporary techniques such as evolutionary algorithms and minimal mutation. As explained in Section 5.3, we found that several studies applied more than one technique in combination. Moreover, we selected studies that had a clear goal of reducing costs, either: (i) by explicitly or implicitly stating this goal in the approach description; (ii) while defining experimental goals; and/or (iii) by experimental measurements and analyses. We discarded studies that simply employed a particular mutation tool, or applied a subset of mutation operators, but did not measure cost reduction either theoretically or empirically.

3.3. Search steps, search string, repositories, and selection procedures

Step 1–Automatic search: This step relied on search strings and search engines. The main terms in the string were “mutation testing” and “cost reduction,” which were used to compose the full search string as follows:

(“mutation testing” or “mutation analysis” or “mutant analysis”) and (“cost reduction” or “sufficient operator” or “sufficient mutation” or “constrained mutation” or “selective mutation” or “weak mutation” or “random selection” or “random mutation” or “random mutants” or “equivalent mutant” or “equivalent mutation”)

The search string, sometimes customized for specific databases, was submitted to the search engines of the following databases:³ IEEE Xplore,⁴ ACM Digital Library,⁵ Elsevier ScienceDirect,⁶ Springer SpringerLink,⁷ and Wiley Online Library.⁸ Additional databases are also sources of primary studies in the snowballing and author survey steps, as described next.

Step 2–Snowballing: The backward snowballing technique (Wohlin, 2014) was applied to the set of studies selected in Step 1. For each selected study, we analyzed the list of references to identify other studies of interest, according to the inclusion and exclusion criteria.

Step 3–Author survey: We contacted every author of a selected study from the previous steps and asked for suggestions of additional studies. We analyzed their suggestions according to the inclusion and exclusion criteria.

Step 4–Post-publication update: Our set of selected studies was updated after our preliminary publication (Ferrari et al., 2018a). Particularly, we analyzed studies published in the 13th International Workshop on Mutation Analysis (Mutation’18), 11th International Conference on Software Testing, Verification and Validation (ICST’18), and 40th International Conference on Software Engineering (ICSE’18). The updated dataset also includes studies suggested by authors previously contacted in our author survey in step 3.

General study selection procedures: For each search step, one researcher performed the preliminary selection by reading specific parts of the retrieved studies (initially, title and abstract, and, if necessary, introduction and conclusion) to check whether they should be selected for full reading. In the final selection step, two researchers fully read each study and made the final selection decision, according to the inclusion and exclusion criteria. All conflicts in the study selection were resolved by the two researchers in order to reduce potential threats.

3.4. Data extraction, synthesis, and analysis

Data of interest were extracted and stored in customized forms. General data such as authors, study title, and year of publication were used to compose references. For this SLR, more specific pieces of information were necessary for the synthesis and analysis steps. The first consists of characteristics of the cost reduction techniques and the metrics used for cost reduction measurement. The second consists of the values obtained with respect to cost reduction savings and the achieved mutation scores.

³ The original repositories of the selected studies can be found in the references at the end of this paper as well in the companion website (Ferrari et al., 2018b). It is important to note that IEEE, Elsevier, and Springer can be considered “pure” (or private) study repositories since they only store and retrieve studies from their own database. ACM and Wiley, on the other hand, have “hybrid” characteristics, since their search engines retrieve studies from a variety of databases, thus overlapping results from other searches.

⁴ <http://ieeexplore.ieee.org> – last accessed on July, 2019.

⁵ <https://dl.acm.org/> – last accessed on July, 2019.

⁶ <http://www.sciencedirect.com/> – last accessed on July, 2019.

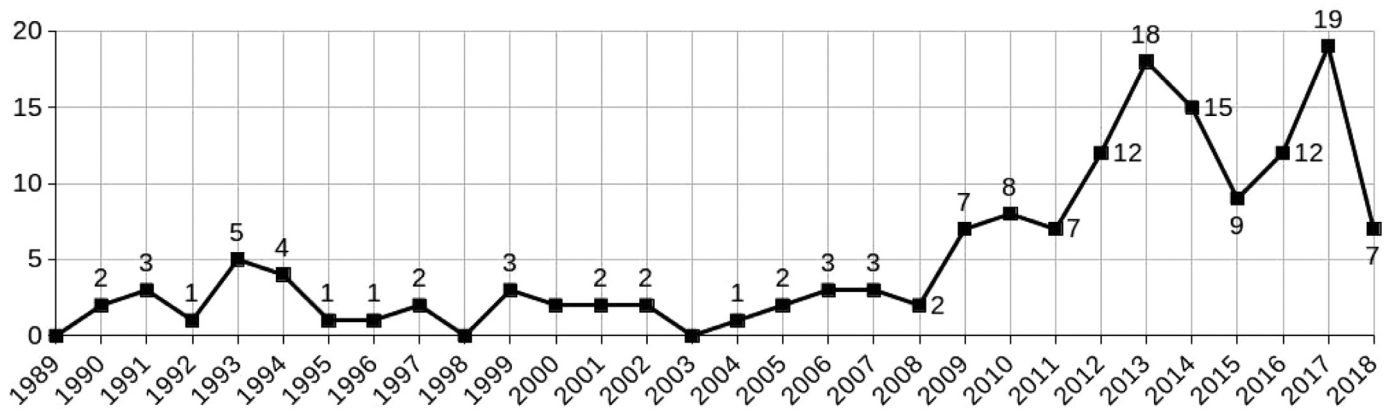
⁷ <http://link.springer.com/> – last accessed on July, 2019.

⁸ <http://onlinelibrary.wiley.com/> – last accessed on July, 2019.

Table 1

Number of studies per search round. Columns labeled with "Sel." include non-duplicated results for final selection. Columns labeled with "Retr." include duplicated search results. Note that the numbers of retrieved studies did not evolve consistently for Rounds 1 and 2 due to changes made in the search engines, and due to specifics of the grammar used to combine search terms in both rounds.

Search round	Database/Search engine												Total	
	IEEE		ACM		Elsevier		Springer		Wiley		Experts		..	
	Retr.	Sel.	Retr.	Sel.	Retr.	Sel.	Retr.	Sel.	Retr.	Sel.	Retr.	Sel.	Retr.	Sel.
1 - Autom (Apr, 2016)	467	27	612	19	606	12	464	0	771	13	19	13	2939	84
	8.88%	17.65%	11.64%	12.42%	11.52%	7.84%	8.82%	0.00%	14.66%	8.50%	0.36%	8.50%	55.89%	54.90%
2 - Autom (Oct, 2017)	303	8	642	10	706	1	78	0	376	1	0	0	2105	20
	5.76%	5.23%	12.21%	6.54%	13.42%	0.65%	1.48%	0.00%	7.15%	0.65%	0.00%	0.00%	40.03%	13.07%
3 - Snowb (Oct, 2017)	-	-	-	-	-	-	-	-	-	-	-	-	72	17
	-	-	-	-	-	-	-	-	-	-	-	-	1.37%	11.11%
4 - Survey (Oct-Dec, 2017)	-	-	-	-	-	-	-	-	-	-	-	-	135	25
	-	-	-	-	-	-	-	-	-	-	-	-	2.57%	16.34%
5 - Update (Aug, 2018)	-	-	-	-	-	-	-	-	-	-	-	-	8	7
	-	-	-	-	-	-	-	-	-	-	-	-	0.15%	4.58%
TOTAL	770	35	1254	29	1312	13	542	0	1147	14	19	13	5259	153
	14.64%	22.88%	23.84%	18.95%	24.95%	8.50%	10.31%	0.00%	21.81%	9.15%	0.36%	8.50%	-	-

**Fig. 1.** Distribution of studies per year. Only part of 2018 is included.

4. Study selection results

This section summarizes our results at each study selection step and the final set of selected studies. [Table 1](#) presents overall numbers for each step. [Fig. 1](#) shows the number of selected cost reduction studies per year from 1989 through (part of) 2018. [Table A.1](#) in [Appendix A](#) summarizes the studies in tabular form.

[Table 1](#) presents the numbers of studies retrieved and selected from each repository and search round. The term *search round* (or *round*) is used to refer to an automatic search round, the snowballing step, the author survey, and the last update performed after our preliminary publication of partial results ([Ferrari et al., 2018a](#)). Notice that [Table 1](#) displays only total numbers for the third, fourth, and fifth rounds since the studies were not retrieved from a particular database via automatic search.

All rounds were performed with the support of the StArt tool ([Hernandes et al., 2012](#)), which supports major steps of the SLR process, with particular help for the preselection and final selection phases. The first round was performed in April 2016 and resulted in 84 selected studies. The second round was performed in October 2017 and resulted in 20 additional studies. Snowballing (third round) was performed over the set of studies selected in the first search round, and resulted in 17 additional studies. The author survey (fourth round) targeted authors of the three previous rounds. It started on 24 October 2017, and responses arrived until 18 December 2017 (54 days). In total, 42 authors replied and

provided 135 suggestions of additional studies. 25 additional, non-duplicate, studies were selected from this group. As described in [Section 3.3](#), our dataset was updated based on the analysis of studies published in three events (Mutation'18, ICST'18, and ICSE'18), and with studies suggested by authors previously contacted in our author survey. This update added 7 new studies ([Search Round 5](#) in [Table 1](#)).

It is important to note that [Table 1](#) only includes total numbers of non-subsumed studies, that is, it does not account for 22 studies that passed our inclusion criteria, but were subsumed (updated or extended) by more recent studies. References to the 22 subsumed studies can be found in [Appendix A \(Table A.2\)](#). Furthermore, as explained in [Section 3.3](#), due to their "hybrid" nature, ACM's and Wiley's search engines retrieved results that overlap results from other queried databases. Therefore, [Table 2](#) provides more precise information regarding the number of studies selected per database than [Table 1](#).

Table 2
Number of studies per database.

Database	# of Studies	Database	# of Studies
IEEE	58 37.91%	Elsevier	17 11.11%
ACM	33 21.57%	Wiley	20 13.07%
Springer	15 9.80%	Others	10 6.54%
Total			153 100.00%

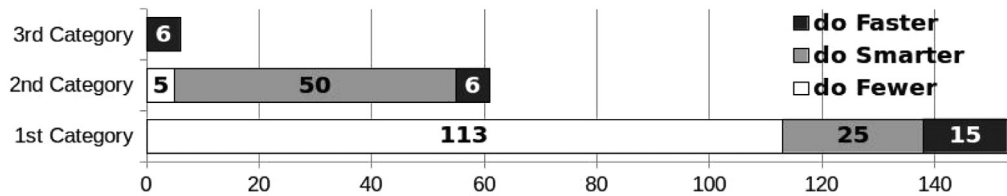


Fig. 2. Number of studies per Offutt and Untch (2000)'s categories.

Fig. 1 shows the distribution of selected studies per year. Note that only part of 2018 is included. It is clear the number of studies has been growing since 2009. Analyses of selected studies are presented in the next sections.

5. An overall characterization

This section presents and discusses an overall characterization of research into cost reduction of mutation testing. It starts in Section 5.1 by presenting the results of an initial classification based on the traditional “do fewer,” “do smarter,” and “do faster” categories (Offutt and Untch, 2000). In Section 5.2, we refine this classification into a set of primary goals associated with cost reduction. Then, Section 5.3 describes techniques that have been developed to achieve those primary goals. Quantitative data are also presented in the following sections.

5.1. Traditional classification: do fewer, do smarter, do faster

Offutt and Untch (2000) grouped cost reduction techniques into three categories: *do fewer*, *do smarter*, and *do faster*. In their words:

“The ‘do fewer’ approaches seek ways of running fewer mutant programs without incurring intolerable information loss. The ‘do smarter’ approaches seek to distribute the computational expense over several machines or factor the expense over several executions by retaining state information between runs or seek to avoid complete execution. The ‘do faster’ approaches focus on ways of generating and running each mutant program as quickly as possible.”

We classified the 153 selected studies according to Offutt and Untch's categories, as summarized in Fig. 2. The bar labelled “1st Category” shows that 113 studies used a *do fewer* approach, 25 used a *do smarter* approach, and 15 used *do faster*. This bar reflects the primary goal of the 153 studies. However, some studies combined more than one goal. The bar labelled “2nd Category” shows that five studies used a second approach of *do fewer*, 50 had a second approach of *do smarter*, and six used *do faster*. For example, to automatically generate test cases, Papadakis and Malevris (2010b) used *control-flow analysis* supported by the *meta-mutants* technique. We classified this study as *do smarter* followed by *do faster*. To obtain the total number of studies that, for example, tried to *do fewer*, one can sum 113 (shown in the “1st Category” bar) with 5 (shown in the “2nd Category” bar).

Fig. 3 shows the distribution of studies by year and category. We found more *do fewer* and *do smarter* approaches. For example, 17 out of 18 studies published in 2013 tried to run fewer mutants without significant effectiveness loss. In the same year, 11 out of 18 studies tried to reduce costs in a “smarter” way, usually by distributing computational expense, processing intermediate execution information from both the original program and its mutants, or by analyzing mutant execution statistics.

5.2. Primary goals for cost reduction

The *do fewer*, *do smarter*, and *do faster* categories have been used for almost 20 years. However, as cost reduction techniques

have become more complicated, this categorization has become harder to apply. Some techniques fit into more than one category and others do not really fit into any of the three. Thus, a major contribution of this paper is a new, and modern, characterization of primary goals for cost reduction techniques.

Our analysis resulted in the six primary goals described as follows. Each is labeled with a symbol that is used as a marker in subsequent figures. We also provide a complete list of references to studies that pursued each primary goal.

PG-1: (▽) *Reducing the number of mutants*: The objective is to reduce the number of mutants that will be executed, preferably without reducing effectiveness (Petrović and Ivanković, 2018; Zhu et al., 2018; McMinn et al., 2018; Abduljadayel and Wedyan, 2018; Sun et al., 2017a; Derezińska and Rudnik, 2017; Delgado-Pérez et al., 2017; Praphamontripong and Offutt, 2017; Gopinath et al., 2017; Delgado-Pérez et al., 2017c; Just et al., 2017; Al-Hajjaji et al., 2017; Delgado-Pérez et al., 2017b; Sun et al., 2017b; Derezińska, 2016; Quyen et al., 2016; Parsai et al., 2016; Kurtz et al., 2016; Lima et al., 2016; Derezińska and Hałas, 2015; Reuling et al., 2015; Bluemke and Kulesza, 2014b; Harman et al., 2014; Lacerda and Ferrari, 2014; Ammann et al., 2014; Delamaro et al., 2014a; 2014c; Madeyski et al., 2014; Delamaro et al., 2014b; Bluemke and Kulesza, 2014a; 2013; Oliveira et al., 2013; Deng et al., 2013; Cachia et al., 2013; Inozemtseva et al., 2013; Reales et al., 2013; Zhang et al., 2013a; Gligoric et al., 2013; Derezińska and Rudnik, 2012; Nobre et al., 2012; Patrick et al., 2012; Omar and Ghosh, 2012; Wedyan and Ghosh, 2012; Kaminski et al., 2011b; Domínguez-Jiménez et al., 2011; Papadakis and Malevris, 2010a; Sridharan and Siami-Namin, 2010; Kintis et al., 2010; Zhang et al., 2010a; Ji et al., 2009; Kaminski and Ammann, 2009; Polo et al., 2009; Untch, 2009; Siami-Namin et al., 2008; Tuya et al., 2007; Siami-Namin and Andrews, 2006; Adamopoulos et al., 2004; Vincenzi et al., 2001; Barbosa et al., 2001; Mresa and Bottaci, 1999; Offutt et al., 1996; Wong and Mathur, 1995; Wong et al., 1994; Offutt et al., 1993; Marcozzi et al., 2018; Zhu et al., 2017).

PG-2: (≡) *Automatically detecting equivalent mutants*: The objective is to automatically identify which mutants are equivalent to the original program, and then eliminate them from consideration (Marcozzi et al., 2018; Devroey et al., 2017; Kintis et al., 2017; Holling et al., 2016; Patel and Hierons, 2016; Kintis et al., 2015; Kintis and Malevris, 2015; Papadakis et al., 2014; Schuler and Zeller, 2013; Ferrari et al., 2013; Kintis and Malevris, 2013; Papadakis and Le Traon, 2013; Schuler et al., 2009; Anbalagan and Xie, 2008; Offutt et al., 2006; Vincenzi et al., 2002; Harman et al., 2000; Hierons et al., 1999; Offutt and Pan, 1997; Offutt and Craft, 1994; McMinn et al., 2018; Delgado-Pérez et al., 2017; Offutt and Lee, 1994).

PG-3: (▷) *Executing faster*: The objective is to reduce execution time by using novel algorithms, tool improvements, or special-purpose hardware. Some techniques analyze each mutant to decide whether it can be partially executed or

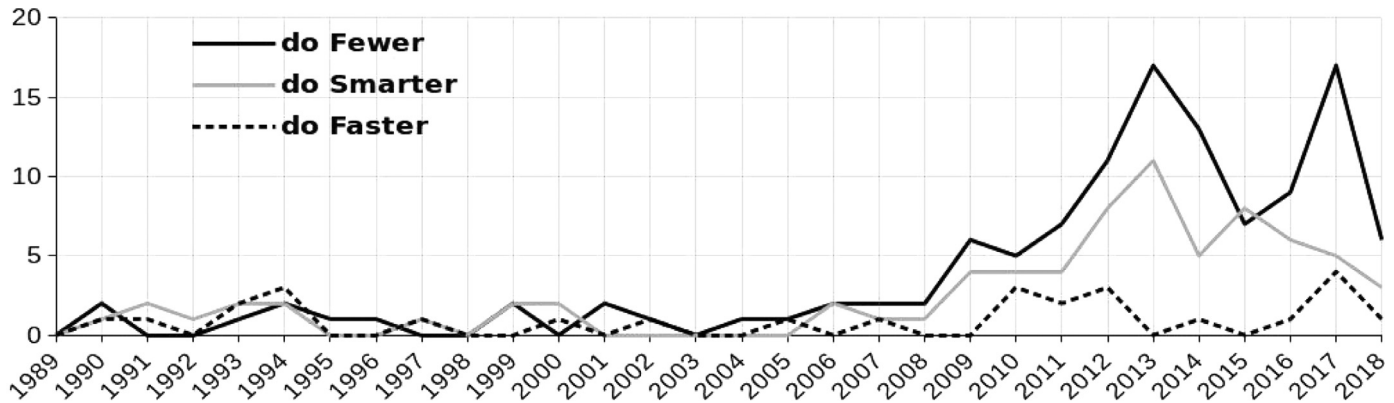


Fig. 3. Distribution of studies per year for Offutt and Untch (2000)'s categories.

even discarded without being executed (Chen and Zhang, 2018; Denisov and Pankevich, 2018; Wang et al., 2017; McMinn et al., 2016; Devroey et al., 2016; Gopinath et al., 2016b; Zhang et al., 2016; Li et al., 2015; Reales and Polo, 2014; Just et al., 2014a; Reales and Polo, 2013; Wright et al., 2013; Zhang et al., 2012; Gligoric et al., 2012; Kim et al., 2012; Durelli et al., 2012; Papadakis and Malevris, 2011b; Just et al., 2011; Gligoric et al., 2010; Bogacki and Walter, 2006a; Ma et al., 2005; Alexander et al., 2002; Jackson and Woodward, 2000; Untch et al., 1997; Fleishgaker and Weiss, 1994; Offutt and Lee, 1994; Choi and Mathur, 1993; Untch et al., 1993; Weiss and Fleishgaker, 1993; Offutt et al., 1992; Krauser et al., 1991; DeMillo et al., 1991; Devroey et al., 2017).

PG-4: (ϵ) *Reducing the number of test cases or the number of executions*: The objective is either to find smaller test sets that are still as effective at killing mutants, or to identify groups of similar mutants to reduce the number of test runs (Gopinath et al., 2018; Zhu et al., 2017; Ma and Kim, 2016; Fraser and Arcuri, 2015; Derezińska, 2013; Zhang et al., 2013b; Papadakis and Malevris, 2012; Kaminski and Ammann, 2011; Ayari et al., 2007; Sahinoğlu and Spafford, 1990; Zhu et al., 2018; Devroey et al., 2016; Derezińska and Hałas, 2015; Harman et al., 2014; Oliveira et al., 2013; Just et al., 2012b; Tuya et al., 2007; Adamopoulos et al., 2004).

PG-5: (ϕ) *Avoiding the creation of certain mutants*: The objective is to define mutation operators or mutation generation algorithms to generate fewer mutants. The general idea is to generate only non-trivial (and non-equivalent) mutants (Delgado-Pérez et al., 2017a; Iida and Takada, 2017; Fernandes et al., 2017; Just and Schweiggert, 2015; Belli and Beyazit, 2015; Kintis and Malevris, 2014; Kaminski et al., 2013; Just et al., 2012b; Hu et al., 2011; Steimann and Thies, 2010; Domínguez-Jiménez et al., 2009a; Marshall et al., 1990).

PG-6: (τ) *Automatically generating test cases*: The objective is to generate test cases automatically, to kill as many mutants as possible. Test case generation is typically guided by characteristics of the mutants, and can substantially reduce the effort required to create tests, which is usually done by hand (Bashir and Nadeem, 2017; Matnei Filho and Vergilio, 2015; Aichernig et al., 2014; Henard et al., 2014; Aichernig et al., 2013; Fraser and Zeller, 2012; Papadakis and Malevris, 2011a; Zhang et al., 2010b; Papadakis and Malevris, 2010b; 2009; Liu et al., 2006; Baudry et al., 2005; DeMillo and Offutt, 1993; Papadakis and Malevris, 2011b; Harman et al., 2000; DeMillo and Offutt, 1991; Offutt et al., 1999).

Approaches that try to avoid the creation of certain mutants (PG-5) typically characterize circumstances during mutant generation that would lead to the creation of equivalent, trivial, and redundant mutants. These rules are then embedded into the code that implements the mutation operators to prevent such mutants from being created. Other approaches try to reduce the number of mutants (PG-1) post-creation. For these, the mutation operators are applied as originally specified; and the reduction of the mutant set is achieved either by constraining the set of operators or by strategically selecting mutants from the generated set (e.g. based on mutant characteristics or randomly). Clearly, achieving PG-5 contributes to achieving PG-1. In other words, some of these goals can be seen as “side-effects” of others. Another example involves PG-1, PG-4 and PG-6. Reducing the number of mutants (PG-1) implicitly reduces the number of test case executions (PG-4) and (possibly) the number of automatically generated test cases (PG-6). Fig. 4 summarizes the relationships among the primary goals. Each edge indicates that the primary goal in the target node is indirectly achieved when the goal in the source node is achieved. For example, if PG-1 is achieved, then PG-6 is indirectly achieved.

We also emphasize that there is no clear disjoint, one-to-one, mapping between our new primary goals and Offutt and Untch (2000)'s categories. For instance, PG-3 (*Executing faster*) can be achieved by (1) by using novel algorithms (e.g., meta-mutants (Untch et al., 1993)), (2) improved tools (e.g., compiler-integrated mutant generator and executor (DeMillo et al., 1991)), or (3) running mutants on special-purpose hardware (e.g. multi-processor machines (Krauser et al., 1991; Offutt et al., 1992; Wang et al., 2017)). These methods were classified by Offutt and Untch as *do smarter* (1 and 2) and *do faster* (3).

Fig. 5 shows the number of studies that used each primary goal. The chart reveals that *reducing the number of mutants* was the most common, followed by *executing faster*, then *automatically detecting equivalent mutants*.

5.3. Cost reduction techniques

This section presents a broad classification of cost reduction-related research according to 21 categories, or *techniques*. Some of the categories rely on mutation-specific research (such as *weak mutation* and *higher order mutation*), and others on classical software execution and analysis techniques (such as *parallel execution* and *control-flow analysis*). The timeline in Fig. 6 shows the year each technique was introduced.

Five techniques appear twice in Fig. 6: *random mutation* (Acree et al., 1979; Mathur and Wong, 1993); *higher order mutation* (Acree et al., 1979; Polo et al., 2009); *weak mutation* (Howden,

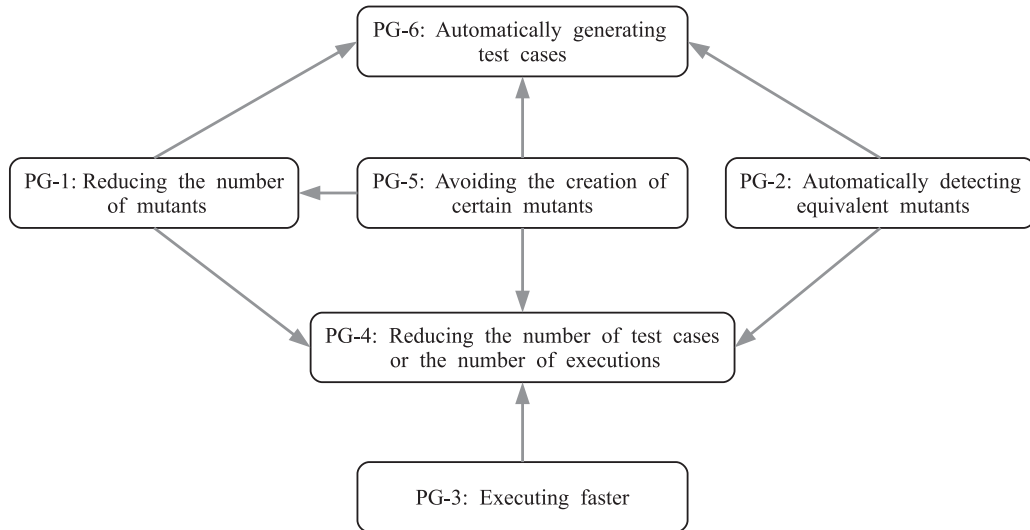


Fig. 4. Relationships among primary goals.

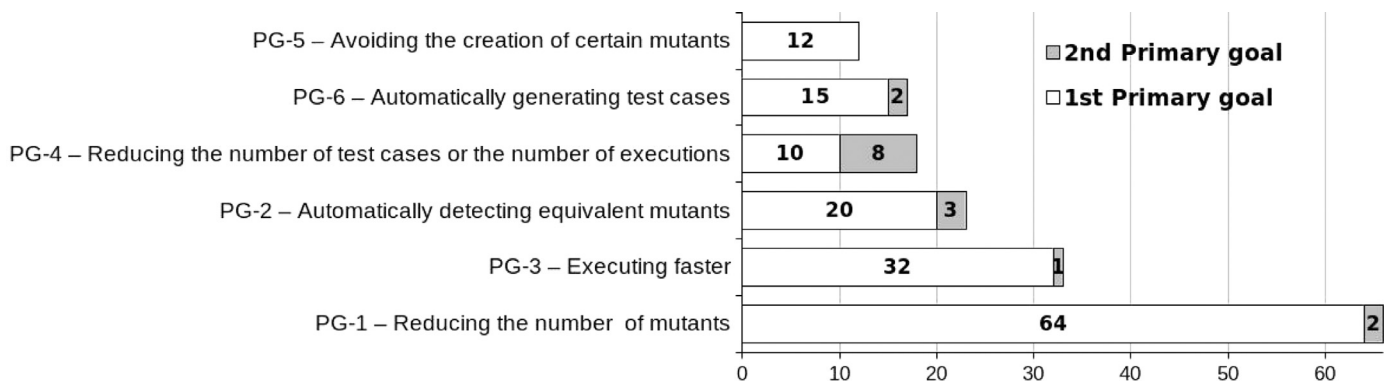


Fig. 5. Number of studies per primary goal.

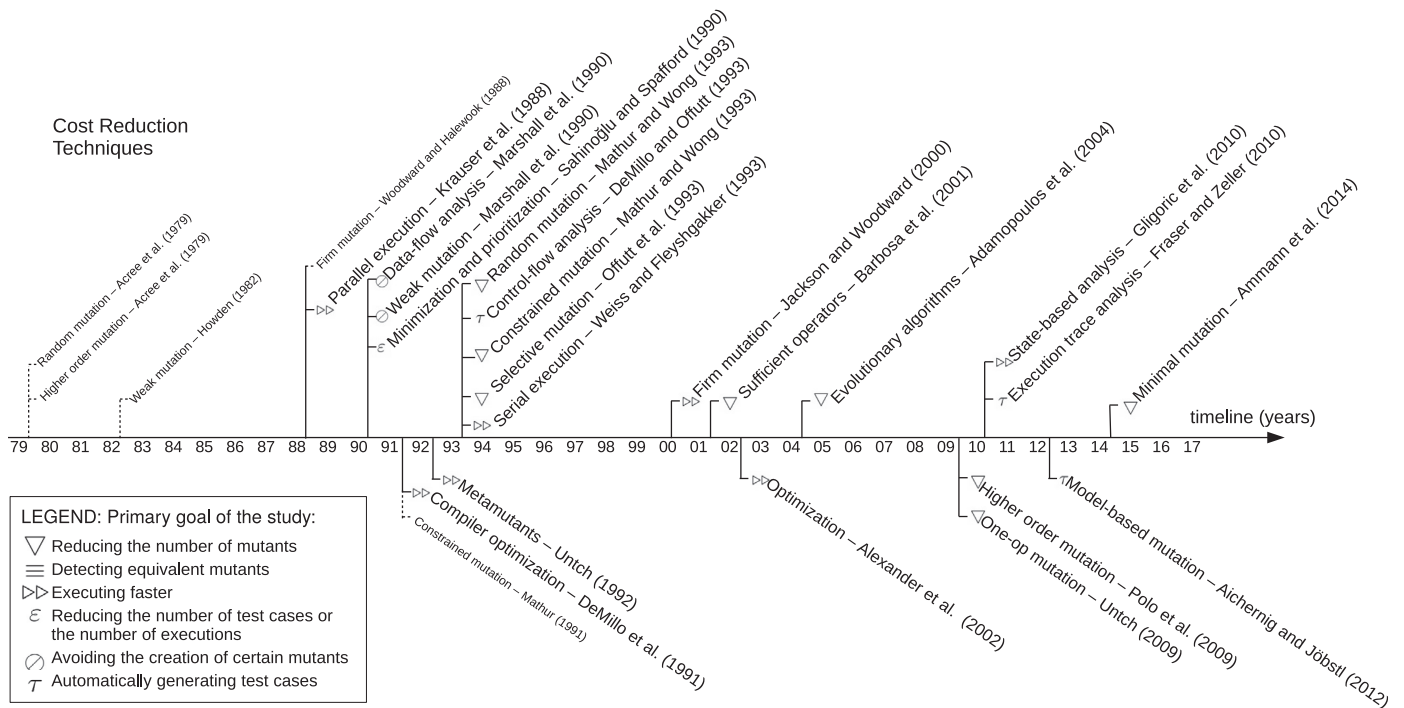


Fig. 6. Timeline for introduction of cost reduction techniques in peer-reviewed studies.

1982; Marshall et al., 1990); *firm mutation* (Woodward and Halewood, 1988; Jackson and Woodward, 2000); and *constrained mutation* (Mathur, 1991; Mathur and Wong, 1993). These techniques were first described either without a focus on cost reduction (Howden, 1982; Woodward and Halewood, 1988), in a high level way (Mathur, 1991), or in a non-peer-reviewed publication (Acree et al., 1979), and were later published with more details in peer-reviewed studies (Marshall et al., 1990; Jackson and Woodward, 2000; Mathur and Wong, 1993; Polo et al., 2009). These initial references (Acree et al., 1979; Howden, 1982; Woodward and Halewood, 1988; Mathur, 1991) appear with different notation (dotted lines and smaller font size). Also, the techniques in Fig. 6 are annotated with the symbols from the list of primary goals. For instance, *data-flow analysis* and *weak mutation* were first explored to *avoid the creation of certain mutants* ($\odot$) (Marshall et al., 1990), whereas *metamutants* was first explored for *executing faster* ($\gg$) (Untch, 1992).

The following list describes the 21 techniques. The descriptions are brief and omit details that were published in later years. Some categories have some overlap (including *selective mutation*, *sufficient operators*, *one-op mutation*, and *minimal mutation*), as discussed in Section 5.4.

- T-1: *Random mutation*: This technique selects randomly from the complete set of mutants according to a predefined probability distribution. Random mutation has been interpreted as “choose X% of all mutants,” “for each mutant, generate it with X% probability,” and as “choose X% of mutants generated by each operator” (Acree et al., 1979; Mathur and Wong, 1993; Derezińska and Rudnik, 2017; Gopinath et al., 2017; Parsai et al., 2016; Bluemke and Kulesza, 2013; Papadakis and Malevris, 2010a; Zhang et al., 2010a; Wong and Mathur, 1995; Zhang et al., 2013a; Petrović and Ivanković, 2018; Kurtz et al., 2016).
- T-2: *Higher order mutation*: This technique combines two or more simple mutations to create a single complex mutant (Acree et al., 1979; Gopinath et al., 2018; Lima et al., 2016; Reuling et al., 2015; Harman et al., 2014; Madeyski et al., 2014; Reales et al., 2013; Omar and Ghosh, 2012; Kaminski et al., 2011b; Kintis et al., 2010; Polo et al., 2009; Abuljadayel and Wedyan, 2018; Derezińska, 2016; Kaminski and Ammann, 2011; Papadakis and Malevris, 2010a; Devroey et al., 2016; Kintis et al., 2015).
- T-3: *Weak mutation*: This execution technique checks whether the state of the mutant is infected shortly after the mutated location has been executed, rather than checking the output after execution ends. If the state is infected, the mutant is killed immediately (Howden, 1982; Zhu et al., 2017; Papadakis and Malevris, 2011b; Offutt and Lee, 1994; Zhu et al., 2018; Devroey et al., 2017; Ma and Kim, 2016; Fraser and Arcuri, 2015; Zhang et al., 2010b; Kintis et al., 2010; Marshall et al., 1990; Kim et al., 2012).
- T-4: *Firm mutation*: This is a variant of weak mutation where execution is allowed to proceed for some pre-defined duration after the state is infected (Woodward and Halewood, 1988; Jackson and Woodward, 2000; Offutt and Lee, 1994).
- T-5: *Parallel execution*: This technique executes mutants in parallel processors, reducing the total time needed to perform mutation analysis (Li et al., 2015; Reales and Polo, 2013; Choi and Mathur, 1993; Offutt et al., 1992; Krauser et al., 1991; Wang et al., 2017; Jackson and Woodward, 2000).
- T-6: *Data-flow analysis*: This technique uses program data flow-related information to decide which mutants to generate and to analyze mutants. It considers whether variables that are more prone to failure during execution are reached and referenced (Al-Hajjaji et al., 2017; Kintis and Malevris, 2015; Papadakis et al., 2014; Kintis and Malevris, 2014; Schuler and Zeller, 2013; Papadakis and Le Traon, 2013; Just et al., 2012b; Patrick et al., 2006; Hierons et al., 1999; Offutt and Craft, 1994; Marshall et al., 1990; Delgado-Pérez et al., 2017a; Kintis et al., 2015; Harman et al., 2000).
- T-7: *Minimization and prioritization of test sets*: This technique analyzes the test suite to score test cases based on their effectiveness at killing mutants, then either eliminates test cases that are ineffective or runs the most effective test cases before the less effective test cases (Sahinoğlu and Spafford, 1990; Derezińska, 2013; Tuya et al., 2007).
- T-8: *Compiler optimization*: This technique uses compiler-related techniques to optimize mutant execution and analysis (for instance, to automatically detect equivalent mutants) (Denisov and Pankevich, 2018; Kintis et al., 2017; DeMillo et al., 1991; Delgado-Pérez et al., 2017; Just et al., 2011; Offutt and Craft, 1994).
- T-9: *Constrained mutation*: This technique chooses a subset of mutation operators to use. The choice relies on testers’ intuition regarding the significance of particular groups of mutants (Mathur, 1991; Mathur and Wong, 1993; Petrović and Ivanković, 2018; Wong et al., 1994; Gopinath et al., 2017; Wong and Mathur, 1995).
- T-10: *Metamutants (or mutant schemata)*: This technique generates and executes mutants by embedding all mutants in one parameterized program called a *metamutant*. The metamutant is then compiled for fast execution. When run, the metamutant takes a parameter that tells it which mutant to run (Untch, 1992; Gopinath et al., 2016b; Reales and Polo, 2014; Wright et al., 2013; Ma et al., 2005; Untch et al., 1997; 1993; Kim et al., 2012; Papadakis and Malevris, 2010b; Weiss and Fleishgaker, 1993; Ma and Kim, 2016; Papadakis and Malevris, 2011b).
- T-11: *Control-flow analysis*: This technique uses program control flow-related information, focusing on execution characteristics to identify branches and commands that help determine which structures are most relevant to the generation and execution of mutants (Marcozzi et al., 2018; Chen and Zhang, 2018; McMinn et al., 2018; Delgado-Pérez et al., 2017a; Just et al., 2017; Sun et al., 2017b; Holling et al., 2016; Patel and Hierons, 2016; Kintis et al., 2015; Zhang et al., 2012; Papadakis and Malevris, 2012; Kaminski and Ammann, 2011; Zhang et al., 2010b; Papadakis and Malevris, 2010b; 2009; Ji et al., 2009; Liu et al., 2006; Harman et al., 2000; Offutt and Pan, 1997; DeMillo and Offutt, 1993; Petrović and Ivanković, 2018; Papadakis et al., 2014; Schuler and Zeller, 2013; Papadakis and Le Traon, 2013; Just et al., 2012b; Patrick et al., 2009; Offutt et al., 2006; Bashir and Nadeem, 2017; Fraser and Arcuri, 2015; DeMillo and Offutt, 1991; Offutt et al., 1999).
- T-12: *Selective mutation*: This technique tries to avoid the application of mutation operators that are responsible for the most mutants or to select mutation operators that result in mutants that are killed by tests that also kill lots of mutants created by other operators. The idea is that if a test set, T_{op} , that is adequate for a subset of mutation operators op , also kills a very high percentage of all mutants, then we can select only the operators in op (Sun et al., 2017a; Praphamontripong and Offutt, 2017; Delgado-Pérez et al., 2017c; Bluemke and Kulesza, 2014b; 2014a; Zhang et al., 2013a; Gligoric et al., 2013; Derezińska and Rudnik, 2012; Sridharan and Siami-Namin, 2010; Mresa and Bottaci, 1999; Offutt et al., 1996; 1993; Kurtz et al., 2016; Zhang et al., 2010a; Tuya et al., 2007; Delgado-Pérez et al., 2017; Gopinath et al., 2017).

- T-13: *Serial execution*: This technique dynamically determines classes of mutants that behave similarly, thus decreasing the number of mutants to be executed. The overhead incurred needs to be small relative to the time saved (Ma and Kim, 2016; Kim et al., 2012; Fleishgaker and Weiss, 1994; Weiss and Fleishgaker, 1993).
- T-14: *Sufficient operators*: This technique tries to determine an essential set of mutation operators by applying customized procedures (Just and Schweiggert, 2015; Lacerda and Ferrari, 2014; Delamaro et al., 2014b; Siami-Namin et al., 2008; Siami-Namin and Andrews, 2006; Vincenzi et al., 2001; Barbosa et al., 2001; Zhang et al., 2010a). It can be seen as a special case of *selective mutation*, where the main difference is the complexity of the procedures applied to identify the final subset of mutation operators, which may be based, for example, on heuristics or statistics).
- T-15: *Optimization of generation, execution, and analysis of mutants*: This technique groups approaches that reduce the cost of mutation testing by exploring strategies that did not fit other categories on this list. This category appears as *Optimization* in Fig. 6 for brevity. More details are given later in this section (Zhu et al., 2018; Iida and Takada, 2017; Fernandes et al., 2017; McMinn et al., 2016; Zhang et al., 2016; Derezińska and Hałas, 2015; Belli and Beyazit, 2015; Just et al., 2014a; Derezińska, 2013; Kaminski et al., 2013; Ferrari et al., 2013; Kintis and Malevris, 2013; Cachia et al., 2013; Inozemtseva et al., 2013; Zhang et al., 2013b; Wedyan and Ghosh, 2012; Gligoric et al., 2012; Durelli et al., 2012; Hu et al., 2011; Just et al., 2011; Steimann and Thies, 2010; Kaminski and Ammann, 2009; Anbalagan and Xie, 2008; Bogacki and Walter, 2006a; Vincenzi et al., 2002; Alexander et al., 2002; Chen and Zhang, 2018; Zhu et al., 2017; Devroey et al., 2016; Gopinath et al., 2016b; Just and Schweiggert, 2015; Reuling et al., 2015; Wright et al., 2013; Zhang et al., 2012).
- T-16: *Evolutionary algorithms*: This technique uses evolutionary algorithms to reduce the number of mutants, to reduce the number of test cases, or to identify equivalent mutants (Abuljadayel and Wedyan, 2018; Delgado-Pérez et al., 2017; Bashir and Nadeem, 2017; Delgado-Pérez et al., 2017b; Quyen et al., 2016; Matnei Filho and Vergilio, 2015; Fraser and Arcuri, 2015; Henard et al., 2014; Oliveira et al., 2013; Nobre et al., 2012; Fraser and Zeller, 2012; Papadakis and Malevris, 2011a; Domínguez-Jiménez et al., 2011; Domínguez-Jiménez et al., 2009a; Ayari et al., 2007; Baudry et al., 2005; Adamopoulos et al., 2004; Harman et al., 2014; Sridharan and Siami-Namin, 2010; Ji et al., 2009; Papadakis and Malevris, 2011b).
- T-17: *One-op mutation*: This technique uses only a single mutation operator, which has the advantage of producing few mutants but also providing comprehensive coverage of the program (Derezińska, 2016; Delamaro et al., 2014a; 2014c; Deng et al., 2013; Untch, 2009; Kurtz et al., 2016).
- T-18: *Execution trace analysis*: This technique uses traces of execution of the original program and some mutants to decide which of the remaining mutants should be executed (Reales and Polo, 2014; Fraser and Zeller, 2012).
- T-19: *Model-based mutation*: This technique mutates formal or informal models of the program, and then uses the mutants to automatically generate test cases that are later used to kill mutants of the program (Devroey et al., 2017; 2016; Aichernig et al., 2014; 2013).
- T-20: *State-based analysis*: This technique compares states of different mutant executions. When two mutants lead to the same mutation state, that is, when the same execution path is observed, only one needs to be executed and the result of

the other can be inferred. Similarly, the technique creates groups of classes that define certain transition sequences such that only one needs to be verified (Wang et al., 2017; Gligoric et al., 2010; Bashir and Nadeem, 2017; Just et al., 2014a; Gligoric et al., 2012).

- T-21: *Minimal mutation*: This technique identifies and eliminates redundant mutants by applying the concepts of mutant subsumption and dominator mutants (Kurtz et al., 2016; Ammann et al., 2014; Just et al., 2017).

Note that T-15 (*optimization of generation, execution, and analysis of mutants*) classifies studies that do not fit into any other category. For instance, Alexander et al. (2002) implemented a mutation tool that uses *Java* reflection to mutate objects at run-time, thus avoiding the need to compile separate mutated programs. Durelli et al. (2012) evolved the *Java* Virtual Machine to embed native support to speed up the execution of the original program and its mutants. Derezińska (2013) proposed a clustering algorithm that uses results of mutant execution on subsets of test cases. A more recent example is Zhang et al. (2016)'s work, which used machine learning to devise predictive models to avoid the execution of some mutants. We classified these and other similar techniques as *optimization*-related. These are described in Section 5.5.

Fig. 7 shows the distribution of studies per technique. Some studies used two or more techniques (discussed further in Section 5.4). Studies that used more than one technique are listed in Fig. 7 with counts. For example, *minimal mutation* (T-21) has two units for *Used as 1st Technique* and one unit for *Used as 2nd Technique*. The amount of emphasis of each technique is based on the focus authors gave to that particular technique. As an example, Marshall et al. (1990) reduced mutation costs by applying *data-flow analysis* to support *weak mutation*. Thus, *data-flow analysis* was the main technique while *weak mutation* was secondary. In another example, Wong and Mathur (1995) compared *random mutation* with *constrained mutation*, without any clear preference for either. In this case, assigning higher participation for a particular technique was arbitrary. Therefore, even though Fig. 7 shows varied levels of participation for most techniques, both techniques might have had similar importance.

Fig. 8 shows the distribution of the nine most investigated techniques per year, split into three charts to improve readability. Even without statistical tests for time series, it is clear that researchers are studying some techniques more in recent years, such as *evolutionary algorithms*, *control-flow analysis*, *higher order mutation*, and *selective mutation*. Fig. 9 illustrates this by showing the distribution of the six techniques most investigated over the five last years (2014–2018).

5.4. Additional notes about the cost reduction techniques and primary goals

As mentioned in Section 5.3, some of the techniques are similar. For instance, *constrained mutation* (Mathur, 1991), *selective mutation* (Offutt et al., 1993), *sufficient operators* (Barbosa et al., 2001), *one-op mutation* (Untch, 2009), and the recent *minimal mutation* technique (Ammann et al., 2014) all reduce the number of mutants in some way, as far as possible without reducing effectiveness. After considering putting them all into one large category, we decided that a fine-grained categorization would be more helpful, and that small categories would be simpler to understand. Thus, we divided them on fairly specific characteristics. For example, *constrained mutation* relies on the testers' intuition, while *one-op mutation* relies on individual mutation operators. Along the same lines, we chose to separate *weak mutation* (Howden, 1982) and *firm mutation* (Woodward and Halewood, 1988), even though

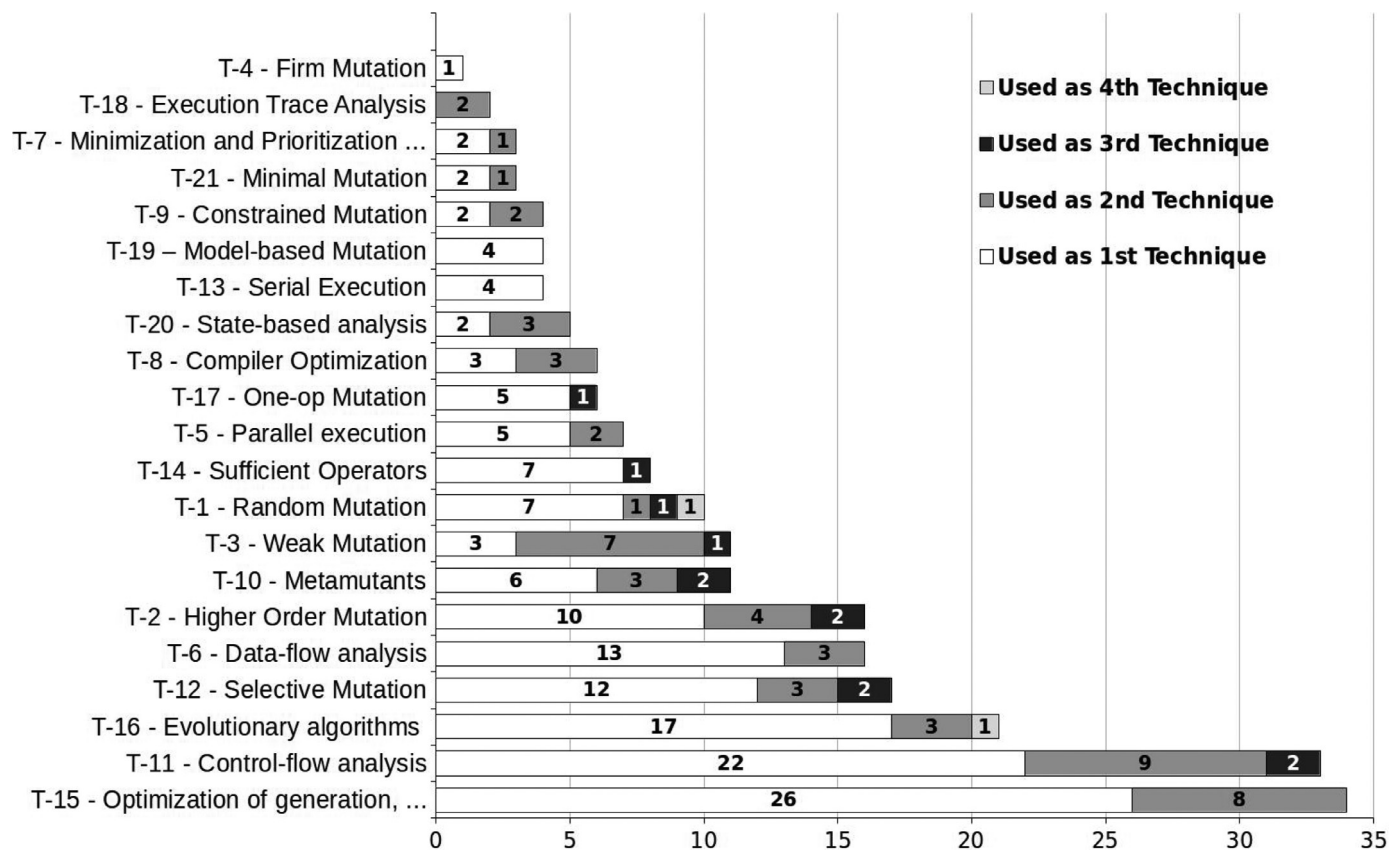


Fig. 7. Number of studies per cost reduction technique.

weak mutation could be viewed as a special case of firm mutation.

