DISSERTAÇÃO DE MESTRADO

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM

CIÊNCIA DA COMPUTAÇÃO

"Investigação de Similaridade entre Programas para Apoiar o Teste de Mutação"

ALUNO: Lucas Diniz Dallilo
ORIENTADOR: Prof. Dr. Fabiano Cutigi Ferrari

São Carlos
Maio/2019

CAIXA POSTAL 676
FONE/FAX: (16) 3351-8233
13565-905 - SÃO CARLOS - SP
BRASIL

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

INVESTIGAÇÃO DE SIMILARIDADE ENTRE PROGRAMAS PARA APOIAR O TESTE DE MUTAÇÃO

LUCAS DINIZ DALLILO

ORIENTADOR: PROF. DR. FABIANO CUTIGI FERRARI

São Carlos - SP

Maio/2019

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

INVESTIGAÇÃO DE SIMILARIDADE ENTRE PROGRAMAS PARA APOIAR O TESTE DE MUTAÇÃO

LUCAS DINIZ DALLILO

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, área de concentração: Engenharia de Software
Orientador: Prof. Dr. Fabiano Cutigi Ferrari

São Carlos - SP

Maio/2019

UNIVERSIDADE FEDERAL DE SÃO CARLOS

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

Folha de Aprovação

Assinaturas dos membros da comissão examinadora que avaliou e aprovou a Defesa de Dissertação de Mestrado do candidato Lucas Diniz Dallilo, realizada em 20/05/2019:

Prof. Dr. Fabiano Cutigi Ferrari
UFSCar

Prof. Dr. Auri Marcelo Rizzo Vincenzi
UFSCar

Prof. Dr. Marcelo Medeiros Eler
USP

Certifico que a defesa realizou-se com a participação à distância do(s) membro(s) Marcelo Medeiros Eler e, depois das arguições e deliberações realizadas, o(s) participante(s) à distância está(ao) de acordo com o conteúdo do parecer da banca examinadora redigido neste relatório de defesa.

Prof. Dr. Fabiano Cutigi Ferrari

Agradecimentos

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior-Brasil (CAPES) - Código de Financiamento 001.

Agradeço à minha família, por tudo.

Aos professores que contribuíram com a minha formação.

Ao meu orientador pela paciência, contribuições no trabalho, ensinamentos e oportunidades.

Aos meus amigos e colegas pela companhia e ajuda.

Ao DC-UFSCar pela estrutura e ensino.

Muito obrigado.

Resumo

Contexto: O critério Análise de Mutação - ou, teste de mutação – possibilita tanto a avaliação de conjuntos de teste quanto a identificação de defeitos presentes no software. O critério é considerado como sendo eficaz, porém apresenta lacunas em relação à sua eficiência. Muitas técnicas de redução de custo foram apresentadas porém os resultados dessas técnicas são pouco generalizáveis para grupos distintos de programas. Alguns estudos nessa linha apresentaram heurísticas para calcular a similaridade entre programas e auxiliar suas propostas, mas o cálculo não era o tema central dos estudos. **Objetivos**: Este trabalho investigou a similaridade entre programas, no contexto do teste de mutação de programas OO, como subsídio primário para apoiar estratégias de redução de custo no critério. **Metodologia**: O trabalho incluiu: (i) uma pesquisa bibliográfica para caracterizar o cálculo de similaridade no contexto apresentado; (ii) a definição de um *framework* conceitual que utiliza a similaridade como suporte a estratégias de redução de custos; (iii) o desenvolvimento de uma ferramenta baseada no *framework*. A similaridade é inferida por meio da *clusterização* de métricas CK. A ferramenta também manipula os cálculos e resultados provenientes da técnica de redução de custo empregada; e (iv) um experimento e análise de resultados. Classes Java foram clusterizadas e os valores de escores de mutação por operador, das classes e clusters foram calculados. No contexto da técnica de redução de custo One-Op, os melhores operadores foram comparados entre os grupos criados e as classes individualmente. **Resultados**: Como resultados, têm-se a descrição do cálculo de similaridade no contexto apresentado; um *framework* conceitual e respectivo ferramental de apoio; e um experimento em 38 classes organizadas em 3 bases de dados. **Conclusões**: A relevância da similaridade foi constatada, possibilitando futuros testes dos operadores candidatos dos grupos similares sendo aplicados em novos programas no contexto do critério e estratégia de redução de custo.