Another interesting observation is that the frequency of studies that combine techniques has increased over time. Fig. 10 shows that techniques have been combined since research on cost reduction started, but this combined usage has increased since 2010. In 2009, two studies combined two techniques (Schuler et al., 2009; Ji et al., 2009), but in 2010 five studies combined two techniques (Kintis et al., 2010; Papadakis and Malevris, 2010a; Sridharan and Siami-Namin, 2010; Papadakis and Malevris, 2010b; Zhang et al., 2010b), and one combined three (Zhang et al., 2010a). By 2017, six studies combined two techniques (Wang et al., 2017; Just et al., 2017; Zhu et al., 2017; Devroey et al., 2017; Delgado-Pérez et al., 2017a; Abuljadayel and Wedyan, 2018), and three combined three (Bashir and Nadeem, 2017; Delgado-Pérez et al., 2017; Gopinath et al., 2017). We also found a few studies that combined four techniques starting in 2011 (Papadakis and Malevris, 2011b; Kurtz et al., 2016). In total, of the 107 studies selected since 2010, 45 combined at least two techniques. Table 3 lists the studies that combined two or more techniques per year (the techniques are shown between parentheses).

Table 4 relates primary goals, cost reduction techniques, and studies. For each primary goal, the table lists all studies and the applied cost reduction techniques. The number of times a study is listed for a given goal corresponds to the number of applied techniques applied in that study, irrespective of which technique was applied to achieve a particular goal. For example, for *reducing the number of mutants* (PG-1), Gopinath et al. (2017) applied three techniques: *t-1* (random mutation), *T-9* (constrained mutation) and *T-12* (selective mutation). Therefore, Gopinath et al.'s study is listed three times for PG-1.

Note that some studies tried to achieve two primary goals and applied two or more techniques. An example is by Just et al. (2012b). That study applied techniques *T-6* (Data-flow analysis) and *T-11* (Control-flow analysis) to reduce the number of test cases or the number of executions (PG-4), and to avoid the creation of certain mutants (PG-5). In cases like this, the combination of techniques may allow one to achieve a combination of primary goals, therefore we do not distinguish between particular techniques applied to achieve particular goals.

Table 5 relates cost reduction techniques, primary goals, and studies. For each technique, the table lists all studies that addressed a particular goal. Similarly to Table 4, the number of times a study is listed corresponds to the number of primary goals it pursued times the number of techniques it applied (e.g. Just et al. (2012b)'s study is listed two times for *T-6* and two times for *T-11*).

The reader should notice that providing a complete list of tools that implement these cost reduction techniques is outside the scope of this paper. Readers could use information in this paper to identify the tools used in the experiments. Examples of useful information are the lists of references to studies with respect to primary goals (Section 5.2), techniques (Section 5.3), and the relationships between them (Tables 4 and 5).

5.5. Overview of selected studies

We describe each of the 153 selected studies in detail on the companion website to this paper (Ferrari et al., 2018b).⁹ We

⁹ <http://goo.gl/edyF1n> –last accessed November 2018.

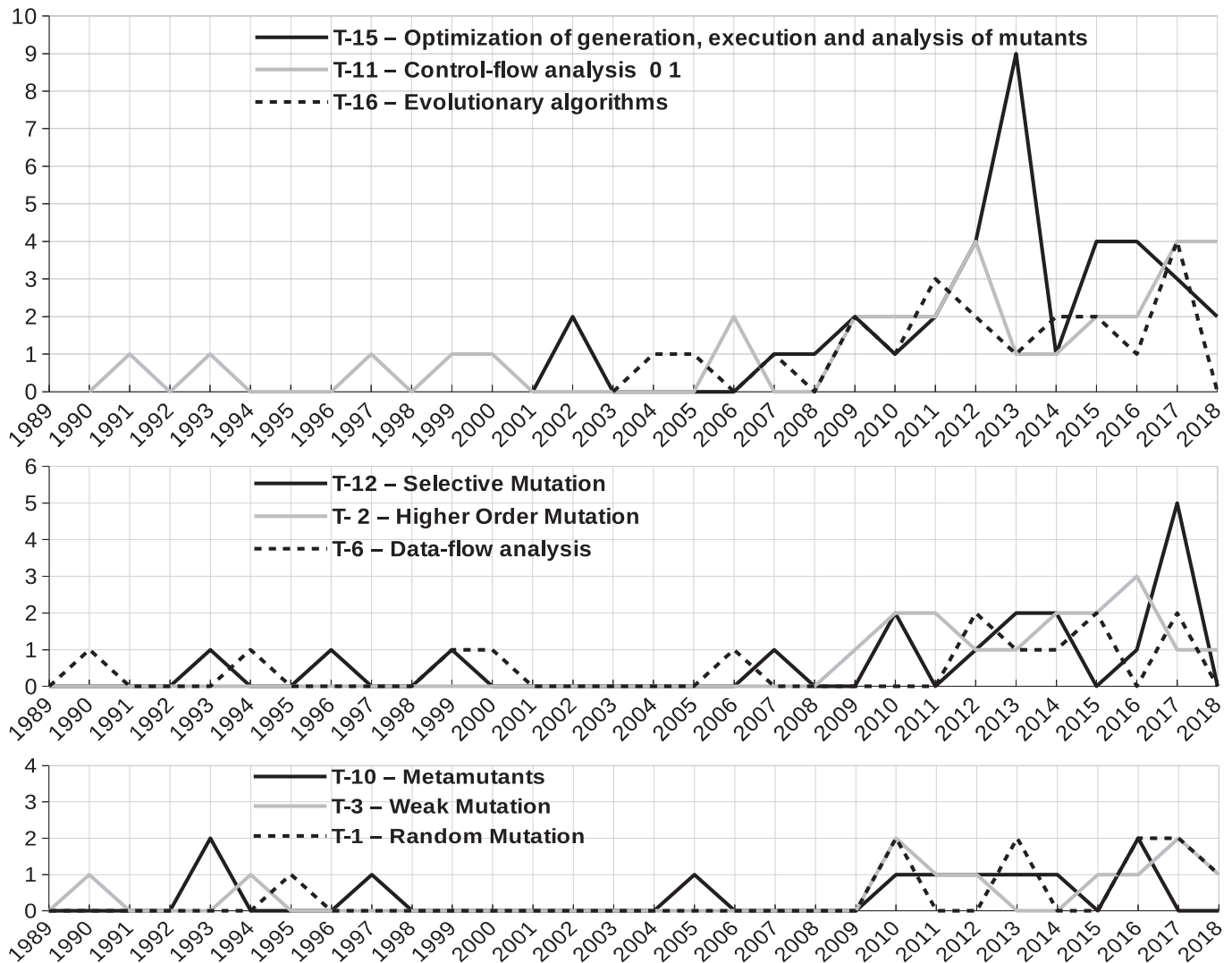


Fig. 8. Distribution of studies of the top nine techniques per year.

include a BibTeX file with an entry for every study, and a description field with more details than in this paper. The descriptions include the cost reduction goals of the studies (either explicit or implicit), and the achieved results. Every description also includes the cost reduction technique as described in Section 5.3, and the technology and artifacts addressed (if provided by the original authors). As an example, the study of Offutt and Craft (1994) is described in the complementary material as: “Offutt and Craft (1994) had the goal of detecting equivalent mutants. They used six techniques developed for compiler optimization. They analyzed the flow of data in mutants to identify equivalent mutants. In their experiment, they used 15 small Fortran programs and 14 mutation operators. The study automatically detected an average of 19.80% of the equivalent mutants.”

The remainder of this section provides a more concise description of the selected studies, grouped by cost reduction goal and applied cost reduction technique. The groups of studies are organized chronologically, with a few exceptions when very interrelated studies are described within a given group (e.g. the studies by Domínguez-Jiménez et al. (2011) and Delgado-Pérez et al. (2017b), which explored Evolutionary Mutation Testing to reduce the number of mutants to be executed). When possible, we established re-

lationships between studies that extended prior studies. For the sake of completeness, studies that pursued two primary goals appear twice, each one in the subsection for each goal.

5.5.1. Studies that tried to reduce the number of mutants (PG-1)

Offutt et al. (1993) extended and used the Mothra (DeMillo and Offutt, 1991) tool to experiment with several selective mutation options on 10 Fortran programs. Offutt et al. (1996) extended their prior work (Offutt et al., 1993) by establishing a reduced set of operators (ABS, UOI, LCR, AOR, and ROR). Mresa and Bottaci (1999) proposed a new type of selective mutation that takes into account the cost of detecting equivalent mutants. Tuya et al. (2007) proposed mutation operators for SQL. They used selective mutation and minimization and prioritization of test sets to reduce the number of mutants and the number of test cases. Zhang et al. (2010a) empirically compared random mutation with selective mutation and sufficient operators. Results of experiments with 7 medium-sized C programs provided hints that 7%-random selection would achieve similar test effectiveness. Sridharan and Siarni-Namin (2010) used the same programs used by Zhang et al. to experiment with a Bayesian (evolutionary) approach to select the operators that

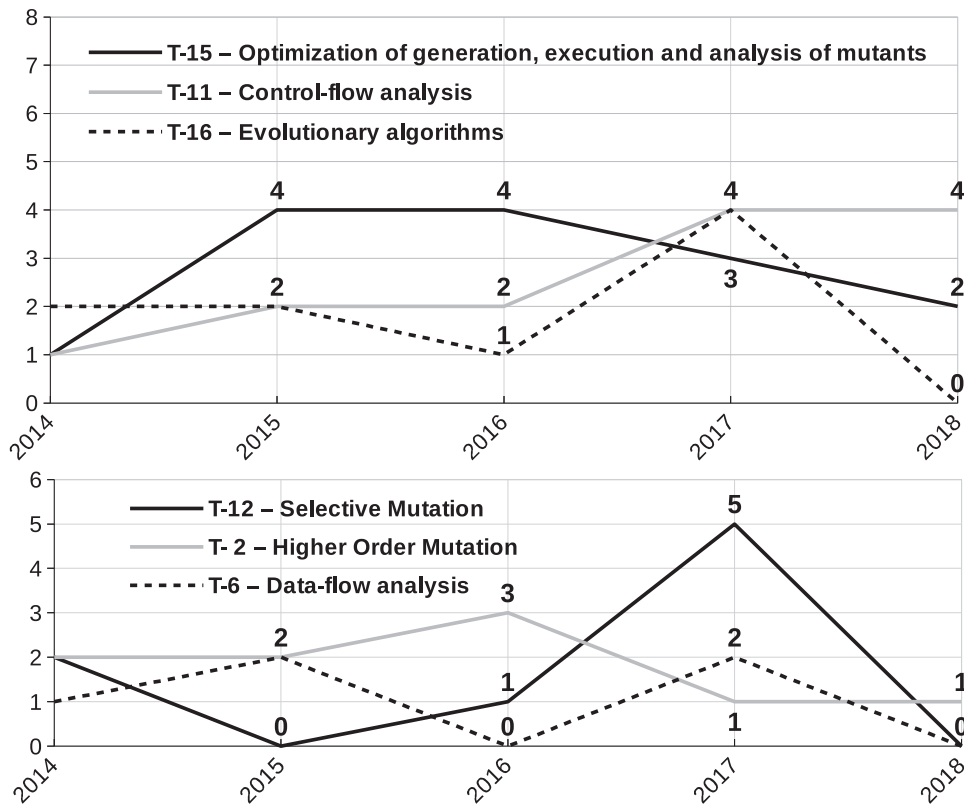


Fig. 9. Distribution of studies of the top six techniques in the last five years.

Table 3

List of studies that applied combined techniques per year (years with no studies are omitted).

Year	Studies and Respective Cost Reduction Techniques
1990	Marshall et al. (1990) (T-6,T-3)
1993	Weiss and Fleishgakkker (1993) (T-13,T-10)
1994	Offutt and Craft (1994) (T-6,T-8)
1995	Wong and Mathur (1995) (T-1,T-9)
2000	Harman et al. (2000) (T-11,T-6); Jackson and Woodward (2000) (T-4,T-5)
2006	Offutt et al. (2006) (T-6,T-11)
2007	Tuya et al. (2007) (T-7,T-12)
2009	Schuler et al. (2009) (T-6,T-11); Ji et al. (2009) (T-11,T-16)
2010	Zhang et al. (2010a) (T-1,T-12,T-14); Kintis et al. (2010) (T-2,T-3); Papadakis and Malevris (2010a) (T-1,T-2); Sridharan and Siami-Namin (2010) (T-12,T-16); Papadakis and Malevris (2010b) (T-11,T-10); Zhang et al. (2010b) (T-11,T-3)
2011	Just et al. (2011) (T-15,T-8); Kaminski and Ammann (2011) (T-11,T-2); Papadakis and Malevris (2011b) (T-3,T-11,T-10,T-16)
2012	Kim et al. (2012) (T-13,T-10,T-3); Gligoric et al. (2012) (T-15,T-20); Zhang et al. (2012) (T-11,T-15); Just et al. (2012b) (T-6,T-11); Patrick et al. (2012) (T-6,T-11); Fraser and Zeller (2012) (T-16,T-18)
2013	Papadakis and Le Traon (2013) (T-6,T-11); Zhang et al. (2013a) (T-12,T-1); Schuler and Zeller (2013) (T-6,T-11); Wright et al. (2013) (T-10,T-15); Derezińska (2013) (T-15,T-7)
2014	Just et al. (2014a) (T-15,T-20); Papadakis et al. (2014) (T-6,T-11); Harman et al. (2014) (T-2,T-16); Reales and Polo (2014) (T-10,T-18)
2015	Reuling et al. (2015) (T-2,T-15); Kintis et al. (2015) (T-11,T-6,T-2); Fraser and Arcuri (2015) (T-16,T-3,T-11); Just and Schweiggert (2015) (T-14,T-15)
2016	Ma and Kim (2016) (T-13,T-3,T-10); Kurtz et al. (2016) (T-21,T-12,T-17,T-1); Gopinath et al. (2016b) (T-10,T-15); Devroey et al. (2016) (T-19,T-15,T-2); Derezińska (2016) (T-17,T-2)
2017	Wang et al. (2017) (T-20,T-5); Just et al. (2017) (T-11,T-21); Zhu et al. (2017) (T-3,T-15); Bashir and Nadeem (2017) (T-16,T-20,T-11); Devroey et al. (2017) (T-19,T-3); Gopinath et al. (2017) (T-1,T-9,T-12); Delgado-Pérez et al. (2017a) (T-11,T-6); Delgado-Pérez et al. (2017) (T-16,T-8,T-12); Abuljadayel and Wedyan (2018) (T-16,T-2)
2018	Chen and Zhang (2018) (T-11,T-15); Zhu et al. (2018) (T-15,T-3); Petrović and Ivanković (2018) (T-9,T-11,T-1)

generated mutants that were hard to kill. Derezińska and Rudnik (2012) investigated *selective mutation* for C# programs. Gligoric et al. (2013) applied *selective mutation* to concurrent Java programs. Zhang et al. (2013a) applied *selective mutation* and subsequently applied *random mutation*. Bluemke and Kulesza (2014a,b) also explored *selective mutation* and a combination of

tools for test case generation, mutant generation, and program analysis. Kurtz et al. (2016) claimed that *selective mutation* must be specialized to avoid noise introduced by redundant mutants. Gopinath et al. (2017) theoretically and empirically compared several cost reduction techniques (*selective mutation*, *constrained mutation*, and *random mutation*), and used statistical analysis to

Table 4

Mapping between primary goals and cost reduction techniques.

Primary Goal	Cost Reduction Techniques and Respective Studies
PG-1 – Reducing the number of mutants	T-1 (Zhang et al., 2010a; 2013a; Wong and Mathur, 1995; Papadakis and Malevris, 2010a; Kurtz et al., 2016; Parsai et al., 2016; Gopinath et al., 2017; Derezińska and Rudnik, 2017; Petrović and Ivanković, 2018), T-2 (Kintis et al., 2010; Kaminski et al., 2011b; Polo et al., 2009; Reales et al., 2013; Reuling et al., 2015; Papadakis and Malevris, 2010a; Omar and Ghosh, 2012; Madeyski et al., 2014; Harman et al., 2014; Lima et al., 2016; Derezińska, 2016; Abuljadayel and Wedyan, 2018), T-3 (Kintis et al., 2010; Zhu et al., 2017; 2018), T-6 (Patrick et al., 2012; Al-Hajjaji et al., 2017), T-7 (Tuya et al., 2007), T-8 (Delgado-Pérez et al., 2017), T-9 (Wong and Mathur, 1995; Wong et al., 1994; Gopinath et al., 2017; Petrović and Ivanković, 2018), T-11 (Sun et al., 2017b; Patrick et al., 2012; Ji et al., 2009; Just et al., 2017; Marcozzi et al., 2018; Petrović and Ivanković, 2018; McMinn et al., 2018), T-12 (Zhang et al., 2010a; Offutt et al., 1993; Tuya et al., 2007; Mresa and Bottaci, 1999; Offutt et al., 1996; Gligoric et al., 2013; Bluemke and Kulesza, 2014a; Zhang et al., 2013a; Sridharan and Siami-Namin, 2010; Kurtz et al., 2016; Delgado-Pérez et al., 2017c; Gopinath et al., 2017; Praphamontipong and Offutt, 2017; Derezińska and Rudnik, 2012; Bluemke and Kulesza, 2014b; Delgado-Pérez et al., 2017; Sun et al., 2017a), T-14 (Zhang et al., 2010a; Siami-Namin et al., 2008; Vincenzi et al., 2001; Barbosa et al., 2001; Delamaro et al., 2014b; Siami-Namin and Andrews, 2006; Lacerda and Ferrari, 2014), T-15 (Wedyan and Ghosh, 2012; Inozemtseva et al., 2013; Reuling et al., 2015; Cachia et al., 2013; Kaminski and Ammann, 2009; Zhu et al., 2017; Derezińska and Hałas, 2015; Zhu et al., 2018), T-16 (Domínguez-Jiménez et al., 2011; Oliveira et al., 2013; Sridharan and Siami-Namin, 2010; Adamopoulos et al., 2004; Ji et al., 2009; Harman et al., 2014; Delgado-Pérez et al., 2017b; Quyen et al., 2016; Nobre et al., 2012; Delgado-Pérez et al., 2017; Abuljadayel and Wedyan, 2018), T-17 (Untch, 2009; Delamaro et al., 2014c; 2014a; Deng et al., 2013; Kurtz et al., 2016; Derezińska, 2016), T-21 (Ammann et al., 2014; Kurtz et al., 2016; Just et al., 2017)
PG-2 – Automatically detecting equivalent mutants	T-2 (Kintis et al., 2015), T-3 (Offutt and Lee, 1994; Devroey et al., 2017), T-6 (Hierons et al., 1999; Papadakis and Le Traon, 2013; Schuler et al., 2009; Schuler and Zeller, 2013; Offutt et al., 2006; Kintis and Malevris, 2015; Papadakis et al., 2014; Offutt and Craft, 1994; Harman et al., 2000; Kintis et al., 2015), T-8 (Offutt and Craft, 1994; Kintis et al., 2017; Delgado-Pérez et al., 2017), T-11 (Offutt and Pan, 1997; Papadakis and Le Traon, 2013; Schuler et al., 2009; Schuler and Zeller, 2013; Offutt et al., 2006; Papadakis et al., 2014; Harman et al., 2000; Kintis et al., 2015; Patel and Hierons, 2016; Holling et al., 2016; Marcozzi et al., 2018; McMinn et al., 2018), T-12 (Delgado-Pérez et al., 2017), T-15 (Kintis and Malevris, 2013; Ferrari et al., 2013; Anbalagan and Xie, 2008; Vincenzi et al., 2002), T-16 (Delgado-Pérez et al., 2017), T-19 (Devroey et al., 2017)
PG-3 – Executing faster	T-2 (Devroey et al., 2016), T-3 (Offutt and Lee, 1994; Kim et al., 2012; Papadakis and Malevris, 2011b; Devroey et al., 2017), T-4 (Jackson and Woodward, 2000), T-5 (Choi and Mathur, 1993; Offutt et al., 1992; Reales and Polo, 2013; Wang et al., 2017; Krauser et al., 1991; Li et al., 2015; Jackson and Woodward, 2000), T-8 (Just et al., 2011; DeMillo et al., 1991; Denisov and Pankevich, 2018), T-10 (Weiss and Fleishgakkker, 1993; Ma et al., 2005; Kim et al., 2012; Untch et al., 1993; 1997; Wright et al., 2013; Papadakis and Malevris, 2011b; Gopinath et al., 2016b; Reales and Polo, 2014), T-11 (Zhang et al., 2012; Papadakis and Malevris, 2011b; Chen and Zhang, 2018), T-13 (Weiss and Fleishgakkker, 1993; Kim et al., 2012; Fleishgakkker and Weiss, 1994), T-15 (Durelli et al., 2012; Gligoric et al., 2012; Zhang et al., 2012; Just et al., 2014a; 2011; Wright et al., 2013; Zhang et al., 2016; Gopinath et al., 2016b; Devroey et al., 2016; McMinn et al., 2016; Alexander et al., 2002; Bogacki and Walter, 2006a; Chen and Zhang, 2018), T-16 (Papadakis and Malevris, 2011b), T-18 (Reales and Polo, 2014), T-19 (Devroey et al., 2016; 2017), T-20 (Gligoric et al., 2012; Just et al., 2014a; Gligoric et al., 2010; Wang et al., 2017)
PG-4 – Reducing the number of test cases or the number of executions	T-2 (Kaminski and Ammann, 2011; Harman et al., 2014; Devroey et al., 2016; Gopinath et al., 2018), T-3 (Fraser and Arcuri, 2015; Ma and Kim, 2016; Zhu et al., 2017; 2018), T-6 (Just et al., 2012b), T-7 (Tuya et al., 2007; Sahinoğlu and Spafford, 1990), T-10 (Ma and Kim, 2016), T-11 (Papadakis and Malevris, 2012; Kaminski and Ammann, 2011; Just et al., 2012b; Fraser and Arcuri, 2015), T-12 (Tuya et al., 2007), T-13 (Ma and Kim, 2016), T-15 (Zhang et al., 2013b; Derezińska, 2013; Devroey et al., 2016; Zhu et al., 2017; Derezińska and Hałas, 2015; Zhu et al., 2018), T-16 (Ayari et al., 2007; Oliveira et al., 2013; Adamopoulos et al., 2004; Fraser and Arcuri, 2015; Harman et al., 2014), T-19 (Devroey et al., 2016)
PG-5 – Avoiding the creation of certain mutants	T-3 (Marshall et al., 1990), T-6 (Marshall et al., 1990; Just et al., 2012b; Kintis and Malevris, 2014; Delgado-Pérez et al., 2017a), T-11 (Just et al., 2012b; Delgado-Pérez et al., 2017a), T-14 (Just and Schweiggert, 2015), T-15 (Steimann and Thies, 2010; Kaminski et al., 2013; Hu et al., 2011; Belli and Beyazit, 2015; Just and Schweiggert, 2015; Fernandes et al., 2017; Iida and Takada, 2017), T-16 (Domínguez-Jiménez et al., 2009a)
PG-6 – Automatically generating test cases	T-3 (Papadakis and Malevris, 2011b; Zhang et al., 2010b), T-6 (Harman et al., 2000), T-10 (Papadakis and Malevris, 2010b; 2011b), T-11 (Offutt et al., 1999; Papadakis and Malevris, 2010b; Harman et al., 2000; Liu et al., 2006; Papadakis and Malevris, 2009; 2011b; DeMillo and Offutt, 1993; Zhang et al., 2010b; Bashir and Nadeem, 2017; DeMillo and Offutt, 1991), T-16 (Papadakis and Malevris, 2011a; Baudry et al., 2005; Papadakis and Malevris, 2011b; Bashir and Nadeem, 2017; Matnei Filho and Vergilio, 2015; Henard et al., 2014; Fraser and Zeller, 2012), T-18 (Fraser and Zeller, 2012), T-19 (Aichernig et al., 2013; 2014), T-20 (Bashir and Nadeem, 2017)

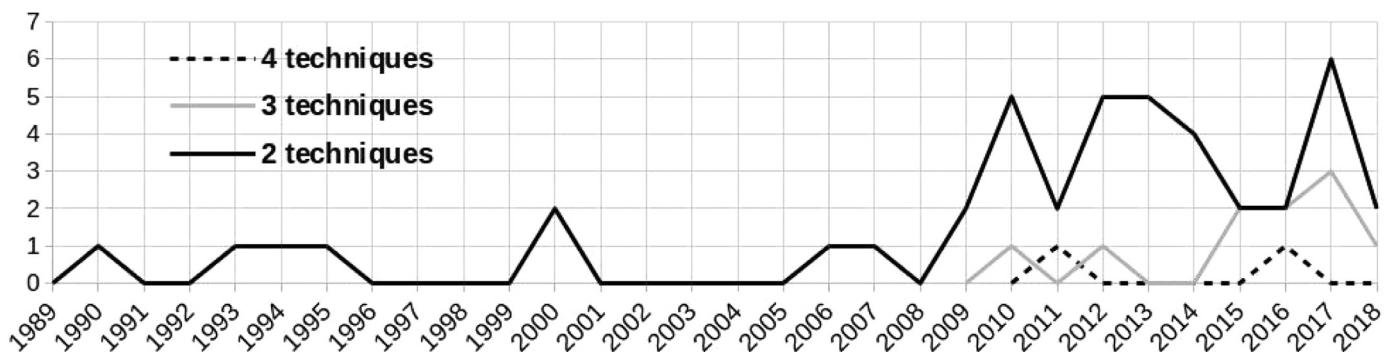
**Fig. 10.** Distribution of studies that applied combined techniques per year.

Table 5
Mapping between cost reduction techniques and primary goals.

Cost reduction technique	Primary goals and respective studies
T-1 – Random mutation	PG-1 (Zhang et al., 2010a; 2013a; Wong and Mathur, 1995; Papadakis and Malevris, 2010a; Kurtz et al., 2016; Parsai et al., 2016; Gopinath et al., 2017; Derezińska and Rudnik, 2017; Petrović and Ivanković, 2018)
T-2 – Higher order mutation	PG-1 (Kintis et al., 2010; Kaminski et al., 2011b; Polo et al., 2009; Reales et al., 2013; Reuling et al., 2015; Papadakis and Malevris, 2010a; Omar and Ghosh, 2012; Madeyski et al., 2014; Harman et al., 2014; Lima et al., 2016; Derezińska, 2016; Abuljadayel and Wedyan, 2018), PG-2 (Kintis et al., 2015), PG-3 (Devroey et al., 2016), PG-4 (Kaminski and Ammann, 2011; Harman et al., 2014; Devroey et al., 2016; Gopinath et al., 2018)
T-3 – Weak mutation	PG-1 (Kintis et al., 2010; Zhu et al., 2017; 2018), PG-2 (Offutt and Lee, 1994; Devroey et al., 2017), PG-3 (Offutt and Lee, 1994; Kim et al., 2012; Papadakis and Malevris, 2011b; Devroey et al., 2017), PG-4 (Fraser and Arcuri, 2015; Ma and Kim, 2016; Zhu et al., 2017; 2018), PG-5 (Marshall et al., 1990), PG-6 (Papadakis and Malevris, 2011b; Zhang et al., 2010b)
T-4 – Firm mutation	PG-3 (Jackson and Woodward, 2000)
T-5 – Parallel execution	PG-3 (Choi and Mathur, 1993; Offutt et al., 1992; Reales and Polo, 2013; Wang et al., 2017; Krauser et al., 1991; Li et al., 2015; Jackson and Woodward, 2000)
T-6 – Data-flow analysis	PG-1 (Patrick et al., 2012; Al-Hajjaji et al., 2017), PG-2 (Hierons et al., 1999; Papadakis and Le Traon, 2013; Schuler et al., 2009; Schuler and Zeller, 2013; Offutt et al., 2006; Kintis and Malevris, 2015; Papadakis et al., 2014; Offutt and Craft, 1994; Harman et al., 2000; Kintis et al., 2015), PG-4 (Just et al., 2012b), PG-5 (Marshall et al., 1990; Just et al., 2012b; Kintis and Malevris, 2014; Delgado-Pérez et al., 2017a), PG-6 (Harman et al., 2000)
T-7 – Minimization and prioritization of test sets	PG-1 (Tuya et al., 2007), PG-4 (Tuya et al., 2007; Sahinoğlu and Spafford, 1990)
T-8 – Compiler optimization	PG-1 (Delgado-Pérez et al., 2017), PG-2 (Offutt and Craft, 1994; Kintis et al., 2017; Delgado-Pérez et al., 2017), PG-3 (Just et al., 2011; DeMillo et al., 1991; Denisov and Pankevich, 2018)
T-9 – Constrained mutation	PG-1 (Wong and Mathur, 1995; Wong et al., 1994; Gopinath et al., 2017; Petrović and Ivanković, 2018)
T-10 – Metamutants	PG-3 (Weiss and Fleyshgaker, 1993; Ma et al., 2005; Kim et al., 2012; Untch et al., 1993; 1997; Wright et al., 2013; Papadakis and Malevris, 2011b; Gopinath et al., 2016b; Reales and Polo, 2014), PG-4 (Ma and Kim, 2016), PG-6 (Papadakis and Malevris, 2010b; 2011b)
T-11 – Control-flow analysis	PG-1 (Sun et al., 2017b; Patrick et al., 2012; Ji et al., 2009; Just et al., 2017; Marcozzi et al., 2018; Petrović and Ivanković, 2018; McMinn et al., 2018), PG-2 (Offutt and Pan, 1997; Papadakis and Le Traon, 2013; Schuler et al., 2009; Schuler and Zeller, 2013; Offutt et al., 2006; Papadakis et al., 2014; Harman et al., 2000; Kintis et al., 2015; Patel and Hierons, 2016; Holling et al., 2016; Marcozzi et al., 2018; McMinn et al., 2018), PG-3 (Zhang et al., 2012; Papadakis and Malevris, 2011b; Chen and Zhang, 2018), PG-4 (Papadakis and Malevris, 2012; Kaminski and Ammann, 2011; Just et al., 2012b; Fraser and Arcuri, 2015), PG-5 (Just et al., 2012b; Delgado-Pérez et al., 2017a), PG-6 (Offutt et al., 1999; Papadakis and Malevris, 2010b; Harman et al., 2000; Liu et al., 2006; Papadakis and Malevris, 2009; 2011b; DeMillo and Offutt, 1993; Zhang et al., 2010b; Bashir and Nadeem, 2017; DeMillo and Offutt, 1991)
T-12 – Selective mutation	PG-1 (Zhang et al., 2010a; Offutt et al., 1993; Tuya et al., 2007; Mresa and Bottaci, 1999; Offutt et al., 1996; Gligoric et al., 2013; Bluemke and Kulesza, 2014a; Zhang et al., 2013a; Sridharan and Siami-Namin, 2010; Kurtz et al., 2016; Delgado-Pérez et al., 2017c; Gopinath et al., 2017; Praphamontipong and Offutt, 2017; Derezińska and Rudnik, 2012; Bluemke and Kulesza, 2014b; Delgado-Pérez et al., 2017; Sun et al., 2017a), PG-2 (Delgado-Pérez et al., 2017), PG-4 (Tuya et al., 2007)
T-13 – Serial execution	PG-3 (Weiss and Fleyshgaker, 1993; Kim et al., 2012; Fleyshgaker and Weiss, 1994), PG-4 (Ma and Kim, 2016)
T-14 – Sufficient operators	PG-1 (Zhang et al., 2010a; Siami-Namin et al., 2008; Vincenzi et al., 2001; Barbosa et al., 2001; Delamaro et al., 2014b; Siami-Namin and Andrews, 2006; Lacerda and Ferrari, 2014), PG-5 (Just and Schweiggert, 2015)
T-15 – Optimization of generation, execution and analysis of mutants	PG-1 (Wedyan and Ghosh, 2012; Inozemtseva et al., 2013; Reuling et al., 2015; Cachia et al., 2013; Kaminski and Ammann, 2009; Zhu et al., 2017; Derezińska and Hałas, 2015; Zhu et al., 2018), PG-2 (Kintis and Malevris, 2013; Ferrari et al., 2013; Anbalagan and Xie, 2008; Vincenzi et al., 2002), PG-3 (Durelli et al., 2012; Gligoric et al., 2012; Zhang et al., 2012; Just et al., 2014a; 2011; Wright et al., 2013; Zhang et al., 2016; Gopinath et al., 2016b; Devroey et al., 2016; McMinn et al., 2016; Alexander et al., 2002; Bogacki and Walter, 2006a; Chen and Zhang, 2018), PG-4 (Zhang et al., 2013b; Derezińska, 2013; Devroey et al., 2016; Zhu et al., 2017; Derezińska and Hałas, 2015; Zhu et al., 2018), PG-5 (Steimann and Thies, 2010; Kaminski et al., 2013; Hu et al., 2011; Belli and Beyazit, 2015; Just and Schweiggert, 2015; Fernandes et al., 2017; Iida and Takada, 2017),
T-16 – Evolutionary algorithms	PG-1 (Domínguez-Jiménez et al., 2011; Oliveira et al., 2013; Sridharan and Siami-Namin, 2010; Adamopoulos et al., 2004; Ji et al., 2009; Harman et al., 2014; Delgado-Pérez et al., 2017b; Quyen et al., 2016; Nobre et al., 2012; Delgado-Pérez et al., 2017; Abuljadayel and Wedyan, 2018), PG-2 (Delgado-Pérez et al., 2017), PG-3 (Papadakis and Malevris, 2011b), PG-4 (Ayari et al., 2007; Oliveira et al., 2013; Adamopoulos et al., 2004; Fraser and Arcuri, 2015; Harman et al., 2014), PG-5 (Domínguez-Jiménez et al., 2009a), PG-6 (Papadakis and Malevris, 2011a; Baudry et al., 2005; Papadakis and Malevris, 2011b; Bashir and Nadeem, 2017; Matnei Filho and Vergilio, 2015; Henard et al., 2014; Fraser and Zeller, 2012)
T-17 – One-op mutation	PG-1 (Untch, 2009; Delamaro et al., 2014c; 2014a; Deng et al., 2013; Kurtz et al., 2016; Derezińska, 2016)
T-18 – Execution trace analysis	PG-3 (Reales and Polo, 2014), PG-6 (Fraser and Zeller, 2012)
T-19 – Model-based mutation	PG-2 (Devroey et al., 2017), PG-3 (Devroey et al., 2016; 2017), PG-4 (Devroey et al., 2016), PG-6 (Aichernig et al., 2013; 2014)
T-20 – State-based analysis	PG-3 (Gligoric et al., 2012; Just et al., 2014a; Gligoric et al., 2010; Wang et al., 2017), PG-6 (Bashir and Nadeem, 2017)
T-21 – Minimal mutation	PG-1 (Ammann et al., 2014; Kurtz et al., 2016; Just et al., 2017)

evaluate operator-based mutant selection. Praphamontipong and Offutt (2017) applied a set of new mutation operators for Web applications and analyzed redundancy among operators. Delgado-Pérez et al. (2017) explored the Trivial Compiler Equivalence technique (Kintis et al., 2017; Papadakis et al., 2015) in combination with *selective mutation* and *evolutionary algorithms*, and Delgado-Pérez et al. (2017c) applied *selective mutation* to 83 classes written in C++. Sun et al. (2017a) investigated mutation testing for WS-

BPEL programs and used *selective mutation* to assess applicability and efficacy of the operators.

Wong et al. (1994) used *constrained mutation* and explored varied strategies to compose subsets of operators, and Wong and Mathur (1995) compared *constrained mutation* with *random mutation*. Recently, Petrović and Ivanković (2018) described an approach for selecting *productive mutants*. These are, to some extent, similar to *constrained mutation* as originally proposed. It relies on the

developers' understanding of the code for selecting either killable or equivalent mutants.

Papadakis and Malevris (2010a) empirically evaluated *random mutation* and *higher order mutation*. Bluemke and Kulesza (2013) explored random mutation with eight Java classes and used various tools (MuClipse, CodePro, and some tailor-made tools). Parsai et al. (2016) proposed to apply weights to randomly selected mutants and to focus on only acceptable mutants. Derezińska and Rudnik (2017) explored random mutation together with equivalence partitioning for object-oriented (C#) programs.

Barbosa et al. (2001) defined a procedure to identify *sufficient operators*. Vincenzi et al. (2001) applied the same procedure (Barbosa et al., 2001) to empirically evaluate unit-level and integration-level mutation operators for C programs. Siami-Namin and Andrews (2006) and Siami-Namin et al. (2008) explored *sufficient operators* for C programs supported by statistical methods (linear regression analysis). Delamaro et al. (2014b) presented a new procedure for defining *sufficient operators*. The process iteratively includes mutation operators until 100% mutation score is achieved. Lacerda and Ferrari (2014) applied Barbosa et al. (2001)'s method in *AspectJ* programs.

Adamopoulos et al. (2004) applied an *evolutionary algorithm* that relies on a fitness function to avoid equivalent mutants during the co-evolution process. Only mutants that are hard to kill and test cases that are good at detecting mutants are selected. Ji et al. (2009) applied an *evolutionary algorithm* to support a domain reduction technique to identify mutant clusters, and then executed one mutant per cluster. Domínguez-Jiménez et al. (2011) proposed Evolutionary Mutation Testing (EMT) to reduce the number of mutants. They used the GAmEra tool to apply the approach to WS-BPEL projects. Delgado-Pérez et al. (2017b) also explored the EMT approach for C++ programs. Nobre et al. (2012) explored three multi-objective algorithms (namely, MTabu, NSGA-II, and PACO). Oliveira et al. (2013) proposed a new genetic co-evolutionary algorithm and compared it with five other evolutionary approaches. Harman et al. (2014) investigated strongly subsuming *higher order mutants* (SSHOMs) and *evolutionary algorithms* to reduce both the number of mutants and the number of test cases. Quyen et al. (2016) explored mutation testing of Simulink models with *evolutionary algorithms*. Abuljadayel and Wedyan (2018) applied genetic algorithms to kill *higher order mutants* for Java programs.

Polo et al. (2009) explored *higher order mutation* for Java programs. Kintis et al. (2010) experimented with *higher order mutation* and *weak mutation*, and compared results against strong mutation for Java programs. Kaminski et al. (2011b) suggested that generating only higher order logical mutants was sufficient. Omar and Ghosh (2012) applied *higher order mutation* to *AspectJ* programs. Reales et al. (2013) explored *higher order mutation* for Java programs by using the Bacterio tool and different algorithms. Madeyski et al. (2014) implemented four strategies for *higher order mutation* of Java programs with support of the JudyDiffOP tool. Reuling et al. (2015) proposed an approach to perform *higher order mutation* testing on feature models of software product lines (SPLs), trying to avoid redundant and equivalent mutants. Lima et al. (2016) experimented with four strategies for *higher order mutation* and compared results with first order mutation. Derezińska (2016) analyzed first order and *higher order mutation* based on statement deletion operators for C# programs, and presented a high-level analysis focused on the benefits of the deletion operators.

Sun et al. (2017b) applied *control-flow analysis* and proposed four strategies that focus on the most diverse mutants. Just et al. (2017) proposed an approach for handling mutants based on abstract syntax tree analysis and on the *minimal mutation* ap-

proach, originally proposed by Ammann et al. (2014). *Minimal mutation* was also explored by Kurtz et al. (2016), together with *selective mutation*, *one-op mutation*, and *random mutation*.

Marcozzi et al. (2018) applied *control-flow analysis* to reduce the number of equivalent mutants based on proofs of logical assertions. McMinn et al. (2018) analyzed paths in SQL expressions to remove ineffective mutants, including non-compilable, impaired, equivalent, and redundant mutants.

Kaminski and Ammann (2009) reduced the mutant set related to logical expressions by *optimizing mutant generation* based on minimal DNF predicates. Wedyan and Ghosh (2012) proposed an preliminary analysis of source code to prevent the generation of equivalent mutants for *AspectJ* programs. Zhu et al. (2017) explored *weak mutation* and Formal Concept Analysis (FCA) to group similar mutants and test cases. Inozemtseva et al. (2013) prioritized mutating parts of the programs that are most fault-prone based on fault history information; they used mutants generated by the PIT tool. Cachia et al. (2013) proposed incremental mutation testing, which tests newly modified code only. Derezińska and Hałas (2015) tried to *optimize* the generation and execution of mutants by applying syntax tree analysis and exploring runtime resources of the Python interpreter. Zhu et al. (2018) evolved their previous work (Zhu et al., 2017) and explored six compression techniques based on *weak mutation* to optimize strong mutation.

Untch (2009) proposed the *one-op mutation* technique to compose a reduced set of mutants. Deng et al. (2013) applied *one-op mutation* to Java classes. Delamaro et al. (2014c) designed new deletion-related mutation operators for C programs, and implemented them in the Proteum tool. In another study, Delamaro et al. (2014a) explored other mutation operators for *one-op mutation*.

Patrick et al. (2012) used *control-flow analysis* and *data-flow analysis* to identify mutants that had little impact on the program output, and hence are harder to kill. Al-Hajjaji et al. (2017) performed static *data-flow analysis* to select reduced sets of mutants for configurable systems.

5.5.2. Studies that tried to automatically detect equivalent mutants (PG-2)

Devroey et al. (2017) detected equivalent mutants in finite automata, i.e. behavioral models, by applying *weak mutation* and language equivalence concepts.

Offutt and Craft (1994) performed *data-flow analysis* in mutants of Fortran programs and used six *compiler optimization* techniques to identify equivalent mutants. Kintis et al. (2017) and Papadakis et al. (2015) also explored *compiler optimization* to detect equivalent mutants. In particular, they proposed the TCE (Trivial Compiler Equivalence) technique to remove equivalent and duplicated mutants. Hierons et al. (1999) used program slicing to create simple mutants for C programs. Harman et al. (2000) explored program dependence (*data-flow* and *control-flow analysis*) to detect equivalent mutants by checking weakly killed mutants, and to generate test data automatically. Offutt et al. (2006) specified heuristics, based on *data-flow analysis*, to avoid equivalent mutants for class-level mutation operators. Schuler et al. (2009) applied the concept of dynamic invariants (derived via *control-flow* and *data-flow analyses*) and showed that invariant-violating mutants are less likely to be equivalent. More recently, Schuler and Zeller (2013) applied the same concepts to classify mutants as killable or equivalent, and to provide hints about the most relevant mutants to be analyzed. Papadakis and Le Traon (2013) and Papadakis et al. (2014) empirically investigated Schuler and Zeller (2013)'s approach as a way to reduce the effects of equivalent mutants in mutation testing. Kintis and Malevris (2015) investigated problematic *data-flow* patterns, through static analysis, to automatically identify equivalent mutants. Kintis et al. (2015)

relied on runtime information to detect equivalent mutants. They explored first and *high order mutation* and applied *control-flow* and *data-flow analyses* to classify killable and equivalent mutants.

Offutt and Pan (1997) devised a constraint-based technique to detect equivalent mutants. They applied *symbolic execution* and heuristics to recognize infeasible constraints among rules to generate test cases—if the constraints cannot be satisfied, the mutant cannot be killed and thus is equivalent. Patel and Hierons (2016) presented Interlocutory Mutation Testing (IMT) as a predicate-based *control-flow analysis* technique to automatically classify equivalent and killable mutants in programs that are non-deterministic and susceptible to coincidental correctness. Holling et al. (2016) used static analysis and symbolic execution to classify mutants as being killable, equivalent, and “don’t know.” More recently, Marcozzi et al. (2018) implemented a technique to identify equivalent mutants by proving the validity of logical assertions.

Vincenzi et al. (2002) optimized the prediction of equivalent mutants by applying Bayesian-learning. The outcomes depend on the size of randomly-generated test sets. Anbalagan and Xie (2008) and Ferrari et al. (2013) implemented an approach to automatically detect equivalent mutants of pointcut expressions in AspectJ programs. Kintis and Malevris (2013) introduced the concept of *mirrored mutants*, which are mutants that exhibit analogous behavior, such that identifying one mirrored mutant as being equivalent could help recognize other equivalent mutants.

5.5.3. Studies that tried to execute faster (PG-3)

Devroey et al. (2016) explored *higher order mutation*, *model-based mutation*, and *optimization* strategies to reduce the number of mutants of formal transition systems.

Offutt and Lee (1994) explored *weak mutation* to speed up mutant execution and to automatically detect equivalent mutants. Papadakis and Malevris (2011b) applied *control-flow analysis* and *evolutionary algorithms* to support test case generation, and performed weak mutation to check the coverage of generated *metamutants*. Kim et al. (2012) proposed optimizing test executions by avoiding redundant executions that were identified using *weak mutation*. Devroey et al. (2017) detected equivalent mutants of finite automata (behavioral) models. They applied *weak mutation* together with language equivalence concepts, to detect equivalent mutants and speed up mutation analysis.

Krauser et al. (1991) unified mutants for *parallel execution* on Single Instruction Multiple Data (SIMD) machines. Offutt et al. (1992) tried similar ideas with Multiple Instruction Multiple Data (MIMD) machines. Similarly to Offutt et al. (1992), Choi and Mathur (1993) developed PMothra, which enabled *parallel execution* of mutants using a hypercube computer. Jackson and Woodward (2000) introduced the *parallel firm mutation* technique for Java programs, using Java threads to increase execution speed. Reales and Polo (2013) reported results from five algorithms for *parallel execution of mutants* and showed that the mutant execution cost is reduced proportionally to the number of nodes being used. Li et al. (2015) explored mutation testing for Ruby programs with *parallel execution* in a cloud infrastructure.

DeMillo et al. (1991) investigated a *compiler optimization* technique that created mutants by applying small patches at compile time. Just et al. (2011) explored conditional mutation to reduce the time to generate and execute the mutants. Denisov and Pankevich (2018) introduced a new tool for mutation testing based on the Low-Level Virtual Machine (LLVM), which compiles only the code fragments that are mutated.

Untch (1992) and Untch et al. (1993) introduced *metamutants* to compile and execute mutants faster. Untch et al. (1997) later implemented a *metamutant-based tool* (TUMs) using Mutant Schema Generation (MSG). Weiss and Fleyshgaker (1993) proposed a

new algorithm for *serial mutation of metamutants* for fast mutant execution. They later invented a new algorithm for *serial execution* and analysis of mutants called Lazy Mutation Analysis (LMA). Ma et al. (2005) applied the *metamutant* concept in the MuJava tool to generate mutants for compiled Java code. Wright et al. (2013) explored four different approaches that combine *metamutants* and *optimization* techniques to speed up mutation testing of database schemas. Reales and Polo (2014) proposed the MUSIC technique, which implements *metamutants* and *execution trace analysis* to speed up mutation. Gopinath et al. (2016b) explored the concepts of *metamutants* and *execution optimization* techniques to speed up mutant execution by avoiding multiple mutant compilation and redundant, partial test case executions.

Zhang et al. (2012) used *control-flow analysis* and *optimization* techniques to speed up mutation testing during regression testing. Chen and Zhang (2018) also applied regression testing ideas, reporting fewer test cases required, fewer mutants executed, and fewer test executions.

Alexander et al. (2002) used *optimization* techniques to mutate Java objects. The study describes a multi-threaded tool that sped up mutation. Bogacki and Walter (2006a) used aspect-oriented programming concepts to perform mutation while executing the original program to try to reduce mutant compilation and execution time. Durelli et al. (2012) modified the Java Virtual Machine to embed native support to speed up execution. Gligoric et al. (2010) presented a method to efficiently explore the states of multithreaded programs. The approach was implemented by Gligoric et al. (2012). Just et al. (2014a) implemented *optimizations* in the Major mutation tool, focusing on infected states and propagation to output. Wang et al. (2017) extended Just et al. (2014a)’s approach by using meta-functions to fork new processes to reduce redundant executions. Zhang et al. (2016) predicted whether mutants would be killed before execution by using a model that relies on features of mutants, tests, and coverage measures. McMinin et al. (2016) virtualized mutation testing of databases, thus reducing communication with the database.

5.5.4. Studies that tried to reduce the number of test cases or the number of executions (PG-4)

Kaminski and Ammann (2011) introduced the minimal-MUMCUT logic-based testing criterion based on *higher order mutation*, *control-flow analysis*, and fault hierarchies. Harman et al. (2014) investigated strongly subsuming *higher order mutants* (SSHOMs) and *evolutionary algorithms* to reduce both the number of mutants and the number of test cases. Devroey et al. (2016) explored *higher order mutation*, *model-based mutation*, and *optimization* strategies to reduce the number of mutants in formal transition systems. Gopinath et al. (2018) reduced the number of test case executions by creating “supermutants,” that combine several first order mutants (i.e., *metamutants*).

Fraser and Arcuri (2015) investigated *weak mutation*, *evolutionary algorithms* and *control-flow analysis* for automatic generation of reduced test sets. Ma and Kim (2016) devised a mutant clustering approach to execute only one mutant from a group of mutants that are weakly killed by a given test case. The approach generates *metamutants* and *serial mutants* to speed up execution. Zhu et al. (2017) explored *weak mutation* (more precisely, state infection analysis) and Formal Concept Analysis (FCA) to group similar mutants and test cases. Zhu et al. (2018) evolved their previous work (Zhu et al., 2017) and explored six compression techniques to improve the efficiency of strong mutation based on *weak mutation*.

Tuya et al. (2007) proposed mutation operators for SQL. They used *selective mutation* and *minimization and prioritization of test sets* to reduce the number of mutants and required test cases. Sahinoğlu and Spafford (1990) addressed *minimization and prioritization of test sets* by applying a sequential statistical procedure

based on prespecified thresholds. Derezińska (2013) defined a mutant clustering approach for C# programs to reduce the numbers of mutants and tests.

Papadakis and Malevris (2012) used control-flow analysis to select test cases. Just et al. (2012b) also used *control-flow analysis* to prioritize test cases.

Zhang et al. (2013b) optimized mutant execution by predicting, without running, which mutants can be killed based on historical test coverage.

Adamopoulos et al. (2004) applied an *evolutionary algorithm* that used a fitness function to avoid equivalent mutants during co-evolution. Only mutants that were hard to kill and test cases that were good at detecting mutants were selected. Ayari et al. (2007) proposed an evolutionary approach based on ant colony optimization to generate tests automatically. Oliveira et al. (2013) explored genetic co-evolutionary algorithms to generate reduced sets of test cases that achieve higher mutation scores.

5.5.5. Studies that tried to avoid creation of certain mutants (PG-5)

Marshall et al. (1990)'s approach predicts the impact of mutations of program variables from the strong mutation and *weak mutation* perspectives, thus avoiding the creation of many mutants. Just et al. (2012b) used *data-flow analysis* to avoid creating redundant relational and conditional operator mutants. They also prioritized test cases based on *control-flow analysis*. Kintis and Malevris (2014) prevented the generation of equivalent mutants by defining data flow patterns that reveal code locations that should not be mutated. Delgado-Pérez et al. (2017a) defined class-level mutation operators for C++ programs and a set of restrictions (based on *data-flow analysis* and *control-flow analysis*) to avoid the creation of unproductive mutants (equivalent, duplicate, invalid, and trivial).

Just and Schweiggert (2015) defined rules to avoid the creation of unnecessary operator mutants (conditional, unary, and relational) mutants by using *sufficient operators*, *optimization*, and prior results (Just et al., 2012b).

Steimann and Thies (2010) explored behavior-preserving constraints (based on refactoring rules) to optimize the generation of killable mutants. Hu et al. (2011) evaluated static and dynamic nature of class-level mutation operators. They proposed new rules to avoid the creation of equivalent mutants. Kaminski et al. (2013) theoretically analyzed and updated fault hierarchies for relational mutation operators (ROR), which eliminate redundancy among mutants. Their analysis showed that only three out of seven possible ROR mutants need to be generated. Inspired by Kaminski et al. (2013)'s, Iida and Takada (2017) introduced the notion of mutant killable preconditions to identify redundant mutants in control-flow statements. Belli and Beyazit (2015) proposed an event-based approach named *k-Reg-based* that avoids creating equivalent and redundant mutants for models written in a regular grammar. Fernandes et al. (2017) proposed a strategy to create rules to avoid useless (equivalent and duplicated) mutants. The rules discard those mutants right before they are generated.

Domínguez-Jiménez et al. (2009a) explored the use of genetic algorithms to avoid creating mutants for WS-BPEL compositions.

5.5.6. Studies that tried to automatically generate test cases (PG-6)

Zhang et al. (2010b) used *control-flow analysis* to introduce mutant-killing constraints into the program under test, and explored *weak mutation* concepts to guide the generation of test inputs. Papadakis and Malevris (2011b) used *control-flow analysis* and *evolutionary algorithms* to generate test cases, and performed *weak mutation* to evaluate coverage on the generated *metamutants*.

Harman et al. (2000) explored program dependence (*data-flow analysis* and *control-flow analysis*) to detect equivalent mutants by checking weakly killed mutants, as well as to automatically generate test cases.

Papadakis and Malevris (2010b) performed *control-flow analysis* of *metamutants* to generate better test cases.

DeMillo and Offutt (1991, 1993) used symbolic execution (*control-flow analysis*) to develop Constraint-Based Testing (CBT) to automatically generate test cases. CBT generated tests that satisfy the conditions needed to reach mutants and infect the program state after the mutant, and discarded tests that did not add to the mutation score. Later, Offutt et al. (1999) evolved the CBT technique by replacing symbolic execution algorithms with dynamic symbolic execution in a technique they called dynamic domain reduction (DDR). Liu et al. (2006) explored path-wise test data generation, claiming it was more efficient than symbolic evaluation. Papadakis and Malevris (2009) applied path selection based on *control-flow analysis* to generate tests.

Bashir and Nadeem (2017) proposed a combined fitness function for *evolutionary algorithms* that uses both *control-flow* and *state-based* properties of the mutants to generate test data automatically. Baudry et al. (2005) explored the concept of ant colonies in *evolutionary algorithms* to generate test cases. Papadakis and Malevris (2011a) used dynamic information from mutant executions to generate test data automatically. Fraser and Zeller (2012) presented *uTest*, which generates unit tests for Java classes based on genetic algorithms and *execution trace analysis*. Henard et al. (2014) explored search-based optimization methods to minimize the number of selected configurations in Software Product Line testing and to maximize the number of mutants killed. Matnei Filho and Vergilio (2015) also tested Software Product Lines using a multi-objective evolutionary approach.

Aichernig et al. (2013) combined *model-based* testing and mutation testing to automatically generate test cases. Their ideas can be used for several languages (e.g. Prolog, Java and C). They also developed a tool named *MoMut* (Aichernig et al., 2014) to generate test cases from UML models.

Answer for RQ1: Regarding RQ1 (*Which techniques support cost reduction of mutation testing?*), we noticed that mutation-related costs can be reduced by applying a wide range of techniques, sometimes individually and sometimes in combination. Some of the most common techniques are traditional software analysis methods such as *control-flow analysis* and *data-flow analysis*. Others, such as *selective mutation* and *higher-order mutation*, are common only within the mutation testing field, whereas still others are widely used in CS and Math (e.g. *compiler optimization*, *evolutionary algorithms*, and *optimization-related* techniques). These observations lead us to conclude that mutation-related cost reduction research is more interdisciplinary than in the past. That implies that collaboration with researchers from other areas can be very productive and should be encouraged.

6. Cost reduction metrics and results

Another contribution of this paper is the analysis of the benefits (in terms of cost savings) versus the loss of effectiveness (in terms of the mutation score or test quality). The analysis concerns research questions RQ2 and RQ3 (defined in Section 3.1) and starts with the characterization of metrics that have been used in the selected studies (Section 6.1). Then, Section 6.2 summarizes the main findings regarding results collected with those metrics.

6.1. Metrics and their timeline

Table 6 lists metrics and studies that applied those metrics. The third column of the table, "*Intent*," shows the intended goal

Table 6

Metrics to measure the cost reduction of mutation testing.

ID	Metric name	Intent	Studies that collected the metric
M-1	<i>Number of test cases required to achieve mutant coverage</i>	Estimate the number of test cases that need to be created (automatically or manually) or executed.	Wong and Mathur (1995), Liu et al. (2006), Tuya et al. (2007), Papadakis and Malevris (2010a), Kaminski and Ammann (2011), Papadakis and Malevris (2012), Derezińska (2013), Harman et al. (2014), Delamaro et al. (2014c,b), Bluemke and Kulesza (2014a), Lima et al. (2016), Reales et al. (2013), Delgado-Pérez et al. (2017c), Chen and Zhang (2018), Delamaro et al. (2014a), Zhu et al. (2017), Wong et al. (1995), Ji et al. (2009), Cachia et al. (2013), Derezińska and Rudnik (2017), Derezińska and Rudnik (2012), Sahinoğlu and Spafford (1990), Oliveira et al. (2013), Wong et al. (1994)
M-2	<i>Number of mutants to be executed</i>	Estimate the number of mutants that need to be executed.	Offutt et al. (1993), Wong and Mathur (1995), Offutt et al. (1996), Barbosa et al. (2001), Vincenzi et al. (2001), Tuya et al. (2007), Siami-Namin et al. (2008), Kaminski and Ammann (2009), Polo et al. (2009), Untch (2009), Sridharan and Siami-Namin (2010), Papadakis and Malevris (2010a), Steimann and Thies (2010), Kaminski et al. (2011b), Domínguez-Jiménez et al. (2011), Omar and Ghosh (2012), Just et al. (2012b), Nobre et al. (2012), Bluemke and Kulesza (2014a), Derezińska (2013), Deng et al. (2013), Reales et al. (2013), Zhang et al. (2013a), Gligoric et al. (2013), Just and Schweiggert (2015), Madeyski et al. (2014), Harman et al. (2014), Lacerda and Ferrari (2014), Ammann et al. (2014), Delamaro et al. (2014a, 2014c, 2014b), Belli and Beyazit (2015), Lima et al. (2016), Delgado-Pérez et al. (2017a), Delgado-Pérez et al. (2017c), Wang et al. (2017), Zhu et al. (2017), Gopinath et al. (2017), Parsai et al. (2016), Praphamontripong and Offutt (2017), Sun et al. (2017a), Derezińska and Hałas (2015), Bluemke and Kulesza (2014b), Chen and Zhang (2018), Zhu et al. (2018), Bluemke and Kulesza (2013), Marshall et al. (1990), Wong et al. (1995), Siami-Namin and Andrews (2006), Ji et al. (2009), Zhang et al. (2010a), Cachia et al. (2013), Inozemtseva et al. (2013), Reuling et al. (2015), Al-Hajjaji et al. (2017), Just et al. (2017), Derezińska and Rudnik (2017), Derezińska (2016), Derezińska and Rudnik (2012), Oliveira et al. (2013), Henard et al. (2014), Marozzi et al. (2018), Kaminski and Ammann (2009), Quyen et al. (2016), Kaminski et al. (2013), Iida and Takada (2017)
M-3	<i>Mutant execution speedup</i>	Estimate how much time is required for mutant execution.	Offutt et al. (1992), Choi and Mathur (1993), Offutt and Lee (1994), Untch et al. (1997), Ma et al. (2005), Gligoric et al. (2010), Papadakis and Malevris (2011b), Just et al. (2012b), Gligoric et al. (2012), Durelli et al. (2012), Reales and Polo (2014), Just and Schweiggert (2015), Just et al. (2014a), Delgado-Pérez et al. (2017a), Wang et al. (2017), Zhu et al. (2017), Zhang et al. (2016), Devroey et al. (2016), Li et al. (2015), Derezińska and Hałas (2015), Krauser et al. (1991), McMinin et al. (2016), Kim et al. (2012), Zhang et al. (2013a), Madeyski et al. (2014), Chen and Zhang (2018), Zhu et al. (2018), Untch et al. (1993), Reales and Polo (2013), Wright et al. (2013), Ma and Kim (2016), Gopinath et al. (2016b), Derezińska and Rudnik (2017), Weiss and Fleishgaker (1993), Fleishgaker and Weiss (1994), Bogacki and Walter (2006a)
M-4	<i>Mutant execution efficiency</i>	Estimate the performance gains or losses when two or more execution configurations (e.g. processor and network infrastructure) are compared.	Offutt et al. (1992)
M-5	<i>Test generation speedup</i>	Estimate how much time is required for test generation.	DeMillo and Offutt (1993), Liu et al. (2006)
M-6	<i>Number of equivalent mutants automatically detected</i>	Estimate the number of mutants that need be handled during mutant analysis.	Offutt and Craft (1994), Offutt and Pan (1997), Anbalagan and Xie (2008), Ferrari et al. (2013), Kintis and Malevris (2013), Wright et al. (2014), Papadakis et al. (2014), Kintis and Malevris (2015), Fernandes et al. (2017), Kintis et al. (2017), Devroey et al. (2017), Offutt et al. (2006), Schuler et al. (2009), Papadakis and Le Traon (2013), Patel and Hierons (2016), Marozzi et al. (2018), McMinin et al. (2018)
M-7	<i>Number of cycles (or generations) in evolutionary algorithms</i>	Estimate the effort required to run a specific evolutionary algorithm.	Bashir and Nadeem (2017), Baudry et al. (2005)
M-8	<i>Mutant compilation speedup</i>	Estimate how much time is required for mutant compilation.	Delgado-Pérez et al. (2017a), Fernandes et al. (2017), Just et al. (2011), Ma et al. (2005)
M-9	<i>Mutant generation speedup</i>	Estimate how much time is required for mutant generation.	Ma et al. (2005), Domínguez-Jiménez et al. (2011), Kim et al. (2012), Just et al. (2011)
M-10	<i>Completeness of generated test suite</i>	Estimate the effort required to evolve the current set suite to achieve the intended mutant coverage.	Papadakis and Malevris (2009), Kaminski and Ammann (2009), Papadakis and Malevris (2012), Matnei Filho and Vergilio (2015), Zhang et al. (2010b)
M-11	<i>Number of killable mutants automatically detected</i>	Estimate the number of mutants that need be handled during mutant analysis.	Devroey et al. (2017), Holling et al. (2016), Schuler et al. (2009), Patel and Hierons (2016)

(continued on next page)

Table 6 (continued)

ID	Metric name	Intent	Studies that collected the metric
M-12	Number of equivalent mutants generated	Estimate the number of mutants that need be handled during mutant analysis.	Papadakis and Malevris (2010a), Kintis et al. (2010), Hu et al. (2011), Kaminski et al. (2011b), Wedyan and Ghosh (2012), Deng et al. (2013), Kintis and Malevris (2014), Delamaro et al. (2014b), Polo et al. (2009), Delamaro et al. (2014a), Delamaro et al. (2014c), Madeyski et al. (2014) Papadakis and Malevris (2010b)
M-13	Number of cycles in test generation algorithms	Estimate the effort required to run a specific test generation algorithm.	Steimann and Thies (2010), Wright et al. (2014), Madeyski et al. (2014), Devroey et al. (2017), Aichernig et al. (2013) Patrick et al. (2012)
M-14	Mutant analysis speedup	Estimate how much time is required for mutant analysis.	
M-15	Probability to kill mutants	To obtain a probability for mutant coverage based on the available test set.	
M-16	Number of test case executions	Estimate the number of test case executions for a given mutant set.	Zhang et al. (2012, 2013b), Bashir and Nadeem (2017), Derezińska and Hałas (2015), Kim et al. (2012), Sahinoğlu and Spafford (1990), Ma and Kim (2016), Reales and Polo (2014), Gopinath et al. (2018), Chen and Zhang (2018)
M-17	Number of duplicated mutants automatically detected	Estimate the number of mutants that need be handled during mutant analysis.	Kintis et al. (2017), Fernandes et al. (2017), McMinn et al. (2018)
M-18	Variation in mutation score	Estimate gains and losses with respect to mutation score.	Sun et al. (2017b)

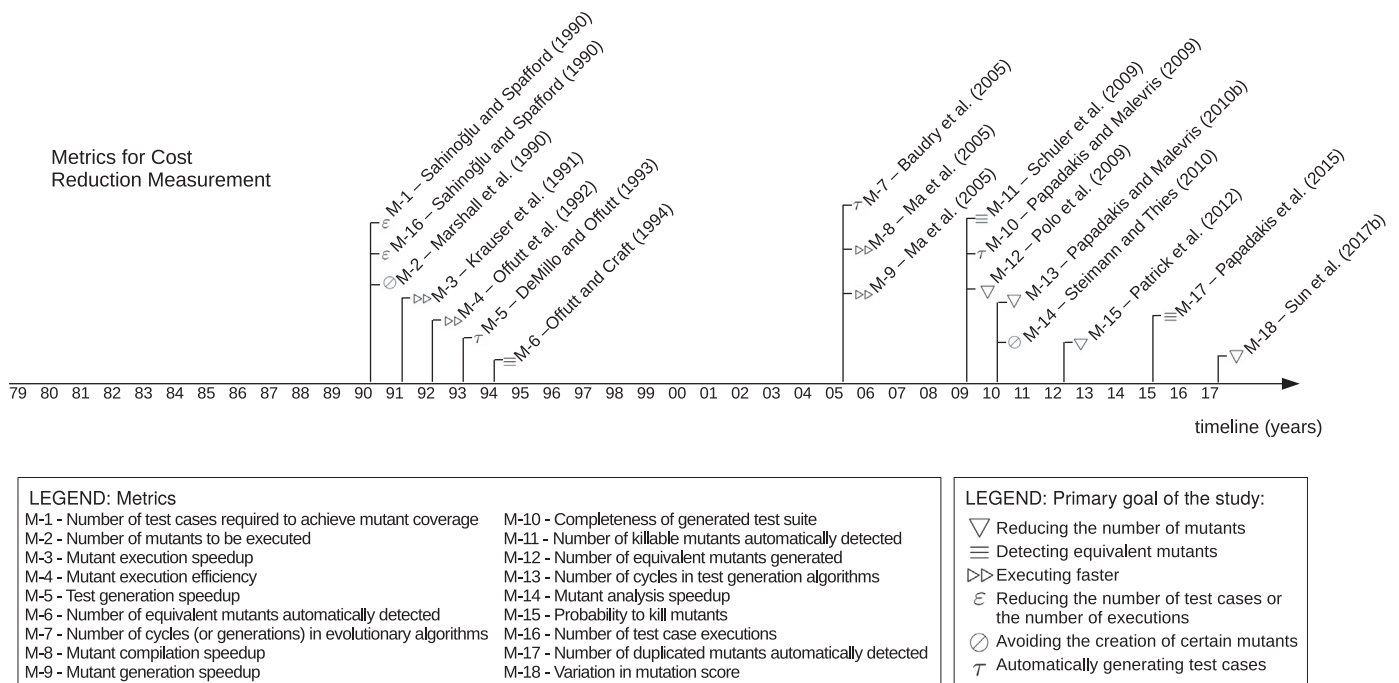


Fig. 11. Timeline for metrics used to measure cost reduction of mutation testing according to their uses in peer-reviewed studies.

of the metrics. Notice that the intents of some metrics overlap. Metrics M-6 (*Number of equivalent mutants automatically detected*), M-11 (*Number of killable mutants automatically detected*), M-12 (*Number of equivalent mutants generated*), and M-17 (*Number of duplicated mutants automatically detected*) have similar intents, despite the fact that they collect different values from the sets of mutants. Also notice that the definition for each metric can be derived from its names and intent. For instance, M-3 (*Mutant execution speedup*) can be defined as the time spent (in a given time unit) for mutant execution when compared to conventional execution (i.e. without applying a cost reduction technique).

Fig. 11 lists all identified metrics in a timeline. The first studies that used the metrics were published in 1990 (Marshall et al., 1990; Sahinoğlu and Spafford, 1990) and 1991 (Krauser et al., 1991). Marshall et al. (1990) estimated how many mutants would be executed (M-2) after using static data-flow analysis to avoid

creating certain mutants. Sahinoğlu and Spafford (1990) applied statistical methods to estimate the number of test cases required to reach a threshold mutation score (M-1), as well as to estimate the number of test case executions (M-16). Both studies used small programs. Krauser et al. (1991) ran simulation experiments to measure mutant execution speedup (M-3) on parallel machines. The timeline also shows that (1) few metrics (M-1 to M-6, and M-16) were used in the early days (1990 to 1994), and (2) many more metrics were used in the subsequent 14 years.

Fig. 12 shows the number of studies that used each metric. *Number of mutants to be executed* (M-2) was used the most (66 studies), followed by *Mutant execution speedup* (M-3) (36 studies) and *Number of test cases required to achieve mutant coverage* (M-1) (25 studies). If we consider only recent research, particularly studies published in the last five years (2014–2018), the most applied

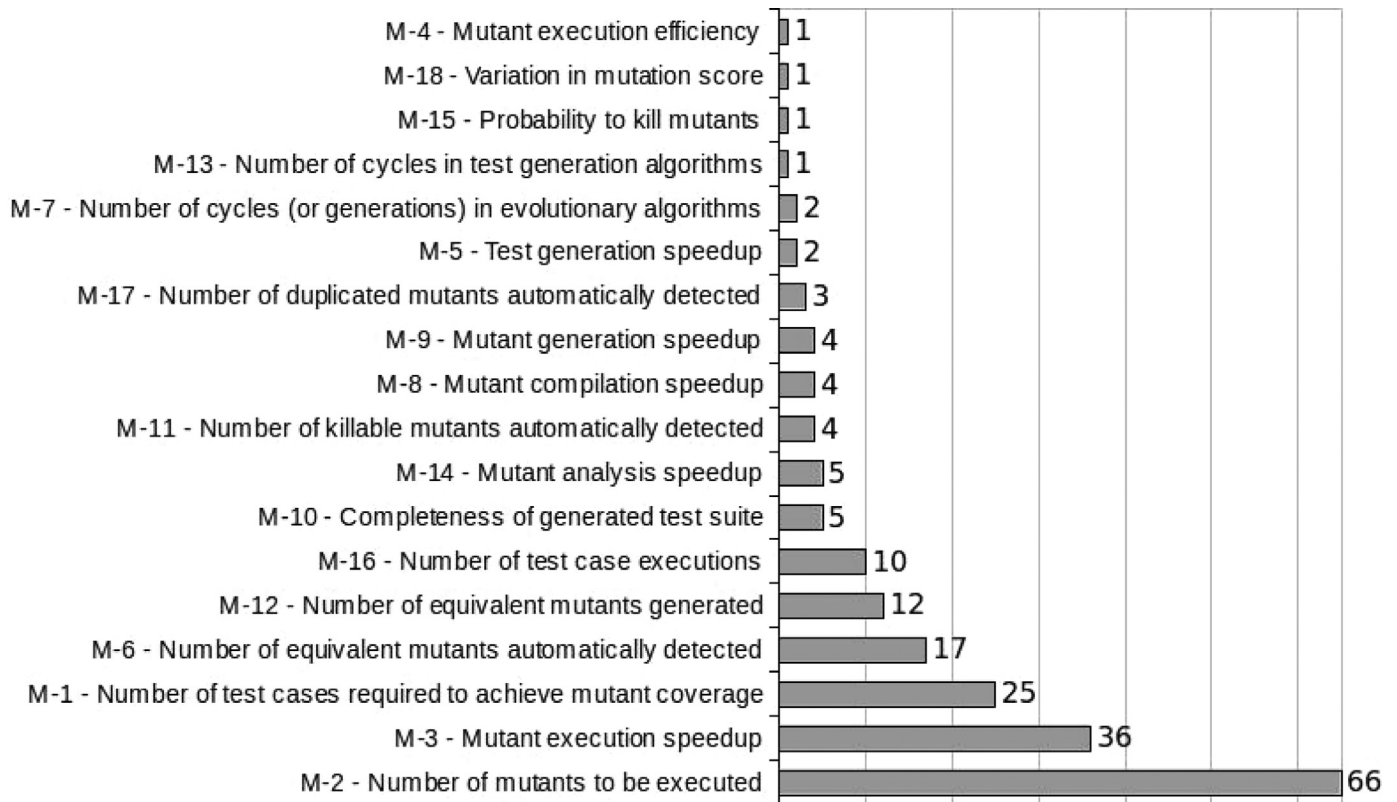


Fig. 12. Number of studies per metric.

metrics are the same as in Fig. 12. The distribution of studies per year is depicted in Fig. 13.

Regarding RQ2 (*Which metrics are used to measure the cost reduction of mutation testing?*), we noticed that a variety of metrics have been used, with emphasis in measuring the *Number of mutants to be executed* (M-2), *Mutant execution speedup* (M-3), and the *Number of test cases required to achieve mutant coverage* (M-1). These are important factors related to the cost of mutation testing. Nonetheless, the results of our SLR for RQ2 also indicates that a major challenge for the practical adoption of mutation testing, *dealing with equivalent mutants*, has not been addressed as much as the others and we recommend more research into the problem. Specifically, only 11% (17 of 151) of the studies used metric M-6 (*Number of equivalent mutants automatically detected*) and 8% (12 of 151) used M-12 (*Number of equivalent mutants generated*).

Note about complex metrics not listed in this section: The list of metrics identified in this SLR is not intended to be complete, because several studies applied customized, and somewhat complex, metrics in the experiments. Examples can be found in the studies by Mresa and Bottaci (1999), Delamaro et al. (2014a), and Delgado-Pérez et al. (2017c), and probably more. Mresa and Bottaci (1999) estimated the cost of test case generation to be a combination of factors such as the number of mutant executions and the number of redundant tests. They also defined metrics such as *relative test generation cost* and *relative equivalence detection*. Delamaro et al. (2014a) devised a weighted cost function that is based on the *relative cost* of a mutation operator instead of being based on its absolute cost. Delgado-Pérez et al. (2017c) measured the *degree of redundancy* of a mutation operator, which is defined as the ratio of the number of redundant mutants generated by an operator to the number of mutants generated by the remaining operators. In our analysis, we focused only on iden-

tifying metrics that are derived from quantifiable elements directly extracted from the mutant creation, execution and analysis steps. Typical examples are the time saved to run a set of mutants (reflected by the *mutant execution speedup* (M-3) metric), and the size of mutation-adequate test sets (reflected by the *number of test cases required to achieve mutant coverage* (M-1) metric). The measurements used most often are presented in the next section.

6.2. Results about cost reduction measurement

Figs. 14–17 show the results achieved by different studies with respect to different cost reduction measurements. These data reflects the first part of RQ3: “*What are the savings (benefit) ... for the techniques?*”. Results for the four most used metrics are displayed (M-2, M-3, M-1, and M-6), grouped by the three most common programming languages (Fortran, Java, and C). Figs. 18 and 19 show the mutation scores achieved in the same studies listed in Figs. 14 and 16 (i.e. for metrics M-2 and M-1). These data address the second part of RQ3: “*What are ... loss of effectiveness (as proxied by mutation score) for the techniques?*”. We do not present mutation score data for metrics M-3 and M-6, since they were not reported for techniques applied to speed up mutant execution and automatically detect equivalent mutants. Furthermore, some values are not available so are missing in Figs. 18 and 19.

In Figs. 14 through 19, the bars show the minimum, mean, and maximum percentages of either cost reduction or mutation score. Note that not all values could be extracted from the studies. From some studies, we extracted only minimum and maximum values (for example, Wong and Mathur (1995) in Fig. 14a). Sometimes we could extract only the mean values (for example, Vincenzi et al. (2001) in Fig. 14b). Furthermore, no values were

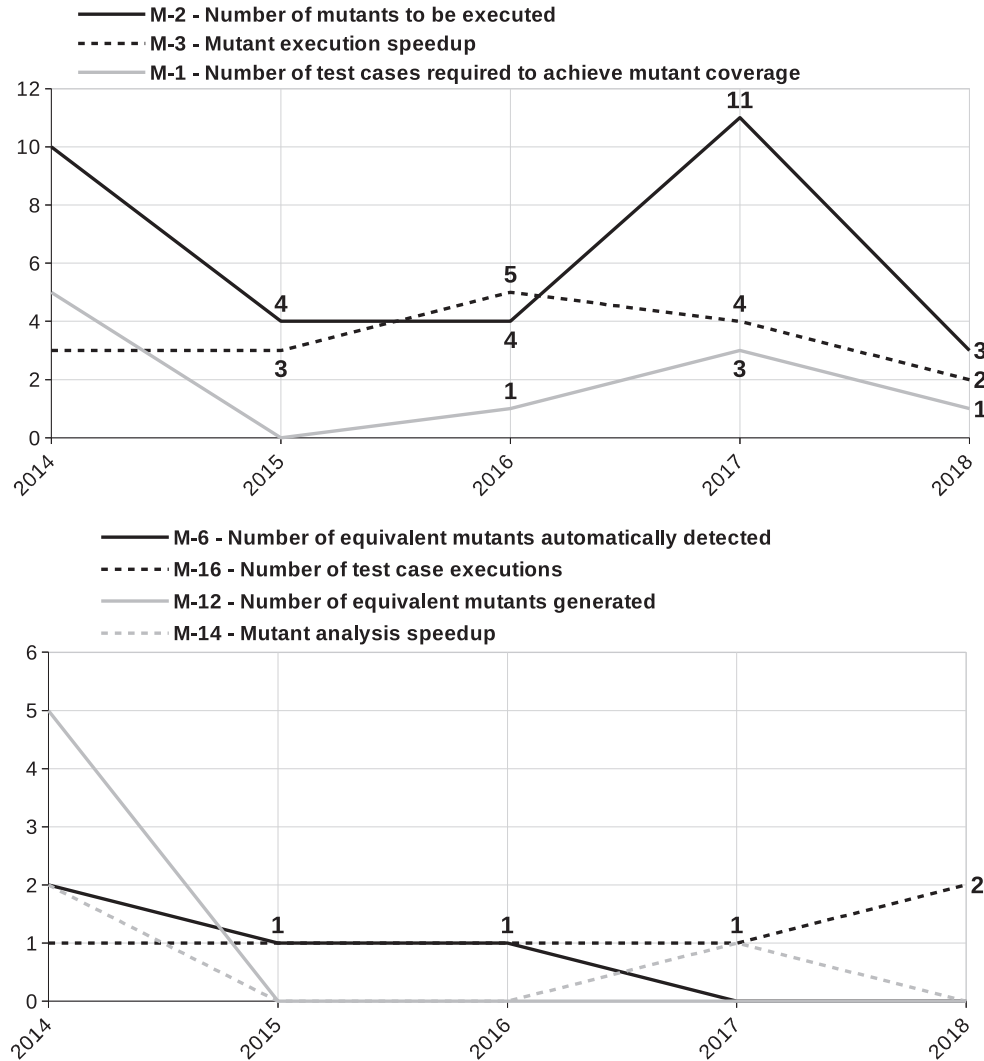


Fig. 13. Distribution of studies that applied the top seven metrics in the last five years.

extracted at all for some studies. This is because of the level of details of reported results, characteristics of the applied cost reduction techniques, and customized experimental settings. This also meant we could not create whisker charts, which would require more detailed data than were presented in several studies. Also note that some studies appear twice in the charts of Figs. 14–19. Authors of such studies applied more than one cost reduction technique, and performed individual measurements regarding cost reduction and mutation score. An example is by Wong and Mathur (1995) (Figs. 14a and 18a), who applied *random mutation* and *constrained mutation*. Additionally, Papadakis and Malevris (2010a) (Figs. 14b and 18b) explored *random mutation* and *higher order mutation*, and Kintis et al. (2010) (Fig. 17c) applied *higher order mutation* and *weak mutation*. Section 6.4 further discusses some characteristics of the measurements we retrieved.

Fig. 14 shows the results of studies that measured cost reduction using metric M-2 (*Number of mutants to be executed*) for Fortran (14a), C (14b), and Java (14c) programs. Fig. 15 shows the results regarding M-3 (*Mutant execution speedup*), while Figs. 16 and 17 show results for metrics M-1 (*Number of test cases required to achieve mutant coverage*) and M-6 (*Number of equivalent mutants*

automatically detected). Notice that M-1 was applied in a single study that targeted Fortran programs, therefore the chart is not shown in Fig. 16.

6.3. Analysis and discussion

Figs. 14 through 17 reveal wide variations in cost reductions from both the inter-study or the intra-study perspectives. Variations among mutation scores (Figs. 18 and 19) were less frequent, but still noticeable. Note that the inter-study perspective can provide an interesting baseline to compare results reported on groups of studies by different scientists who used the same cost reduction techniques. The intra-study perspective, on the other hand, provides a fine-grained notion of how varied results can be for different subjects of a particular study.

Regarding the *Number of mutants to be executed*, average reductions ranged from 26.70% (Just et al., 2012b) (Fig. 14c) to 99.77% (Steimann and Thies, 2010) (Fig. 14c). With respect to *Mutant execution speedup*, the average reductions ranged from 25.00% (Just and Schweiggert, 2015) (Fig. 15c) to 97.78% (Choi and Mathur, 1993) (Fig. 15a). For *Number of test cases required to achieve*

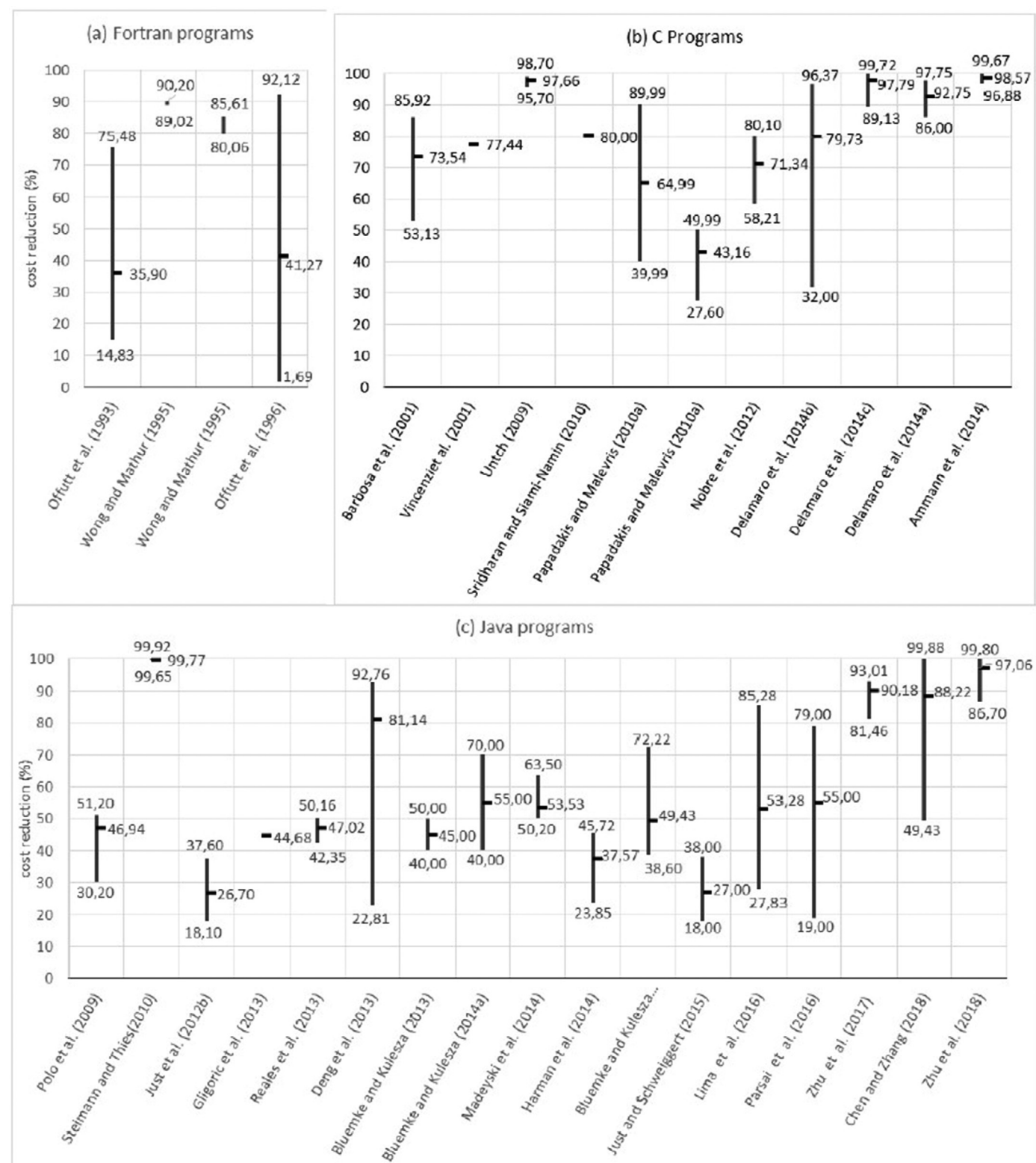


Fig. 14. Cost reduction regarding M-2 – Number of mutants to be executed.

mutant coverageand Number of equivalent mutants automatically detected, the averages ranged from 7.80% (Delamaro et al., 2014b) (Fig. 16a) to 92.60% (Siami-Namin et al., 2008) (Fig. 16a), and from 1.78% (Fernandes et al., 2017) (Fig. 17c) to 90.51% (Deng et al., 2013)(Fig. 17c).

Regarding mutation scores, for both groups of studies (namely, the ones that measured Number of mutants to be executed, and Number of test cases required to achieve mutant coverage), the averages ranged from 0.800 (Reales et al., 2013) (Fig. 18c) to 1.000 (Delamaro et al., 2014b) (Fig. 18b). Before choosing a cost

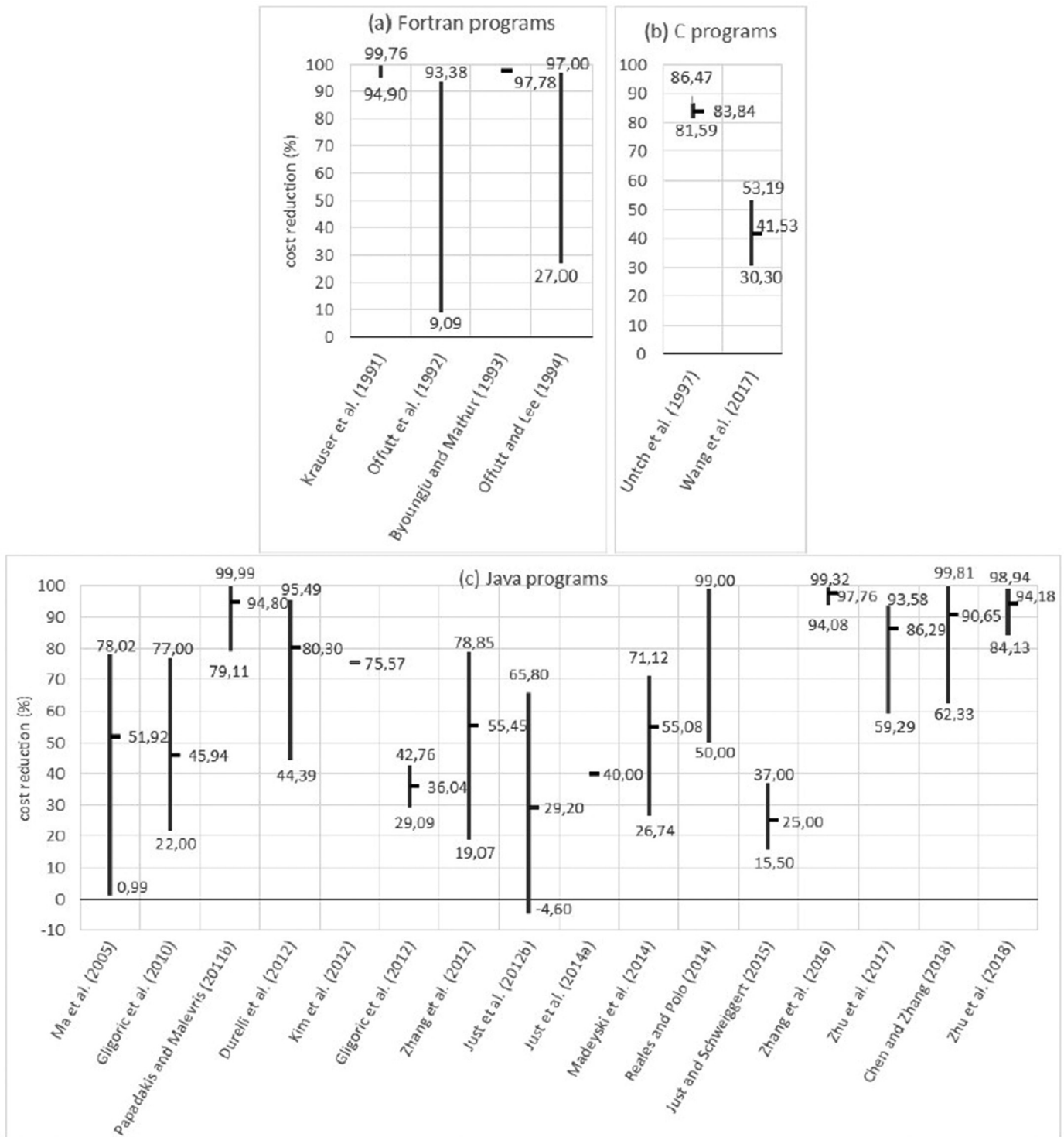


Fig. 15. Cost reduction regarding M-3 - Mutant execution speedup.

reduction approach to adopt, it is important to consider the quality of the resulting test set regarding mutation-effectiveness, and hence its fault-revealing capability. Overall, the results presented in the charts show that while some studies achieved very high, even full, mutation score, some others did not satisfactorily maintain test effectiveness.

Taking an inter-study perspective reveals that the cost reduction variations presented in Figs. 14 through 17 are mainly due

to the range of techniques that have been used, or to the experimental settings of studies that used the same technique and tools to a common set of programs. For example, the cost reductions shown in Fig. 14a were obtained by applying *selective mutation* (Offutt et al., 1993; 1996), *random mutation*, and *constrained mutation* (Wong and Mathur, 1995). These three studies targeted the same language (*Fortran*) with the same tool (*Mothra*). However, Offutt et al. (1993, 1996)'s studies addressed different

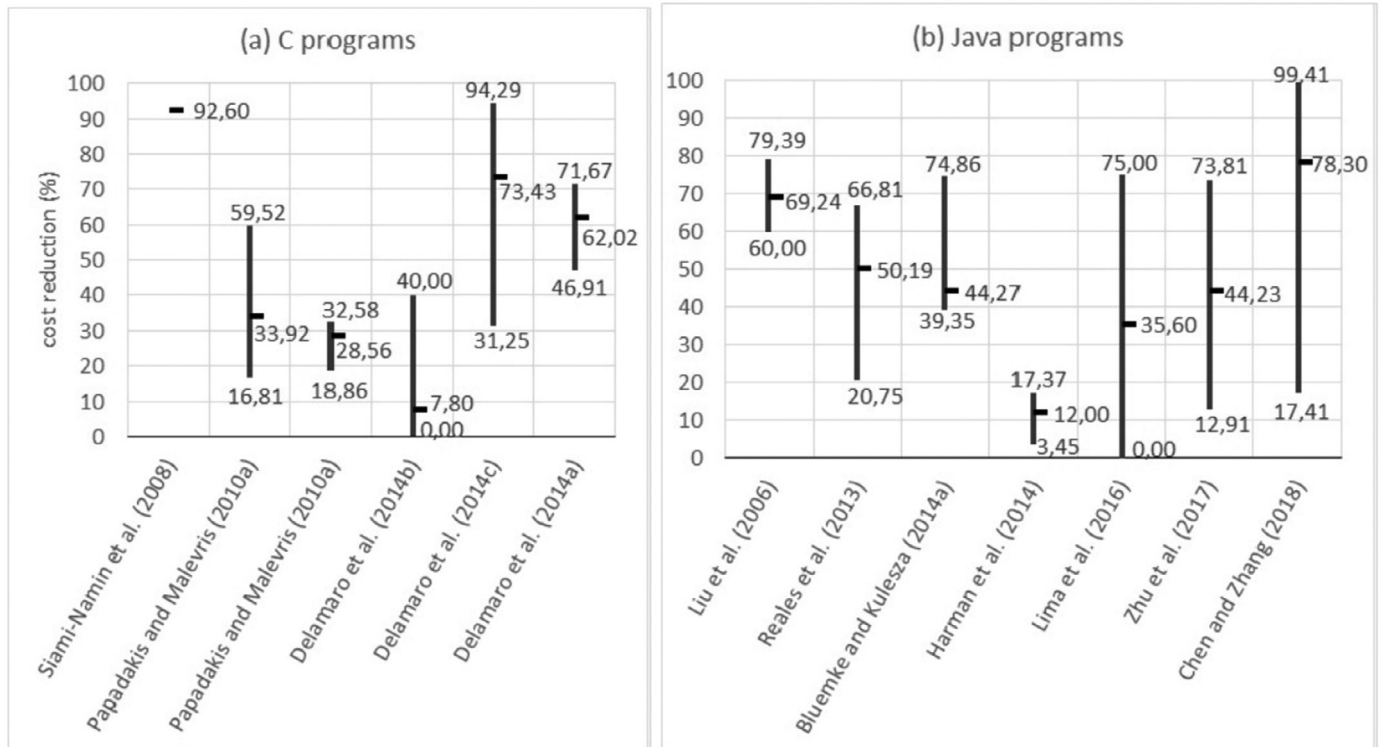


Fig. 16. Cost reduction regarding M-1 – Number of test cases required to achieve mutant coverage.

selective mutation strategies (even though being applied to the same set of programs), while Wong and Mathur (1995) used another set of programs and a different test generation tool.

Regarding RQ3 (What are the savings (benefits) and loss of effectiveness (regarding mutation score) for the techniques?), we found wide variations in results from both an inter-study and an intra-study perspective. Such variations, together with the number of metrics applied (see Section 6.1), make it harder to establish a baseline of studies for comparison in new experiments. Nevertheless, researchers can benefit from our results by using them as a reference to obtain more detailed information from specific studies, and to better design experiments that are comparable and reproducible.

6.4. Further discussion about results for cost reduction measurement

This section presents more details about the measurements the studies used. These details affect how the results should be interpreted. We discuss six types of results: (1) results that represent close to (or even exactly) 100% cost reduction, (2) results that depend on either randomly or systematically defined thresholds, (3) results that are very context-specific, (4) imprecise results, and (5) results obtained from inconsistent subjects.

(i) Results that are close to (or even exactly) 100% cost reduction

Some studies reached 100% cost reduction or very close (Anbalagan and Xie, 2008; Papadakis and Malevris, 2009; Steimann and Thies, 2010; Kintis and Malevris, 2014). Although appealing, some were not generalizable and some did not measure all cost factors. For instance, Anbalagan and Xie (2008) automatically identified all equivalent mutants for a particular category of mutants: mutants of pointcut expressions of aspect-oriented programs written in the AspectJ language. Other costs that were not considered include the cost of running the code weaving compilation step (re-

quired by compilers of aspect-oriented languages), and the cost of running the mutant equivalence checking algorithm.

Another example is Papadakis and Malevris (2009)'s approach, which automatically generates tests to kill all non-equivalent mutants. That is, it produces mutation-adequate test suites. On the surface, this appears to eliminate the human cost of test data generation. However, the study did not include the manual mutant analysis cost, or the time to generate, compile, and execute mutants.

Steimann and Thies (2010) reached 99.77% cost reduction using the Number of mutants to be executed, but only for a single type of mutant (access modifier change). Kintis and Malevris (2014) reported cost reductions using Number of equivalent mutants automatically detected that widely ranged from 0 to 100% for Java programs. Patel and Hierons (2016) investigated mutation testing for non-deterministic programs. The authors performed a preliminary evaluation based on a single program and identified 100% of equivalent mutants. Both studies omitted other costs.

(ii) Results that depend on either randomly or systematically defined thresholds

We did not present the cost reduction for some studies that used evolutionary algorithms (T-16), or for optimization-related techniques (T-15). This is because the results depended largely on the number of iterations of the algorithm, which is usually predefined arbitrarily based on factors such as timeout, achieved mutant coverage, and size of subjects. Some examples are the studies by Domínguez-Jiménez et al. (2009a), Fraser and Arcuri (2015), and Reuling et al. (2015).