Palavras-Chave: Teste de software; Análise de Mutação; Teste de mutação; Técnicas de redução de custo; Similaridade; One-Op; Clusterização

Abstract

Context: The Mutation Analysis criterion – or, mutation testing – allows both the evaluation of test sets and the identification of faults present in the software. The criterion is considered effective, however has gaps in relation to its efficiency. Many cost reduction techniques have been presented however, the results yielded by these techniques are little generalizable to different groups of programs. Some studies in this context presented heuristics to calculate similarity between programs as a way to support cost reduction, but the calculation was not the central theme of the studies. **Goals**: This work investigated the similarity between programs, in the context of mutation testing of object-oriented programs, as primary information source to support the definition of a strategy to reduce the cost of the criterion. **Methodology**: The work included: (i) A literature research to characterize the similarity calculation in the presented context; (ii) The definition of a conceptual framework to apply similarity as a supportive technique for cost reduction strategies; and (iii) automation of the framework. The similarity is inferred through the clustering of CK metrics information. The tool also handles calculation and results from the employed cost reduction techniques; and (iv) An experiment and results analysis. Java classes were clustered and the values of mutation scores per operator of the classes and clusters were calculated. In the context of the One-Op cost reduction technique, the best candidate operators were compared between the generated groups and the classes individually. **Results**: As results, we have the description of similarity calculation in the presented context; a conceptual framework and respective support tool; and a experiment in 38 classes organized in 3 databases. **Conclusion**: The relevance of similarity was observed, thus allowing for further experiments involving operators obtained from similar programs to be applied to untested programs in the context of the criterion and strategy of cost reduction.

Keywords: Software test; Mutation Analysis; Mutation Test; Cost Reduction Techniques; Similarity; One-Op; Clustering;

Lista de Figuras

Lista de Tabelas

Lista de abreviaturas e siglas

(V&V) - Verificação e Validação
OO - Orientação a Objetos
OA - Orientação a Aspectos
POO - Programação Orientada a Objetos
CK - Chidamber e Kemerer
CFG - *Control Flow Graph*
SLR - *Systematic Literature Review*
MS - *Mapeamento Sistemático*
StArt - *State of Art through Systematic Review*
WMC - *Weighted methods per class*
DIT - *Depth of inheritance tree*
RFC - *Response for class*
CBO - *Couplig between Object Classes*
LCOM - *Lack of Cohesion of Methods*
NOC - *Number of Children*
AST - Árvore Sintática Abstrata
LOC - *Lines of code*
BIC - *Bayesian Information Criterion*
BPMN - *Business Process Model and Notation*
BPD - *Business Process Diagram*
LaPES - Laboratório de Pesquisa em Engenharia de Software
GNU - *General Public License*
Weka - *Waikato Environment for Knowledge Analysis*
ARFF - *Attribute-Relation File Format*
SDL - *Statment deletion*

Sumário

Capítulo

1

Introdução

1.1 Contexto e Motivação

A garantia da qualidade é uma necessidade na produção de software (Sommerville, 2011). Para ajudar a suprir essa demanda, as atividades de Verificação e Validação (V&V) são empregadas para garantir que os programas tenham sido construídos de forma correta, e que funcionem como especificados, assim atendendo às expectativas dos clientes. No contexto de validação de software, o teste é uma atividade fundamental para revelar defeitos (Myers et al., 2011), os quais ocorrem independentemente das técnicas rigorosas de desenvolvimento aplicadas. A atividade de teste pode ser dividida em fases como unidade, integração e sistema (Myers et al., 2011; Pressman, 2010). Várias técnicas podem ser empregadas, sendo tradicionalmente classificadas como teste funcional, teste estrutural, e teste baseado em defeitos.