Domínguez-Jiménez et al. (2009a) based their experiment on parameters such as population size and new individuals generated between different generations. Fraser and Arcuri (2015) focused on generating tests to achieve mutant coverage by applying a combination of techniques (evolutionary algorithms, weak mutation, and

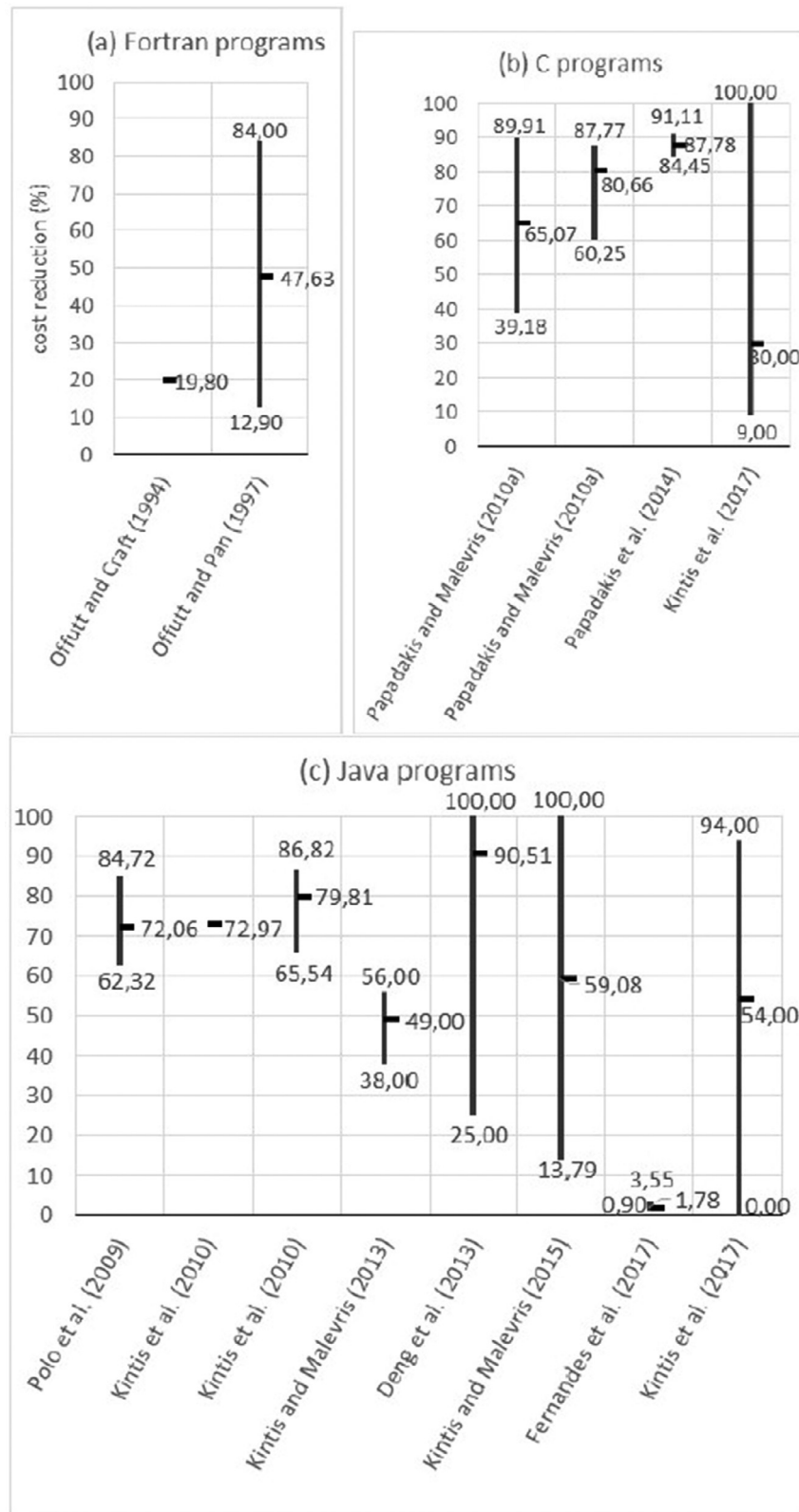


Fig. 17. Cost reduction regarding M-6 – Number of equivalent mutants automatically detected.

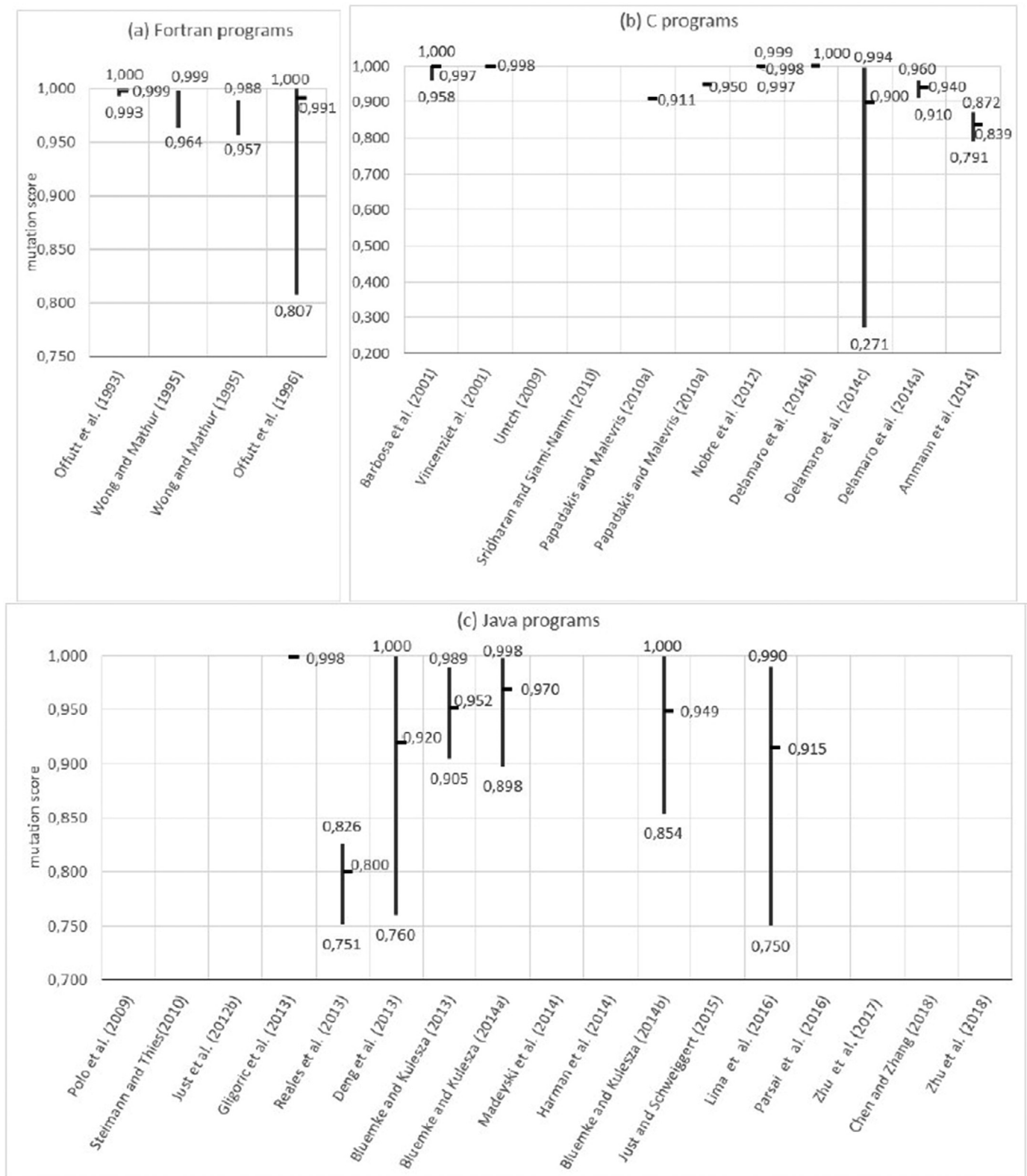


Fig. 18. Mutation scores achieved in studies that applied metric M-2 – Number of mutants to be executed.

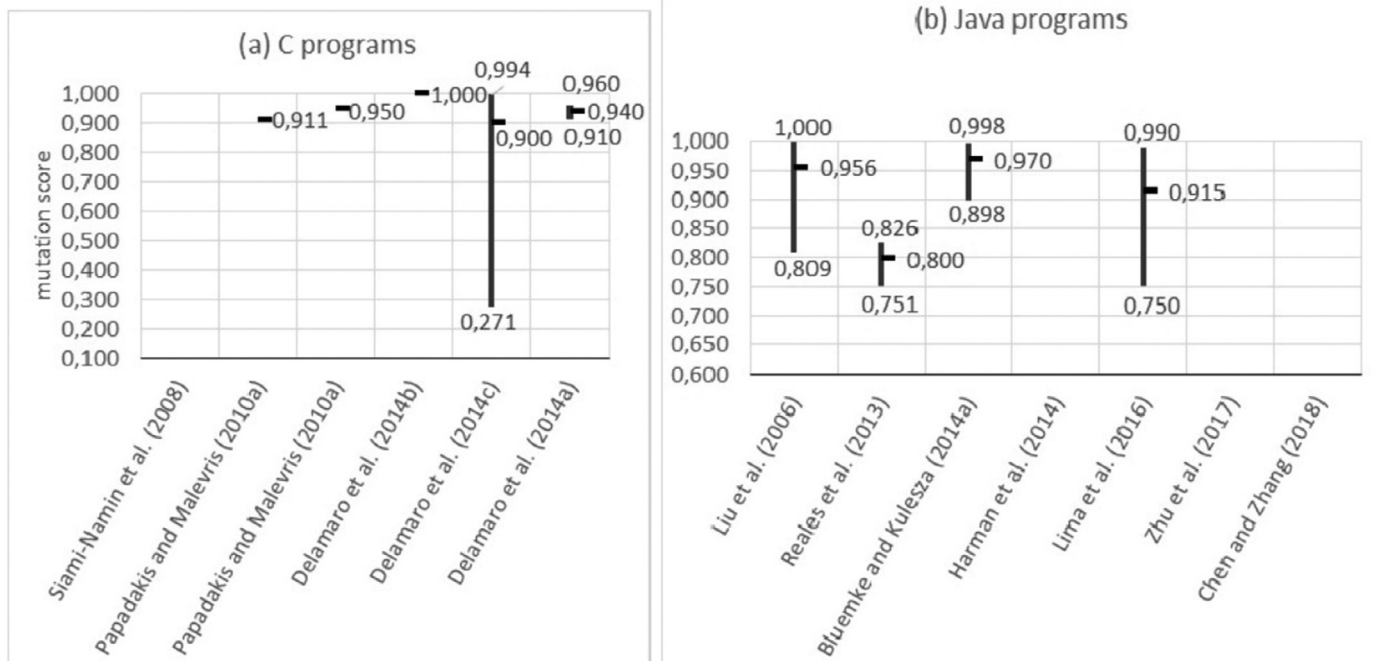


Fig. 19. Mutation scores achieved in studies that applied metric M-1 – Number of test cases required to achieve mutant coverage.

control flow analysis). Their studies compared mutation-based test case generation with branch coverage-based tests. One evaluation compared their approach with conventional mutation testing. They found an increase in mutation score, but the execution of the test generation algorithm was limited to four minutes.

Reuling et al. addressed mutation testing for software product lines (SPL). The authors set up three levels of SPL feature combinations, and were not able to use subjects that had more than 200 features (they ran out of time). Zhang et al. (2010b) investigated automatic test generation for Java programs and set a timeout after a predefined execution time or number of iterations, so it was impossible to extract cost reduction and mutation score values.

(iii) Results that are very context-specific

Some studies not listed in Section 6.2 were done in very particular contexts. Examples include specific tools (Zhang et al., 2010b; Aichernig et al., 2013; Zhang et al., 2012), regression testing (Cachia et al., 2013; Chen and Zhang, 2018; Zhang et al., 2012) specific algorithms (Reales and Polo, 2013), and unusual sets of mutation operators (Kaminski and Ammann, 2011; Gopinath et al., 2018), or programming constructs (Wedyan and Ghosh, 2012; Ferrari et al., 2013). For instance, Zhang et al. (2010b) compared results from two test generation tools (*Pex* and *PexMutator*), but only *PexMutator* was designed for mutation testing, so there was no basis for comparison. Aichernig et al. (2013) explored test case generation based on state machine models and measured cost reduction based on results produced by a prior implementation by the same authors (Aichernig and Jöbstl, 2012).¹⁰ Zhang et al. (2012) studied mutation testing for regression testing of Java programs and compared their results with results produced by the *JAVALANCHE* tool (Schuler et al., 2009).

Another example is by Reales and Polo (2013), who studied *Mutant execution speedup*. The authors discussed results for five alter-

native parallel execution implementations, but they were not compared with sequential mutant execution.

Kaminski and Ammann (2011) measured the *Number of test cases required to achieve mutant coverage* but only for logical expression mutants. Gopinath et al. (2018) created higher-order mutants but only used the statement deletion mutation operator. Wedyan and Ghosh (2012) and Ferrari et al. (2013) investigated mutation testing for *AspectJ* programs. They calculated cost reduction with respect to automatic detection of equivalent mutants of pointcut expressions, which are specific to aspect-oriented programming.

Other examples of varied contexts for mutation testing in which cost reduction has been measured are formal behavioural models (Devroey et al., 2017), and variability points in configurable systems (Al-Hajjaji et al., 2017).

(iv) Imprecise results

Some studies reduced costs of mutation testing, but we could not extract precise cost reduction values from the reported results. An example was Kintis et al.'s study (Kintis et al., 2015), which classified 81% of mutants as killable with 71% precision. Even though this was 20% "superior" to previous approaches, we could not define a percentage of cost reduction from the reported data. Schuler et al.'s studies were similar (Schuler et al., 2009; Schuler and Zeller, 2013).

Kurtz et al. (2016) presented several cross-comparison experiments involving redundant, equivalent, and dominator mutants. However, they did not present precise values for cost reduction or mutation scores. Marcozzi et al. (2018) addressed the automatic identification of equivalent, redundant, and trivial mutants, which they called *polluting test objectives*. They did not compare their results with baseline or ground-truth numbers of equivalent or trivial mutants.

(v) Results obtained from inconsistent subjects

We also found inconsistencies among the sets of subject programs used in some studies. For example, Papadakis and

¹⁰ Aichernig and Jöbstl's study (Aichernig and Jöbstl, 2012) was subsumed by their 2013 study (Aichernig et al., 2013), as shown in Table A.2.

Malevris (2011b) measured *Mutant execution speedup* based on seven Java programs. In the same study, however, the authors compared results for automatic test generation based on 10 programs that included only five of the original seven programs. Ma and Kim (2016) studied cost reduction with *Number of test case executions* and *Mutant execution speedup*. While the cost reduction based on the first metric (*Number of test case executions*) was collected for three groups of subjects, the measurement of results for *Mutant execution speedup* was presented for a single group of programs. As another example, Chen and Zhang (2018) measured cost reduction based on four metrics: *Number of test cases required to achieve mutant coverage*; *Number of mutants to be executed*; *Mutant execution speedup*; and *Number of test case executions*. However, the paper presented results only for a subset of subjects for *Mutant execution speedup*.

7. Threats to validity

This section discusses threats to the validity of our SLR and how we mitigated them. The two main threats to validity are (i) the completeness of the results; and (ii) the classification of studies based on the lists of categories. Regarding the first, the results come from an analysis of 175 peer-reviewed studies published in journals, conference, and workshop proceedings. These studies subsumed other 22 studies. These studies do not include “gray literature,” including Masters and PhD theses, technical reports, technical specifications, and other unrefereed documents. This same focus on selecting only peer-reviewed studies was adopted in other SLRs (Wohlin, 2014; MacDonell et al., 2010). Furthermore, other surveys on mutation testing (summarized in Section 8) show similar trends, particularly with respect to the growing number of published studies in the last 15 years (Papadakis et al., 2019; Jia and Harman, 2011; Madeyski et al., 2014; Silva et al., 2017). The inclusion of additional, non-peer-reviewed, studies would probably not change these results, although it would increase the numbers in this paper.

The backward snowballing technique was used after the first search round with only one level of depth. Wohlin (2014) suggested snowballing should be an iterative process that ends only when no new studies are found. Therefore, our procedure to apply snowballing may have created another threat to completeness. To mitigate this threat, we used Wohlin’s alternative suggestion for contacting authors of primary studies to ask for additional primary studies. In particular, we contacted every author of selected primary studies to ask for additional papers to be analyzed (see Section 3.3 for more details). As a result, we analyzed 135 more studies suggested in 42 replies, and added 25 items to our final set of studies. The same search procedure was adopted by Jia and Harman (2011), who called it a ‘transitive closure’ on the literature.

For threat (ii), the initial classification of studies based on the *do fewer*, *do smarter*, and *do faster* categories (Offutt and Untch, 2000) is no longer adequate. When these categories were first defined, fewer techniques were known. More recent techniques are hard to categorize into the *do fewer*, *do smarter*, and *do faster* groups. Category *T-15* for cost reduction techniques reflects this complexity. In addition, since 2000 it has become more common to combine multiple techniques, as can be seen in Figs. 6 and 10. As far as possible, the classification we present in this paper was performed in a discerning and unbiased manner.

Still regarding threat (ii), as discussed in Section 5.3, several studies were assigned category *T-15* (*optimization of generation, execution and analysis of mutants*) as an “others” category. It might be possible to further analyze those techniques to identify subcategories, increasing the list of cost reduction techniques. It might

also be reasonable to combine some categories, as discussed in Section 5.4.

8. Related work

In some sense, every paper referenced here is “related.” For our related work section, therefore, we focus on other surveys on mutation testing, with a specific emphasis on surveys that focus on cost reduction.

Polo and Reales (2010) summarized a small set of reference papers that tried to reduce costs of the main steps in mutation testing: (1) mutant generation, (2) test case generation and execution, and (3) result analysis. The authors suggested techniques such as *T-1* (*random mutation*) and *T-12* (*selective mutation*) for step (1); *T-7* (*minimization and prioritization of test sets*) and *T-3* (*weak mutation*) for step (2); and *T-2* (*higher order mutation*) for step (3). Polo and Reales did not use systematic search criteria to identify the techniques and they did not classify studies by technique. They only analyzed 10 cost reduction-related studies, all of which we include here. One interesting note regarding Polo and Reales’s work is that the authors grouped together *test case generation* and *number of test case executions* as a common goal for cost reduction, as we did with our primary goal *PG-4* (Section 5.2).

Jia and Harman (2011) published what is probably the most extensive survey of mutation testing in general to date. They identified several cost reduction techniques, including equivalent mutant detection. Jia and Harman identified the first peer-reviewed cost reduction paper as the 1982 paper that introduced *weak mutation* (Howden, 1982). We include this paper in Fig. 6, but not as a selected paper, as it did not focus on cost reduction, did not implement the idea, and included no empirical validation. Our SLR search found the first study that used *weak mutation* to reduce cost to be Marshall et al. (1990) in 1990. Jia and Harman summarized 66 mutation testing publications they classified as cost reduction-related, published from 1982 through 2009, including 52 peer-reviewed papers and 14 non-indexed (mostly, gray literature) items such as PhD and Masters Theses, technical reports, and Doctoral Symposium papers. In the same timeframe, our paper includes 44 peer-reviewed studies specifically focused on cost reduction.

Madeyski et al. (2014) published results of an SLR on the equivalent mutant problem. The authors classified 24 studies in three categories, based on the goal: *detecting equivalent mutants*, *avoiding equivalent mutant generation*, and *suggesting equivalent mutants*. Moreover, they identified 17 methods that contribute to achieve such goals. This paper includes 15 of Madeyski et al.’s studies; the others either did not focus on cost reduction, were not peer-reviewed, or are included here as related work.

Silva et al. (2017) performed an SLR on the use of Search-based Software Testing (SBST) techniques in mutation testing. The authors identified 69 studies published between 1998 and 2014, focusing primarily on the meta-heuristics applied. As opposed to this study, Silva et al. provided an overall characterization of their topic without focusing on cost reduction-related studies. They also included non-peer-reviewed studies.

In the most recent related work, Papadakis et al. (2019) updated Jia and Harman (2011)’s survey to include studies published from 2008 to 2017 (inclusive). The authors stated they selected a total of 502 publications (although only 405 appear in the reference list), of which 260 addressed mutation testing problems (including supporting tools), while 242 either used mutation testing to assess test quality, or tackled problems unrelated to mutation testing. According to Papadakis et al., their study focused on conferences, symposia, workshops, and journals. When contrasted with our work, we searched the same databases that index the conference proceedings and journals they analyzed. Therefore,

papers in their dataset that are not in our dataset are probably due to differences in our selection criteria. Papadakis et al.'s paper did not report the selection criteria so we could not compare directly.

9. Conclusion, recommendations, and future work

Mutation testing is a highly investigated and very effective way to generate tests and to assess test quality. However, it is also very expensive. The four major cost factors are (i) a large number of mutants are generated and executed, even for small programs; (ii) test data generation; (iii) test suites are large; and (iv) equivalent mutants. In response, researchers have developed many approaches to reduce the cost of mutation. This paper summarized results of a systematic literature review that characterized the history and the state-of-the-art of cost reduction for mutation testing. We updated and extended our prior paper (Ferrari et al., 2018a), analyzed the evolution of research on the topic, and summarized its underlying goals, techniques, metrics used, and results achieved.

We analyzed a total of 175 peer-reviewed studies, of which 153 present either original or updated approaches and results for mutation cost reduction. The 21 techniques identified use six main goals to reduce mutation cost. Apart from the *Optimization*-related category (which groups a set of techniques that could not be otherwise classified), the techniques investigated the most are *control-flow analysis*, *evolutionary algorithms*, *selective mutation*, *higher order mutation*, and *data-flow analysis*.

Experimental results were measured with 18 metrics, plus a range of complex, study-specific metrics that were not addressed in this review. The mostly commonly used metrics are the *Number of mutants to be executed*, *Mutant execution speedup*, and the *Number of test cases required to achieve mutant coverage*.

Summary of recommendations: Based on the research questions investigated in this work, we summarize several recommendations.

- **Targeting the primary goals:** Our results reveal that *reducing the number of mutants* (PG-1) was the most common primary goal, followed by *executing faster* (PG-3), then *automatically detecting equivalent mutants* (PG-2). If we look at the last 10 years considered in this research (2009 to 2018), we found 113 studies, 15 of which focused on PG-2 (Papadakis and Le Traon, 2013; Schuler et al., 2009; Schuler and Zeller, 2013; Kintis and Malevris, 2015; Papadakis et al., 2014; Kintis and Malevris, 2013; Ferrari et al., 2013; Kintis et al., 2015; Patel and Hierons, 2016; Holling et al., 2016; Kintis et al., 2017; Devroey et al., 2017; Marcozzi et al., 2018; Delgado-Pérez et al., 2017; McMinn et al., 2018). Thus, only 13% of the studies from the past decade directly addressed a problem that is widely considered the major obstacle to practical adoption of mutation testing (Jia and Harman, 2011; Madeyski et al., 2014). The total number of studies related to equivalent mutants corroborates this observation. In a recent survey, Papadakis et al. (2019) also found that equivalent mutant detection remains an open problem. Therefore, we recommend more research into the problem. We found 12 studies (11 in the past decade) that addressed *avoiding the creation of certain mutants* (PG-5). Rules to prevent creation of equivalent and redundant mutant can be embedded into mutation operators. Given that this goal impacts the largest number of other goals, including reducing the number of equivalent mutants, we recommend more research on this problem.
- **Applying combined techniques:** We found that mutation cost reduction research has become more interdisciplinary over time.

It is true that traditional mutation-specific techniques such as changing the set of operators (e.g. *selective mutation* and *one-op mutation*), *weak mutation*, *higher-order mutation*, and *minimal mutation* are still studied. However, we also found frequent and increasing interest in techniques borrowed from other Computer Science subareas such as software analysis and artificial intelligence.

We also found several studies that combined techniques to achieve substantial savings. For example, *weak mutation* and *optimization* were applied together to reduce the number of mutants to be executed and mutant execution time, with 97% and 94% savings (Zhu et al., 2018). As another example, *control-flow analysis* has been combined with *constrained mutation* and *random mutation* in industrial setting (Petrović and Ivanković, 2018). This suggests that stronger collaboration with researchers from different subareas can be very productive.

- **Standardizing and sharing experimental methods, materials and reports:** While collecting cost reduction results from the selected studies, we found lots of differences in experimental design that affected the results. The level of detail in the papers also varied greatly. As a consequence, we could not compare studies with similar goals (e.g. to run meta-analysis of results). Moreover, very few experimental materials are available online. Therefore, we recommend the community to create benchmarks that include all software artifacts, including software under test, test cases that reveal actual faults, test cases that kill mutants, mutant programs, mutation execution results in the form of killing matrices, as well as tools to produce and handle such artifacts and directions on their use. This would greatly help researchers design experiments that can be compared and reproduced.

Future work: As with any SLR, it is unlikely that we found all primary studies, as acknowledged as a validity threat in Section 7. Even though several search strategies were used, well-known limitations of scientific writing and repositories mean no search can be perfect. Thus we hope to see future updates to this systematic literature review, performed by either ourselves or other researchers.

Acknowledgements

We would like to thank all 42 authors that suggested studies to be included into the SLR dataset. We also thank George Mason University for hosting Fabiano Ferrari during part of his participation in this research. Finally, we are also thankful for the financial support, detailed as follows: Fabiano Ferrari was supported by the *Fundação de Amparo Pesquisa do Estado de São Paulo* - Brasil, grant #2016/21251-0; and *Conselho Nacional de Desenvolvimento Científico e Tecnológico* - Brasil, grant #306310/2016-3; Alessandro Pizzoleto was supported by the *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior* - Finance Code 001. Jeff Offutt was partly supported by KKS (The *Knowledge Foundation*), by project 20130085, Testing of Critical System Characteristics (TOCSYC); and partly by a Google educational grant, Self-Paced Learning Increases Retention and Capacity (SPARC).

Appendix A. Selected studies and subsumed studies

This appendix includes tables that list all primary studies that were selected in this SLR (Table A.1), and a table that list all subsumed studies (Table A.2). Tables of selected studies are sorted by author name and year of study publication, while tables of subsumed studies are sorted by year of publication of the updated or extended studies.

Table A.1
Selected studies.

Author/Year	Title	Publisher	Journal/Event
Abuljadayel and Wedyan (2018)	An Approach for the Generation of Higher Order Mutants Using Genetic Algorithms	Modern Education and Computer Science Press	International Journal of Intelligent Systems and Applications
Aichernig et al. (2013)	Incremental Refinement Checking for Test Case Generation	Springer	International Conference on Quality Software (QSIC)
Aichernig et al. (2014)	Model-Based Mutation Testing of an Industrial Measurement Device	Springer	International Conference Tests and Proofs (TAP)
Al-Hajjaji et al. (2017)	Efficient Mutation Testing in Configurable Systems	IEEE	International Workshop on Variability and Complexity in Software Design (VACE)
Alexander et al. (2002)	Mutation of Java Objects	IEEE	International Symposium on Software Reliability Engineering (ISSRE)
Ammann et al. (2014)	Establishing Theoretical Minimal Sets of Mutants	IEEE	International Conference on Software Testing, Verification and Validation (ICST)
Anbalagan and Xie (2008)	Automated Generation of Pointcut Mutants for Testing Pointcuts in AspectJ Programs	IEEE	International Symposium on Software Reliability Engineering (ISSRE)
Adamopoulos et al. (2004)	How to Overcome the Equivalent Mutant Problem and Achieve Tailored Selective Mutation Using Co-evolution	Springer	Genetic and Evolutionary Computation Conference (GECCO)
Ayari et al. (2007)	Automatic Mutation Test Input Data Generation via Ant Colony	ACM	Genetic and Evolutionary Computation Conference (GECCO)
Barbosa et al. (2001)	Toward the Determination of Sufficient Mutant Operators for C	Wiley	Software Testing, Verification and Reliability
Bashir and Nadeem (2017)	Improved Genetic Algorithm to Reduce Mutation Testing Cost	IEEE	IEEE Access
Baudry et al. (2005)	From Genetic to Bacteriological Algorithms for Mutation-Based Testing	Wiley	Software Testing, Verification and Reliability
Belli and Beyazit (2015)	Exploiting Model Morphology for Event-Based Testing	IEEE	IEEE Transactions on Software Engineering
Bluemke and Kulesza (2013)	Reduction of Computational Cost in Mutation Testing by Sampling Mutants	Springer	International Conference on Dependability and Complex Systems (DepCoS-RELCOMEX)
Bluemke and Kulesza (2014b)	Reductions of Operators in Java Mutation Testing	Springer	International Conference on Dependability and Complex Systems (DepCoS-RELCOMEX)
Bluemke and Kulesza (2014a)	Reduction in Mutation Testing of Java Classes	IEEE	International Conference on Software Engineering and Applications (ICSOFT-EA)
Bogacki and Walter (2006a)	Aspect-oriented Response Injection: an Alternative to Classical Mutation Testing	Springer	IFIP Working Conference on Software Engineering Techniques: Design for Quality (SET)
Choi and Mathur (1993)	High-Performance Mutation Testing	Elsevier	Journal of Systems and Software
Cachia et al. (2013)	Towards Incremental Mutation Testing	Elsevier	Validation Strategies for Software Evolution Workshop (VSSE)
Chen and Zhang (2018)	Speeding up Mutation Testing via Regression Test Selection: An Extensive Study	IEEE	International Conference on Software Testing, Verification and Validation (ICST)
Delamaro et al. (2014a)	Experimental Evaluation of SDL and One-Op Mutation for C	IEEE	International Conference on Software Testing, Verification and Validation (ICST)
Delamaro et al. (2014c)	Designing Deletion Mutation Operators	IEEE	International Conference on Software Testing, Verification and Validation (ICST)
Delamaro et al. (2014b)	Growing a Reduced Set of Mutation Operators	IEEE	Brazilian Symposium on Software Engineering (SBES)
Delgado-Pérez et al. (2017b)	GiGAn: Evolutionary Mutation Testing for C++ Object-oriented Systems	ACM	ACM Symposium on Applied Computing (SAC)
Delgado-Pérez et al. (2017c)	Assessment of C++ Object-Oriented Mutation Operators: A Selective Mutation Approach	Wiley	Software Testing, Verification and Reliability
Delgado-Pérez et al. (2017a)	Assessment of Class Mutation Operators for C++ with the MuCPP Mutation System	Elsevier	Information and Software Technology
Delgado-Pérez et al. (2017)	Using Evolutionary Computation to Improve Mutation Testing	Springer	International Work-Conference on Artificial Neural Networks (IWANN)
DeMillo et al. (1991)	Compiler-integrated Program Mutation	IEEE	IEEE Annual Computer Software and Applications Conference (COMPSAC)
DeMillo and Offutt (1991)	Constraint-based Automatic Test Data Generation	IEEE	IEEE Transactions on Software Engineering
DeMillo and Offutt (1993)	Experimental Results from an Automatic Test Case Generator	ACM	ACM Transactions on Software Engineering and Methodology
Deng et al. (2013)	Empirical Evaluation of the Statement Deletion Mutation Operator	IEEE	International Conference on Software Testing, Verification and Validation (ICST)
Denisov and Pankevich (2018)	Mull It Over: Mutation Testing Based on LLVM	IEEE	International Workshop on Mutation Analysis (Mutation)
Derezińska and Rudnik (2012)	Quality Evaluation of Object-Oriented and Standard Mutation Operators Applied to C# Programs	Springer	International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS)
Derezińska (2013)	A Quality Estimation of Mutation Clustering in C# Programs	Springer	International Conference on Dependability and Complex Systems (DepCoS-RELCOMEX)
Derezińska and Hałas (2015)	Improving Mutation Testing Process of Python Programs	Springer	Computer Science On-line Conference Software Engineering in Intelligent Systems (CSOC)

(continued on next page)

Table A.1 (continued)

Author/Year	Title	Publisher	Journal/Event
Derezińska (2016)	Evaluation of Deletion Mutation Operators in Mutation Testing of C# Programs	Springer	International Conference on Dependability and Complex Systems (DepCoS-RELCOMEX)
Derezińska and Rudnik (2017)	Evaluation of Mutant Sampling Criteria in Object-Oriented Mutation Testing	Polskie Towarzystwo Informatyczne	Federated Conference on Computer Science and Information Systems (ACSIS)
Devroey et al. (2016)	Featured Model-based Mutation Analysis	ACM	International Conference on Software Engineering (ICSE)
Devroey et al. (2017)	Automata Language Equivalence vs. Simulations for Model-Based Mutant Equivalence: An Empirical Evaluation	IEEE	International Conference on Software Testing, Verification and Validation (ICST)
Domínguez-Jiménez et al. (2009a)	GAmara: An Automatic Mutant Generation System for WS-BPEL Compositions	IEEE	IEEE European Conference on Web Services (ECOWS)
Domínguez-Jiménez et al. (2011)	Evolutionary Mutation Testing	Elsevier	Information and Software Technology
Durelli et al. (2012)	Toward Harnessing High-Level Language Virtual Machines for Further Speeding Up Weak Mutation Testing	IEEE	International Conference on Software Testing, Verification and Validation (ICST)
Fernandes et al. (2017)	Avoiding Useless Mutants	ACM	International Conference on Generative Programming: Concepts and Experiences (GPCE)
Ferrari et al. (2013)	Towards the Practical Mutation Testing of AspectJ Programs	Elsevier	Science of Computer Programming
Matnei Filho and Vergilio (2015)	A Mutation and Multi-objective Test Data Generation Approach for Feature Testing of Software Product Lines	IEEE	Brazilian Symposium on Software Engineering (SBES)
Fleishgaker and Weiss (1994)	Efficient Mutation Analysis: A New Approach	ACM	International Symposium on Software Testing and Analysis (ISSTA)
Fraser and Zeller (2012)	Mutation-Driven Generation of Oracles and Unit Tests	IEEE	IEEE Transactions on Software Engineering
Fraser and Arcuri (2015)	Achieving Scalable Mutation-based Generation of Whole Test Suites	Springer	Empirical Software Engineering
Gligoric et al. (2010)	MuTMuT: Efficient Exploration for Mutation Testing of Multithreaded Code	IEEE	International Conference on Software Testing, Verification and Validation (ICST)
Gligoric et al. (2012)	Efficient Mutation Testing of Multithreaded Code	Wiley	Software Testing, Verification and Reliability
Gligoric et al. (2013)	Selective Mutation Testing for Concurrent Code	ACM	International Symposium on Software Testing and Analysis (ISSTA)
Gopinath et al. (2016b)	Topsy-Turvy: A Smarter and Faster Parallelization of Mutation Analysis	ACM	International Conference on Software Engineering (ICSE)
Gopinath et al. (2017)	Mutation Reduction Strategies Considered Harmful	IEEE	IEEE Transactions on Software Reliability
Gopinath et al. (2018)	If You Can't Kill a Supermutant, You Have a Problem	IEEE	International Workshop on Mutation Analysis (Mutation)
Harman et al. (2000)	The Relationship Between Program Dependence and Mutation Analysis	Kluwer	Mutation 2000 Symposium
Harman et al. (2014)	Angels and Monsters: An Empirical Investigation of Potential Test Effectiveness and Efficiency Improvement from Strongly Subsuming Higher Order Mutation	ACM	International Conference on Automated Software Engineering (ASE)
Hierons et al. (1999)	Using Program Slicing to Assist in the Detection of Equivalent Mutants	Wiley	Software Testing, Verification and Reliability
Henard et al. (2014)	Mutation-Based Generation of Software Product Line Test Configurations	Springer	International Symposium Search-Based Software Engineering (SSBSE)
Holling et al. (2016)	Nequivack: Assessing Mutation Score Confidence	IEEE	International Workshop on Mutation Analysis (Mutation)
Hu et al. (2011)	An Analysis of OO Mutation Operators	IEEE	International Workshop on Mutation Analysis (Mutation)
Iida and Takada (2017)	Reducing Mutants with Mutant Killable Precondition	IEEE	International Workshop on Mutation Analysis (Mutation)
Inozemtseva et al. (2013)	Using Fault History to Improve Mutation Reduction	ACM	International Symposium on the Foundations of Software Engineering (FSE)
Jackson and Woodward (2000)	Parallel Firm Mutation of Java Programs	Kluwer	Mutation 2000 Symposium
Ji et al. (2009)	A Novel Method of Mutation Clustering Based on Domain Analysis	Knowledge Systems Institute Graduate School	International Conference on Software Engineering and Knowledge Engineering (SEKE)
Just et al. (2011)	Using Conditional Mutation to Increase the Efficiency of Mutation Analysis	ACM	International Workshop on Automation of Software Test (AST)
Just et al. (2012b)	Using Non-Redundant Mutation Operators and Test Suite Prioritization to Achieve Efficient and Scalable Mutation Analysis	IEEE	International Symposium on Software Reliability Engineering (ISSRE)
Just et al. (2014a)	Efficient Mutation Analysis by Propagating and Partitioning Infected Execution States	ACM	International Symposium on Software Testing and Analysis (ISSTA)
Just and Schweiggert (2015)	Higher Accuracy and Lower Run Time: Efficient Mutation Analysis Using Non-redundant Mutation Operators	Wiley	Software Testing, Verification and Reliability

(continued on next page)

Table A.1 (continued)

Author/Year	Title	Publisher	Journal/Event
Just et al. (2017)	Inferring Mutant Utility from Program Context	ACM	International Symposium on Software Testing and Analysis (ISSTA)
Kaminski and Ammann (2009)	Using a Fault Hierarchy to Improve the Efficiency of DNF Logic Mutation Testing	IEEE	International Conference on Software Testing, Verification and Validation (ICST)
Kaminski and Ammann (2011)	Reducing Logic Test Set Size while Preserving Fault Detection	Wiley	Software Testing, Verification and Reliability
Kaminski et al. (2011b)	A Logic Mutation Approach to Selective Mutation for Programs and Queries	Elsevier	Information and Software Technology
Kaminski et al. (2013)	Improving logic-based testing	Elsevier	Journal of Systems and Software
Kim et al. (2012)	Combining Weak and Strong Mutation for a Noninterpretive Java Mutation System	Wiley	Software Testing, Verification and Reliability
Kintis et al. (2010)	Evaluating Mutation Testing Alternatives: A Collateral Experiment	IEEE	Asia-Pacific Software Engineering Conference (APSEC)
Kintis and Malevris (2013)	Identifying More Equivalent Mutants via Code Similarity	IEEE	Asia-Pacific Software Engineering Conference (APSEC)
Kintis and Malevris (2014)	Using Data Flow Patterns for Equivalent Mutant Detection	IEEE	International Workshop on Mutation Analysis (Mutation)
Kintis et al. (2015)	Employing Second-order Mutation for Isolating First-order Equivalent Mutants	Wiley	Software Testing, Verification and Reliability
Kintis and Malevris (2015)	MEDIC: A Static Analysis Framework for Equivalent Mutant Identification	Elsevier	Information and Software Technology
Kintis et al. (2017)	Detecting Trivial Mutant Equivalences via Compiler Optimisations	IEEE	IEEE Transactions on Software Engineering
Krauser et al. (1991)	High Performance Software Testing on SIMD Machines	IEEE	IEEE Transactions on Software Engineering
Kurtz et al. (2016)	Analyzing the Validity of Selective Mutation with Dominator Mutants	ACM	International Symposium on the Foundations of Software Engineering (FSE)
Lacerda and Ferrari (2014)	Towards the Establishment of a Sufficient Set of Mutation Operators for AspectJ Programs	SBC	Brazilian Symposium on Systematic and Automated Software Testing (SAST)
Li et al. (2015)	Mutation Testing in Practice Using Ruby	IEEE	International Workshop on Mutation Analysis (Mutation)
Lima et al. (2016)	Evaluating Different Strategies for Reduction of Mutation Testing Costs	ACM	Brazilian Symposium on Systematic and Automated Software Testing (SAST)
Liu et al. (2006)	An Approach to Test Data Generation for Killing Multiple Mutants	IEEE	International Conference on Software Maintenance (ICSM)
Ma et al. (2005)	MuJava: An Automated Class Mutation System	Wiley	Software Testing, Verification and Reliability
Ma and Kim (2016)	Mutation Testing Cost Reduction by Clustering Overlapped Mutants	Elsevier	Journal of Systems and Software
Madeyski et al. (2014)	Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation	IEEE	IEEE Transactions on Software Engineering
Marcozzi et al. (2018)	Time to Clean Your Test Objectives	IEEE	International Conference on Software Engineering (ICSE)
Marshall et al. (1990)	Static Dataflow-aided Weak Mutation Analysis (SDAWM)	Elsevier	Information and Software Technology
Reales and Polo (2013)	Parallel Mutation Testing	wiley	Software Testing, Verification and Reliability
Reales et al. (2013)	Validating Second-Order Mutation at System Level	IEEE	IEEE Transactions on Software Engineering
Reales and Polo (2014)	Reducing Mutation Costs Through Uncovered Mutants	Wiley	Software Testing, Verification and Reliability
McMinn et al. (2016)	Virtual Mutation Analysis of Relational Database Schemas	IEEE	International Workshop on Automation of Software Test (AST)
McMinn et al. (2018)	Automatic Detection and Removal of Ineffective Mutants for the Mutation Analysis of Relational Database Schemas	IEEE	IEEE Transactions on Software Engineering
Mresa and Bottaci (1999)	Efficiency of Mutation Operators and Selective Mutation Strategies: an Empirical Study	Wiley	Software Testing, Verification and Reliability
Siarni-Namin and Andrews (2006)	Finding Sufficient Mutation Operators via Variable Reduction	IEEE	International Workshop on Mutation Analysis (Mutation)
Siarni-Namin et al. (2008)	Sufficient Mutation Operators for Measuring Test Effectiveness	ACM	International Conference on Software Engineering (ICSE)
Nobre et al. (2012)	Reducing Interface Mutation Costs with Multiobjective Optimization Algorithms	IGI Global	International Journal of Natural Computing Research
Offutt et al. (1992)	Mutation Testing of Software Using a MIMD Computer	Wiley	International Conference on Parallel Processing (ICPP)
Offutt et al. (1993)	An Experimental Evaluation of Selective Mutation	IEEE	International Conference on Software Engineering (ICSE)
Offutt and Craft (1994)	Using Compiler Optimization Techniques to Detect Equivalent Mutants	Wiley	Software Testing, Verification and Reliability
Offutt and Lee (1994)	An Empirical Evaluation of Weak Mutation	IEEE	IEEE Transactions on Software Engineering
Offutt et al. (1996)	An Experimental Determination of Sufficient Mutant Operators	ACM	ACM Transactions on Software Engineering and Methodology

(continued on next page)

Table A.1 (continued)

Author/Year	Title	Publisher	Journal/Event
Offutt and Pan (1997)	Automatically Detecting Equivalent Mutants and Infeasible Paths	Wiley	Software Testing, Verification and Reliability
Offutt et al. (1999)	The Dynamic Domain Reduction Procedure for Test Data Generation	Wiley	Software Practice and Experience
Offutt et al. (2006)	The Class-Level Mutants of MuJava	ACM	International Workshop on Automation of Software Test (AST)
Oliveira et al. (2013)	A Coevolutionary Algorithm to Automatic Test Case Selection and Mutant in Mutation Testing	IEEE	IEEE Congress on Evolutionary Computation (CEC)
Omar and Ghosh (2012)	An Exploratory Study of Higher Order Mutation Testing in Aspect-Oriented Programming	IEEE	International Symposium on Software Reliability Engineering (ISSRE)
Papadakis and Malevris (2009)	An Effective Path Selection Strategy for Mutation Testing	IEEE	Asia-Pacific Software Engineering Conference (APSEC)
Papadakis and Malevris (2010a)	An Empirical Evaluation of the First and Second Order Mutation Testing Strategies	IEEE	International Workshop on Mutation Analysis (Mutation)
Papadakis and Malevris (2010b)	Automatic Mutation Test Case Generation via Dynamic Symbolic Execution	IEEE	International Symposium on Software Reliability Engineering (ISSRE)
Papadakis and Malevris (2011a)	Automatic Mutation based Test Data Generation	ACM	Genetic and Evolutionary Computation Conference (GECCO)
Papadakis and Malevris (2011b)	Automatically Performing Weak Mutation with the Aid of Symbolic Execution, Concolic Testing and Search-based Testing	Springer	Software Quality Journal
Papadakis and Malevris (2012)	Mutation Based Test Case Generation Via a Path Selection Strategy	Elsevier	Information and Software Technology
Papadakis and Le Traon (2013)	Mutation Testing Strategies using Mutant Classification	ACM	ACM Symposium on Applied Computing (SAC)
Papadakis et al. (2014)	Mitigating the Effects of Equivalent Mutants with Mutant Classification Strategies	Elsevier	Science of Computer Programming
Parsai et al. (2016)	Evaluating Random Mutant Selection at Class-level in Projects with Non-adequate Test Suites	ACM	International Conference on Evaluation and Assessment in Software Engineering (EASE)
Patel and Hierons (2016)	Resolving the Equivalent Mutant Problem in the Presence of Non-determinism and Coincidental Correctness	Springer	International Conference on Testing Software and Systems (ICTSS)
Patrick et al. (2012)	MESSI: Mutant Evaluation by Static Semantic Interpretation	IEEE	International Conference on Software Testing, Verification and Validation (ICST)
Petrović and Ivanković (2018)	State of Mutation Testing at Google	IEEE	International Conference on Software Engineering - Software Engineering in Practice Track (ICSE-SEIP)
Prapamontripong and Offutt (2017)	Finding Redundancy in Web Mutation Operators	IEEE	International Workshop on Mutation Analysis (Mutation)
Quyen et al. (2016)	Improving Mutant Generation for Simulink Models Using Genetic Algorithm	IEEE	International Conference on Electronics, Information, and Communications (ICEIC)
Reuling et al. (2015)	Fault-based Product-line Testing: Effective Sample Generation Based on Feature-Diagram Mutation	ACM	International Conference on Software Product Line (SPLC)
Sahinoğlu and Spafford (1990)	A Bayes Sequential Statistical Procedure for Approving Software Products	Elsevier	IFIP Conference on Approving Software Products (ASP)
Schuler et al. (2009)	Efficient Mutation Testing by Checking Invariant Violations	ACM	International Symposium on Software Testing and Analysis (ISSTA)
Schuler and Zeller (2013)	Covering and Uncovering Equivalent Mutants	Wiley	Software Testing, Verification and Reliability
Sridharan and Siami-Namin (2010)	Prioritizing Mutation Operators based on Importance Sampling	IEEE	International Symposium on Software Reliability Engineering (ISSRE)
Steimann and Thies (2010)	From Behaviour Preservation to Behaviour Modification: Constraint-Based Mutant Generation	ACM	International Conference on Software Engineering (ICSE)
Sun et al. (2017a)	An Empirical Study on Mutation Testing of WS-BPEL Programs	Oxford University Press	Computer Journal
Sun et al. (2017b)	A Path-aware Approach to Mutant Reduction in Mutation Testing	Elsevier	Information and Software Technology
Tuya et al. (2007)	Mutating database queries	Elsevier	Information and Software Technology
Untch et al. (1993)	Mutation Analysis Using Mutant Schemata	ACM	International Symposium on Software Testing and Analysis (ISSTA)
Untch et al. (1997)	TUMS: Testing Using Mutant Schemata	ACM	Annual Southeast Regional Conference (ACM-SE)
Untch (2009)	On Reduced Neighborhood Mutation Analysis Using a Single Mutagenic Operator	ACM	Annual Southeast Regional Conference (ACM-SE)
Polo et al. (2009)	Decreasing the Cost of Mutation Testing with Second-Order Mutants	Wiley	Software Testing, Verification and Reliability
Vincenzi et al. (2001)	Unit and Integration Testing Strategies for C Programs Using Mutation	Wiley	Software Testing, Verification and Reliability
Vincenzi et al. (2002)	Bayesian-Learning Based Guidelines to Determine Equivalent Mutants	World Scientific Publishing	International Journal of Software Engineering and Knowledge Engineering

(continued on next page)

Table A.1 (continued)

Author/Year	Title	Publisher	Journal/Event
Wang et al. (2017)	Faster Mutation Analysis via Equivalence Modulo States	ACM	International Symposium on Software Testing and Analysis (ISSTA)
Wedyan and Ghosh (2012)	On Generating Mutants for AspectJ Programs	Elsevier	Information and Software Technology
Weiss and Fleyshgaker (1993)	Improved Serial Algorithms for Mutation Analysis	ACM	International Symposium on Software Testing and Analysis (ISSTA)
Wong et al. (1994)	Constrained Mutation in C Programs	Brazilian Computer Society	Brazilian Symposium on Software Engineering (SBES)
Wong and Mathur (1995)	Reducing the Cost of Mutation Testing: An Empirical Study	Elsevier	Journal of Systems and Software
Wright et al. (2013)	Efficient Mutation Analysis of Relational Database Structure Using Mutant Schemata and Parallelisation	IEEE	International Workshop on Mutation Analysis (Mutation)
Zhang et al. (2010a)	Is Operator-Based Mutant Selection Superior to Random Mutant Selection?	ACM	International Conference on Software Engineering (ICSE)
Zhang et al. (2010b)	Test Generation via Dynamic Symbolic Execution for Mutation Testing	IEEE	International Conference on Software Maintenance (ICSM)
Zhang et al. (2012)	Regression Mutation Testing	ACM	International Symposium on Software Testing and Analysis (ISSTA)
Zhang et al. (2013b)	Faster Mutation Testing Inspired by Test Prioritization and Reduction	ACM	International Symposium on Software Testing and Analysis (ISSTA)
Zhang et al. (2013a)	Operator-Based and Random Mutant Selection: Better Together	IEEE	International Conference on Automated Software Engineering (ASE)
Zhang et al. (2016)	Predictive Mutation Testing	ACM	International Symposium on Software Testing and Analysis (ISSTA)
Zhu et al. (2017)	Speeding-Up Mutation Testing via Data Compression and State Infection	IEEE	International Workshop on Mutation Analysis (Mutation)
Zhu et al. (2018)	An Investigation of Compression Techniques to Speed up Mutation Testing	IEEE	International Conference on Software Testing, Verification and Validation (ICST)

Table A.2
Subsumed studies.

Updated / Extended Studies		Subsumed studies	
Author/Year	Title	Author/Year	Title
Krauser et al. (1991)	High Performance Software Testing on SIMD Machines	Krauser et al. (1988)	High Performance Testing on SIMD Machines
Offutt and Lee (1994)	An Empirical Evaluation of Weak Mutation	Offutt and Lee (1991)	How Strong is Weak Mutation?
Untch et al. (1993)	Mutation Analysis Using Mutant Schemata	Untch (1992)	Mutation-based Software Testing Using Program Schemata
Wong and Mathur (1995)	Reducing the Cost of Mutation Testing: An Empirical Study	Mathur and Wong (1993)	Evaluation of the Cost of Alternative Mutation Testing Strategies
Wong and Mathur (1995)	Reducing the Cost of Mutation Testing: An Empirical Study	Wong et al. (1995)	Mutation Versus All-uses: An Empirical Evaluation of Cost, Strength and Effectiveness
Offutt and Pan (1997)	Automatically Detecting Equivalent Mutants and Infeasible Paths	Offutt and Pan (1996)	Detecting Equivalent Mutants and the Feasible Path Problem
Alexander et al. (2002)	Mutation of Java Objects	Bieman et al. (2001)	A Technique for Mutation of Java Objects
Bogacki and Walter (2006a)	Aspect-oriented Response Injection: an Alternative to Classical Mutation Testing	Bogacki and Walter (2006b)	Evaluation of Test Code Quality with Aspect-Oriented Mutations
Domínguez-Jiménez et al. (2009a)	GAmber: An Automatic Mutant Generation System for WS-BPEL Compositions	Domínguez-Jiménez et al. (2009b)	A Framework for Mutant Genetic Generation for WS-BPEL
Papadakis and Malevris (2011b)	Automatically Performing Weak Mutation with the Aid of Symbolic Execution, Concolic Testing and Search-based Testing	Papadakis et al. (2010)	Towards Automating the Generation of Mutation Tests
Fraser and Zeller (2012)	Mutation-Driven Generation of Oracles and Unit Tests	Fraser and Zeller (2010)	Mutation-Driven Generation of Oracles and Unit Tests
Schuler and Zeller (2013)	Covering and Uncovering Equivalent Mutants	Schuler and Zeller (2010)	(Un-)Covering Equivalent Mutants
Kaminski et al. (2013)	Improving logic-based testing	Kaminski et al. (2011a)	Better Predicate Testing
Reales and Polo (2014)	Reducing Mutation Costs Through Uncovered Mutants	Reales and Polo (2012)	Mutant Execution Cost Reduction: Through MUSIC (Mutant Schema Improved with Extra Code)
Just and Schweiggert (2015)	Higher Accuracy and Lower Run Time: Efficient Mutation Analysis Using Non-redundant Mutation Operators	Just et al. (2012a)	Do Redundant Mutants Affect the Effectiveness and Efficiency of Mutation Analysis?
Aichernig et al. (2013)	Incremental Refinement Checking for Test Case Generation	Aichernig and Jöbstl (2012)	Efficient Refinement Checking for Model-Based Mutation Testing
Kintis et al. (2015)	Employing Second-order Mutation for Isolating First-order Equivalent Mutants	Kintis et al. (2012)	Isolating First Order Equivalent Mutants via Second Order Mutation
Just et al. (2014a)	Efficient Mutation Analysis by Propagating and Partitioning Infected Execution States	Just (2014)	The Major Mutation Framework: Efficient and Scalable Mutation Analysis for Java

(continued on next page)

Table A.2 (continued)

Updated / Extended Studies		Subsumed studies	
Author/Year	Title	Author/Year	Title
Kintis et al. (2017)	Detecting Trivial Mutant Equivalences via Compiler Optimisations	Papadakis et al. (2015)	Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique
Gopinath et al. (2017)	Mutation Reduction Strategies Considered Harmful	Gopinath et al. (2016a)	On the Limits of Mutation Reduction Strategies
Petrović and Ivanković (2018)	State of Mutation Testing at Google	Petrović et al. (2018)	An Industrial Application of Mutation Testing: Lessons, Challenges, and Research Directions
McMinn et al. (2018)	Automatic Detection and Removal of Ineffective Mutants for the Mutation Analysis of Relational Database Schemas	Wright et al. (2014)	The Impact of Equivalent, Redundant and Quasi Mutants on Database Schema Mutation Analysis

References

- Abuljadayel, A., Wedyan, F., 2018. An approach for the generation of higher order mutants using genetic algorithms. *Int. J. Intell. Syst. Appl.* 10 (1), 34–45.
- Acree, A.T., Budd, T.A., DeMillo, R.A., Lipton, R.J., Sayward, F.G., 1979. Mutation Analysis. Technical Report GIT-ICS-79/08. School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, USA.
- Adamopoulos, K., Harman, M., Hierons, R.M., 2004. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In: *Proceedings of the 3rd Genetic and Evolutionary Computation Conference (GECCO)*. Springer, Seattle, WA, USA. II–1338–1349 (LNCS v.3103).
- Aichernig, B.K., Auer, J., Jöbstl, E., Korošec, R., Krenn, W., Schlick, R., Schmidt, B.V., 2014. Model-based mutation testing of an industrial measurement device. In: *Proceedings of the 8th International Conference Tests and Proofs (TAP)*. Springer, York, UK, pp. 1–19.
- Aichernig, B.K., Jöbstl, E., 2012. Efficient refinement checking for model-based mutation testing. In: *Proceedings of the 12th International Conference on Quality Software (QASIC)*. IEEE Computer Society, Xi'an, China, pp. 21–30.
- Aichernig, B.K., Jöbstl, E., Kegele, M., 2013. Incremental refinement checking for test case generation. In: *Proceedings of the 7th International Conference Tests and Proofs (TAP)*. Springer, Budapest, Hungary, pp. 1–19.
- Al-Hajjaji, M., Krüger, J., Benduhn, F., Leich, T., Saake, G., 2017. Efficient mutation testing in configurable systems. In: *Proceedings of the IEEE/ACM 2nd International Workshop on Variability and Complexity in Software Design (VACE)*. IEEE, Buenos Aires, Argentina, pp. 2–8.
- Alexander, R.T., Bieman, J.M., Ghosh, S., Ji, B., 2002. Mutation of Java objects. In: *Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, Annapolis, MD, USA, pp. 341–351.
- Ammann, P., Delamaro, M.E., Offutt, A.J., 2014. Establishing theoretical minimal sets of mutants. In: *Proceedings of the 7th Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Cleveland, OH, USA, pp. 21–30.
- Anbalagan, P., Xie, T., 2008. Automated generation of pointcut mutants for testing pointcuts in AspectJ programs. In: *Proceeding of the 19th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, pp. 239–248.
- Andrews, J.H., Briand, L.C., Labiche, Y., 2005. Is mutation an appropriate tool for testing experiments? In: *Proceedings of the 27th International Conference on Software Engineering (ICSE)*. ACM, St. Louis, MO, USA, pp. 402–411.
- Ayari, K., Bouktif, S., Antoniol, G., 2007. Automatic mutation test input data generation via ant colony. In: *Proceedings of the 9th Genetic and Evolutionary Computation Conference (GECCO)*. ACM, London, UK, pp. 1074–1081.
- Barbosa, E.F., Maldonado, J.C., Vincenzi, A.M.R., 2001. Toward the determination of sufficient mutant operators for C. *Softw. Test. Verif. Reliab.* 11 (2), 113–136.
- Bashir, M.B., Nadeem, A., 2017. Improved genetic algorithm to reduce mutation testing cost. *IEEE Access* 5, 3657–3674.
- Baudry, B., Fleurey, F., Jézéquel, J.-M., Le Traon, Y., 2005. From genetic to bacteriological algorithms for mutation-based testing. *Softw. Test. Verif. Reliab.* 15 (2), 73–96.
- Belli, F., Beyazit, M., 2015. Exploiting model morphology for event-based testing. *IEEE Trans. Softw. Eng.* 41 (2), 113–134.
- Bieman, J.M., Ghosh, S., Alexander, R.T., 2001. A technique for mutation of Java objects. In: *Proceedings of the 16th International Conference on Automated Software Engineering (ASE) - Short Paper*. IEEE, San Diego, CA, USA, pp. 337–340.
- Biolchini, J., Mian, P.G., Natali, A.C.C., Travassos, G.H., 2005. Systematic Review in Software Engineering. Tech. Report RT-ES 679/05. Systems Engineering and Computer Science Dept., COPPE/UF RJ, Rio de Janeiro, RJ, Brazil.
- Bluemke, I., Kulesza, K., 2013. Reduction of computational cost in mutation testing by sampling mutants. In: *Proceedings of the 8th International Conference on Dependability and Complex Systems (DepCoS-RELCOMEX)*. Springer, Brunów, Poland, pp. 41–51.
- Bluemke, I., Kulesza, K., 2014a. Reduction in mutation testing of Java classes. In: *Proceedings of the 9th International Conference on Software Engineering and Applications (ICSOE-EA)*. IEEE, Vienna, Austria, pp. 297–304.
- Bluemke, I., Kulesza, K., 2014b. Reductions of operators in Java mutation testing. In: *Proceedings of the 9th International Conference on Dependability and Complex Systems (DepCoS-RELCOMEX)*. Springer, Brunów, Poland, pp. 93–102.
- Bogacki, B., Walter, B., 2006. Aspect-oriented response injection: an alternative to classical mutation testing. In: *Proceedings of the IFIP Working Conference on Software Engineering Techniques: Design for Quality (SET)*. Springer, Warsaw, Poland, pp. 273–282.
- Bogacki, B., Walter, B., 2006. Evaluation of test code quality with aspect-oriented mutations. In: *Proceedings of the 7th International Conference Extreme Programming and Agile Processes in Software Engineering (XP) - Posters and Demonstrations*. Springer, Oulu, Finland, pp. 202–204.
- Budd, T., Angluin, D., 1982. Two notions of correctness and their relation to testing. *Acta Inform.* 8 (1), 31–45.
- Cachia, M.A., Micallef, M., Colombo, C., 2013. Towards incremental mutation testing. In: *Proceedings of the 2013 Validation Strategies for Software Evolution Workshop (VSSE)*. Elsevier, Rome, Italy, pp. 2–11.
- Chen, L., Zhang, L., 2018. Speeding up mutation testing via regression test selection: an extensive study. In: *Proceedings of the 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Västerås, Sweden, pp. 58–69.
- Choi, B., Mathur, A.P., 1993. High-performance mutation testing. *J. Syst. Softw.* 20 (2), 135–152.
- Delamaro, M.E., Deng, L., Durelli, V.H.S., Li, N., Offutt, A.J., 2014. Experimental evaluation of SDL and One-Op mutation for C. In: *Proceedings of the 7th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Cleveland, OH, USA, pp. 203–212.
- Delamaro, M.E., Deng, L., Li, N., Durelli, V.H.S., Offutt, A.J., 2014. Growing a reduced set of mutation operators. In: *Proceedings of the 28th Brazilian Symposium on Software Engineering (SBES)*. IEEE, Maceió, AL, Brazil, pp. 81–90.
- Delamaro, M.E., Offutt, A.J., Ammann, P., 2014. Designing deletion mutation operators. In: *Proceedings of the 7th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Cleveland, OH, USA, pp. 11–20.
- Delgado-Pérez, P., Medina-Bulo, I., Merayo, M.G., 2017. Using evolutionary computation to improve mutation testing. In: *Proceedings of the 14th International Work-Conference on Artificial Neural Networks (IWANN)*. Springer, Cadiz, Spain, pp. 381–391.
- Delgado-Pérez, P., Medina-Bulo, I., Palomo-Lozano, F., García-Domínguez, A., Domínguez-Jiménez, J.J., 2017. Assessment of class mutation operators for C++ with the MuCPP mutation system. *Inf. Softw. Technol.* 169–184.
- Delgado-Pérez, P., Medina-Bulo, I., Segura, S., García-Domínguez, A., Juan, J., 2017. GiGAn: evolutionary mutation testing for C++ object-oriented systems. In: *Proceedings of the 32nd ACM Symposium on Applied Computing (SAC)*. ACM, Marrakech, Morocco, pp. 1387–1392.
- Delgado-Pérez, P., Segura, S., Medina-Bulo, I., 2017. Assessment of C++ object-oriented mutation operators: a selective mutation approach. *Softw. Test. Verif. Reliab.* 27 (4–5), 1630–1649.
- DeMillo, R.A., Krauser, E.W., Mathur, A.P., 1991. Compiler-integrated program mutation. In: *Proceedings of the 15th IEEE Annual Computer Software and Applications Conference (COMPSAC)*. IEEE, Tokyo, Japan, pp. 351–356.
- DeMillo, R.A., Lipton, R.J., Sayward, F.G., 1978. Hints on test data selection: help for the practicing programmer. *IEEE Comput.* 11 (4), 34–43.
- DeMillo, R.A., Offutt, A.J., 1991. Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng.* 17 (9), 900–910.
- DeMillo, R.A., Offutt, A.J., 1993. Experimental results from an automatic test case generator. *ACM Trans. Softw. Eng. Methodol.* 2 (2), 109–127.
- Deng, L., Offutt, A.J., Li, N., 2013. Empirical evaluation of the statement deletion mutation operator. In: *Proceedings of the 6th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Luxembourg City, Luxembourg, pp. 84–93.
- Denisov, A., Pankevich, S., 2018. Mull it over: mutation testing based on LLVM. In: *Proceedings of the 13th International Workshop on Mutation Analysis (Mutation)*. IEEE, Västerås, Sweden, pp. 25–31.
- Derezińska, A., 2013. A quality estimation of mutation clustering in C# programs. In: *Proceedings of the 8th International Conference on Dependability and Complex Systems (DepCoS-RELCOMEX)*. Springer, Brunów, Poland, pp. 119–129 (AISC v.224).
- Derezińska, A., 2016. Evaluation of deletion mutation operators in mutation testing of C# programs. In: *Proceedings of the 11th International Conference on*

- Dependability and Complex Systems (DepCoS-RELCOMEX). Springer, Brunów, Poland, pp. 97–108.
- Derezińska, A., Hałas, K., 2015. Improving mutation testing process of python programs. In: Proceedings of the 4th Computer Science On-line Conference Software Engineering in Intelligent Systems (CSOC). Springer, pp. 233–242.
- Derezińska, A., Rudnik, M., 2012. Quality evaluation of object-oriented and standard mutation operators applied to C# programs. In: Proceedings of the 50th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS). Springer, Prague, Czech Republic, pp. 42–57.
- Derezińska, A., Rudnik, M., 2017. Evaluation of mutant sampling criteria in object-oriented mutation testing. In: Proceedings of the Federated Conference on Computer Science and Information Systems (ACSIS). Polskie Towarzystwo Informatyczne, Prague, Czech Republic, pp. 1315–1324.
- Devroey, X., Perrouin, G., Papadakis, M., Legay, A., Schobbens, P.-Y., Heymans, P., 2016. Featured model-based mutation analysis. In: Proceedings of the 38th International Conference on Software Engineering (ICSE). ACM, Austin, TX, USA, pp. 655–666.
- Devroey, X., Perrouin, G., Papadakis, M., Legay, A., Schobbens, P.Y., Heymans, P., 2017. Automata language equivalence vs. simulations for model-based mutant equivalence: an empirical evaluation. In: Proceedings of the 10th IEEE International Conference on Software Testing, Verification and Validation (ICST). IEEE, Tokyo, Japan, pp. 424–429.
- Do, H., Rothermel, G., 2006. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Trans. Softw. Eng.* 32 (9), 733–752.
- Domínguez-Jiménez, J.J., Estero-Botaro, A., García-Domínguez, A., Medina-Bulo, I., 2009. GAmara: an automatic mutant generation system for WS-BPEL compositions. In: Proceedings of the 7th IEEE European Conference on Web Services (ECOWS). IEEE, Eindhoven, The Netherlands, pp. 97–106.
- Domínguez-Jiménez, J.J., Estero-Botaro, A., García-Domínguez, A., Medina-Bulo, I., 2011. Evolutionary mutation testing. *Inf. Softw. Technol.* 53, 1108–1123.
- Domínguez-Jiménez, J.J., Estero-Botaro, A., Medina-Bulo, I., 2009. A framework for mutant genetic generation for WS-BPEL. In: Proceedings of the 35th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM). IEEE, Spindler Mlyn, Czech Republic, pp. 229–240.
- Durelli, V.H.S., Offutt, A.J., Delamaro, M.E., 2012. Toward harnessing high-level language virtual machines for further speeding up weak mutation testing. In: Proceedings of the 5th International Conference on Software Testing, Verification and Validation (ICST). IEEE, Montreal, QC, Canada, pp. 681–690.
- Fernandes, L., Ribeiro, M., Carvalho, L., Gheyi, R., Mongiovi, M., Santos, A., Cavalcanti, A., Ferrari, F.C., Maldonado, J.C., 2017. Avoiding useless mutants. In: Proceedings of the 16th ACM International Conference on Generative Programming: Concepts and Experiences (GPCE). ACM, Vancouver, BC, Canada, pp. 187–198.
- Ferrari, F.C., Pizzoleto, A.V., Offutt, A.J., 2018. A systematic review of cost reduction techniques for mutation testing: preliminary results. In: Proceedings of the 13th International Workshop on Mutation Analysis (Mutation). IEEE, Västerås, Sweden, pp. 1–10.
- Ferrari, F.C., Pizzoleto, A.V., Offutt, A.J., Fernandes, L., Ribeiro, M., 2018. SLR on Cost Reduction for Mutation Testing – Companion Website. <http://goo.gl/edyF1n>.
- Ferrari, F.C., Rashid, A., Maldonado, J.C., 2013. Towards the practical mutation testing of AspectJ programs. *Sci. Comput. Programm.* 78 (9), 1639–1662.
- Fleyshgakkher, V.N., Weiss, S.N., 1994. Efficient mutation analysis: a new approach. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA). ACM, Seattle, WA, USA, pp. 185–195.
- Frankl, P.G., Weiss, S.N., Hu, C., 1997. All-uses vs mutation testing: an experimental comparison of effectiveness. *J. Syst. Softw.* 38 (3), 235–253.
- Fraser, G., Arcuri, A., 2015. Achieving scalable mutation-based generation of whole test suites. *Empir. Softw. Eng.* 20 (3), 783–812.
- Fraser, G., Zeller, A., 2010. Mutation-driven generation of oracles and unit tests. In: Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA). ACM, Trento, Italy, pp. 147–158.
- Fraser, G., Zeller, A., 2012. Mutation-driven generation of oracles and unit tests. *IEEE Trans. Softw. Eng.* 38 (2), 278–292.
- Gligoric, M., Jagannath, V., Luo, Q., Marinov, D., 2012. Efficient mutation testing of multithreaded code. *Softw. Test. Verif. Reliab.* 23 (5), 375–403.
- Gligoric, M., Jagannath, V., Marinov, D., 2010. MuTmUT: efficient exploration for mutation testing of multithreaded code. In: Proceedings of the 3th International Conference on Software Testing, Verification and Validation (ICST). IEEE, Paris, France, pp. 55–64.
- Gligoric, M., Zhang, L., Pereira, C., Pokam, G., 2013. Selective mutation testing for concurrent code. In: Proceedings of the 22nd International Symposium on Software Testing and Analysis (ISSTA). ACM, Lugano, Switzerland, pp. 224–234.
- Gopinath, R., Ahmed, I., Alipour, M.A., Jensen, C., Groce, A., 2017. Mutation reduction strategies considered harmful. *IEEE Trans. Softw. Reliab.* 66 (3), 854–874.
- Gopinath, R., Alipour, M.A., Ahmed, I., Jensen, C., Groce, A., 2016. On the limits of mutation reduction strategies. In: Proceedings of the 38th International Conference on Software Engineering (ICSE). ACM, Austin, TX, USA, pp. 511–522.
- Gopinath, R., Jensen, C., Groce, A., 2016. Topsy-turvy: a smarter and faster parallelization of mutation analysis. In: Proceedings of the 38th International Conference on Software Engineering Companion (ICSE). ACM, Austin, TX, USA, pp. 740–743.
- Gopinath, R., Mathis, B., Zeller, A., 2018. If you can't kill a supermutant, you have a problem. In: Proceedings of the 13th International Workshop on Mutation Analysis (Mutation). IEEE, Västerås, Sweden, pp. 18–24.
- Hamlet, R.G., 1977. Testing programs with the aid of a compiler. *IEEE Trans. Softw. Eng.* 3 (4), 279–290.
- Harman, M., Hierons, R.M., Danicic, S., 2000. The relationship between program dependence and mutation analysis. In: Proceedings of the Mutation 2000 Symposium. Kluwer Academic Publishers, San Jose, CA, USA, pp. 5–13.
- Harman, M., Jia, Y., Reales, P., Polo, M., 2014. Angels and monsters: an empirical investigation of potential test effectiveness and efficiency improvement from strongly subsuming higher order mutation. In: Proceedings of the 29th International Conference on Automated Software Engineering (ASE). ACM, Vasteras, Sweden, pp. 397–408.
- Henard, C., Papadakis, M., Le Traon, Y., 2014. Mutation-based generation of software product line test configurations. In: Proceedings of the 6th International Symposium Search-Based Software Engineering (SSBSE). Springer, Fortaleza, CE, Brazil, pp. 92–106.
- Hernandes, E.C.M., Zamboni, A.B., Fabbri, S.C.P.F., Di Thommazo, A., 2012. Using GQM and TAM to evaluate StArt – a tool that supports systematic review. *CLEI Electron. J.* 15 (1), 13–25.
- Hierons, R.M., Harman, M., Danicic, S., 1999. Using program slicing to assist in the detection of equivalent mutants. *Softw. Test. Verif. Reliab.* 9 (4), 233–262.
- Holling, D., Banescu, S., Probst, M., Petrovska, A., Pretschner, A., 2016. Nequivack: assessing mutation score confidence. In: Proceedings of the 11th International Workshop on Mutation Analysis (Mutation). IEEE, Chicago, IL, USA, pp. 152–161.
- Howden, W.E., 1982. Weak mutation testing and completeness of test sets. *IEEE Trans. Softw. Eng.* 8 (4), 371–379.
- Hu, J., Li, N., Offutt, A.J., 2011. An analysis of OO mutation operators. In: Proceedings of the 6th International Workshop on Mutation Analysis (Mutation). IEEE, Berlin, Germany, pp. 334–341.
- Iida, C., Takada, S., 2017. Reducing mutants with mutant killable precondition. In: Proceedings of the 12th International Workshop on Mutation Analysis (Mutation). IEEE, Tokyo, Japan, pp. 128–133.
- Inozemtseva, L., Hemmati, H., Holmes, R., 2013. Using fault history to improve mutation reduction. In: Proceedings of the 9th Joint Meeting on Foundations of Software Engineering: New Ideas Track (ESEC/FSE). ACM, Saint Petersburg, Russia, pp. 639–642.
- Jackson, D., Woodward, M.R., 2000. Parallel firm mutation of Java programs. In: Proceedings of the Mutation 2000 Symposium. Kluwer Academic Publishers, San Jose, CA, USA, pp. 55–61.
- Ji, C., Chen, Z., Xu, B., Zhao, Z., 2009. A novel method of mutation clustering based on domain analysis. In: Proceedings of the 21th International Conference on Software Engineering and Knowledge Engineering (SEKE). Knowledge Systems Institute Graduate School, Boston, MA, USA, pp. 1–6.
- Jia, Y., Harman, M., 2011. An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.* 37 (5), 649–678.
- Just, R., 2014. The major mutation framework: efficient and scalable mutation analysis for Java. In: Proceedings of the 23rd International Symposium on Software Testing and Analysis (ISSTA). ACM, San Jose, CA, USA, pp. 433–436.
- Just, R., Ernst, M.D., Fraser, G., 2014. Efficient mutation analysis by propagating and partitioning infected execution states. In: Proceedings of the 23rd International Symposium on Software Testing and Analysis (ISSTA). ACM, San Jose, CA, USA, pp. 315–326.
- Just, R., Jalali, D., Inozemtseva, L., Ernst, M.D., Holmes, R., Fraser, G., 2014. Are mutants a valid substitute for real faults in software testing? In: Proceedings of the 22nd International Symposium on Foundations of Software Engineering (FSE). ACM, Hong Kong, China, pp. 654–665.
- Just, R., Kapfhammer, G.M., Schweiggert, F., 2011. Using conditional mutation to increase the efficiency of mutation analysis. In: Proceedings of the 6th International Workshop on Automation of Software Test (AST). ACM, Waikiki, Honolulu, HI, USA, pp. 50–56.
- Just, R., Kapfhammer, G.M., Schweiggert, F., 2012. Do redundant mutants affect the effectiveness and efficiency of mutation analysis? In: Proceedings of the 5th IEEE International Conference on Software Testing, Verification and Validation (ICST). IEEE, Montreal, QC, Canada, pp. 720–725.
- Just, R., Kapfhammer, G.M., Schweiggert, F., 2012. Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis. In: Proceedings of the IEEE 23th International Symposium on Software Reliability Engineering (ISSRE). IEEE, Dallas, TX, USA, pp. 11–20.
- Just, R., Kurtz, B., Ammann, P., 2017. Inferring mutant utility from program context. In: Proceedings of the 26th International Symposium on Software Testing and Analysis (ISSTA). ACM, Santa Barbara, CA, USA, pp. 284–294.
- Just, R., Schweiggert, F., 2015. Higher accuracy and lower run time: efficient mutation analysis using non-redundant mutation operators. *Softw. Test. Verif. Reliab.* 25 (5–7), 490–507.
- Kaminski, G., Ammann, P., 2009. Using a fault hierarchy to improve the efficiency of DNF logic mutation testing. In: Proceedings of the 2nd International Conference on Software Testing, Verification and Validation (ICST). IEEE, Denver, CO, USA, pp. 386–395.
- Kaminski, G., Ammann, P., 2011. Reducing logic test set size while preserving fault detection. *Softw. Test. Verif. Reliab.* 21 (3), 155–193.
- Kaminski, G., Ammann, P., Offutt, A.J., 2011. Better predicate testing. In: Proceedings of the 6th International Workshop on Automation of Software Test (AST). ACM, Honolulu, HI, USA, pp. 57–63.
- Kaminski, G., Ammann, P., Offutt, A.J., 2013. Improving logic-based testing. *J. Syst. Softw.* 86 (8), 2002–2012.
- Kaminski, G., Praphamontipong, U., Ammann, P., Offutt, A.J., 2011. A logic mutation approach to selective mutation for programs and queries. *Inf. Softw. Technol.* 53 (10), 1137–1152.

- Kim, S.-W., Ma, Y.-S., Kwon, Y.-R., 2012. Combining weak and strong mutation for a noninterpretive Java mutation system. *Softw. Test. Verif. Reliab.* 23 (8), 647–668.
- Kintis, M., Malevris, N., 2013. Identifying more equivalent mutants via code similarity. In: *Proceedings of the 20th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, Bangkok, Thailand, pp. 180–188.
- Kintis, M., Malevris, N., 2014. Using data flow patterns for equivalent mutant detection. In: *Proceedings of the 9th International Workshop on Mutation Analysis (Mutation)*. IEEE, Cleveland, OH, USA, pp. 196–205.
- Kintis, M., Malevris, N., 2015. MEDIC: a static analysis framework for equivalent mutant identification. *Inf. Softw. Technol.* 68, 1–17.
- Kintis, M., Papadakis, M., Jia, Y., Malevris, N., Le Traon, Y., Harman, M., 2017. Detecting trivial mutant equivalences via compiler optimisations. *IEEE Trans. Softw. Eng.* 44 (4), 1–25.
- Kintis, M., Papadakis, M., Malevris, N., 2010. Evaluating mutation testing alternatives: a collateral experiment. In: *Proceedings of the 17th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, Sydney, Australia, pp. 300–309.
- Kintis, M., Papadakis, M., Malevris, N., 2012. Isolating first order equivalent mutants via second order mutation. In: *Proceedings of the 5th IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Montreal, QC, Canada, pp. 701–710.
- Kintis, M., Papadakis, M., Malevris, N., 2015. Employing second-order mutation for isolating first-order equivalent mutants. *Softw. Test. Verif. Reliab.* 25 (5–7), 508–535.
- Kitchenham, B., 2004. Procedures for Performing Systematic Reviews. Joint Technical Report TR/SE-0401 (Keele) - 0400011T1 (NICTA). Software Engineering Group - Department of Computer Science - Keele University; and Empirical Software Engineering - National ICT Australia Ltd, Staffordshire, UK; and Eversleigh, Australia.
- Kitchenham, B.A., Dybb, T., Jørgensen, M., 2004. Evidence-based software engineering. In: *Proceedings of the 26th International Conference on Software Engineering (ICSE)*. IEEE, Edinburgh, Scotland, pp. 273–281.
- Krauser, E.W., Mathur, A.P., Rego, V.J., 1988. High performance testing on SIMD machines. In: *Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis*. IEEE, Banff, AB, Canada, pp. 171–177.
- Krauser, E.W., Mathur, A.P., Rego, V.J., 1991. High performance software testing on SIMD machines. *IEEE Trans. Softw. Eng.* 17 (5), 403–423.
- Kurtz, B., Ammann, P., Offutt, A.J., Delamaro, M.E., Kurtz, M., Gökçe, N., 2016. Analyzing the validity of selective mutation with dominator mutants. In: *Proceedings of the 24th International Symposium on Foundations of Software Engineering (FSE)*. ACM, Seattle, WA, USA, pp. 571–582.
- Lacerda, J.T.S., Ferrari, F.C., 2014. Towards the establishment of a sufficient set of mutation operators for AspectJ programs. In: *Proceedings of the 8th Brazilian Workshop on Systematic and Automated Software Testing (SAST)*. Brazilian Computer Society, Maceio, AL, Brazil, pp. 21–30.
- Li, N., Praphamontipong, U., Offutt, A.J., 2009. An experimental comparison of four unit test criteria: mutation, edge-pair, all-uses and prime path coverage. In: *Proceedings of the 4th International Workshop on Mutation Analysis (Mutation)*. IEEE, Denver, CO, USA, pp. 220–229.
- Li, N., West, M., Escalona, A., Durelli, V.H.S., 2015. Mutation testing in practice using ruby. In: *Proceedings of the 10th International Workshop on Mutation Analysis (Mutation)*. IEEE, Graz, Austria, pp. 1–6.
- Lima, J.A.P., Guizzo, G., Vergilio, S.R., Silva, A.P.C., Filho, H.L.J., Ehrenfried, H.V., 2016. Evaluating different strategies for reduction of mutation testing costs. In: *Proceedings of the 1st Brazilian Symposium on Systematic and Automated Software Testing (SAST)*. ACM, Maringa, PR, Brazil, pp. 4:1–4:10.
- Liu, M.-H., Gao, Y.-F., Shan, J.-H., Liu, J.-H., Zhang, L., Sun, J.-S., 2006. An approach to test data generation for killing multiple mutants. In: *Proceedings of the 22th International Conference on Software Maintenance (ICSM)*. IEEE, Philadelphia, PA, USA, pp. 113–122.
- Ma, Y.-S., Kim, S.-W., 2016. Mutation testing cost reduction by clustering overlapped mutants. *J. Syst. Softw.* 115 (C), 18–30.
- Ma, Y.-S., Offutt, A.J., Kwon, Y.R., 2005. MuJava: an automated class mutation system. *Softw. Test. Verif. Reliab.* 15 (2), 97–133.
- MacDonell, S., Shepperd, M., Kitchenham, B., Mendes, E., 2010. How reliable are systematic reviews in empirical software engineering? *IEEE Trans. Softw. Eng.* 36 (5), 676–687.
- Madeyski, L., Orzeszyna, W., Torkar, R., Józala, M., 2014. Overcoming the equivalent mutant problem: a systematic literature review and a comparative experiment of second order mutation. *IEEE Trans. Softw. Eng.* 40 (1), 23–42.
- Marcozzi, M., Bardin, S., Kosmatov, N., Papadakis, M., Prevosto, V., Correnson, L., 2018. Time to clean your test objectives. In: *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. IEEE, Gothenburg, Sweden, pp. 456–467.
- Marshall, A.C., Hedley, D., Riddell, I.J., Hennell, M.A., 1990. Static dataflow-aided weak mutation analysis (SDAWM). *Inf. Softw. Technol.* 32 (1), 99–104.
- Mathur, A.P., 1991. Performance, effectiveness, and reliability issues in software testing. In: *Proceedings of the 15th IEEE Annual Computer Software and Applications Conference (COMPSAC)*. IEEE, Tokyo, Japan, pp. 604–605.
- Mathur, A.P., 2007. *Foundations of Software Testing*, 1st Addison-Wesley Professional, Toronto, ON, Canada.
- Mathur, A.P., Wong, W.E., 1993. Evaluation of the cost of alternative mutation testing strategies. In: *Proceedings of the 7th Brazilian Symposium on Software Engineering (SBES)*. Brazilian Computer Society, João Pessoa, PB, Brazil, pp. 320–335.
- Mathur, A.P., Wong, W.E., 1994. An empirical comparison of data flow and mutation based test adequacy criteria. *Softw. Test. Verif. Reliab.* 4 (1), 9–31.
- Matnei Filho, R.A., Vergilio, S.R., 2015. A mutation and multi-objective test data generation approach for feature testing of software product lines. In: *Proceedings of the 29th Brazilian Symposium on Software Engineering (SBES)*. IEEE, Belo Horizonte, MG, Brazil, pp. 21–30.
- McMinn, P., Kapfhammer, G.M., Wright, C.J., 2016. Virtual mutation analysis of relational database schemas. In: *Proceedings of the 11th International Workshop on Automation of Software Test (AST)*. ACM, Austin, TX, USA, pp. 36–42.
- McMinn, P., Wright, C.J., McCurdy, C.J., Kapfhammer, G., 2018. Automatic detection and removal of ineffective mutants for the mutation analysis of relational database schemas (in press). *IEEE Trans. Softw. Eng.* 1–1.
- Mresa, E.S., Bottaci, L., 1999. Efficiency of mutation operators and selective mutation strategies: an empirical study. *Softw. Test. Verif. Reliab.* 9 (4), 205–232.
- Nobre, T., Vergilio, S.R., Pozo, A., 2012. Reducing interface mutation costs with multi-objective optimization algorithms. *Int. J. Nat. Comput. Res.* 21–40.
- Offutt, A.J., Craft, W.M., 1994. Using compiler optimization techniques to detect equivalent mutants. *Softw. Test. Verif. Reliab.* 4 (3), 131–154.
- Offutt, A.J., Jin, Z., Pan, J., 1999. The dynamic domain reduction approach to test data generation. *Softw. Pract. Exp.* 29 (2), 167–193.
- Offutt, A.J., Lee, A., Rothermel, G., Untch, R.H., Zapf, C., 1996. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.* 5 (2), 99–118.
- Offutt, A.J., Lee, S.D., 1991. How strong is weak mutation? In: *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. ACM, Victoria, BC, Canada, pp. 200–213.
- Offutt, A.J., Lee, S.D., 1994. An empirical evaluation of weak mutation. *IEEE Trans. Softw. Eng.* 20 (5), 337–344.
- Offutt, A.J., Ma, Y.-S., Kwon, Y.-R., 2006. The class-level mutants of MuJava. In: *Proceedings of the International Workshop on Automation of Software Test (AST)*. ACM, Shanghai, China, pp. 78–84.
- Offutt, A.J., Pan, J., 1996. Detecting equivalent mutants and the feasible path problem. In: *Proceedings of the 11th Annual Conference on Computer Assurance (COMPASS)*. ACM, Gaithersburg, MD, USA, pp. 224–236.
- Offutt, A.J., Pan, J., 1997. Automatically detecting equivalent mutants and infeasible paths. *Softw. Test. Verif. Reliab.* 7 (3), 165–192.
- Offutt, A.J., Pargas, R.V., Fichter, S.V., Khambekar, P.K., 1992. Mutation testing of software using a MIMD computer. In: *Proceedings of the International Conference on Parallel Processing (ICPP)*. John Wiley & Sons, Chicago, Illinois, USA.
- Offutt, A.J., Rothermel, G., Zapf, C., 1993. An experimental evaluation of selective mutation. In: *Proceedings of the 15th International Conference on Software Engineering (ICSE)*. IEEE, Baltimore, MD, USA, pp. 100–107.
- Offutt, A.J., Untch, R.H., 2000. *Mutation 2000: uniting the orthogonal*. In: *Proceedings of the Mutation 2000 Symposium*. Kluwer Academic Publishers, San Jose, CA, USA, pp. 34–44.
- Oliveira, A.A.L., Camilo-Junior, C.G., Vincenzi, A.M.R., 2013. A coevolutionary algorithm to automatic test case selection and mutant in mutation testing. In: *Proceedings of the 2013 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, Cancun, Mexico, pp. 829–836.
- Omar, E., Ghosh, S., 2012. An exploratory study of higher order mutation testing in aspect-oriented programming. In: *Proceedings of the 23th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, Dallas, TX, USA, pp. 1–10.
- Papadakis, M., Delamaro, M., Le Traon, Y., 2014. Mitigating the effects of equivalent mutants with mutant classification strategies. *Sci. Comput. Program.* 298–319.
- Papadakis, M., Jia, Y., Harman, M., Le Traon, Y., 2015. Trivial compiler equivalence: a large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In: *Proceedings of the 37th International Conference on Software Engineering (ICSE)*. ACM, Florence, Italy, pp. 936–946.
- Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Le Traon, Y., Harman, M., 2019. Mutation testing advances: an analysis and survey. In: *Memor. A.M. (Ed.), Advances in Computers*, 112. Elsevier, Amsterdam, The Netherlands, pp. 275–378.
- Papadakis, M., Le Traon, Y., 2013. Mutation testing strategies using mutant classification. In: *Proceedings of the 28th ACM Symposium on Applied Computing (SAC)*. ACM, Coimbra, Portugal, pp. 1223–1229.
- Papadakis, M., Malevris, N., 2009. An effective path selection strategy for mutation testing. In: *Proceedings of the 16th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, Batu Ferringhi, Penang, Malaysia, pp. 422–429.
- Papadakis, M., Malevris, N., 2010. An empirical evaluation of the first and second order mutation testing strategies. In: *Proceedings of the 5th International Workshop on Mutation Analysis (Mutation)*. IEEE, Paris, France, pp. 90–99.
- Papadakis, M., Malevris, N., 2010. Automatic mutation test case generation via dynamic symbolic execution. In: *Proceedings of the IEEE 21th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, San Jose, CA, USA, pp. 121–130.
- Papadakis, M., Malevris, N., 2011. Automatic mutation based test data generation. In: *Proceedings of the 13th Genetic and Evolutionary Computation Conference (GECCO): Search-Based Software Engineering Track (Poster Session)*. ACM, Dublin, Ireland, pp. 247–248.
- Papadakis, M., Malevris, N., 2011. Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing. *Softw. Qual. J.* 19 (4), 691–723.
- Papadakis, M., Malevris, N., 2012. Mutation based test case generation via a path selection strategy. *Inf. Softw. Technol.* 54 (9), 915–932.
- Papadakis, M., Malevris, N., Kallia, M., 2010. Towards automating the generation of mutation tests. In: *Proceedings of the 5th Workshop on Automation of Software Test (AST)*. ACM, Cape Town, South Africa, pp. 111–118.
- Parsai, A., Murguia, A., Demeyer, S., 2016. Evaluating random mutant selection at class-level in projects with non-adequate test suites. In: *Proceedings of the 20th*

- International Conference on Evaluation and Assessment in Software Engineering (EASE). ACM, Limerick, Ireland, pp. 1–10.
- Patel, K., Hierons, R.M., 2016. Resolving the equivalent mutant problem in the presence of non-determinism and coincidental correctness. In: Proceedings of the 28th International Conference on Testing Software and Systems (ICTSS). Springer, Graz, Austria, pp. 123–138 (LNCS v.9976).
- Patrick, M., Oriol, M., Clark, J.A., 2012. MESSI: mutant evaluation by static semantic interpretation. In: Proceedings of the 5th International Conference on Software Testing, Verification and Validation (ICST). IEEE, Montreal, QC, Canada, pp. 711–719.
- Petrović, G., Ivanković, M., 2018. State of mutation testing at Google. In: Proceedings of the 40th International Conference on Software Engineering - Software Engineering in Practice Track (ICSE-SEIP) (to appear). IEEE, Gothenburg, Sweden, pp. 163–171.
- Petrović, G., Ivanković, M., Kurtz, B., Ammann, P., Just, R., 2018. An industrial application of mutation testing: lessons, challenges, and research directions. In: Proceedings of the 13th International Workshop on Mutation Analysis (Mutation). IEEE, Västerås, Sweden, pp. 47–53.
- Polo, M., Piattini, M., García-Rodríguez, I., 2009. Decreasing the cost of mutation testing with second-order mutants. *Softw. Test. Verif. Reliab.* 19 (2), 111–131.
- Polo, M., Reales, P., 2010. Mutation testing cost reduction techniques: a survey. *IEEE Softw.* 27 (3), 80–86.
- Praphamontpong, U., Offutt, A.J., 2017. Finding redundancy in web mutation operators. In: Proceedings of the 12th International Workshop on Mutation Analysis (Mutation). IEEE, Tokyo, Japan, pp. 134–142.
- Quyen, N.T.H., Tung, K.T., Hanh, L.T.M., Binh, N.T., 2016. Improving mutant generation for simulink models using genetic algorithm. In: Proceedings of the International Conference on Electronics, Information, and Communications (ICEIC). IEEE, Da Nang, Vietnam, pp. 1–4.
- Reales, P., Polo, M., 2012. Mutant execution cost reduction: through MUSIC (Mutant schema improved with extra code). In: Proceedings of the 5th International Conference on Software Testing, Verification and Validation (ICST). IEEE, Montreal, QC, Canada, pp. 664–672.
- Reales, P., Polo, M., 2013. Parallel mutation testing. *Softw. Test. Verif. Reliab.* 23 (4), 315–350.
- Reales, P., Polo, M., 2014. Reducing mutation costs through uncovered mutants. *Softw. Test. Verif. Reliab.* 25 (5–7), 464–489.
- Reales, P., Polo, M., Alemán, J.L.F., 2013. Validating second-order mutation at system level. *IEEE Trans. Softw. Eng.* 39 (4), 570–587.
- Reuling, D., Bürdek, J., Rotärmel, S., Lochau, M., Kelter, U., 2015. Fault-based product-line testing: effective sample generation based on feature-diagram mutation. In: Proceedings of the 19th International Conference on Software Product Line (SPLC). ACM, Nashville, TN, USA, pp. 131–140.
- Sahingözü, M., Spafford, E.H., 1990. A Bayes sequential statistical procedure for approving software products. In: Proceedings of the IFIP Conference on Approving Software Products (ASP). Elsevier, Banff, AB, Canada, pp. 43–56.
- Schuler, D., Dallmeier, V., Zeller, A., 2009. Efficient mutation testing by checking invariant violations. In: Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA). ACM, Chicago, IL, USA, pp. 69–80.
- Schuler, D., Zeller, A., 2010. (Un-)Covering equivalent mutants. In: Proceedings of the 3th International Conference on Software Testing, Verification and Validation (ICST). IEEE, Paris, France, pp. 45–54.
- Schuler, D., Zeller, A., 2013. Covering and uncovering equivalent mutants. *Softw. Test. Verif. Reliab.* 23 (5), 353–374.
- Siami-Namin, A., Andrews, J.H., 2006. Finding sufficient mutation operators via variable reduction. In: Proceedings of the 2nd Workshop on Mutation Analysis (Mutation). IEEE, Raleigh, NC, USA, pp. 1–10.
- Siami-Namin, A., Andrews, J.H., Murdoch, D.J., 2008. Sufficient mutation operators for measuring test effectiveness. In: Proceedings of the 30th International Conference on Software Engineering (ICSE). ACM, Leipzig, Germany, pp. 351–360.
- Silva, R.A., Souza, S.R.S., Souza, P.S.L., 2017. A systematic review on search based mutation testing. *Inf. Softw. Technol.* 81, 19–35.
- Sridharan, M., Siami-Namin, A., 2010. Prioritizing mutation operators based on importance sampling. In: Proceedings of the IEEE 21th International Symposium on Software Reliability Engineering (ISSRE). IEEE, San Jose, CA, USA, pp. 378–387.
- Steimann, F., Thies, A., 2010. From behaviour preservation to behaviour modification: constraint-based mutant generation. In: Proceedings of the 32th International Conference on Software Engineering (ICSE). ACM, Cape Town, South Africa, pp. 425–434.
- Sun, C., Pan, L., Wang, Q., Liu, H., Zhang, X., 2017. An empirical study on mutation testing of WS-BPEL programs. *Comput. J.* 60 (1), 143–158.
- Sun, C., Xue, F., Liu, H., Zhang, X., 2017. A path-aware approach to mutant reduction in mutation testing. *Inf. Softw. Technol.* 81, 65–81.
- Tuya, J., Suárez-Cabal, M.J., de la Riva, C., 2007. Mutating database queries. *Inf. Softw. Technol.* 49 (4), 398–417.
- Untch, R.H., 1992. Mutation-based software testing using program schemata. In: Proceedings of the 30th Annual Southeast Regional Conference (ACM-SE). ACM, Raleigh, NC, USA, pp. 285–291.
- Untch, R.H., 2009. On reduced neighborhood mutation analysis using a single mutagenic operator. In: Proceedings of the 47th Annual Southeast Regional Conference (ACM-SE). ACM, Clemson, SC, USA, pp. 71–75.
- Untch, R.H., Harrold, M.J., Offutt, A.J., 1997. TUMS: testing using mutant schemata. In: Proceedings of the 35th Annual Southeast Regional Conference (ACM-SE). ACM, Murfreesboro, TN, USA, pp. 174–181.
- Untch, R.H., Offutt, A.J., Harrold, M.J., 1993. Mutation analysis using mutant schemata. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA). ACM, Cambridge, MA, USA, pp. 139–148.
- Vincenzi, A.M.R., Maldonado, J.C., Barbosa, E.F., Delamaro, M.E., 2001. Unit and integration testing strategies for C programs using mutation. *Softw. Test. Verif. Reliab.* 11 (4), 249–268.
- Vincenzi, A.M.R., Nakagawa, E.Y., Maldonado, J.C., Delamaro, M.E., Romero, R.A.F., 2002. Bayesian-learning based guidelines to determine equivalent mutants. *Int. J. Softw. Eng. Knowl. Eng.* 12 (6), 675–689.
- Wang, B., Xiong, Y., Shi, Y., Zhang, L., Hao, D., 2017. Faster mutation analysis via equivalence modulo states. In: Proceedings of the 26th International Symposium on Software Testing and Analysis (ISSTA). ACM, Santa Barbara, CA, USA, pp. 295–306.
- Wedyan, F., Ghosh, S., 2012. On generating mutants for AspectJ programs. *Inf. Softw. Technol.* 900–914.
- Weiss, S.N., Fleishgaker, V.N., 1993. Improved serial algorithms for mutation analysis. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA). ACM, Cambridge, MA, USA, pp. 149–158.
- Wohlin, C., 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE). ACM, London, UK, pp. 1–10.
- Wong, W.E., Maldonado, J.C., Delamaro, M.E., Mathur, A.P., 1994. Constrained mutation in C programs. In: Proceedings of the 8th Brazilian Symposium on Software Engineering (SBES). Brazilian Computer Society, pp. 439–452.
- Wong, W.E., Mathur, A.P., 1995. Reducing the cost of mutation testing: an empirical study. *J. Syst. Softw.* 31 (3), 185–196.
- Wong, W.E., Mathur, A.P., Maldonado, J.C., 1995. Mutation versus all-uses: an empirical evaluation of cost, strength and effectiveness. *Softw. Qual. Product.* 258–265.
- Woodward, M.R., Halewood, K., 1988. From weak to strong, dead or alive? An analysis of some mutation testing issues. In: Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis. IEEE, Banff, AB, Canada, pp. 152–158.
- Wright, C.J., Kapfhammer, G.M., McMinn, P., 2013. Efficient mutation analysis of relational database structure using mutant schemata and parallelisation. In: Proceedings of the 8th International Workshop on Mutation Analysis (Mutation). IEEE, Luxembourg City, Luxembourg, pp. 63–72.
- Wright, C.J., Kapfhammer, G.M., McMinn, P., 2014. The impact of equivalent, redundant and quasi mutants on database schema mutation analysis. In: Proceedings of the 14th International Conference on Quality Software (QSIQ). IEEE, Dallas, TX, USA, pp. 57–66.
- Zhang, J., Wang, Z., Zhang, L., Hao, D., Zang, L., Cheng, S., Zhang, L., 2016. Predictive mutation testing. In: Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA). ACM, Saarbrücken, Germany, pp. 342–353.
- Zhang, L., Gligoric, M., Marinov, D., Khurshid, S., 2013. Operator-based and random mutant selection: better together. In: Proceedings of the 28th International Conference on Automated Software Engineering (ASE). IEEE, Palo Alto, CA, USA, pp. 92–102.
- Zhang, L., Hou, S., Hu, J., Xie, T., Mei, H., 2010. Is operator-based mutant selection superior to random mutant selection? In: Proceedings of the 32th International Conference on Software Engineering (ICSE). ACM, Cape Town, South Africa, pp. 435–444.
- Zhang, L., Marinov, D., Khurshid, S., 2013. Faster mutation testing inspired by test prioritization and reduction. In: Proceedings of the 22nd International Symposium on Software Testing and Analysis (ISSTA). ACM, Lugano, Switzerland, pp. 235–245.
- Zhang, L., Marinov, D., Zhang, L., Khurshid, S., 2012. Regression mutation testing. In: Proceedings of the 21st International Symposium on Software Testing and Analysis (ISSTA). ACM, Minneapolis, MN, USA, pp. 331–341.
- Zhang, L., Xie, T., Zhang, L., Tillmann, N., de Halleux, J., Mei, H., 2010. Test generation via dynamic symbolic execution for mutation testing. In: Proceedings of the 26th International Conference on Software Maintenance (ICSM). IEEE, Timisoara, Romania, pp. 1–10.
- Zhu, Q., Panichella, A., Zaidman, A., 2017. Speeding-up mutation testing via data compression and state infection. In: Proceedings of the 12th International Workshop on Mutation Analysis (Mutation). IEEE, Tokyo, Japan, pp. 103–109.
- Zhu, Q., Panichella, A., Zaidman, A., 2018. An investigation of compression techniques to speed up mutation testing. In: Proceedings of the 11th International Conference on Software Testing, Verification and Validation (ICST). IEEE, Västerås, Sweden, pp. 274–284.

MSc. Alessandro Viola Pizzoleto is a PhD candidate at the Computing Department of the Universidade Federal de São Carlos (UFSCar), Brazil. He received a Master's degree in computer science from the Universidade Estadual Paulista (UNESP), Brazil. His main research interests are related to software testing and experimental software engineering.

Dr. Fabiano Cutigi Ferrari is an associate professor at the Computing Department of the Universidade Federal de São Carlos (UFSCar), Brazil. He received a PhD degree in computer science from the Universidade de São Paulo (USP). His PhD has been supported by FAPESP, and he was a visiting student at Lancaster University/UK supported by CAPES, AOSD-Europe Project (2007–2008), and DiVA Project (2010). He has also been a visiting researcher at George Mason University (2017–2018). His main research interests are related to software testing, experimental

software engineering, adaptive systems, and software metrics. He is a member of the ACM. A list of his main publications can be found at <http://lattes.cnpq.br/3154345471250570>.

Dr. Jeff Offutt is a Professor of Software Engineering at George Mason University. He has published over 185 refereed research papers (h-index of 64), and invented numerous widely used test techniques. Offutt is editor-in-chief of Wiley's journal of Software Testing, Verification and Reliability, co-founded the IEEE International Conference on Software Testing (ICST), and co-authored Introduction to Software Testing. He was awarded the Outstanding Faculty Award from the State Council of Higher Education for Virginia in 2019, GMU's Teaching Excellence Award, Teaching With Technology, in 2013, and his 2014 software engineering education paper was chosen by ACM as a notable paper. Current projects include the CS4all educational project, the SPARC educational project, muJava, Testing of Critical System Characteristics (TOCSYC) and PILOT projects at University of Skovde, automatic repair of SQL queries, testing mobile and web applications, test automation, and usable se-

curity. Offutt received the PhD in Computer Science from the Georgia Institute of Technology in 1988. He is on the web at <https://cs.gmu.edu/~offutt/>.

Leo Fernandes is a Ph.D. candidate at the Informatics Center of the Universidade Federal de Pernambuco (CIn/UFPE), Brazil. He received a Master's degree in computer science from the Universidade Federal de Pernambuco (UFPE). His main research interests are related to software testing, functional programming, parallel programming and software architecture.

Dr. Márcio Ribeiro is an assistant professor at the Computing Institute of the Universidade Federal de Alagoas (UFAL), Brazil. He received a Ph.D. degree in computer science from the Universidade Federal de Pernambuco (UFPE). His main research interests are related to software product lines and families, aspect-oriented and object-oriented programming, experimental software engineering, refactoring, software modularity and software static analysis. A list of his main publications can be found at <http://lattes.cnpq.br/9300936571715992>.

APÊNDICE C

SiMut: Exploring Program Similarity to Support the Cost Reduction of Mutation Testing

Neste apêndice apresenta-se o conteúdo completo do artigo publicado no *15th International Workshop on Mutation Analysis (Mutation'20)* (PIZZOLETO et al., 2020). Um resumo deste artigo é apresentado no Capítulo 4 desta tese.

IEEE COPYTIGHT NOTICE:¹ “©IEEE, 2020. Reprinted, with permission, from the *Proceedings of the 15th International Workshop on Mutation Analysis (Mutation'20)*, <<https://doi.org/10.1109/ICSTW50294.2020.00052>>”

¹ <<https://journals.ieeeauthorcenter.ieee.org/choose-a-publishing-agreement/avoid-infringement-upon-ieee-copyright/>> - acessado em novembro/2022.

SiMut: Exploring Program Similarity to Support the Cost Reduction of Mutation Testing

Alessandro V. Pizzoleto, Fabiano C. Ferrari
Computing Department
Federal University of São Carlos
São Carlos, SP, Brazil
{alessandro.pizzoleto, fcferrari}@ufscar.br

Lucas D. Dallilo
Department of Computer Systems
University of São Paulo
São Carlos, SP, Brazil
lucasdallilo@usp.br

Jeff Offutt
Department of Computer Science
George Mason University
Fairfax, VA, USA
offutt@gmu.edu

Abstract—Scientists have created many cost reduction techniques for mutation testing, and most of them reduce cost with minor losses of effectiveness. However, many of these techniques are difficult to generalize, difficult to scale, or both. Published results are usually limited to a modest collection of programs. Therefore, an open question is whether the results of a given cost reduction technique on programs studied in the paper will hold true for other programs. This paper introduces a conceptual framework, named *SiMut*, to support the cost reduction of mutation testing based on historical data and program similarity. Given a new, untested program u , the central idea is applying to u the same cost reduction strategy applied to a group G of programs that are similar to u and have already been tested with mutation, and check for consistency of results in terms of reduced costs and quality of test sets. *SiMut* includes activities to compute program abstractions and similarity. Based on this information, it supports the application of mutation cost reduction techniques to both G and u . This paper presents the concepts behind *SiMut*, a proof-of-concept implementation of *SiMut*, and results from a pilot study. Finally, we discuss some issues related to the use of *SiMut*, focusing on the composition of a representative dataset to properly explore the potential of our framework.

Index Terms—mutation testing; cost reduction; program similarity; reuse of historical data

I. INTRODUCTION

Mutation testing [14] (or simply *mutation*) helps testers create high quality test sets. It takes a program p and systematically generates slightly modified versions of p , called *mutants*. *Mutation operators* $O = \{o_1, o_2, \dots, o_n\}$ are used to create mutants. Ideally, each mutant is a faulty program and should behave differently from p for at least one test case. When such a test is found, the mutant is said to be *killed* by the test case. Until then, the mutant remains *alive* and either a test case must be found or designed to kill the mutant, or the mutant should be determined to be equivalent. An *equivalent mutant* behaves the same as p on all tests, so cannot be killed. The *mutation score* (MS) is a value in the interval $[0, 1]$ that represents the quality of the test set with respect to a given group of mutants. The closer MS is to 1, the better the test set is. The MS formula $MS = K/(M-E)$ uses the dead (K), equivalent (E), and the total number of mutants (M) generated. We also consider mutants generated by a single operator o_i . A set of tests that kills all o_i mutants is said to be *o_i -adequate*, and the mutation score for o_i is the score of the *o_i -adequate* tests on all mutants created for the program.

Empirical studies have found that mutation yields tests that detect more faults than tests derived with most other structural criteria [7, 18, 27, 30, 31], and that can reveal real faults [3, 13, 22]. Mutation is also very often used as a test quality assessment instrument to evaluate tests derived using other testing techniques and criteria. For example, as recently highlighted in a literature review [40] that selected 502 mutation-related studies, all published between 2008 and 2017, 217 used mutants only for test assessment purposes.

Although effective, mutation is also expensive [21, 24, 40, 42]. Two major factors affecting the cost are (i) the large number of mutants (even for small programs), and (ii) the manual analysis required for unkillable mutants. Since detecting equivalent mutants is generally undecidable [5], equivalence detection can only be partially automated [29, 35, 39]. These factors have limited the adoption of mutation in industry, leading to almost four decades of research into reducing the cost of mutation, as reported in recent literature surveys [40, 42].

Experimental results indicate that many mutation cost reduction techniques reduce costs with minor losses of effectiveness [42], however many techniques are difficult to generalize, difficult to scale, or both. For example, Kurtz et al. [26] found significant cost savings, but only if each program is analyzed individually. Thus, a significant open question is whether the results of *any cost reduction technique* on previous programs will hold true for new programs.

For example, *selective mutation* [4, 36] defines an empirically determined subset of mutation operators, a *sufficient* set O_s , that dramatically reduces the number of mutants (and thereby the cost) without reducing the test quality. However, Kurtz et al. [26] found that the ideal sufficient set is different for each program. This problem can be generalized in terms of particular sets of programs, or even in terms of technologies. For instance, while investigating the ability of mutation to reveal real faults, Siami-Namin and Kakarla [46] concluded that “*the major question to address is whether the researchers need to apply exactly the same set of operators or the same number of mutants as used*” in previous experiments, “*for any other experimentations regardless of underlying programming languages*.” Papadakis et al. [40] also observed that “*when using mutation testing, it is important to carefully select*

mutant operators that are appropriate to the programming language studied”.

We are currently investigating whether we can establish similarities among programs such that the most effective sufficient set of mutation operators for a prior program will likely also be an effective sufficient set for a similar program. In a previous paper [12], we reported results from an experiment with a small set of 38 Java programs. We *clustered* programs based on internal metrics, and asked whether sufficient sets of mutation operators that were previously determined to be effective for programs in the cluster are also effective for new programs in the cluster. That is, can internal metrics be used to determine similarity among programs that can be used to predict an effective sufficient set of mutation operators. Notice that such a program similarity-based approach can use any cost reduction technique that relies on empirical training, including but not limited to, supervised Machine Learning algorithms and operator-based mutant selection,

This paper extends our prior paper in three ways. It:

- 1) Externalizes a conceptual framework, named *SiMut*, to support the cost reduction of mutation testing based on historical data and program similarity. *SiMut* uses varied program abstractions beyond internal metrics, varied program similarity calculations beyond clustering, and the application of varied cost reduction techniques.
- 2) Presents a proof-of-concept implementation of the main activities defined in *SiMut*, as well as results from a pilot study with a dataset that is larger than in our prior work. The artifacts included in the database were used and produced in prior experiments with mutation testing.
- 3) Discusses key issues that should be handled while composing a dataset to apply the approach embedded in *SiMut*.

We analyzed three extensive literature reviews on mutation testing [21, 40, 42]. Two [21, 40] summarized mutation-related research published from 1977 to 2017; they account for more than 800 analyzed studies, and provided a broad picture of the area. The other [42] analyzed 175 studies and focused on characterizing goals and techniques for cost reduction, including used metrics and achieved results. None of studies analyzed in the reviews had goals similar to ours. Therefore, to the best of our knowledge, this is the first approach that combines the use of information from previously tested programs and program similarity to support the testing of new programs using mutation at reduced cost.

This section summarizes the context, motivation, goals and contributions of this paper. Basic background is presented in Section II. Section III describes the *SiMut* framework, Section IV describes our implementation, and Section V reports on results of a pilot study. Section VI discusses key issues related to the application of *SiMut*, as well as limitations of this work. Section VII summarizes related work. Finally, Section VIII brings our conclusions and points out future work.

II. BACKGROUND

This section describes basic background on program abstractions and program similarity. Both concepts are used in the *SiMut* framework, described in Section III.

A. Program Abstractions

Program abstraction is used to simplify the process of analyzing complex programs. The abstraction tool creates a relationship between a concrete program and an abstract program [10]. We summarize various forms of program abstractions, including structural graphs, sequence diagrams, obfuscated source code, simplified source code, and static and dynamic internal metrics.

In the context of program abstractions, our two experiments (one from our prior work [12] and another here) used the Chidamber and Kemerer (CK) static internal metrics [9] as program abstractions. We then used them to compute similarity among programs.

B. Program Similarity

The literature has several definitions for program similarity. For example, in a comparison study of software clone detection techniques, Roy et al. [45] defined two types of similarity between code fragments, one based on text (similar program code), and based the other on functionality (similar program behavior).

Another definition comes from Walenstein et al. [49]. The definitions are similar to the ones proposed by Roy et al. [45]. Walenstein et al. conceptually defined two types of similarity: behavioral (semantic) and representational (syntactic). Semantic similarity compares the meaning not the structure. Examples are the comparison of program input and output, and the program execution with searching for statements correspondence (in relation to the order of statement executions). Syntactic similarity, on the other hand, compares the structure not the meaning, in different levels of abstraction, from code blocks to architectural levels.

Representational similarity can be computed based on statically computed internal software metrics, which map code features or structures to the domain of real numbers. We use this technique in this paper, as described in Section V. Another example is “feature-based” similarity, which determines the similarity between programs using the number of feature matches or properties identified in the programs.

III. THE *SiMut* FRAMEWORK

This section describes the conceptual framework named *SiMut* (Program **S**imilarity to support Cost Reduction of **M**utation). *SiMut* supports the cost reduction of mutation testing based on historical data and on program similarity as a core element. It is comprised of a main process (*overall process*) and a sub-process that is responsible for composing groups of similar programs. Both processes, and some usage scenarios, are described below.

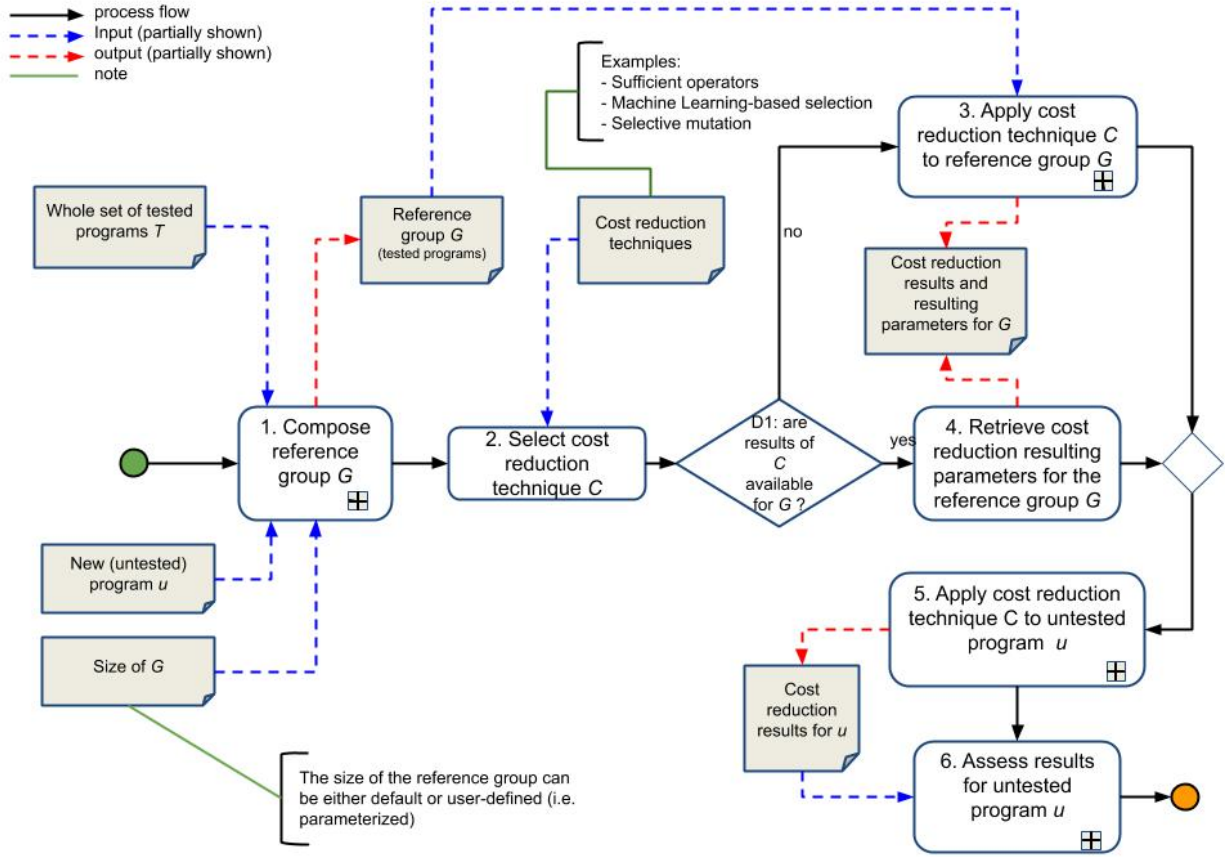


Fig. 1. Overall process.

A. Overall Process

Figure 1 represents the overall process of *SiMut*. We use Business Process Model and Notation (BPMN) to represent the process. Key items (e.g. sub-processes, tasks, inputs, output, and additional notes) are described next.

1. Compose reference group G : This sub-process composes a group G of tested programs that are similar to a new (untested) program u . A *tested program* has a mutation-adequate test suite. The number of programs in G can be either user-defined or default (e.g., defined by the similarity algorithm).

- Inputs: A set T of programs tested with mutation, the untested program u , and (optional) the size of G .
- Output: The reference group G .
- Notes: For more details, see the *Composition of the Reference Group (G)* process in Section III-B.

2. Select cost reduction technique C : A cost reduction technique C to apply to G is chosen from existing techniques such as *selective mutation* [36], *optimization of analysis of mutants* based on Machine Learning algorithms [6, 8], *sufficient operators* [4] and *minimal mutation* [2]. Users decide which technique to use. If results for C applied to G are already

available, they are retrieved (process 4). Otherwise, they are calculated using process 3.

- Input: A collection of available cost reduction techniques.
- Output: The selected cost reduction technique C .

3. Apply cost reduction technique C to reference group G : This sub-process applies C to the programs in G . The resulting measures R_G and the resulting cost reduction parameters S are stored. Examples of parameters are a sufficient set of mutation operators or a trained Machine Learning algorithm. Parameters S will be next applied to the untested program u .

- Input: The reference group G and the selected cost reduction technique C .
- Output: The cost reduction results R_G and the cost reduction resulting parameters S .

4. Retrieve cost reduction resulting parameters S for the reference group G .

5. Apply cost reduction technique C to untested program u : The parameters S from applying C to the reference group G are used to apply C to the new program u .

- Inputs: The untested program u , the selected cost reduction technique C , and the cost reduction parameters S .

- Output: R_u , the results of applying C to u based on S .

6. Assess results for untested program u : Here we evaluate the results R_u . The evaluation includes the mutation score produced by S -adequate tests when all mutants are considered. The evaluation can also compare cost reduction savings with respect to the reference group G .

- Inputs: The cost reduction results R_u , all mutants, and (optional) the cost reduction results R_G .
- Output: Varied (depends on the evaluation goals).

Notes about the overall process: Sub-process 6 is for experimental purposes only. This step would not be executed when *SiMut* is used in practice.

B. Choosing Programs in the Reference Group

This section describes how programs are chosen for the reference group G , as illustrated in Figure 2.

1.1. Computing measures for u : We need to apply the same measures to u that will be used to calculate similarity. Measures that may be calculated (depending on current experimental goals) include internal program metrics [9], control flow graphs [34], and obfuscated source code. We refer to these as *pre-processed abstractions* in the rest of the paper.

- Input: The untested program u .
- Output: A set of measures A_u for u .
- Notes: We assume that the same measures have already been gathered for programs in T . The process of gathering such measures is the same as process #1.1 described previously.

1.2. Select similarity approach S : Next we select a similarity calculation approach S to apply to u with respect to the tested programs T . Example similarity calculations include clustering (applied in the exploratory study presented in Section V), lexical distance calculation, and information diversity calculation.

- Input: A set of available similarity calculation approaches.
- Output: The selected similarity calculation approach S .

1.3. Apply approach S : Next we calculate similarity between u and T to determine the reference group G of programs that are similar to u .

- Inputs: The selected similarity calculation approach S , the set of measures A_u for u , the sets of measures for the programs included in the set of tested programs T , and (optional) the size of G .
- Output: A preliminary version of the reference group G (this group may be refined in the next process).

1.4. Generate reference group G : G is a subset of the programs in T that will be used in the overall process described in Section III-A. G contains programs from T that are most similar to u .

- Input: The selected similarity calculation approach S ; the initial reference group G ; and the size of G .
- Output: The final reference group G .

Notes about the process for computing G : As noted in sub-process 1.1, the composition of the reference group requires that the same measures gathered for u have already been gathered for programs in T . This information must be available to be retrieved and used in sub-process 1.3 (*Apply approach S*).

C. Examples of Usage Scenarios

This section illustrates the use of *SiMut* with two scenarios, each applying a different mutation cost reduction technique. The first uses *sufficient operators* [4]. The second uses *optimization of mutant analysis* through automatic mutant classification based on Machine Learning (ML) algorithms [6, 8]. Both scenarios require the execution of particular paths of the process depicted in Figure 1. A third technique, *one-op mutation* [48], was used in the study described in Section V.

Scenario 1—Mutant classification based on sufficient operators: This scenario computes a sufficient set of mutation operators for u using a procedure such as the *Sufficient* procedure by Barbosa et al. [4]). The sufficient set of operators will be obtained based on the reference group G using the following steps:

- Sub-process 1: Compose the reference group G of similar programs.
- Task 2: Select the cost reduction technique C (sufficient operators).
- Decision D1: Answer is *no* (results for C applied to G are not available).
- Sub-process 3: Apply technique C to G , getting results R_G and cost reduction parameters S (sufficient operators).
- Sub-process 5: Apply the sufficient operators identified for G to the untested program u , obtaining results R_u .
- Sub-process 6 (optional): Assess results R_u , for example, with respect to test set quality (full mutation score), and with respect to R_G .

Scenario 2—Mutant classification based on machine learning algorithms: The tester reuses a trained ML algorithm that automatically classifies mutants based on their utility. An example approach is by Chekam et al. [6, 8], who applied supervised learning to train an algorithm to automatically classify mutants based on their probability of revealing faults. This scenario assumes a reference group G that was already used to train the ML algorithm. The algorithm parameters are retrieved and reused to classify mutants of the untested program u using the following steps:

- Sub-process 1: Compose the Reference group G of similar programs.
- Task 2: Select the cost reduction technique C (mutant classification based on ML).
- Decision D1: Answer is *yes* (results for C applied to G are available).
- Sub-process 4: Retrieve the parameters S from the trained ML algorithm.
- Sub-process 5: Apply the ML-based classifier to mutants of the untested program u , obtaining results R_u .

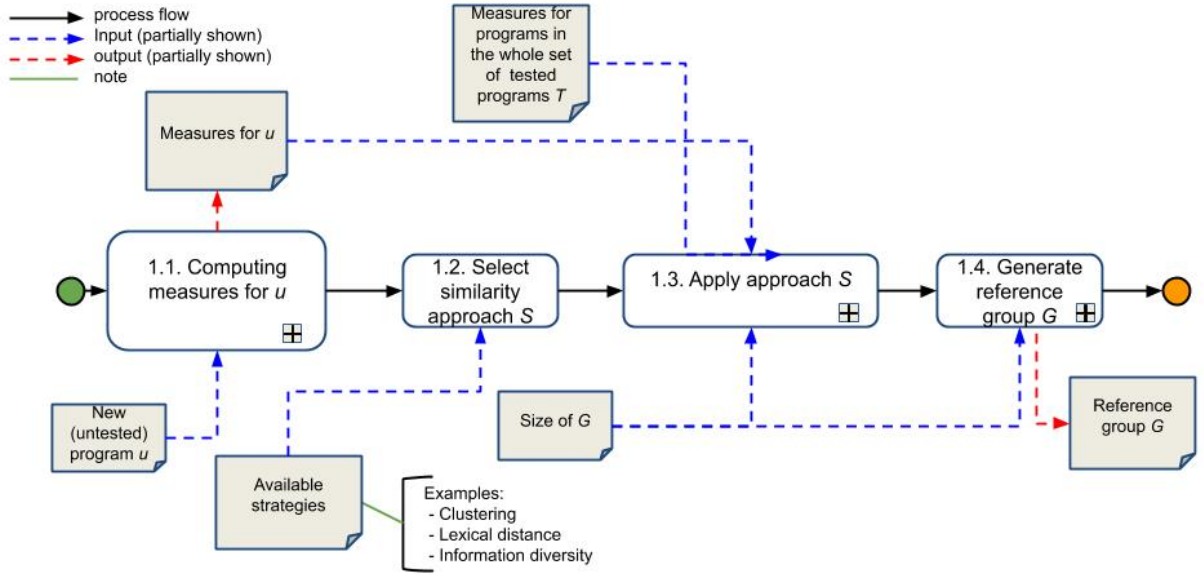


Fig. 2. Composition of the reference group of tested programs.

- Sub-process 6 (optional): Assess results R_u , for example, with respect to precision of the classifier, and with respect to the full mutation score produced by test cases that are adequate for the classified mutants.

IV. PROOF-OF-CONCEPT IMPLEMENTATION

This section describes a proof-of-concept implementation of *SiMut*. The prototype includes three main modules, *Metrics Collector*, *Clustering Calculator*, and *Mutation Score Calculator*. It is available at <https://doi.org/10.6084/m9.figshare.11787474.v1>. *Metrics Collector* and *Clustering Calculator* are Java programs that use an external tool to collect metrics and an API to cluster information. *Mutation Score Calculator* is written in SQL and C#. Details are provided next.

Metrics Collector: We use Analizo¹ [47] to calculate metrics. Analizo computes 10 project level metrics and 16 class level metrics, including the CK metrics [9]. Metrics were gathered in three steps. First, the programs to be analyzed are identified and stored in a file system. Second, the metrics are collected and stored in memory for each source file identified in step one. Third, the metrics are stored in a relational database.

Clustering Calculator: The clustering process runs in four steps. First, all values from the Collector module are normalized in a linear fashion and placed into the interval [0, 1]. Second, the normalized values are formatted for the X-Means algorithm [41]. Third, we use the X-means algorithm, as implemented in the Weka API [17], to produce the clusters. Fourth, the results are presented in terms of the clusters generated and which programs belong to each cluster.

Mutation Score Calculator: Program mutation usually finds a test that kills a mutant, then does not run that mutant against

subsequent tests. However, we need to know all tests that kill all mutants. This information allows us to create a complete *killing matrix* for each program, where rows represent all the mutants, columns represent all the tests, and a cell is marked if that test killed that mutant.

We use a database of mutation-related artifacts (described in Section V-B) to identify mutants generated by each operator, and calculate mutation scores for each operator. For the experiment described in Section V, the mutation scores of each operator were calculated for the programs separately, for each cluster created, and for the entire set of programs. This allowed us to determine the best mutation operator (One-Op) for each option (program, cluster, and total set). Note that our database includes information about mutants and tests that kill them; therefore, we do not re-run the mutants on the test sets.

V. PILOT STUDY

We ran a pilot study to demonstrate the feasibility of our process, as well as to illustrate how results can be interpreted. At this point, we do not analyze statistical significance because of limitations on the data, as discussed in Section VI-A.

A. Study Overview

Our research goal is to determine if internal program information (as measured by internal metrics) can be used to predict mutation testing results. More specifically, for an untested program u , we evaluate whether the best mutation operators for programs similar to u are also the best operators for u as well. We measure operators based on their mutation scores. This study uses the same procedures as in our previous study [12], but considers a substantially larger database of mutation-related artifacts.

This study followed the *SiMut* process from Section III. We selected CK metrics as the program abstraction, clustering as

¹<http://www.analizo.org/>, accessed in February, 2020.

TABLE I
SUMMARY OF ARTIFACTS INCLUDED IN THE DATABASE.

Source	# Programs	# Tests	# Mutants	# Operators	Operators	Tool
[15]	29	18	4,161	15	LOI; SDL; AOIS; VDL; COI; AOIU; ROR; AOR; ODL; CDL; COR; ASRS; AODU; AODS; COD	muJava
[38, 43]	9	303	1,347	10	AOIS; AOIU; AOR; ROR; LOI; AODS; COR; AODU; COD; COI	muJava
[16]	13	11	2,418	18	JID; JSD; JSE; AOIS; AOIU; AOR; ASRS; COI; LOI; ROR; EAM; JDC; JTD; JTI; IOD; AODU; COD; COR	muJava
[37]	21	105	10,147	30	EAM; EOC; IOD; IOR; IPC; ISI; JSD; JSE; PCI; OMR; PCC; ISD; PRV; OAN; PMD; PPD; EMM; IOP; PCD; JID; ROR; UOI; SWA; LCR; NUF; ABS; TEX; AOR; DEC; INC	muJava
[19]	211	934	19,851	10	DEC; SWA; AOR; ABS; INC; UOI; NUF; ROR; LCR; TEX	Bacterio

the similarity calculation technique, and one-op [48] as the mutation cost reduction technique.

B. Subject Programs

We used Java programs from previous mutation-related research. We used our previous Systematic Literature Review [42] to find subject programs that are available online. We also contacted all authors of studies that used Java programs to request access to the artifacts from their experiments.

After analyzing these materials, we had a collection of artifacts that included programs, mutants, mutation operators used, test cases, and information on which tests killed which mutants. We had to omit artifacts that did not have enough information for our research.

Table I summarizes our subject artifacts, as drawn from six prior experiments [15, 16, 19, 37, 38, 43]. These artifacts are available at <https://tinyurl.com/mutation2020>.

The database includes 283 Java programs, most of which have a single class. We used 37,924 first-order mutants generated from the mutation operators listed in the *Operators* column by the muJava [33] and Bacterio [44] tools. We also used 1,371 test cases, which are linked to the mutants they killed. The studies indicated the tests were mutation-adequate, so mutants not killed are assumed to be equivalent. Disregarding duplicate programs, the database includes 218 items.

As shown in Table I, the 15 mutation operators in muJava were applied to the 29 programs in study [15]. However, the next row shows that only 10 muJava operators were applied to the 9 programs. In general, different programs and different studies might use different mutation operators within the same dataset. This issue imposes restrictions to the reuse of historical information and is further discussed in Section VI.

C. Used Metrics

Chidamber and Kemerer proposed six metrics for object-oriented software [9]. Table II defines the metrics, and Table III shows the values gathered using the Analizo tool for some of the programs². Given that most programs only have a single class, we determined that only the ACCM and RFC metrics were meaningful. The other CK metrics are related to class hierarchies and structures that are not common in our subject programs. Impacts of this 2-value (ACCM and RFC) representation of each program is discussed in Section VI.

²Metrics for other programs are omitted for space but included in the repository <https://tinyurl.com/mutation2020>.

D. Program Sets and Clusters

We separated the programs into three sets, based on which mutation operators were applied to them. This allowed us to combine programs used in previous studies, and to create groups of programs that are homogeneous in terms of applied mutation operators. For instance, consider a set of programs $B = \{P_1, P_2, P_3, P_4, P_5\}$. If the LOI and AOR operators were applied to P_1 , P_2 and P_5 , and the AOR and ROR operators were applied to P_1 , P_3 and P_4 , we created two sets of programs $B_1 = \{P_1, P_2, P_5\}$ and $B_2 = \{P_1, P_3, P_4\}$. Then, for B_1 we considered mutants generated by LOI and AOR, and for B_2 we considered mutants generated by AOR and ROR. The sizes of the program sets are shown in the second column of Table IV.

The numbers of programs per cluster formed by the X-Means algorithm are shown in the *# Programs per Cluster* columns in Table IV. Two clusters were formed for B_1 , 3 for B_2 , and 5 for B_3 . The numbers of mutation operators applied to each program set are shown in the *# Operators* column and the mutation operators are listed in the *Operators* column.

E. Summary of Results

Table V summarizes the results of this pilot study. $R1$ and $R2$, defined below, may have positive (V), negative (X), or neutral results (N). Note that, to achieve such results, from each set (B_1 , B_2 , B_3), we took each program as an untested program u (for experimental purposes only), and computed the best mutation operators for u , for the cluster to which u is associated with (except u), for the other clusters formed for the program set (called *remaining clusters*), and for all programs in the program set.

- $R1$: We compute $R1$ to evaluate if the cluster of similar programs is a good predictor of the best mutation operator for an untested program u , when compared to clusters of programs that are less similar to u . O_u is the set of best operators for u , and OC_i is the set of best operators for each cluster i . $R1$ evaluates the intersection of O_u and OC_i , considering the cluster that u is associated with, and the remaining clusters formed from the program set.
- $R2$: We next compute $R2$ to evaluate if the cluster of similar programs is a good predictor of the best mutation operator for an untested program u , when compared to the entire set of programs. OP_u is the set of best operators for the entire set of programs. $R2$ evaluates the intersection between the sets O_u and OP_u , then compares this result

TABLE II
CK METRICS COLLECTED BY ANALIZO.

ACCM (<i>Average of Cyclomatic Complexity per Method</i>):	Used as substitute to WMC (Weighted Methods per Class), consists of summing up all separate paths within a method (cyclomatic complexity proposed by McCabe [32]), and then summing up all method-specific complexities.
CBO (<i>Coupling between Object Classes</i>):	Corresponds to the number of interactions of a chosen class with other classes at the level of methods and variables.
DIT (<i>Depth of Inheritance Tree</i>):	Measures the depth of the inheritance tree of a given class.
LCOM4 (<i>Lack of Cohesion of Methods version 4</i>):	The methods are organized in pairs, say $m1$ and $m2$. The number of class attributes that $m1$ and $m2$ have in common are counted, then the number of class attributes $m1$ and $m2$ do not have in common are counted. LCOM4 is the difference between those counts.
NOC (<i>Number of Children</i>):	Count the number of subclasses of a class.
RFC (<i>Response for Class</i>):	The number of methods inside the class added to the number of external methods used by the class.

TABLE III
CK METRICS FROM A SUBSET OF PROGRAMS.

Source	Program	LOC	CK Metrics					
			ACCM	CBO	DIT	LCOM4	NOC	RFC
[38, 43]	Bisec	24	2	0	0	3	0	5
[38, 43]	BubCorrecto	35	1.57	0	0	2	0	14
[38, 43]	Find	44	3.25	0	0	1	0	11
[38, 43]	Fourballs	28	2.5	0	0	1	0	11
[38, 43]	Mid	41	1.44	0	0	4	0	18
[38, 43]	Triangulo	54	3.6	0	1	2	0	16
[38, 43]	PluginTokenizer	103	1.62	0	1	6	0	30
[38, 43]	Ciudad	262	2.65	0	0	1	0	55
[38, 43]	IgnoreList	36	2.66	0	0	1	0	7
[15]	BoundedQueue	78	2	0	0	1	0	26
[15]	countPositive	63	3.5	0	0	2	0	2
...								

TABLE IV
PROGRAM SETS AND CLUSTERS.

Program Sets	# Programs	# Programs per Cluster					# Operators	Operators
		C_1	C_2	C_3	C_4	C_5		
B_1	36	17	19				10	AOIS; AOIU; AOR; CDL; COI; COR; LOI; ROR; SDL; VDL
B_2	82	47	29	6			10	ABS; AOR; DEC; INC; LCR; NUF; ROR; SWA; TEX; UOI
B_3	120	22	74	13	3	8	5	AOR; LOI; NUF; ROR; VDL

TABLE V
SUMMARY OF RESULTS.

Results	Program Sets		
	B_1	B_2	B_3
$R1$	V(14), N(10), X(12)	V(0), N(30), X(52)	V(0), N(90), X(30)
$R2$	V(14), N(22), X(0)	V(0), N(82), X(0)	V(1), N(110), X(9)

with OC_u , which is the set of best operators for the cluster u is associated with.

For both $R1$ and $R2$, positive (P) and neutral (N) results favor the similarity approach, given that the cluster was a predictor at least as good as the other clusters, or at least as good as the entire set of programs. Note that positive results regarding $R2$ motivate the evolution of the research, since one might consider just a subset of programs to calculate cost reduction parameters for applying a particular mutation cost reduction technique.

In this pilot study, for B_1 the clusters were good predictors of best mutation operators in 67% (24/36) of the cases when compared with the other clusters, and in 100% (36/36) of the cases when compared with all programs. These percentages were 37% (30/82) and 100% (82/82) for B_2 , and 75% (90/120) and 93% (111/120) for B_3 .

As we are trying to show feasibility as well as describing how the approach works via example, we did not compute significance in this pilot study, nor compare results with

our prior work [12]. Next we discuss issues that should be handled while identifying or finding artifacts for this type of experiment or when applying this approach in practice.

VI. DISCUSSIONS AND LIMITATIONS

This section discusses issues related to the selection and preparation of artifacts and techniques to be used in an approach such as the one introduced by *SiMut*. We highlight three issues: (i) how representative the set of programs in the population is, (ii) how comprehensive the mutation operator set is, and (iii) the tools used to generate the mutants.

(i) *How representative the set of programs in the population is*: We used programs that had been used in previous studies [15, 16, 19, 37, 38, 43]. Despite the relatively large number of programs (218), they are small and simple, and most did not use OO-specific constructs such as class hierarchies and composition. We only used programs that came with mutants whose operators were identified and test cases that killed all non-equivalent mutants. These constraints make it

difficult to create a representative population of programs, and it is unlikely our programs were. In the future, we hope to either select metrics that are appropriate for the population of programs, or create a larger and more representative population of programs.

(ii) *How comprehensive the mutation operator set is:* Questions such as “Are there enough operators?”, or “Are there too few operators?” should be considered when selecting the cost reduction technique to use. The set of operators should be aligned with the intent of the technique. For instance, if using sufficient operators [4], we should analyze many operators. In our pilot study, the programs used relatively few operators, making it harder to explore different scenarios, for example, trying subsets of sufficient operators of varying sizes. In this study, the same few operators (ROR and SDL) dominated.

(iii) *The tools used to generate mutants:* The mutation operators implemented in each tool vary, and sometimes vary in different releases of the same tool. The details of how some operators are implemented also vary, depending on design and implementation decisions.

In this context, Kintis et al. [25] compared the operators from three mutation tools for Java programs: PIT, muJava and Major. For example, consider Table VI. PIT implements the Math operator (M) as described in the first row. muJava and Major implements arithmetic mutation using several individual operators (AORB, SOR, and LOR in muJava, and AOR, SOR and LOR in Major). Kintis et al. [25] also found differences in the same operator. For example, conditional operator replacement (COR) for muJava and Major is illustrated in Table VI.

TABLE VI
MUTATION OPERATORS IMPLEMENTED IN DIFFERENT TOOLS (ADAPTED FROM KINTIS ET AL. [25])

Operator		Description
PIT	M	$\{(op1, op2) (op1, op2) \in \{(+, -), (-, +), (*, /), (/ , *), (% , *), (\&,), (, \&), (\wedge, \&), (< <, > >), (> >, < <), (> > >, < < <)\}\}$
muJava	AORB	$\{(op1, op2) op1, op2 \in \{+, -, *, /, \% \} \wedge op1 \neq op2\}$
	SOR	$\{(op1, op2) op1, op2 \in \{> >, > > >, < < < \} \wedge op1 \neq op2\}$
	LOR	$\{(op1, op2) op1, op2 \in \{\&, , \wedge \} \wedge op1 \neq op2\}$
	COR	$\{(op1, op2) op1, op2 \in \{\& \&, , \wedge \} \wedge op1 \neq op2\}$
Major	AOR	$\{(op1, op2) op1, op2 \in \{+, -, *, /, \% \} \wedge op1 \neq op2\}$
	SOR	$\{(op1, op2) op1, op2 \in \{> >, > > >, < < < \} \wedge op1 \neq op2\}$
	LOR	$\{(op1, op2) op1, op2 \in \{\&, , \wedge \} \wedge op1 \neq op2\}$
	COR	$\{(\& \&, op2), (, op2) op1 \in \{==, LHS, RHS, false\}, op2 \in \{!=, LHS, RHS, true\}\}$

These differences in mutation operators may mean that effective tests for mutants created by one tool may not be effective for mutants created from the same operators in a different tool. Thus, the tools used to create the mutants in the program population must be carefully considered.

A. Limitations

SiMut is a conceptual framework that currently has one proof-of-concept implementation. The *SiMut* framework has three variables: (1) the program abstraction, (2) the similarity

calculation, and (3) the cost reduction technique. Our implementation and pilot study used internal metrics for (1), clustering for (2), and one-op mutation for (3). Other configurations can be used for more extensive experimentation.

SiMut can also be expensive to use. Specific costs include executing *SiMut*’s sub-processes to generate the reference group of programs G , and computing the cost reduction technique for G . However, computing cost reduction for an entire set of programs, and obtaining cost reduction parameters to apply to a new, untested program may be even more costly.

VII. RELATED WORK

Despite a comprehensive analysis of the literature on software testing [21, 40, 42], we were not able to find prior approaches that use historical information to reduce the cost of applying mutation testing. This section describes recent papers that explore program similarity to automatically classify equivalent and redundant mutants [23, 24, 39], use internal program characteristics to reduce the cost of mutation testing [6, 8], and to correlate such characteristics with program testability [11].

Several authors used program similarity to automatically classify equivalent [23, 24, 39] and redundant mutants [24, 39]. At the binary code level, Papadakis et al. [39] developed *Trivial Compiler Equivalence* (TCE), which uses a simple diff program to check equivalence between an original program and its mutants, and to check for redundancy among mutants. If the binary code is identical, the mutant is confirmed to be equivalent or redundant.

A recent evaluation of TCE [24] considered programs written in Java and C. For each language, two sets of programs were analyzed (6 small, and 6 medium or large Java programs; and 18 small, and 6 medium and large C programs). On average, for the first sets of programs, for which ground-truth information was available (that is, all equivalent mutants were manually classified), TCE identified 30% of the equivalent mutants for C and 54% for Java. For medium and large programs, considering all mutants, TCE classified 5.7% of Java mutants and 7.4% of C mutants as equivalent, and 5.4% of Java mutants and 21% of C mutants as duplicate. The study did not report the true percent of equivalent and duplicate mutants for medium and large programs, as that would have taken extensive (and time-consuming) hand analysis.

At the source code level, Kintis and Malevris [23] explored the concept of *mirrored mutants*; if two or more mutants are present in similar code fragments, and particularly in identical code locations inside these fragments, defining one of those mutants as equivalent to the original program may imply that the others are also equivalent. The authors used a clone detection tool to detect the mirrored mutants. An evaluation with six small Java programs resulted in average 49% automatically classified equivalent mutants, with precision ranging from 88% to 100%. Also at the source code level, Ji et al. [20] used a domain reduction technique to compute a distance value between two mutants to classify mutants into specific clusters. They experimented with 12 mutation operators from the muJava tool [28] and one small

Java program. The predefined number of algorithm executions did not allow for precise extraction of results. In our work, sub-processes 1.1, 1.2, and 1.3 in Figure 2 can handle either source code or binary code as program abstractions to calculate similarity. As opposed to previous research [20, 23], *SiMut* does not compute the similarity between mutants of a single program; instead, it computes the similarity among various programs.

Chekam et al.’s FaRM approach [6, 8] relies on internal program information and supervised ML algorithms to automatically classify mutants that lead to more effective test cases in terms of probability of revealing real unknown faults. FaRM uses 28 program features to train the classifier. The features are derived from the control-flow graph, the abstract syntax tree, and the mutation type. Chekam et al. found that FaRM outperforms random mutation and selective mutation in terms of fault revealing ability, ranging from 23% to 34% more revealed faults. When compared with our work, FaRM uses program features that encompasses complexity elements, which are also reflected by internal metrics. As described in Section III-C, our approach could be combined with FaRM; from a large set of input programs, *SiMut* could find the most similar programs to be used during training, with the expectation of achieving more precise mutant classification.

Cruz and Eler [11] explored program clustering based on CK metrics to estimate program testability. They studied Java programs and generated clusters using the Expectation Maximization (EM) algorithm, based on the CBO, LCOM, RFC, and WMC metrics. The testability of a class was characterized as high or low, according to the structural coverage and mutation score produced by the associated test sets. The precision of the EM algorithm was assessed with the kNN (*k* Nearest Neighbors) algorithm [1]. In particular, Cruz and Eler used $k = 3$ and in 81.5% of cases, kNN confirmed the clusters produced by the EM algorithm precisely grouped classes with the same level of testability.

VIII. CONCLUSIONS AND FUTURE WORK

This paper introduced the *SiMut* conceptual framework to support cost reduction of mutation testing based on historical data and program similarity. Although numerous papers have reported success using cost reduction techniques when averaged over many programs, Kurtz et al. [26] demonstrated that, although the techniques work well on average, they do not work well on individual programs. They suggested the idea of *individualized* cost reduction, where the cost reduction technique is tailored to individual programs. This leaves the problem: How do we know which cost reduction technique will work well with a given program before we test it? Thus, the key goal of *SiMut* is to support the application of mutation testing to a new, untested program u based on knowledge obtained from testing programs that are similar to u . The framework has three main variation points: the program abstraction that represents the programs, the approach to be used to calculate similarity among programs, and the mutation cost reduction technique that to apply to u .

We demonstrated the feasibility of implementing and running *SiMut* with a proof-of-concept implementation and a pilot study. We used internal program metrics as the program abstraction, clustering as the similarity calculation approach, and one-op mutation as the cost reduction technique. We also presented some usage scenarios, and discussed some issues related to the use of the framework, focusing on the composition of a representative dataset to properly explore the potential of *SiMut*.

For future work, the implementation can be reused and customized for other experiments, including replications, and for properly supporting the evolution of *SiMut*. We also plan to analyze how the mechanisms *SiMut* uses can be used in other contexts. Examples are the metrics collector and the clustering calculator, which may also be used for program analysis purposes. Last but not least, we would also like to compare programs at the method level.

ACKNOWLEDGEMENTS

Alessandro Pizzoleto and Lucas Dallilo were supported by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, and Federal University of São Carlos, Brazil. Fabiano Ferrari was partly supported by Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP, Brazil) - grant #2016/21251-0, Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq, Brazil) - grant #306310/2016-3, and Federal University of São Carlos, Brazil. Jeff Offutt was partly supported by KKS (The Knowledge Foundation), by project 20130085, Testing of Critical System Characteristics (TOCSYC); and partly by a Google educational grant, Self-Paced Learning Increases Retention and Capacity (SPARC).

REFERENCES

- [1] D. W. Aha, D. Kibler, and M. K. Albert, “Instance-based Learning Algorithms,” *Machine Learning*, vol. 6, no. 1, 1991.
- [2] P. Ammann, M. E. Delamaro, and A. J. Offutt, “Establishing Theoretical Minimal Sets of Mutants,” in *Proc. the 7th Internl. Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2014.
- [3] J. H. Andrews, L. C. Briand, and Y. Labiche, “Is Mutation an Appropriate Tool for Testing Experiments?” in *Proc. the 27th Internl. Conf. on Software Engineering (ICSE)*. ACM, 2005.
- [4] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi, “Toward the Determination of Sufficient Mutant Operators for C,” *Soft. Testing, Verification and Reliability*, vol. 11, no. 2, 2001.
- [5] T. Budd and D. Angluin, “Two Notions of Correctness and their Relation to Testing,” *Acta Informatica*, vol. 8, no. 1, 1982.
- [6] T. Chekam, M. Papadakis, T. Bisseyandé, and Y. Le Traon, “Predicting the Fault Revelation Utility of Mutants,” in *Proc. the 40th Internl. Conference on Software Engineering (ICSE) - Posters Track*. IEEE, 2018.
- [7] T. T. Chekam, M. Papadakis, Y. Le Traon, and M. Harman, “An Empirical Study on Mutation, Statement and Branch Coverage Fault Revelation That Avoids the Unreliable Clean Program Assumption,” in *Proc. the 39th Internl. Conference on Software Engineering (ICSE)*. IEEE, 2017.
- [8] T. T. Chekam, M. Papadakis, T. F. Bisseyandé, Y. Le Traon, and K. Sen, “Selecting Fault Revealing Mutants,” SnT Centre, University of Luxembourg; and University of California, Joint Technical Report arXiv:1803.07901, 2018.
- [9] S. R. Chidamber and C. F. Kemerer, “A metrics suite for object oriented design,” *IEEE Trans. on Soft. Eng.*, vol. 20, no. 6, 1994.

- [10] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," in *Proc. the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM, 1977.
- [11] R. C. Cruz and M. M. Eler, "Using a Cluster Analysis Method for Grouping Classes According to their inferred Testability: An Investigation of CK Metrics, Code Coverage and Mutation Score," in *Proc. the 36th Internl. Conf. of the Chilean Computer Science Society (SCCC)*. Chilean Comp. Science Society, 2017.
- [12] L. D. Dallilo, A. V. Pizzoleto, and F. C. Ferrari, "An Evaluation of Internal Program Metrics as Predictors of Mutation Operator Score," in *Proc. the 4th Brazilian Symposium on Systematic and Automated Software Testing (SAST)*. ACM, 2019.
- [13] M. Daran and P. Thévenod-Fosse, "Software error analysis: A real case study involving real faults and mutations," in *Proc. the 1996 ACM SIGSOFT Internl. Symposium on Software Testing and Analysis (ISSTA)*. ACM, 1996.
- [14] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *IEEE Computer*, vol. 11, no. 4, 1978.
- [15] L. Deng, A. J. Offutt, and N. Li, "Empirical Evaluation of the Statement Deletion Mutation Operator," in *Proc. the 6th Internl. Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2013.
- [16] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, 2005.
- [17] E. Frank, M. A. Hall, and I. H. Witten, *Data Mining: Edition Practical Machine Learning Tools and Techniques*, 4th ed. Morgan Kaufmann, 2016.
- [18] P. G. Frankl, S. N. Weiss, and C. Hu, "All-uses vs Mutation Testing: An Experimental Comparison of Effectiveness," *Journal of Systems and Software*, vol. 38, no. 3, 1997.
- [19] M. Harman, Y. Jia, P. Reales, and M. Polo, "Angels and Monsters: An Empirical Investigation of Potential Test Effectiveness and Efficiency Improvement from Strongly Subsuming Higher Order Mutation," in *Proc. the 29th Internl. Conference on Automated Software Engineering (ASE)*. ACM, 2014.
- [20] C. Ji, Z. Chen, B. Xu, and Z. Zhao, "A Novel Method of Mutation Clustering Based on Domain Analysis," in *Proc. the 21th Internl. Conference on Software Engineering and Knowledge Engineering (SEKE)*. Knowledge Systems Institute Graduate School, 2009.
- [21] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *IEEE Trans. on Soft. Eng.*, vol. 37, no. 5, 2011.
- [22] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are Mutants a Valid Substitute for Real Faults in Software Testing?" in *Proc. the 22nd Internl. Symposium on Foundations of Software Engineering (FSE)*. ACM, 2014.
- [23] M. Kintis and N. Malevris, "Identifying More Equivalent Mutants via Code Similarity," in *Proc. the 20th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2013.
- [24] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. Le Traon, and M. Harman, "Detecting Trivial Mutant Equivalences via Compiler Optimisations," *IEEE Trans. on Soft. Eng.*, vol. 44, no. 4, 2017.
- [25] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis, N. Malevris, and Y. Le Traon, "How Effective are Mutation Testing Tools? An Empirical Analysis of Java Mutation Testing Tools with Manual Analysis and Real Faults," *Empirical Software Engineering*, vol. 23, no. 4, 2018.
- [26] B. Kurtz, P. Ammann, A. J. Offutt, M. E. Delamaro, M. Kurtz, and N. Gökçe, "Analyzing the Validity of Selective Mutation with Dominator Mutants," in *Proc. the 24th Internl. Symposium on Foundations of Software Engineering (FSE)*. ACM, 2016.
- [27] N. Li, U. Praphamontipong, and A. J. Offutt, "An Experimental Comparison of Four Unit Test Criteria: Mutation, Edge-Pair, All-Uses and Prime Path Coverage," in *Proc. the 4th Internl. Workshop on Mutation Analysis (Mutation)*. IEEE, 2009.
- [28] Y. S. Ma, A. J. Offutt, and Y. R. Kwon, "MuJava: An Automated Class Mutation System," *Soft. Testing, Verification and Reliability*, vol. 15, no. 2, 2005.
- [29] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Józala, "Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation," *IEEE Trans. on Soft. Eng.*, vol. 40, no. 1, 2014.
- [30] A. P. Mathur, *Foundations of Software Testing*, 1st ed. Addison-Wesley Professional, 2007.
- [31] A. P. Mathur and W. E. Wong, "An Empirical Comparison of Data Flow and Mutation Based Test Adequacy Criteria," *Soft. Testing, Verification and Reliability*, vol. 4, no. 1, 1994.
- [32] T. J. McCabe, "A Complexity Measure," *IEEE Trans. on Soft. Eng.*, vol. SE-2, no. 4, 1976.
- [33] MuJava, "muJava Home Page," Online, 2015, <https://cs.gmu.edu/~offutt/mujava/> - accessed in December, 2019.
- [34] G. J. Myers, T. Badgett, and C. Sandler, *The Art of Software Testing*, 3rd ed. John Wiley & Sons, 2011.
- [35] A. J. Offutt and J. Pan, "Detecting Equivalent Mutants and the Feasible Path Problem," in *Proc. the 11th Annual Conference on Computer Assurance (COMPASS)*. ACM, 1996.
- [36] A. J. Offutt, G. Rothermel, and C. Zapf, "An Experimental Evaluation of Selective Mutation," in *Proc. the 15th Internl. Conference on Software Engineering (ICSE)*. IEEE, 1993.
- [37] A. J. Offutt, Y.-S. Ma, and Y.-R. Kwon, "The Class-Level Mutants of MuJava," in *Proc. the Internl. Workshop on Automation of Software Test (AST)*. ACM, 2006.
- [38] A. A. L. Oliveira, C. G. Camilo-Junior, and A. M. R. Vincenzi, "A Coevolutionary Algorithm to Automatic Test Case Selection and Mutant in Mutation Testing," in *Proc. the 2013 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2013.
- [39] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon, "Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique," in *Proc. the 37th Internl. Conference on Software Engineering (ICSE)*. ACM, 2015.
- [40] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation Testing Advances: An Analysis and Survey," in *Advances in Computers*, A. M. Memon, Ed. Elsevier, 2019, vol. 112.
- [41] D. Pelleg and A. W. Moore, "X-means: Extending K-means with Efficient Estimation of the Number of Clusters," in *Proc. the 17th Internl. Conference on Machine Learning*. Morgan Kaufmann Publishers, 2000.
- [42] A. V. Pizzoleto, F. C. Ferrari, A. J. Offutt, L. Fernandes, and M. Ribeiro, "A Systematic Literature Review of Techniques and Metrics to Reduce the Cost of Mutation Testing," *Journal of Systems and Software*, vol. 157, 2019.
- [43] M. Polo, M. Piattini, and I. García-Rodríguez, "Decreasing the Cost of Mutation Testing with Second-Order Mutants," *Soft. Testing, Verification and Reliability*, vol. 19, no. 2, 2009.
- [44] P. Reales, "Bacterio Mutation Test System - User Manual," Online, 2011, <https://alarcos.esi.uclm.es/per/preales/bacterio/BacterioManual.pdf> - accessed in February, 2020.
- [45] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach," *Science of Computer Programming*, vol. 74, no. 7, 2009.
- [46] A. Siami-Namin and S. Kakarla, "The use of mutation in testing experiments and its sensitivity to external threats," in *Proc. the 20th Internl. Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2011.
- [47] A. Terceiro, J. Costa, J. Miranda, P. Meirelles, L. R. Rios, L. Almeida, C. Chavez, and F. Kon, "Analizo: An Extensible Multi-Language Source Code Analysis and Visualization Toolkit," in *Proc. the 1st Brazilian Conference on Software: Theory and Practice (CBSOFT) - Tools Session*, 2010.
- [48] R. H. Untch, "On Reduced Neighborhood Mutation Analysis Using a Single Mutagenic Operator," in *Proc. the 47th Annual Southeast Regional Conference (ACM-SE)*. ACM, 2009.
- [49] A. Walenstein, M. El-Ramly, J. R. Cordy, W. S. Evans, K. Mahdavi, M. Pizka, G. Ramalingam, and J. W. Von-Gudenberg, "Similarity in Programs," in *Duplication, Redundancy, and Similarity in Software*, ser. Dagstuhl Seminar Proceedings, 2007.

APÊNDICE D

An Automated Framework for Cost Reduction of Mutation Testing Based on Program Similarity

Neste apêndice apresenta-se o conteúdo completo do artigo publicado no *17th International Workshop on Mutation Analysis (Mutation'22)* (GUARNIERI; PIZZOLETEO; FERRARI, 2022). Informações parciais deste artigo são apresentadas no Capítulo 4 e no Capítulo 6 desta tese.

IEEE COPYRIGHT NOTICE:¹ “©IEEE, 2022. Reprinted, with permission, from the *Proceedings of the 17th International Workshop on Mutation Analysis (Mutation'22)*, <<https://doi.org/10.1109/ICSTW55395.2022.00041>>”

¹ <<https://journals.ieeeauthorcenter.ieee.org/choose-a-publishing-agreement/avoid-infringement-upon-ieee-copyright/>> - acessado em novembro/2022.

An Automated Framework for Cost Reduction of Mutation Testing Based on Program Similarity

Giovanni F. Guarnieri
Federal University of São Carlos
Sorocaba, SP, Brazil
gfguarnieri@hotmail.com

Alessandro V. Pizzoleto
Federal University of São Carlos
São Carlos, SP, Brazil
alessandro.pizzoleto@ufscar.br

Fabiano C. Ferrari
Federal University of São Carlos
São Carlos, SP, Brazil
fcferrari@ufscar.br

Abstract—This paper presents an implementation and assessment of a framework named SiMut. The framework was introduced in a previous paper with the objective of helping reducing the cost for testing a program based on groups of similar programs previously tested with mutation. The implementation presented in the paper handles Java programs and includes a set of variants that relate to three types of program abstraction (original source code, processed source code, and internal complexity metrics), three similarity calculation strategies (clustering, information diversity, and plagiarism), and one mutation cost reduction approach (inspired by the One-Op mutation technique). Our evaluation encompasses 20 variant combinations, also referred to as *SiMut configurations*, and 35 small Java programs. A cross-comparison involving the formed clusters and a comparison with randomly formed clusters points to configurations that tend to reach high effectiveness in foreseeing the best mutation operators for untested programs.

Index Terms—mutation testing; cost reduction; program similarity

I. INTRODUCTION

Mutation testing [7, 11] is able to yield tests that detect more faults than tests derived with most other structural criteria [3, 10, 18, 19], and that can reveal real faults [2, 14]. Performing mutation testing consists in systematically generating slightly modified versions of a program p , called *mutants*. *Mutation operators* embed the modification rules. Ideally, each mutant is a faulty program and should behave differently from p for at least one test case. When the different behavior is observed, the mutant is said to be *killed* by the test case. Until then, the mutant remains *alive* and either an additional test case must be found to kill the mutant, or the mutant should be determined to be equivalent. An *equivalent mutant* behaves the same as p on all tests, and hence cannot be killed. The *mutation score* (MS) is the ratio of killed mutants with respect to the total number of generated mutants except the equivalent ones. MS is a value in the interval $[0, 1]$ that represents the quality of the test set with respect to a given group of mutants. A set of tests that kills all killable mutants of a particular mutation operator o_i is said to be *o_i -adequate*, and the mutation score for o_i (named *mutation score of o_i*) is the score of the *o_i -adequate* tests on all mutants created for the program.

Despite being effective, mutation testing is expensive due to the large number of mutants (even for small programs), and the manual analysis that is required for unkillable mutants (e.g. to decide if a mutant is equivalent to the program under test). Therefore, reducing the cost of mutation testing has been on

focus by the research community since the late 70's [20, 24]. Cost reduction is considered mandatory to allow for the adoption of mutation testing in industrial software projects. With that in mind, researchers have proposed many techniques to achieve this goal. Earlier, the techniques have been classified as “do fewer”, “do faster”, and “do smarter” [20], and more recently classified according to the typical cost reduction goals such as *reducing the number of mutants*, *automatically detecting equivalent mutants*, and *avoiding the creation of certain mutants* (e.g. *redundant and equivalent*) [24].

Selective mutation [21], evolutionary algorithms (e.g. based on machine learning [30]) and higher-order mutation [27] are examples of either classical or currently largely explored (or both) techniques for reducing the cost of mutation. However, these and many other techniques are difficult to generalize. For example, Kurtz et al. [17] found significant cost savings while investigating sufficient sets of mutation operators, but only if each program is analyzed individually; according to the authors, the ideal sufficient set is different for each program.

In this context, a relevant open question is whether the results from the application of a given cost reduction technique will hold true for new programs. In a prior study, Pizzoleto et al. [25] introduced the SiMut (Program Similarity to support Cost Reduction of Mutation) conceptual framework and hypothesized that through SiMut one might compute mutation cost reduction parameters based on a *reference group* G of programs that are similar to a new, untested program u ; then, testing u with the same cost reduction parameters would lead to consistent results for G and u . In this paper, this approach is called the *SiMut-based approach*. By cost reduction parameters we mean, for example, a sufficient set of mutation operators, or a machine learning algorithm trained to classify mutants (e.g. as killable, equivalent and redundant).

The preliminary implementation and experiments with SiMut [6, 25] were incipient. The implementation was realized as proof-of-concepts to allow exploiting ideas, whereas the experiments consisted in simple experimental setups and relied on imprecise data obtained from studies performed by others. This paper substantially evolves prior research [25] mainly by describing a well-designed implementation of SiMut and by reporting on a set of experiments with various SiMut configurations. Our current implementation handles Java programs and is able to handle three types of program abstraction (original source code, processed source code, and internal

complexity metrics), three similarity calculation strategies (clustering, information diversity, and plagiarism), and one mutation cost reduction technique (One-Op mutation [31]).

For evaluation purposes, the ability of the reference group G to precisely foresee the cost reduction parameters for u is compared with the parameters computed for other groups of programs that are less similar to u . In our experiments, we also use program groups formed randomly as a baseline for comparison of results. We call these groups *random clusters*. For each experiment run, 10 random clusters are formed with approximately 20% of the classes present in our database, and the average results obtained for the random clusters are used in the comparison. Our results indicate that, for particular SiMut configurations, the reference group helps identifying a small group of top-ranked mutation operators that are consistent with the best operators identified for the untested classes.

To the best of our knowledge,¹ the SiMut-based approach is the first one that combines the use of information from previously tested programs and program similarity to support the testing of new programs using mutation at reduced cost. Compared to previous research [6, 25] and to the state-of-the-art in mutation testing, this paper brings the following contributions:

- It presents an implementation of the conceptual SiMut framework that automates a program similarity-based approach to help reducing the cost of mutation testing.
- It experiments with 20 possible configurations of SiMut variants and reports on the achieved results.
- It points out the most promising configurations of SiMut for the prediction of the mutation operators with highest mutation scores for untested classes.

The next section presents basic background. Section III describes the *SiMut* implementation, and Section IV details our experiments. Section V analyzes the obtained results. Section VI discusses the threats to validity. Section VII summarizes related work, and Section VIII concludes the paper.

II. BACKGROUND

Conceptually, the application of SiMut relies on a set of programs P that are already tested with mutation [25]; that is, P only includes programs for which mutation-related artifacts such as test cases, mutants, and mutation execution results are available. Given P , an untested program u , and a particular program abstraction (*e.g.* internal metrics or processed source code) the approach embedded in SiMut consists in identifying the group of programs that are most similar to u . We call this group the *reference group* G . SiMut then computes the mutation cost reduction parameters for G and suggests the same parameters to be applied to u .

In what follows, we summarize the key elements of SiMut in terms of its main processes. A detailed description of the process is provided by Pizzoleto et al. [25]. In short, SiMut comprises an overall process and a key sub-process that is responsible for composing groups of similar programs. Both processes are depicted in Figure 1 (adapted from [25]).

¹We are currently updating a systematic literature review on mutation testing cost reduction [24] and preliminary results are consistent with previous observations regarding the originality of our research.

Overall process: The overall process encompasses the following five² sub-processes or activities:

- 1- *Compose reference group* G
- 2- *Select cost reduction technique* C
- 3- ³*Apply* C to G
- 4- ⁴*Retrieve cost reduction resulting parameters for* G
- 5- *Apply* C to untested program u using the computed or retrieved cost reduction parameters

Examples of *cost reduction parameters* are a sufficient set of mutation operators or a trained Machine Learning algorithm. These parameters will be applied to the untested program u .

Sub-process 1- *Compose reference group* G : It is split in four sub-processes or activities:

- 1.1- *Compute measures for untested program* u
- 1.2- *Select similarity approach* S
- 1.3- *Apply approach* S
- 1.4- *Generate reference group* G

Measures may be computed from varied program abstractions. Examples of *program abstractions* are static and dynamic internal metrics, a simplified source code, or a higher level software specification such as a state machine or a UML sequence diagram. Note that a program abstraction is used to simplify the process of analyzing complex programs. The abstraction tool creates a relationship between a concrete program and an abstract program. In the SiMut-based approach, the measures must also be available for the programs that compose the database, since they serve as input for the similarity calculation step. Thus, the measures for the programs in the database must be either computed in advanced (and be stored in the database), or be computed on-the-fly.

Finally, the *similarity approach* is intended to apply one or more similarity calculation algorithms and produce the reference group G of programs that are similar to u . The approaches currently embedded in the SiMut implementation are described in Section III.

III. IMPLEMENTATION

The SiMut implementation we describe in this section performs the steps described in Figure 1. The components that compute the abstractions, the measures and the similarity among programs are developed in Java, whereas the components that compute the mutation cost reduction are currently implemented in C# with future plans to be ported to Java, thus enabling SiMut to work on different operating systems. The Maven tool⁵ was chosen for project management, making it easier to resolve the dependencies (*i.e.* to import libraries). To be executed, SiMut receives as a parameter the path of an XML file that defines the intended SiMut execution. This XML file must inform: (i) the set T of tested classes from the database which will be used to compose the reference group G ;

²For experimental purposes, the overall process also includes a last sub-process named *Assess results for untested program* u . It aims to support the assessment of results produced with the SiMut-based approach.

³This is performed only if the cost reduction technique C has not yet been computed for the reference group G .

⁴This is performed only if the cost reduction technique C has already been computed for the reference group G and is stored for future use.

⁵<https://maven.apache.org/index.html> - accessed in Jan, 2022.

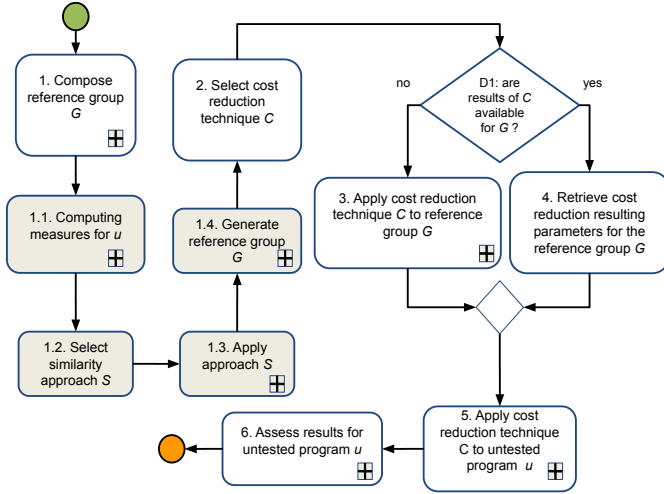


Fig. 1. Simplified view of the SiMut process

(ii) the untested classes u ; (iii) the abstraction A ; and (iv) the similarity computation approach S . It is also possible to indicate how many clusters the user wants as output. At the end of the execution, SiMut stores in the database the generated clusters, from which it is possible to identify the reference group G . Figure 2 presents the possibilities of abstraction and similarity approaches currently implemented in the framework. In our experiment, 20 possible SiMut configurations were performed and the tables of results are presented in Section V. Figure 3 shows a schematic view of the SiMut implementation. A description of the components is next provided.

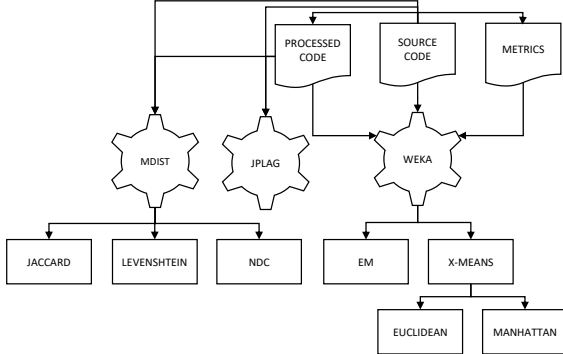


Fig. 2. SiMut possibilities tree.

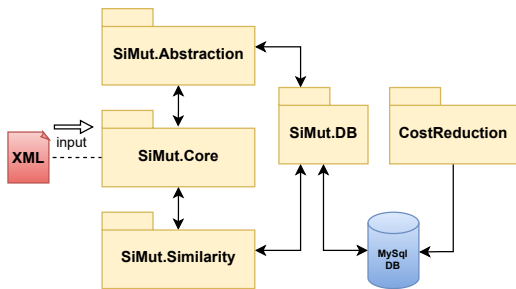


Fig. 3. SiMut architecture

The Abstraction component: It handles the computation of program abstractions and measures. Given the process shown Figure 1, the abstractions implemented for step 1.1 are:

- **Processed code:** Based on the application bytecode, a code obfuscator reduces the size of the selected classes. This reduction happens because, through the obfuscation process, the name of the variables can be shortened, and, according to the obfuscator, the code can go through an optimization process, hence reducing the size or complexity of some parts. The obfuscation results in a smaller bytecode that faithfully represents the original bytecode.
- **Original source code:** For the effectiveness comparison with a processed – *i.e.* obfuscated – source code, SiMut also allows for using the original source code as an abstraction to serve as input for the similarity step.
- **Internal metrics:** Given an application code, it is also possible to calculate several internal metrics, such as: number of lines of code, cyclomatic complexity of each method, and the number of methods in the class. Getting the value of one or more metrics makes it possible to characterize the structure of the selected source code.

The Abstraction component invokes the following tools to automate the calculation of the abstractions and measures:

- **ProGuard**⁶ (for obfuscating code): it is a tool that optimizes, shortens and obfuscates the bytecode of Java applications, thus considerably reducing the size of classes, reducing redundant codes or removing snippets of unused code. In our implementation, we used Proguard version 6.2.2. In the SiMut execution file, it is possible choose which actions should be performed by ProGuard. Through the Java Decompiler⁷ tool (version 1.1.3), SiMut decompiles the bytecode generated by ProGuard and uses the resulting, simplified source code in the similarity step.
- **CKJM extended**⁸ (for calculating internal metrics): it is able to calculate 19 internal metrics from the bytecode of the classes [13]. The metrics include the object-oriented metrics proposed by Chidamber and Kemerer [4]. In the SiMut execution file, it is possible to select which metrics should be used to calculate the similarity. In our implementation, we used CKJM-extended version 2.2.

The Similarity component: It is responsible for computing program similarity. As a way to identify classes that are more similar to the untested class (*i.e.* to reference group G indicated in Figure 1), clustering can be applied through unsupervised machine learning algorithms. With this technique, it is possible to identify the cluster to which the untested class u is allocated. Alternatively, to create the reference group G one may also use plagiarism identification tools in source code; in that case, a set of source codes is analyzed and at the end the percentage of similarity between them is presented, being the higher the percentage of identified plagiarism, the greater the similarity between the codes. As another alternative approach, we may compute information

⁶<https://www.guardsquare.com/proguard> - accessed in Jan, 2022.

⁷<http://java-decompiler.github.io/> - accessed in Jan, 2022.

⁸http://gromit.iar.pwr.wroc.pl/p_inf/ckjm/ - accessed in Jan, 2022.

diversity by receiving a set of texts and at the end of the process the distance between the texts is obtained; in this case, the smaller the diversity distance, the more similar are the texts compared. Using plagiarism identification tools and information diversity calculation algorithms, it is possible to rank programs and group them, thus enabling the identification of the reference group of the untested class.

For step 1.2 of the SiMut process (see Figure 1), according to the selected abstraction, SiMut allows the selection of a tool to identify more similar classes. If the choice is internal metrics, the measures are processed by the Weka library.⁹ If the chosen abstraction is either a processed code or the original source code, Weka, JPlag¹⁰ and mdist¹¹ may be used. Note that mdist is a Docker container invoked by SiMut. Those tools were integrated to our implementation because they are ongoing projects. Details of each are provided next.

- **Weka:** it is a library that includes Machine Learning algorithms for data mining tasks.¹² It provides resources for automatic classification, regression, clustering and feature selection. Through Weka, SiMut enables class clustering using the X-Means [23] or the Expectation Maximization (EM) [8] algorithms. Both are unsupervised machine learning algorithms that generate K clusters of a dataset. As shown in Figure 2, it is possible to choose the Manhattan or Euclidean distance algorithms used by X-Means; this is defined in the SiMut XML execution file. In our implementation, we used Weka version 6.8.
- **mdist:** it is a tool capable of calculating distances among strings. These distances can show how similar or different two strings are. mdist contains several distance functions, such as: Levenshtein, Jaccard, and NCD (Normalized Compression Distance) [9]. In the SiMut execution file it is possible to define the desired distance. In our implementation, we used mdist version 0.1.7.
- **JPlag:** it is a plagiarism detection tool that identifies similarities between source code files. This tool does not simply compare the bytes of the texts, but it can also identify and compare the syntax of the programming language of the selected files. In doing so, being the code the same or not, if the logic is the same, JPlag is able to identify similarity [28]. In our implementation, we used version JPlag 2.12.1.

As ProGuard and CKJM extended process Java bytecode, it is a prerequisite that the classes selected for execution are inside a Maven project, thus SiMut automatically compiles the programs to get the bytecode of the selected classes. All programs, source code of classes and processed data are stored in a MySQL relational database, thus allowing queries to retrieve the reference group of the class under analysis. Furthermore, with the purpose of optimizing the execution, the tool automatically reuses the calculated abstractions for classes already used in previous executions.

The Core component: It is responsible for handling the XML execution file and starting the desired computations. In the XML file it is possible to define the number of clusters desired and, in the case of mdist or JPlag, it is possible to define the percentage of classes that each cluster should have. However, if any Weka clustering algorithm is selected, it is not possible to define the percentage, since the clustering algorithm itself will form the clusters. In that case, in the XML file it is possible to define a limit maximum percentage of classes that each cluster can have; if the cluster exceeds it, a reclustering is automatically performed, generating a final number of clusters given by the formula $(N * 2) - 1$, where N is the number of groups defined in the XML execution file.

Figure 4 shows an example of an input XML execution file. In this example, the chosen abstraction is internal metrics (to be gathered with the CKJM Extended tool), the specific metrics are listed in the options group, and the similarity approach is the clustering (to be computed with the X-means algorithm of Weka). Other details of the XML are not described here due to space limitations.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<project description="Example" name="Example">
  <programs>
    <program name="testedProgram1" path="c:/testedProgram1" />
    <program name="testedProgram2" path="c:/testedProgram2" />
    <program name="untestedProgram" path="c:/untestedProgram" />
  </programs>
  <abstractions>
    <abstraction name="CKJM-Extended">
      <options>
        <option name="dit"/>
        <option name="noc"/>
        <option name="cbo"/>
        <option name="rfc"/>
        <option name="lcom"/>
      </options>
    </abstraction>
  </abstractions>
  <similarities groups="3">
    <sources>
      <source name="CKJM-Extended"/>
    </sources>
    <nprograms>
      <nprogram name="untestedProgram">
        <class name="untestedClass"/>
      </nprogram>
    </nprograms>
    <similarity algorithm="xmeans" name="weka">
      <group level="0"/>
      <group level="1"/>
      <group level="2"/>
    </similarity>
  </similarities>
</project>
```

Fig. 4. Input XML example

The CostReduction component: This component is currently implemented in C# and is responsible for computing the steps related to the mutation cost reduction embedded in SiMut. After obtaining the distribution of the classes in their respective clusters, the next step is to apply the selected mutation cost reduction technique (step 3 in Figure 1).¹³ The currently implemented technique is the One-op mutation [31] that, in simple words, identifies the mutation operator with the best score. To obtain the mutation scores, the database developed and updated by Pizzoleto et al. [26] was used. The

¹³The current SiMut implementation does not include functionalities to perform step 4 of the overall process.

⁹<https://www.cs.waikato.ac.nz/ml/weka/> - accessed in Jan, 2022.

¹⁰<https://jplag.ipd.kit.edu/> - accessed in Jan, 2022.

¹¹<https://hub.docker.com/r/robertfeldt/mdist> - accessed in Jan, 2022.

¹²To use Weka algorithms, the "StringToWordVector" process is performed, which converts the source code into a set of numeric attributes representing word occurrence information from the text contained in the source code.

artifacts used in this step are the same as those configured in the XML execution files. By extracting data from the database, the `CostReduction` component dynamically creates a killing matrix that encompasses mutants and test cases for all the classes handled in the SiMut execution. Among the information that can be extracted from the killing matrix, the following can be highlighted: the mutation score of a particular program, for a group (*e.g.* a cluster) of programs, and for all programs in the database; and the mutation score of a particular mutation operator, considering a specific program or groups of programs. All generated results are stored in CSV files, so for each of the runs a CSV file is generated that can be further processed by other tools.

IV. EXPERIMENT

This section presents the goal, the set of used programs, the combinations of abstractions and similarities selected for the experiments, and the procedures for generating and analyzing the results. All used artifacts are available online.¹⁴

A. Study Goal and Research Questions

Previous studies [6, 25] produced preliminary evidence that the use of similarity can bring benefits for mutation testing cost reduction. In the experiment described in this section, our objective is to analyze the benefits that different abstractions and similarity techniques can bring. For that, we exploit a variety of SiMut execution configurations to answer the following main research question: *Does the use software abstractions induce the formation of groups of similar programs, which consequently can bring benefits to reduce the cost of mutation testing?* We organize our discussion in terms of three more specific questions, as follows:

- **RQ1** - Does the use of the complexity internal metrics abstraction induce the formation of groups of similar programs, which consequently can bring benefits to reduce the cost of mutation testing?
- **RQ2** - Does the use of the source code (either original or processed) abstraction induce the formation of groups of similar programs, which consequently can bring benefits to reduce the cost of mutation testing?
- **RQ3** - Does the use of alternative source code representations (*e.g.* complexity internal metrics or obfuscated code) lead to a more precise formation of groups of similar programs when compared to the original source code, which consequently can bring benefits to reduce the cost of mutation testing?

B. Used Dataset

We selected 35 out of 38 programs used by Dallilo et al. [6]. They are small programs mostly consisting of a single Java class. Only 35 programs were used as they were buildable projects. We combined all programs in a single Maven project, called *simplePrograms*, thus facilitating its operation in the framework and also the generation of test cases and mutants.

Test cases were generated by EvoSuite¹⁵ version 1.0.6. Mutants were generated and executed by PIT¹⁶ version 1.6.2. Both tools are widely used for experimentation in software testing research. More details about the used database can be found elsewhere [26].

Table I shows descriptive metrics of the programs and test cases. It is divided into two parts: Application Code, which concerns the application source code, and EvoSuite/PIT, which concerns tests and mutants generated by EvoSuite and PIT, respectively. In the table, TLOC is the total lines of code, NOM is the number of methods, CC is the average cyclomatic complexity of the class methods, TCs represents the total test cases generated for the class, M means the amount of mutants generated by PIT, and MS represents the mutation score for the class. The application code metrics were calculated using the Eclipse Metrics plugin¹⁷ version 1.3.8.

TABLE I
DESCRIPTIVE METRICS OF THE USED PROGRAMS.

Application / Class	Application Code			EvoSuite/PIT		
	TLOC	NOM	CC	# TCs	# M	MS (%)
DigitReverser	25	2	2.000	7	29	27.58
QuickSort	45	4	3.750	4	96	41.66
power	45	2	4.000	10	42	50.00
findVal	58	3	3.333	9	46	47.84
Bisect	30	3	2.000	3	30	73.33
Heap	48	8	2.125	26	100	74.00
TestPat	34	2	3.500	7	35	91.42
countPositive	38	2	3.500	7	34	61.76
twoPred	38	2	4.000	9	39	51.28
numZero	37	2	3.500	8	33	60.60
InversePermutation	28	2	4.000	7	62	56.45
stats	53	2	2.500	6	89	47.19
Triangle	50	9	2.111	29	139	87.76
CheckPalindrome	24	2	2.500	9	38	57.89
cal	97	3	7.000	15	147	16.32
Find	52	4	3.250	12	84	69.04
Triangulo	88	5	5.000	25	122	85.24
Fourballs	33	2	2.500	10	43	95.34
printPrimes	54	3	3.000	9	46	41.30
MergeSort	37	3	3.333	8	95	48.42
oddOrPos	38	2	4.000	8	40	57.50
RecursiveSelectionSort	27	3	2.333	8	55	36.36
findLast	61	3	3.333	9	48	50.00
sum	35	2	3.000	8	29	55.17
lastZero	37	2	3.500	8	32	59.37
Gaussian	40	7	1.714	25	96	89.58
checkIt	29	2	3.500	9	28	64.28
Mid	60	9	1.667	18	51	84.31
Queue	63	6	1.833	9	85	65.88
BubCorrecto	51	7	1.571	11	48	64.58
BoundedQueue	63	6	2.000	17	83	89.15
trashAndTakeOut	27	2	2.500	9	33	81.81
Calculation	79	9	3.222	38	201	31.84
LRS	33	3	2.667	9	52	94.23
Stack	37	7	1.714	18	58	84.48
AVG	45.543	3.857	3.013	12.114	-	62.66

The number of test cases for each class ranges from 3 to 29, whereas the number of mutants ranges from 28 to 201, and the mutation score ranges from 16.32% to 95.34%, with an average of 62.66%. As reported by Pizzoleto et al. [26], EvoSuite was configured to apply the complete set of test coverage generation criteria [29], whereas PIT was configured to apply the full set of 28 mutation operators. We did not add any manually created test cases nor identified any equivalent mutants (more details in Section VI).

C. Setup of SiMut Variants

The choice of selected combinations to run in SiMut was based on the research questions, as follows:.

¹⁵<https://www.evosuite.org/> - accessed in Jan, 2022.

¹⁶<http://pitest.org/> - accessed in Jan, 2022.

¹⁷<http://metrics2.sourceforge.net/> - accessed in Jan, 2022.

¹⁴<https://tinyurl.com/mutation2022>

RQ1: Table II lists seven metrics (out of 19) that are calculated by CKJM extended. Those seven metrics are considered for some experiment executions. These executions are labeled with *Selected metrics* in tables IV and V that summarize the achieved results. We selected those metrics because most programs used in the experiments do not have object-oriented characteristics (*e.g.* inheritance and encapsulation), and hence might not be appropriate targets of other metrics implemented by the tool. To form the groups, the Weka X-Means and EM algorithms were selected. For X-Means, both distance algorithms were applied (Euclidean and Manhattan). These techniques were selected because they have already been used in previous experiments in the area of software testing [5, 6]. Executions were also created considering all the metrics available in CKJM extended to verify if the choice based on the characteristics of the classes would yield more accurate results.

TABLE II
SELECTED METRICS COMPUTED BY CKJM EXTENDED.

METRIC ID	METRIC NAME
WMC	Weighted methods per class
LCOM	Lack of cohesion in methods
LCOM3	Lack of cohesion in methods Henderson-Sellers version
LOC	Lines of Code
CAM	Cohesion Among Methods of Class
AMC	Average Method Complexity
CC	McCabe's Cyclomatic Complexity

RQ2: The shrinker, optimizer and obfuscator features of ProGuard were applied to the bytecode of the programs. In our experiment executions, to form the clusters the source code processed by ProGuard was used as input to the following tools: mdist, JPlag and Weka. For mdist, we applied Levenshtein, Jaccard and NCD distance functions. For Weka, we applied EM and X-Means (with its both distance algorithms).

RQ3: The same SiMut configurations we used for processed code were used to form clusters based on the original source code of the programs. This enabled us compare the relevance of the processed source code in the SiMut-based approach.

We defined the creation of three clusters for all executions. For executions that used the mdist or JPlag similarity techniques, we defined the following cluster sizes: 20% for the cluster to which the untested class u was allocated, 30% for an intermediate cluster (*i.e.* the set of classes that are moderately similar to u), and 50% for the cluster with the most different classes with respect to u . In experiments that applied Weka, in which we could not indicate the percentage of classes that each cluster would contain, as a way to avoid clusters with many elements, a recluster was automatically executed in the event of a cluster with more than 60% of classes. This recluster occurs only once, so if it results in any new cluster with more than 60% elements, it would not be reclustered again.

For providing a baseline for comparison of the effectiveness of the SiMut-based approach, we randomly formed 10 clusters; each of these *random clusters* is respective to one of the 35 programs used in our experiment (350 clusters, in total). These random clusters contains approximately 20% of the programs and are used across the 20 SiMut configurations. How the results are interpreted is discussed in Section IV-D.

D. Procedures for Generation and Analysis of Results

For experimental purposes, each program in our database was considered an untested program u . This allows us to simulate a real case scenario in which a new class is fed to SiMut, and the framework helps the tester to apply mutation testing at reduced cost. Since we do have mutation-related artifacts for u , we are able to assess the results in case the same cost reduction parameters are applied to u .

To analyze the obtained results and identify the combinations that bring more benefits, we adopted the same interpretation model used by Dallilo et al. [6]. For each program and for each SiMut configuration, a results table was created to store and list of best mutation operators and to show the overlapping of results. Table III exemplifies the results for one of the experiment iterations. The elements shown in the table are defined as follows:

- u : represents the program considered untested in a given iteration
 - C : represents a cluster. In particular,
 - C_u is the cluster to which u is associated (except u); and
 - C_i ($i=1..4$) is one of the remaining clusters.
 - P_u : represents all programs in the database, except u
 - Ran_i : represents randomly formed clusters, except u
 - O : represents a list of one-op candidates. In particular,
 - O_u is the list for u
 - OC_u is the list for C_u
 - OC_i is the list for C_i
 - $ORan_i$ ($i=1..10$) is the list for Ran_i
 - and OP_u is the list for P_u
 - I : represents the size of the intersection between a given list of one-op candidates and the list of one-op candidates for u . In particular,
 - IC_u is the size of the intersection between O_u and OC_u
 - IP_u is the size of the intersection between O_u and OP_u
 - $IRanAvg_u$ is the mean of the size of the intersection of O_u with all $ORan_i$ ($i = 1..10$)
 - IC_i ($i=1..4$) is the size of the intersection between O_u and a particular OC_i
 - m : represents a function that gets the bigger value between a group of values
- A numeric value in parentheses is the size of the intersection. $R1$, $R2$ and $R3$, defined in the sequence, may have a positive (V), neutral (N), or negative (X) result.
- $R1$: compares the results obtained for C_u with respect to the other formed clusters (*i.e.* C_i). The result is positive if IC_u is higher than all other results for IC_i ; negative if IC_u is lower than any other result for IC_i ; or neutral if the results are the same in all cases.
 - $R2$: similarly to $R1$, it compares the results obtained for C_u with respect to the complete set of programs except u (*i.e.* P_u). Positive, neutral and negative results are equally computed as done for $R1$.
 - $R3$: similarly to $R1$ and $R2$, it compares the results obtained for C_u with respect to the mean result obtained for the random clusters (*i.e.* Ran_i).

TABLE III
EXCERPT OF DETAILED RESULTS.

Combination - Original code / mdist - Levenshtein	
u	numZero
O_u	ConstructorCallMutator, RemoveConditionalMutator_ORDER_ELSE, ReturnValsMutator, IncrementsMutator, RemoveIncrementsMutator
OC_u	ConstructorCallMutator, ReturnValsMutator, IncrementsMutator, RemoveIncrementsMutator, ConditionalsBoundaryMutator
OP_u	ConstructorCallMutator, IncrementsMutator, RemoveIncrementsMutator, NonVoidMethodCallMutator
OC_1	MathMutator, ConstructorCallMutator, IncrementsMutator, ReturnValsMutator
OC_2	ConstructorCallMutator, IncrementsMutator, RemoveIncrementsMutator, NonVoidMethodCallMutator
R1	V: $IC_u(4) > m(IC_1(3), IC_2(3))$
R2	V: $IC_u(4) > IP_u(3)$
$ORan_1$	ConstructorCallMutator, IncrementsMutator, RemoveIncrementsMutator, MathMutator
$ORan_2$	ConstructorCallMutator, NonVoidMethodCallMutator, NegateConditionalsMutator, IncrementsMutator
$ORan_3$	ConstructorCallMutator, MemberVariableMutator, ReturnValsMutator, NonVoidMethodCallMutator
$ORan_4$	ConstructorCallMutator, IncrementsMutator, RemoveIncrementsMutator, MemberVariableMutator
$ORan_5$	ConstructorCallMutator, RemoveIncrementsMutator, IncrementsMutator, MathMutator
$ORan_6$	ConstructorCallMutator, IncrementsMutator, MathMutator, ReturnValsMutator
$ORan_7$	ConstructorCallMutator, IncrementsMutator, ReturnValsMutator, RemoveIncrementsMutator
$ORan_8$	ConstructorCallMutator, IncrementsMutator, RemoveIncrementsMutator, ReturnValsMutator
$ORan_9$	ConstructorCallMutator, RemoveIncrementsMutator, IncrementsMutator, RemoveConditionalMutator_EQUAL_IF
$ORan_{10}$	ConstructorCallMutator, RemoveConditionalMutator_ORDER_ELSE, IncrementsMutator, MathMutator
$IRanAvg_u$	3
R3	V: $IC_u(4) > IRanAvg_u(3)$

Table III shows results for one iteration of a SiMut configuration. For the numZero program, the results are:

- O_u has 5 operators (see list in the table);
- OC_u has 5 operators (see list in the table);
- OP_u has 4 operators (see list in the table);
- R1: has positive result, given that IC_u is larger than $m(IC_1, IC_2)$
- R2: has positive result, given that IC_u is larger than IP_u .
- $IRanAvg_u$: is equal to 3 because this is the result of the average of all intersections between O_u with all $ORan_i(i=1..10)$
- R3: has positive result, given that $IC_u > IRanAvg_u$

This process was performed for all classes in all combinations mentioned in the previous subsection. The next section describes and analyzes the achieve results.

V. RESULTS AND ANALYSIS

We split our experiment runs in two parts. The first considers, for each SiMut configuration, a single best mutation operator, whereas the second considers a group with the four top-ranked mutation operators based in their mutation score.

Results for a single best operator: The first part of the experiment relied on the the best operator for each group of programs, namely, the program u , the cluster C_u the clusters C_i , the complete set of programs P , and the 10 randomly formed clusters $Ran_i(i = 1..10)$. Table IV presents the results. Once again, note that the program groups do not include u which is handled as an untested program in our experiments.

We noticed that the vast majority of intersections were either empty or of minimal size. In-depth analysis pointed to one predominant mutation operator (namely, *ConstructorCallMutator*). This naturally turned out to be the single operator present in OC_u and OP_u on most occasions. In Table IV it is possible to verify the neutral result as the most frequent in the majority of the SiMut configurations. especially in respect to R2 results, where IC_u is compared with IP_u . This behavior also impacted the results regarding R1 and R3; for example, using a code processed by ProGuard as an abstraction and the

NCD as a similarity measure computed by mdist, in R1 and R3 there were 31 and 22 neutral results, respectively.

Results for the four best operators: Given the inconclusive results when we considered a single best mutation operator, we decided to expand the group of best operators; more specifically, we ranked the mutation operators according to their mutation score for the considered set of programs (*e.g.*, u , C_u etc.) and picked 25% of the operators with the highest scores. Given that 17 PIT mutation operators produced mutants for the target programs, the groups of best operators had four operators each. In case of ties between operators, those groups could be larger than that. The increase in the number of the best operators in each group allowed for a better balance in the results, with higher amounts of positive and negative results. Table V presents the results using this approach.

The analysis presented in the next section relies on the second set of results, *i.e.* results for the four best operators.

A. Analysis

R1 evaluates whether C_u (*i.e.* the cluster to which the class u is associated) is a better (or at least as good as) predictor of mutation operator scores than the other formed clusters C_i . Therefore, a neutral result can be considered satisfactory for R1. Likewise, a neutral result for R2 is also satisfactory, given that C_u is a mutation operator score predictor as good as the full set of programs, since computing the best operators for a single cluster is less costly than for the set of programs as a whole. For R3, a neutral result is clearly unsatisfactory. This is due to the low cost to randomly form the cluster without the need of computing abstractions, measures and similarity. Based on this interpretation rationale, Table VI presents a ranking in which the values for R1, R2, and R3 may range from 1 to 35. The values of columns R1 and R2 are the sum of the positive and neutral results, column R3 accounts only for positive results, and column T sums up R1, R2 and R3.

Analysis based on R1, R2 and R3, in conjunction: Considering the three sets of results summarized in Table VI, it is possible to notice that a substantial set of SiMut config-

TABLE IV
SUMMARY OF RESULTS WHEN WE CONSIDER ONLY THE BEST OPERATOR FOR EACH GROUP OF PROGRAMS.

Selected metrics / X-Means - Euclidean	Processed code / X-Means - Manhattan	All metrics / EM	All metrics / X-Means - Euclidean
R1 V(11), N(12), X(12)	R1 V(15), N(14), X(6)	R1 V(10), N(18), X(7)	R1 V(0), N(32), X(3)
R2 V(2), N(25), X(8)	R2 V(0), N(33), X(2)	R2 V(1), N(29), X(5)	R2 V(0), N(32), X(3)
R3 V(8), N(18), X(9)	R3 V(10), N(22), X(3)	R3 V(9), N(20), X(6)	R3 V(10), N(21), X(4)
All metrics / X-Means - Manhattan	Selected metrics / EM	Selected metrics / X-Means - Manhattan	Original code / JPlag
R1 V(0), N(33), X(2)	R1 V(0), N(34), X(1)	R1 V(0), N(34), X(1)	R1 V(1), N(34), X(0)
R2 V(0), N(33), X(2)	R2 V(0), N(34), X(1)	R2 V(0), N(34), X(1)	R2 V(0), N(35), X(0)
R3 V(11), N(21), X(3)	R3 V(11), N(22), X(2)	R3 V(11), N(22), X(2)	R3 V(12), N(22), X(1)
Original code / mdist - Jaccard	Original code / mdist - Levenshtein	Original code / EM	Original code / X-Means - Euclidean
R1 V(0), N(33), X(2)	R1 V(7), N(24), X(4)	R1 V(17), N(17), X(1)	R1 V(17), N(14), X(4)
R2 V(0), N(34), X(1)	R2 V(0), N(32), X(3)	R2 V(0), N(35), X(0)	R2 V(0), N(35), X(0)
R3 V(11), N(22), X(2)	R3 V(10), N(21), X(4)	R3 V(12), N(22), X(1)	R3 V(12), N(22), X(1)
Original code / X-Means - Manhattan	Processed code / JPlag	Processed code / mdist - Jaccard	Processed code / mdist - Levenshtein
R1 V(17), N(14), X(4)	R1 V(0), N(35), X(0)	R1 V(0), N(35), X(0)	R1 V(10), N(22), X(3)
R2 V(0), N(35), X(0)	R2 V(0), N(35), X(0)	R2 V(0), N(35), X(0)	R2 V(2), N(30), X(3)
R3 V(12), N(22), X(1)	R3 V(12), N(22), X(1)	R3 V(12), N(22), X(1)	R3 V(12), N(20), X(3)
Processed code / EM	Processed code / X-Means - Euclidean	Processed code / mdist - NCD	Original code / mdist - NCD
R1 V(17), N(13), X(5)	R1 V(16), N(13), X(6)	R1 V(3), N(31), X(1)	R1 V(2), N(32), X(1)
R2 V(0), N(35), X(0)	R2 V(0), N(34), X(1)	R2 V(0), N(35), X(0)	R2 V(0), N(34), X(1)
R3 V(12), N(22), X(1)	R3 V(11), N(22), X(2)	R3 V(12), N(22), X(1)	R3 V(11), N(22), X(2)

TABLE V
SUMMARY OF RESULTS WHEN WE CONSIDER THE FOUR BEST OPERATORS FOR EACH GROUP OF PROGRAMS.

Selected metrics / X-Means - Euclidean	Processed code / X-Means - Manhattan	All metrics / EM	All metrics / X-Means - Euclidean
R1 V(19), N(3), X(13)	R1 V(17), N(1), X(17)	R1 V(21), N(2), X(12)	R1 V(21), N(1), X(13)
R2 V(16), N(13), X(6)	R2 V(12), N(13), X(10)	R2 V(19), N(10), X(6)	R2 V(17), N(12), X(6)
R3 V(21), N(3), X(11)	R3 V(15), N(2), X(18)	R3 V(22), N(3), X(10)	R3 V(22), N(1), X(12)
All metrics / X-Means - Manhattan	Selected metrics / EM	Selected metrics / X-Means - Manhattan	Original code / JPlag
R1 V(21), N(0), X(14)	R1 V(21), N(4), X(10)	R1 V(20), N(4), X(11)	R1 V(19), N(5), X(11)
R2 V(17), N(12), X(6)	R2 V(17), N(14), X(4)	R2 V(16), N(15), X(4)	R2 V(17), N(10), X(8)
R3 V(21), N(1), X(13)	R3 V(23), N(3), X(9)	R3 V(22), N(3), X(10)	R3 V(19), N(1), X(15)
Original code / mdist - Jaccard	Original code / mdist - Levenshtein	Original code / EM	Original code / X-Means - Euclidean
R1 V(19), N(7), X(9)	R1 V(21), N(6), X(8)	R1 V(21), N(2), X(12)	R1 V(18), N(2), X(15)
R2 V(17), N(15), X(3)	R2 V(19), N(12), X(4)	R2 V(16), N(11), X(8)	R2 V(11), N(22), X(2)
R3 V(21), N(2), X(12)	R3 V(22), N(3), X(10)	R3 V(19), N(2), X(14)	R3 V(20), N(4), X(11)
Original code / X-Means - Manhattan	Processed code / JPlag	Processed code / mdist - Jaccard	Processed code / mdist - Levenshtein
R1 V(19), N(1), X(15)	R1 V(17), N(5), X(13)	R1 V(23), N(6), X(6)	R1 V(22), N(4), X(9)
R2 V(13), N(20), X(2)	R2 V(16), N(11), X(8)	R2 V(20), N(12), X(3)	R2 V(22), N(7), X(6)
R3 V(22), N(3), X(10)	R3 V(17), N(3), X(15)	R3 V(27), N(0), X(8)	R3 V(24), N(0), X(11)
Processed code / EM	Processed code / X-Means - Euclidean	Processed code / mdist - NCD	Original code / mdist - NCD
R1 V(19), N(0), X(16)	R1 V(17), N(0), X(18)	R1 V(19), N(7), X(9)	R1 V(19), N(10), X(6)
R2 V(14), N(11), X(10)	R2 V(13), N(12), X(10)	R2 V(19), N(11), X(5)	R2 V(19), N(13), X(3)
R3 V(15), N(2), X(18)	R3 V(15), N(2), X(18)	R3 V(23), N(1), X(11)	R3 V(24), N(3), X(8)

TABLE VI
RANKED RESULTS (R_i MAY RANGE FROM 1 TO 35).

#	COMBINATION	R1	R2	R3	T
1	Processed code / mdist - Jaccard	29	32	27	88
2	Original code / mdist - NCD	29	32	24	85
3	Original code / mdist - Levenshtein	27	31	22	80
4	Selected metrics / EM	25	31	23	79
5	Original code / mdist - Jaccard	26	32	21	79
6	Processed code / mdist - Levenshtein	26	29	24	79
7	Processed code / mdist - NCD	26	30	23	79
8	Selected metrics / X-Means - Manhattan	24	31	22	77
9	Original code / X-Means - Manhattan	20	33	22	75
10	All metrics / EM	23	29	22	74
11	All metrics / X-Means - Euclidean	22	29	22	73
12	Original code / X-Means - Euclidean	20	33	20	73
13	Selected metrics / X-Means - Euclidean	22	29	21	72
14	All metrics / X-Means - Manhattan	21	29	21	71
15	Original code / JPlag	24	27	19	70
16	Original code / EM	23	27	19	69
17	Processed code / JPlag	22	27	17	66
18	Processed code / EM	19	25	15	59
19	Processed code / X-Means - Manhattan	18	25	15	58
20	Processed code / X-Means - Euclidean	17	25	15	57

urations led to promisingly good results. Examples are the configurations *Processed code / mdist - Jaccard* and *Original code / mdist - NCD*, which, respectively, summed up 88 and 85 positive results from a maximum of 105 possible.

This initial analysis suggests that similarity can indeed be an ally to reduce the cost of mutation testing. Particularly,

the configuration that combined code processed by ProGuard and similarity computed with the Jaccard distance measure implemented by the mdist tool presented the best performance in relation to the others (effectiveness of approximately 84% to indicate the best mutation operators). We also highlight the configurations that yielded at least 79 positive results, since this value represents an effectiveness of at least 75%.

Analysis based on R1, R2 and R3, individually: Regarding R1, we observe some configurations had the effectiveness above 75%, all including mdist as the similarity tool (even though with varying abstractions and distance measures). They are (i) *Processed code / mdist - Jaccard*, (ii) *Original code / mdist - NCD*, and (iii) *Original code / mdist - Levenshtein*. Many others have effectiveness above 60%. Regarding R2, except for the last three configurations shown in Table VI, all configurations presented effectiveness above 75% to better foresee the best mutation operators when compared to the whole set of programs. Finally, considering only R3, the effectiveness of five configurations was 66% or higher, and for three configurations it was above 69%, namely (i) *Processed code / mdist - Jaccard*, (ii) *Original code / mdist - NCD*, and (iii) *Processed code / mdist - Levenshtein*. Once again, all configurations that stood out when compared to random clusters include mdist as the similarity tool.

B. Revisiting the Research Questions

Regarding RQ1, the configurations that used internal metrics yielded varied results. Among them, (i) *Selected metrics / EM* and (ii) *Selected metrics / X-Means - Manhattan* presented the best results, reaching, respectively, approximately 75% and 73% effectiveness. These two best configurations used only seven metrics computed by CKJM extended. All other configurations that also used internal metrics as a form of abstraction achieved effectiveness greater than 67%.

Therefore, for RQ1 we conclude that the use of internal complexity metrics as a form of abstraction, combined with clustering algorithms, **can positively** induce the formation of groups of similar programs that contributes for the cost reduction of mutation testing.

The configurations that used source code as the software abstraction produced varied results according to the applied similarity technique. In particular, the formation of groups of similar programs by the mdist tool showed to be promising with both processed and original code; in all executions that used mdist, the effectiveness was above 75%. When Weka was used to group programs based on their source code, the highest reached effectiveness (71%) came from the *Original code / X-Means - Manhattan* configuration, followed by *Original code / X-Means - Euclidean* (effectiveness of 70%). Other Weka-based configurations to process source code abstractions did not produce significant results. When using JPlag in the similarity step, the effectiveness ranged from 63% to 67%.

Regarding RQ2, we conclude that the use of the source code abstraction (either original or processed), for particular similarity tools/approaches (namely, mdist and Weka-X-Means), **can positively** induce the formation of similar groups that contributes for the cost reduction of mutation testing.

When we compare the results for configurations that used alternative source code representations (namely, complexity internal metrics and obfuscated code) with the ones that used original code, we could not observe a clear advantage for any. According to our results, while processed code is present in the most effective configuration (note that in Table VI, original code appears immediately below in terms of effectiveness (note that both abstractions are processed by the mdist tool). Indeed, we see an alternation of types of abstraction in the table of ranked results. Thus, there is no clear trend for the advantageous use of any alternative source code representation beyond the original source code.

Regarding RQ3, we conclude that the original source code, when used in a similarity-based approach as the one defined in the SiMut framework, **can be as effective as** some alternative form of source code representation.

VI. THREATS TO VALIDITY

This section discusses the threats to the validity of our experiment according to three categories. as follows:

Internal: When using EM and X-Means clustering algorithms from Weka, there is no guarantee that the cluster division is optimal in terms of proximity between the class under analysis and the central cluster element (*i.e.* the centroid). Furthermore, the quality of the database of artifacts (programs, test cases, mutants etc.) used in the experiments relies on the accuracy of the procedures and tools used to create it. To reduce threats of this category, we applied the same clustering algorithms used in previous experiments in the context of software testing [5, 6]. Besides that, we used a database that has been systematically created with widely used tools for test generation and mutant generation and execution [25].

Construct: The choice of the minimum number of clusters for all experiments was defined by the authors. This choice, although satisfactory according to the achieved results, may not represent the best option. The percentage of programs to compose each cluster, when applicable, was also a decision that may not represent the best setting. Different numbers of clusters and different sizes of clusters might have led to different results. Another threat concerns the use of programs and associated artifacts (*i.e.* test cases and mutants) that are available in our database “*as is*”, that is, as generated by EvoSuite and PIT, without any further test case addition or equivalent mutant detection. We believe this approach brings a more “practical” nature to our study, making it comparable to a scenario in which software engineers adopt a fully automated process to apply mutation testing to a new program.

External: The experiments reported in this paper were carried out based on 35 simple programs. Therefore, our results may not translate to other settings that, for instance, use different target programs (*e.g.* more complex class structures that involve inheritance or polymorphism relationships), or even settings that use other software abstractions and similarity calculation algorithms and tools.

VII. RELATED WORK

This section presents some of the most recent studies that have relied on similarity as a way of aiding in software testing and, more specifically, in reducing the cost of mutation testing.

Semantic similarity has been used to help identifying equivalent mutants. Jammalamadaka and Parveen [12] proposed a hybrid strategy known as *hybrid wavelet convolutional rain optimization* (HWCRO) to classify the equivalent mutants. For this, features such as semantic similarity and entropy of information between codes and mutants are extracted, then these features are used in a wavelet convolutional neural network (wCNN) classifier. This classifier, after going through performance and efficiency optimizations, says whether the mutant is equivalent (or not) to the original program.

Other studies [15, 16, 22] have also used similarity as a resource to identify equivalent mutants and redundant mutants. A key difference between our study and the others [12, 15, 16, 22] relies on the goal to be achieved with the application of program similarity. While other approaches aim to compare mutants with the original program and among themselves, we compare programs that are (or will be tested) with mutation.

Cruz and Eler [5] explored program clustering based on CK metrics to estimate program testability. They studied Java programs and generated clusters using the EM algorithm, based on the CBO, LCOM, RFC, and WMC metrics. The testability of a class was characterized as high or low, according to the structural coverage and mutation score produced by the associated test sets. The precision of the EM algorithm was assessed with the kNN (*k Nearest Neighbors*) algorithm [1]. In particular, Cruz and Eler used $k = 3$ and in 81.5% of cases, kNN confirmed the clusters produced by the EM algorithm precisely grouped classes with the same level of testability. In our work, the EM algorithm also contributed for the formation of relevant clusters that tend to bring benefits with respect to the approach embedded in the SiMut framework.

VIII. CONCLUSIONS AND FUTURE WORK

In this paper we presented an implementation of SiMut. It consists in a framework that aims to help reducing the cost of mutation testing based on results available for groups of similar programs already tested with mutation. Our implementation includes variants for computing program abstractions and for calculating similarity among programs, and supports a mutation cost reduction technique that identifies the best mutation operators according to their mutation scores.

Our experiments included 20 combinations of variants and 35 Java programs for which mutation-related information was available. The results point to a number of combinations of variants that are effective to foresee the best mutation operators for untested classes. Regarding the used program abstractions to compute similarity, the source code seems to be as relevant as the obfuscated code and complexity internal metrics.

As a last point, we highlight that some algorithms that may be integrated into SiMut consists of machine learning algorithms (e.g. the ones implemented in Weka). In spite of that, as shown in this paper, SiMut does not restrict other algorithms to be applied to compute the program similarity.

In future work we intend to enlarge the program database with more complex programs, thus allowing us to reproduce this experiment with a larger dataset. We also plan to explore other cost reduction techniques (apart from One-op) such as selective mutation and techniques that classify mutants based on machine learning algorithms.

ACKNOWLEDGEMENTS

This work has been financially supported by Federal University of São Carlos, Brazil, Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, and Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP, Brazil) - grant #2016/21251-0.,

REFERENCES

- [1] D. W. Aha, D. Kibler, and M. K. Albert, "Instance-based Learning Algorithms," *Machine Learning*, vol. 6, no. 1, 1991.
- [2] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is Mutation an Appropriate Tool for Testing Experiments?" in *ICSE*, 2005.
- [3] T. T. Chekam, M. Papadakis, Y. Le Traon, and M. Harman, "An Empirical Study on Mutation, Statement and Branch Coverage Fault Revelation That Avoids the Unreliable Clean Program Assumption," in *ICSE*, 2017.
- [4] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. on Soft. Eng.*, vol. 20, no. 6, 1994.
- [5] R. C. Cruz and M. M. Eler, "Using a Cluster Analysis Method for Grouping Classes According to their inferred Testability: An Investigation of CK Metrics, Code Coverage and Mutation Score," in *SCCC*, 2017.
- [6] L. D. Dallilo, A. V. Pizzoleto, and F. C. Ferrari, "An Evaluation of Internal Program Metrics as Predictors of Mutation Operator Score," in *SAST*, 2019.
- [7] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *IEEE Computer*, vol. 11, no. 4, 1978.
- [8] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the EM algorithm," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 39, no. 1, 1977.
- [9] R. Feldt, S. Poulding, D. Clark, and S. Yoo, "Test Set Diameter: Quantifying the Diversity of Sets of Test Cases," in *ICST*, 2016.
- [10] P. G. Frankl, S. N. Weiss, and C. Hu, "All-uses vs Mutation Testing: An Experimental Comparison of Effectiveness," *Journal of Systems and Software*, vol. 38, no. 3, 1997.
- [11] R. G. Hamlet, "Testing Programs with the Aid of a Compiler," *IEEE Trans. on Soft. Eng.*, vol. 3, no. 4, Jul. 1977.
- [12] K. Jammalamadaka and N. Parveen, "Equivalent Mutant Identification Using Hybrid Wavelet Convolutional Rain Optimization," *Software: Practice and Experience*, vol. 52, no. 2, 2022.
- [13] M. Jureczko and D. Spinellis, "Using object-oriented design metrics to predict software defects," in *DepCoS-RELCOMEX*, 2010.
- [14] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are Mutants a Valid Substitute for Real Faults in Software Testing?" in *FSE*, 2014.
- [15] M. Kintis and N. Malevris, "Identifying More Equivalent Mutants via Code Similarity," in *APSEC*, 2013.
- [16] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. Le Traon, and M. Harman, "Detecting Trivial Mutant Equivalences via Compiler Optimisations," *IEEE Trans. on Soft. Eng.*, vol. 44, no. 4, 2017.
- [17] B. Kurtz, P. Ammann, A. J. Offutt, M. E. Delamaro, M. Kurtz, and N. Gökçe, "Analyzing the Validity of Selective Mutation with Dominator Mutants," in *FSE*, 2016.
- [18] N. Li, U. Praphamontriping, and A. J. Offutt, "An Experimental Comparison of Four Unit Test Criteria: Mutation, Edge-Pair, All-Uses and Prime Path Coverage," in *Mutation*, 2009.
- [19] A. P. Mathur and W. E. Wong, "An Empirical Comparison of Data Flow and Mutation Based Test Adequacy Criteria," *Soft. Testing, Verification and Reliability*, vol. 4, no. 1, 1994.
- [20] A. J. Offutt and R. H. Untch, "Mutation 2000: Uniting the Orthogonal," in *Proc. the Mutation 2000 Symposium*, 2000.
- [21] A. J. Offutt, G. Rothermel, and C. Zapf, "An Experimental Evaluation of Selective Mutation," in *ICSE*, 1993.
- [22] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon, "Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique," in *ICSE*, 2015.
- [23] D. Pelleg and A. W. Moore, "X-means: Extending K-means with Efficient Estimation of the Number of Clusters," in *ICML*, 2000.
- [24] A. V. Pizzoleto, F. C. Ferrari, A. J. Offutt, L. Fernandes, and M. Ribeiro, "A Systematic Literature Review of Techniques and Metrics to Reduce the Cost of Mutation Testing," *Journal of Systems and Software*, vol. 157, 2019.
- [25] A. V. Pizzoleto, F. C. Ferrari, L. D. Dallilo, and J. Offutt, "SiMut: Exploring Program Similarity to Support the Cost Reduction of Mutation Testing," in *Mutation*, 2020.
- [26] A. V. Pizzoleto, G. F. Guarnieri, and F. C. Ferrari, "Definition of a Knowledge Base Towards a Benchmark for Experiments with Mutation Testing," in *SBES-IIER*, 2021.
- [27] M. Polo, M. Piattini, and I. García-Rodríguez, "Decreasing the Cost of Mutation Testing with Second-Order Mutants," *Soft. Testing, Verification and Reliability*, vol. 19, no. 2, 2009.
- [28] L. Prechelt, G. Malpohl, M. Philippsen *et al.*, "Finding plagiarisms among a set of programs with jplag," *J. UCS*, vol. 8, no. 11, 2002.
- [29] J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, and A. Arcuri, "Combining Multiple Coverage Criteria in Search-Based Unit Test Generation," in *Search-Based Software Engineering*, Cham, 2015.
- [30] T. Titcheu Chekam, M. Papadakis, T. F. Bissyandé, Y. Le Traon, and K. Sen, "Selecting Fault Revealing Mutants," *Empirical Software Engineering*, vol. 25, 2020.
- [31] R. H. Untch, "On Reduced Neighborhood Mutation Analysis Using a Single Mutagenic Operator," in *ACM-SE*, 2009.

APÊNDICE E

Definition of a Knowledge Base Towards a Benchmark for Experiments with Mutation Testing

Neste apêndice apresenta-se o conteúdo completo do artigo publicado no *XXXV Brazilian Symposium on Software Engineering (SBES'21), Insightful Ideas and Emerging Results Track* (PIZZOLETO; GUARNIERI; FERRARI, 2021). Um resumo deste artigo é apresentado no Capítulo 5 desta tese.

ACM COPYRIGHT NOTICE:¹ “© ACM 2021. This is the author’s version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in the *Proceedings of the XXXV Brazilian Symposium on Software Engineering (SBES'21)* <<https://doi.org/10.1145/3474624.3477060>>.”

¹ <<https://www.acm.org/publications/policies/copyright-policy#Retained>>

Definition of a Knowledge Base Towards a Benchmark for Experiments with Mutation Testing

Alessandro V. Pizzoleto
Federal University of São Carlos
São Carlos, SP, Brazil
alessandro.pizzoleto@ufscar.br

Fabiano C. Ferrari
Federal University of São Carlos
São Carlos, SP, Brazil
fcferrari@ufscar.br

Giovanni F. Guarnieri
Federal University of São Carlos
Sorocaba, SP, Brazil
gfguarnieri@hotmail.com

ABSTRACT

Context: Mutation testing has been investigated since the late 70's. Ever since, researchers have devised dozens of mutation approaches, including ways of generating, executing, and analyzing mutants, as well as ways of reducing the costs for its application. However, research on this field falls short when it comes to producing a representative and manageable set of artifacts to enable experiments with the plethora of existing mutation approaches. *Objective:* In this paper, we describe the process and current results of creating a knowledge base of mutation-related artifacts to support experiments with mutation testing. *Method:* We setup the Evosuite tool for generating test cases, and the PIT tool for generating and running mutants. We also created scripts to import the results to a relational database. The database includes some procedures to generate killing matrices for the tested Java classes. *Results:* Beyond establishing the tooling infrastructure, we populated our database with classes extracted from five Java projects, from which four are open source projects hosted in GitHub and the fifth one is composed by simple Java programs. Currently, the database includes around 2,000 classes, 50,000 test cases, and 195,000 mutants. *Conclusion:* The database structure eases the addition of other Java programs and the related mutation artifacts. Furthermore, it allows for tasks such as minimizing test sets and mutant sets (e.g. by removing redundant tests and trivial mutants), thus providing researchers with a well-established and extensible basis to support varied experiments.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging.**

KEYWORDS

mutation testing; experiments; artifact database; benchmark

ACM Reference Format:

Alessandro V. Pizzoleto, Fabiano C. Ferrari, and Giovanni F. Guarnieri. 2021. Definition of a Knowledge Base Towards a Benchmark for Experiments with Mutation Testing. In *Brazilian Symposium on Software Engineering (SBES '21)*, September 27–October 1, 2021, Joinville, Brazil. ACM, New York, NY, USA, 6 pages. <https://doi.org/10.1145/3474624.3477060>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SBES '21, September 27–October 1, 2021, Joinville, Brazil

© 2021 Association for Computing Machinery.

ACM ISBN 978-1-4503-9061-3/21/09...\$15.00

<https://doi.org/10.1145/3474624.3477060>

1 INTRODUCTION

Mutation testing [4] – or simply *mutation* – has been extensively investigated as a testing selection criterion for over 40 years. It has also been widely used to evaluate tests developed with a variety of testing approaches other than mutation, as recently reported by Papadakis et al. [14]. As a testing criterion based on faults, it generates new versions of the program under test with changes guided by the *mutation operators*; the operators modify the program to create the so-called *mutants* [4]. When the test set distinguishes the mutant behavior from the behavior of the program under test, the mutant is said to be *killed*.

As originally conceived [4], mutation testing is applied in four steps: (1) execution of the original program; (2) generation of mutants; (3) execution of the mutants; and (4) analysis of the mutants. After each cycle, the *mutation score* (MS) can be calculated through the following formula: $MS = K / (M - E)$. In the formula, K is the number of mutants of a program P killed by a test set T ; M is the total number of mutants created from P ; and E is the number of mutants of P that are equivalent to P . Equivalence is established when there is no input that lead a mutant m to produce a different output when compared to P . In other words, equivalence between P and m happens when m cannot be killed. Scores close to 1 indicate the test set is of good quality with respect to M .

In spite of the large body of knowledge on mutation testing [7, 14], a recent literature review on cost reduction techniques [17] found that experimental material used in the primary studies is typically non-standardized, lacks organization and reusability, and are rarely available to be shared and support other experiments. These observations come to prove the difficulty in reproducing prior research and also to compare results with new research initiatives.

Another problem recently reported was an attempt by Pizzoleto et al. [16] to form a set of artifacts either produced or used by others. It took them a hard effort to obtain the artifacts and understand clearly how they were organized. As reported by the authors, they struggled to define the characteristics and data of each of the artifacts and generate the import of the data into a relational database. Pizzoleto et al. [16] concluded that merging artifacts from varied experimental sources led to a limited database in terms of homogeneity of elements such as the used tools and mutation operators (and hence the resulting mutants), the representativeness of the tested programs, the quality of the test data, and the precision of the information regarding mutant coverage.

By analyzing the entire scenario that makes up the process of developing academic research, with particular focus on mutation-related research, proving the efficiency and effectiveness of any approach is a key factor towards its success. Such proof is only obtained through the use of a set of clearly defined and organized

set of artifacts, data and information that allow the comparison of the obtained results from the current approach with results from similar ones. Aligned with this reasoning, this paper presents, as a result, a coherent and concise knowledge base composed by a set of Java software projects and a relational database that aims to help researchers on mutation testing experiment with and, eventually, validate their approaches. In summary, the main characteristics of the knowledge base introduced in this paper are:

- For all targeted software projects, compilation, test case generation, and mutant execution are Maven-based;
- Test cases for each class are automatically generated by the EvoSuite tool and by developers;
- The set of mutants for each class is generated by the PIT tool;
- The status of each mutant is established, including their relationship with test cases when killed; and
- The possibility of generating mutation scores according to the information selected by the user.

Among other queries that the relational database allows for, some key information that can be extracted by the users are: the original source code of the classes; the mutants created by specific mutation operators or by a group of operators; the bytecode of all created mutants; the source code of test cases defined for each class; and the set of dead mutants and the related test cases.

The remainder of this paper is organized as follows: Section 2 summarizes related work. The process of defining and creating the knowledge base, as well the definition of the initial set of artifacts and used tools are presented in Section 3. Section 5 describes the procedures and summarizes the results currently produced. Finally, Section 6 brings our conclusions, discusses limitations of our work, and points out future work.

2 RELATED WORK

Research on mutation testing lacks initiatives with particular focus on creating databases of artefacts to support experimentation. In spite of this, some pieces of research are at some extent related with our work. In particular, in what follows we describe the Siemens suite, the SIR repository, and the Defects4J database.

The Siemens suite of programs, also referred to as the Siemens benchmark [8, 15], was originally composed and used by Hutchins et al. [6] to allow for the comparison of structural-based test selection criteria. The suite is composed of seven small-sized C programs (ranging from 131 to 512 lines of code) with associated test sets and, ever since, has become one of the most used set of programs in mutation testing experiments [7], including highly relevant and cited research in the field [1, 9]. In spite of that, to the best of our knowledge, the Siemens suite does not include any “official” mutation-related set of artefacts. Instead, the suite has been used with a variety of mutation testing tools.

The Software-artifact Infrastructure Repository (SIR)¹ was originally reported by Do et al. [5] and consists in a repository of software-related artefacts created to support research and education in experimentation with program analysis and software testing techniques. Currently, programs in SIR are written either in Java (68 projects, either single or multi-class) or C (15, in total, including the Siemens suite), and bring additional artefacts such as test suites,

fault data, scripts and documentation to support experimentation. After its release in 2005, SIR artefacts have been used by researchers from hundreds of institutions and have supported experiments that have been described in hundreds of publications,² including several publications of mutation-related research. Differently from the SIR repository, our repository has a relational database as a central element and focus on mutation testing experimentation; furthermore, it is consistently fed with data (test cases, mutants, and mutant execution results) produced by a predefined set of tools.

Defects4J³ is a database of testing-related artefacts with focus on real faults extracted from open source software projects [8]. The aim of Defects4J is supporting reproducible testing experiments not solely based on artificially introduced faults (e.g. mutants). Each fault within the Defects4J database has associated test cases that reveal the fault and succeed in the fixed version of the program. The first version of Defects4J [8] had 357 real faults from five open-source project. The current version contains 835 faults from 17 open-source projects. Defects4J contains an integrated test execution framework that includes a test generation component and a mutation analysis component. Such components use, respectively, EvoSuite and Major as default tools for their purposes. Differently from our database, Defects4J is not meant to store results from test case generation tools and mutation testing tools. Instead, it provides support for running those tools on the stored faulty programs.

3 KNOWLEDGE BASE CREATION PROCESS

As presented in Section 1, it is essential to have a set of intact and reusable artifacts for carrying out experiments. The quality of the results obtained is directly related to the quality of the artifacts used, regardless of the domain of the problem analyzed.

In the context of mutation testing, it has been observed that many researchers, in their projects, use simple artifacts of questionable quality, thus making it difficult to prove the efficiency and effectiveness of the process evaluated in their experiments [17]. One very common observed problem was the impossibility of replicating their experiments due to the absence of shareable artifacts. For those cases in which artifacts are available, information about them are imprecise or incomplete, thus hardening the execution of new experiments and the merging of artifacts and results [16].

Considering all the factors presented above that negatively influence the replication and evolution of research carried out, we decided to define, build and maintain a knowledge base of artifacts, data and information for carrying out experiments in the mutation testing domain. It is worth mentioning that this knowledge base can be used in experiments that make use of other testing criteria since it contains sufficient information for this purpose.

The process of defining the artifact knowledge base was considered a priority to facilitate access to all information contained therein, from the source code of the tested classes to the obtained mutation score. Thus, we opted for using Apache Maven⁴ as a standard project management tool, as well as a relational database since it concentrates in a single local the source code of the classes, test

¹<https://sir.csc.ncsu.edu/portal/index.php> - accessed in May, 2021.

²<https://sir.csc.ncsu.edu/portal/usage.php> - accessed in May, 2021.

³<https://github.com/rjust/defects4j> - accessed in May, 2021.

⁴<https://maven.apache.org/> - accessed in May, 2021.

cases, mutants, mutation operators, and killing matrices according to the experimental needs.

The database model is shown in Figure 1. It was defined to meet the aforementioned characteristics and, mainly, to allow the traceability of the information, that is, from the program or class it is possible to filter the mutants generated and the operators used, and the test cases executed on each mutant and its status for each test case. The model also allows the user to do the reverse traceability, that is, from the test case or the mutation operator he/she can identify the program or class to which they are linked.

For initially populating the knowledge base, we carried out some analyzes and searches for artifacts already used in other studies. This is discussed in the next section. It should also be noted that the knowledge base was defined so that new updates could be carried out without major difficulties by other researchers who wish to collaborate, including the insertion of new artifacts.

3.1 Definition of Artifacts

We carefully selected the artifacts used to initially compose our knowledge base. Only compilable projects were included since they would be used in automatic tools for generating test cases and mutants. For the selection of these projects, an informal consultation was carried out with some researchers working in the mutation testing domain. As a result, we concluded that a good initial set should have different artifacts and of variable sizes, so we decided to include small projects with only one class and also projects composed of several classes. There was no concern about the number of lines of code contained in the classes. One mandatory requirement was that all projects must be Maven-based in order to allow for easy integration with other tools such as Evosuite and PIT.

Regarding small projects, we selected 28 of the 38 projects used by Dallilo et al. [3]. Only 28 projects were used, as they were compilable projects. For reasons of convenience and practicality, we decided to combine these 28 projects in a single project, hereafter called *simplePrograms*, thus facilitating their operation.

For large projects, we chose at this first moment to include four projects all extracted from GitHub repositories. They are listed in the first four rows of Table 1. The Apache Commons Lang⁵ (version 3-3.10) offers several auxiliary utilities such as string manipulation methods and basic numerical methods. Apache Commons Math⁶ (version 3-3.6.1) is a library of lightweight, self-contained mathematics and statistics components. Javassist⁷ (version 3.27.0-GA) is a library that facilitates the manipulation of the application's bytecode. Finally, Joda-Time⁸ (version 2.10.7) offers a high-level solution for handling date/time. One factor we considered for the selection of these four projects was the existence of an initial set of test cases implemented by their developers.

Note that some of these projects have already been used in other mutation testing experiments [9–13]. For example, Oliveira [12] made use of 12 classes from the *commons-math* project, 12 classes from the *commons-lang3* project, 12 classes from the *joda-time* project, and two classes from *simplePrograms*.

Table 1 shows some descriptive metrics of the applications and test cases. It is divided into three parts: Application Code, which concerns the application source code, Developers' Test Code, which regards original test code (when available) and Evosuite Test Code, which concerns test code generated by the Evosuite tool (Sections 4 and 5 contain details of the generation of these test cases by the tool). In the table, TLOC represents the total lines of code, TCs stands for test cases, AVG CC represents the average cyclomatic complexity of the classes, and AVG StdDev CC informs the average standard deviation of the cyclomatic complexity among the classes. The numbers of Java files and classes are also shown, from which we can notice that some Java source code files define more than one class. The metrics were calculated using the Eclipse Metrics plugin version 1.3.8.⁹, which computes numerous metrics in varied granularity such as project, package and class. To facilitate the extraction of the desired metrics by parts, as shown in Table 1, a script was created in Node.js capable of generating a summary in XML format based on the XML file exported by the plugin.

4 USED TOOLS

We used the Evosuite tool¹⁰ in version 1.0.6, which automatically generates test cases following the JUnit standard. As for the generation of mutants for the selected projects and consequently their execution, we chose the PIT tool¹¹ in version 1.6.2. Both tools are widely used for experimentation in software testing projects. Additionally, we implemented an auxiliary tool to support the preparation of the information that is processed in each of the stages of creating our knowledge base. Details of the auxiliary tool are presented in Section 4.2. Next we provide key information regarding the setup and use of Evosuite and PIT.

4.1 Setup of Evosuite and PIT

Evosuite: It uses a hybrid approach to generate and optimize the test cases so that the coverage criterion is met. It also adds a set of possible oracles with small assertions that summarize the current behavior of the tested class; this allows developers to identify possible deviations in behavior. For each class identified with the used command line, Evosuite creates a class containing the suite of generated test cases, which follows the JUnit standard. Beyond the command-line mode, Evosuite also operates with plugins for popular Java development IDEs, such as IntelliJ and Eclipse, or on Maven-based mode.

Evosuite implements set of coverage criteria such as method coverage, line coverage, branch coverage, and weak mutation coverage. Details of Evosuite's coverage criteria can be found elsewhere [18]. For this work we used the complete criteria set, since the goal was to generate the most complete test suite possible. This is achieved with the command line shown in Figure 2. A key parameter for Evosuite is the `-Dsearch_budget`, which establishes the time, in milliseconds, Evosuite will have to generate the test cases. For making use of research-based techniques, this time is called the research budget. The larger the research budget available, the greater the quantity and the better the quality of the generated test cases.

⁵<http://commons.apache.org/proper/commons-lang/> - accessed in May, 2021.

⁶<http://commons.apache.org/proper/commons-math/> - accessed in May, 2021.

⁷<https://www.javassist.org/> - accessed in May, 2021.

⁸<https://www.joda.org/joda-time/> - accessed in May, 2021.

⁹<http://metrics2.sourceforge.net/> - accessed in May, 2021.

¹⁰<https://www.evosuite.org/> - accessed in May, 2021.

¹¹<http://pitest.org/> - accessed in May, 2021.

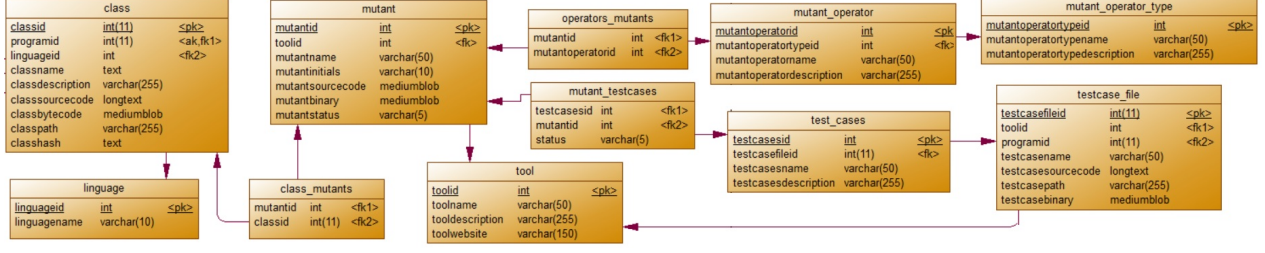


Figure 1: Database diagram/schema.

Table 1: Metrics

Application	Application Code					Developers' Test Code					Evosuite Test Code						
	# Java files	# classes	TLOC	AVG CC	AVG StdDev CC	# Java files	# classes	TLOC	AVG CC	AVG StdDev CC	# TCs	# Java files	# classes	TLOC	AVG CC	AVG StdDev CC	# TCs
commons-lang	161	258	28,497	2.433	3.297	179	442	49,864	1.139	0.673	5,172	222	222	176,466	1.398	0.491	6,655
commons-math	715	1,130	99,453	2.266	4.293	611	863	105,330	1.822	2.187	4,165	940	940	264,179	1.399	0.490	10,023
javassist	234	398	37,765	2.362	4.696	298	570	44,107	1.338	0.870	4,368	460	460	570,670	1.449	0.514	7,385
joda-time	204	245	28,790	1.770	2.422	159	226	55,608	1.292	1.025	4,238	282	282	280,305	1.374	0.486	7,761
simplePrograms	28	28	1,337	2.885	2.505	0	0	0	0	0	0	56	56	5,122	1.154	0.361	318
Total	1,342	2,059	195,842			1,247	2,101	254,909				17,943	1,960	1,960	1,296,742		32,142

```
java -Dtools_jar_location=... -jar evosuite-1.0.6.jar -target <project path .jar> -Duse_separate_classloader=false -Dsearch_budget=6000 -Dstopping_condition=MaxTime
```

Figure 2: Defined Evosuite command line.

PIT: It stood out as a mutation testing tool aimed at the industry, but it has been used in several academic research studies. It performs the mutation at the level of bytecode, making the process of creating and executing mutants faster and less costly since the entire process is executed in memory. The source code is only requested to generate a report containing the mutants and their status.

PIT implements 28 mutation operators. To facilitate the classification and future statistical analyses, we classified the operators in four categories, just as Barbosa et al. [2] classified the Proteum mutation operators used for the C language. The categories of elements upon which the mutation operators actuate are: Variables (5 operators, in total); Constants (1 operator, in total); Operators (9 operators, in total); and Commands (13 operators, in total).

The execution of PIT was carried out by command lines, as exemplified in Figure 3. Among the parameters, mutators indicates the set of mutation operators to be applied, timeoutFactor establishes the timeout for stopping a test execution (e.g. to avoid infinite loop), and features enables the access to relevant mutation information such as the mutant code.

4.2 Auxiliary Tool

In addition to using Evosuite and PIT, we needed to implement an auxiliary tool that aims to prepare the information that will be processed in each of the stages of creating the knowledge base. The purpose of the auxiliary tool is structuring the raw data generated by the other tools so that we can have the largest set of and information related to each of the artifacts contained in the base. Moreover, it seeks to optimize their preparation to be executed in their respective tools and also to be included in the database. In summary, the tool has four necessary functions for the execution of knowledge base creation process, as follows:

(1) **Preparation of test cases:** in order to be aware of the relevance each test case, it was deemed necessary to work with each one individually. Evosuite generates a test class containing all test cases related to a given class from the application. This information, although valuable, was not enough to define the scope of each test case with respect to a precise mapping between test cases and killed mutants. Therefore, we implemented a function that separates each test case into a new class, irrespective of the origin of the test class (i.e. from Evosuite or from the developers).

(2) **Generation of the command line for PIT:** We implemented a function that generates the command line used in PIT for each test class produced by function (1). This function has the purpose of defining all the dependencies of the project, the test file that will be used and the classes that this test file will impact. This process is repeated for all test classes.

(3) **Creation of killing matrix:** The execution of PIT produces a CVS file that contains all the information related to the source code class, the mutants generated for that class, dead mutants, live mutants, and mutants not covered by the test case. In this context, we implemented a function that aims to join all these files in a single one. The function also performs a pre-evaluation of the generated information, and any abnormality is discarded.

(4) **Inclusion of killing matrix in the relational database:** Once the file containing killing matrix is generated, the next step is to include this information in the database. In doing so, the information is centralized in and allows users to process and extract all the information necessary for the execution of their experiments and validation of the data generated by it.

5 PROCEDURES AND RESULTS

```
java -cp <dependency of project> org.pitest.mutationtest.commandline.MutationCoverageReport --timestampedReports true
--mutators ALL --sourceDirs [<source code, test code>],packages=[program.*] --reportDir <result path> --targetClasses program.*
--targetTests <test case file> --verbose true --outputFormats CSV --timeoutFactor 3 --features=+EXPORT
```

Figure 3: Defined PIT command line.

The results obtained with the use EvoSuite and PIT, and the information that can be extracted from the database, are presented in this section. To generate artifacts, we followed a process that is divided into six phases. This enables the evolution of the database (e.g. through the insertion of new classes, test cases and mutants). Each of these phases represents actions that have been analyzed, discussed and must be carried out so that the information generated at the end is complete and compatible with the information already stored into the database. Note that for each of the initial projects, this process was strictly followed and adjusted when necessary. The definition and presentation of the phases is necessary to maintain the integrity of the knowledge base, followed by the inclusion of new artifact projects. The knowledge base is able to accept artifacts from different programming languages; for this, the steps herein described should be followed and, after completion, inclusion scripts should be generated in conformance with the database scheme.

Phase 1 – Execution of Evosuite: The execution of Evosuite on the five projects presented in Table 1 generated a substantial number of test cases. The tool was configured to use all of its test case generation criteria. The goal is to have the most complete set of test cases possible for all classes of all projects. Table 1 shows the quantitative results of this execution. At this stage, we are not concerned with separating the test cases according to the criteria available in the tool; this was left for future work.

Phase 2 – Preparation of Test sets: Evosuite generated test classes for each class of the source code. To ease the process of identifying the mutants that are killed by each test case, our auxiliary tool separates each test case into a specific class. This same procedure was applied to the test classes generated by the developers, when available. It is worth mentioning that at each stage of the process, we checked the integrity of the created files.

Phase 3 – Definitions of the command lines to run PIT: The command line follows the structure presented in Section 4.1. At this stage of the process, we decided to create an execution command line for each test case, despite the possibility of executing them all with a single command line. This decision is directly related to the purpose of identifying which test cases are able to kill each of the mutants, since this is crucial information for obtaining the mutation score. Another objective is linked to the operational process of allowing the execution of the next phase in several workstations, speeding up the completion of processing. This entire process was automated in the auxiliary tool described in Section 4.2.

Phase 4 – Execution of PIT: Each PIT execution generated as a result a CSV file with information regarding the generated mutants, the mutated source code line, the used mutation operator, the executed test case and the information about the status of the mutant, whether dead, alive, or not covered. An example with partial contents of this file is shown in Figure 4. In total, for each test case (be it an Evosuite test or a developer test) a CSV file was produced.

One of the command line parameters aimed to inform PIT that it should generate and save the file of the class under test containing the mutated code. As PIT generates and executes the mutants from the project's bytecode, the file of the mutated class was also generated in this format. After the execution of all command lines, result is a set of files representing the mutants and a set of files representing the status of the mutants for each executed test case.

Table 2 presents the number of mutants generated for each of the projects, and the respective mutation scores. It is worth mentioning that all mutation operators were used to generate mutants.

Table 2: PIT results.

Application	# Mutants	Mutation Score
commons-lang3	40,301	64.52%
commons-math3	60,697	69.79%
javassist	44,461	64.54%
joda-time	46,999	68.54%
simplePrograms	1,888	68.70%
Total	194,346	–

Phase 5 – Generation of the killing matrix: In this phase, the auxiliary tool merges all the generated CSV files in a single file, thus obtaining the killing matrix for the considered classes (e.g. classes of a particular project listed in Table 1). The tool also improves the presentation of the information as follows: semicolons are used to separate the information; removal of redundancies (e.g. mutation information that appears in all files), and a new representation for the status of the mutants, namely: *Y* for KILLED, *OUT* for TIMED_OUT, *N* for SURVIVED, and *N_C* for NO_COVERED. Figure 5 shows a partial example of this file. The first line of the represents the header. The remaining lines represent information of individual mutants, including the results of the test case execution (e.g. the figure shows results for the *Bisect_ESTest* test case).

Phase 6 – Inclusion of the information in the database: In this last phase of the process, all generated artifacts were included in the defined relational database. The inclusion is also handled by the auxiliary tool. According to the database model, the tool runs the entire data inclusion process in the following order: (1) Inclusion of all classes belonging to the targeted project; (2) Inclusion of all used mutation operators; (3) Inclusion of the tool that generated the mutants (i.e. PIT); (4) Inclusion of all mutants generated by PIT. At this time, the mutant bytecode and the relationship with the class that generated the mutant are also included in the database; (5) Inclusion of all test cases generated by Evosuite, as well as developers' test cases. It also includes the source code of the class that contains the test case and the relationship to the class that generated the test case; and (6) Inclusion of the relationship between the test case and the mutant when the test case executes the mutant. Killed and alive are mutant status considered at this point.

5.1 Extraction of Database Information

As previously described, the artifact knowledge base presented in this paper has been defined to assist in experiments, whether

```

Bisect.java,program.Bisect,'...'.MathMutator,sqrt,41,KILLED,program.Bisect_ESTest.program.Bisect_ESTest
Bisect.java,program.Bisect,'...'.MathMutator,sqrt,49,TIMED_OUT,none
Bisect.java,program.Bisect,'...'.RemoveConditionalMutator_ORDER_IF,sqrt,42,SURVIVED,none
Bisect.java,program.Bisect,'...'.ReturnValsMutator,sqrt,56,NO_COVERAGE,none
...

```

Figure 4: Example of a CSV file that represents the results of a PIT execution.

```

Mutation Operator ;Function;line;TC_1;TC_2;TC_3;'...';Bisect_ESTest;.....
InlineConstantMutator ;toString;77 ;N_C ;N_C ;N_C ;'...';Y ;.....
MathMutator ;toString;76 ;N_C ;N_C ;N_C ;'...';N_C ;.....
NegateConditionalsMutator;toString;74 ;N_C ;N_C ;N_C ;'...';Y ;.....
NonVoidMethodCallMutator ;toString;76 ;N_C ;N_C ;N_C ;'...';OUT ;.....
...

```

Figure 5: Example of a CSV file that represents a killing matrix.

or not they are aimed at the software testing domain using the mutation criterion. Particularly regarding the relational database, it provides full access to the original source code of the classes or to a set of classes of the users' choice, since the code for all classes is stored in the DBMS. From the definition of the classes, all generated mutants, mutation operators that generated these mutants and related test cases can be accessed. The user will also be able to perform various calculations related to: class (or set of classes) mutation score, mutation score for specific mutation operators, and so on. The base will also allow that, even without the re-execution of all tests and mutants, users make comparisons between the various mutation cost reduction techniques.

6 CONCLUSIONS AND FUTURE WORK

This paper introduced a new concept of knowledge base for general standardisation of artefacts that can be used for verification and validation of research proposals in the domain of quality and software testing, particularly in the domain of mutation testing. As it is implemented using a DBMS, it can be easily shared, thus allowing easy reproduction of the research that used it and facilitating research evolution innovation. Since it has a stable and accessible set of artefacts, it allows research results comparison, providing more reliability to them, in addition to allowing numerous configurations and the use of artifact selection parameters.

The current version of the knowledge base has some limitations that are related to its structure and some artifacts included in it. Amongst them, we can mention: the user has access to all classes, their mutants and test cases, but cannot access the original project and the classes it contains; with respect to mutants, the base only has the mutant's bytecode file (source code is not available); still regarding mutants, the equivalent one were not identified. but the database structure is prepared for this identification, both in the mutants and mutants_testcases tables.

Future work will address the inclusion of new artefacts, thus providing researchers with a larger set of experimental material. We also plan to validate whether the listed artefacts meet the needs of particular mutation subdomains such as reduction of test suites using mutation, determination of sufficient operators, and optimization techniques to detect equivalent mutants. In the future we also intend to make this knowledge available online so that other researchers can use it and reference it, thus facilitating the replication

of experiments already carried out, evolution of research, comparison between existing techniques and new developed techniques.

ACKNOWLEDGEMENTS: We were supported by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, Fundação de Amparo à Pesquisa do Estado de São Paulo (#2016/21251-0), Conselho Nacional de Desenvolvimento Científico e Tecnológico (#306310/2016-3), and Federal University of São Carlos, Brazil.

REFERENCES

- [1] J. H. Andrews, L. C. Briand, and Y. Labiche. 2005. Is Mutation an Appropriate Tool for Testing Experiments?. In *ICSE'05*.
- [2] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi. 2001. Toward the Determination of Sufficient Mutant Operators for C. *STVR* 11, 2 (2001).
- [3] L. D. Dallilo, A. V. Pizzoleto, and F. C. Ferrari. 2019. An Evaluation of Internal Program Metrics as Predictors of Mutation Operator Score. In *SAST'19*.
- [4] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer* 11, 4 (1978).
- [5] H. Do, S. Elbaum, and G. Rothermel. 2005. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and Its Potential Impact. *Empirical Software Engineering* 10, 4 (2005).
- [6] M. Hutchins et al. 1994. Experiments on the Effectiveness of Dataflow- and control-flow-based test adequacy criteria. In *ICSE'94*.
- [7] Y. Jia and M. Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. on Soft. Eng.* 37, 5 (2011).
- [8] R. Just, D. Jalali, and M. D. Ernst. 2014. Defects4J: A Database of existing faults to enable controlled testing studies for Java programs. In *ISSTA'14 - Tools Track*.
- [9] R. Just et al. 2014. Are Mutants a Valid Substitute for Real Faults in Software Testing?. In *FSE'14*.
- [10] M. Kintis and N. Malevris. 2015. MEDIC: A Static Analysis Framework for Equivalent Mutant Identification. *Information and Software Technology* 68 (2015).
- [11] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. Le Traon, and M. Harman. 2018. Detecting Trivial Mutant Equivalences via Compiler Optimisations. *IEEE Transactions on Software Engineering* 44, 4 (2018).
- [12] L. F. M. Oliveira. 2010. *Tackling The Useless Mutants Problem*. Tese. Centro de Informática, Universidade Federal de Pernambuco.
- [13] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon. 2015. Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique. In *ICSE'15*.
- [14] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Yves Le Traon, and M. Harman. 2019. Mutation Testing Advances: An Analysis and Survey. In *Advances in Computers*.
- [15] D. K. Phan, Y. Kim, and M. Kim. 2018. MUSIC: Mutation Analysis Tool with High Configurability and Extensibility. In *Mutation'18*.
- [16] A. V. Pizzoleto, F. C. Ferrari, L. D. Dallilo, and J. Offutt. 2020. SiMut: Exploring Program Similarity to Support the Cost Reduction of Mutation Testing. In *Mutation'20*.
- [17] A. V. Pizzoleto, F. C. Ferrari, A. J. Offutt, L. Fernandes, and M. Ribeiro. 2019. A Systematic Literature Review of Techniques and Metrics to Reduce the Cost of Mutation Testing. *Journal of Systems and Software* 157 (2019).
- [18] J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, and A. Arcuri. 2015. Combining Multiple Coverage Criteria in Search-Based Unit Test Generation. In *Search-Based Software Engineering*. Springer, Cham, 93–108.