Independentemente das atividades de garantia de qualidade que podem ser realizadas durante o desenvolvimento de um software, a construção em si do software pode se basear em paradigmas distintos. Por exemplo, a programação orientada a objetos (OO) é utilizada para desenvolver um código mais adaptável, reutilizável, e de fácil manutenção. Outro exemplo é a programação orientada a aspectos (OA) que, por sua vez, busca resolver os problemas de modularização enfrentados pela POO (Kiczales et al., 1997), os quais dificultam a reutilização e manutenção do código. Nota-se, a despeito dos benefícios que

novas formas – ou até mesmo, novas tecnologias – podem trazer ao desenvolvimento de software, características inerentes a elas representam novos desafios para o teste de software, os quais podem ser considerados novas fontes de defeitos. Tomando-se novamente os paradigmas OO e OA como exemplo, conceitos e elementos de programação como as hierarquias de classe, polimorfismo, inconsciência e quantificação geram novos tipos de defeitos nos programas (Alexander et al., 2004; Ferrari et al., 2010; Offutt et al., 2001). Para identificar e reduzir o número de defeitos, o teste de software e suas técnicas e critérios associados requerem contínua evolução.

Dentre os diversos critérios de teste, o teste de mutação (DeMillo et al., 1978; Hamlet, 1977) é um critério da técnica de teste baseada em defeitos considerado, segundo a comunidade de teste de software, muito efetivo para avaliar a qualidade dos casos de teste. O critério funciona, de forma resumida e respectiva, na aplicação dos casos de teste no programa original; na geração de um grande grupo de programas pontualmente alterados, chamados mutantes, por meio de operadores de mutação, que alteram o código original em relação, por exemplo, a características da linguagem e paradigma de programação; e na execução dos casos de teste sobre os mutantes e a comparação de suas saídas à saída da versão original. Um escore pode ser calculado, e quanto mais mutantes mortos (isto é, ao menos um caso de teste detecta o defeito) em relação ao total de mutantes, maior será seu valor.

O teste de mutação é utilizado como um "padrão" para verificar a efetividade de novos critérios de teste, tendo sido avaliado em diversas iniciativas ao longo dos anos (Andrews et al., 2005, 2006; Do e Rothermel, 2006), e assim constatado em uma recente e extensa pesquisa bibliográfica realizada por Papadakis et al. (2019). Esses autores selecionaram e descreveram mais de 240 artigos científicos publicados de 2009 a 2018, no quais se utilizou o teste de mutação como ferramenta para avaliar testes gerados a partir de outros critérios. A despeito disso, apesar de eficaz, o teste de mutação não é eficiente, devido ao seu alto custo de aplicação. Conforme constatado e relatado na literatura (Ferrari et al., 2018; Papadakis et al., 2019), alguns fatores contribuem para o elevado custo. Exemplos desses fatores são a grande quantidade de mutantes gerados, a grande quantidade de execuções de testes necessárias, e o intrínseco trabalho manual de análise de mutantes remanescentes (seja para a criação de novos casos de teste, seja para classificar os mutantes como equivalentes). Dessa forma, as pesquisas sobre esse critério geralmente focam na redução direta ou indireta do custo, sendo esse relacionado tanto a tarefas automatizáveis quanto a tarefas manuais.

Dos variados estudos sobre a redução de custo do teste de mutação surgiram diversas técnicas para se atingir o objetivo pretendido. Por exemplo, Ferrari et al. (2018) relataram

que em 165 estudos por eles analisados, 21 diferentes técnicas foram empregadas para atingir 6 diferentes objetivos de redução de custo. A despeito dessa variedade de técnicas disponíveis, ressalta-se que, em geral, os resultados da aplicação das técnicas são pouco generalizáveis. Por exemplo, Siami-Namin et al. (2008) afirmaram que as técnicas por eles propostas, se aplicadas a um conjunto distinto de programas (por exemplo, programas com estruturas lógica e de dados mais complexas), poderiam produzir diferentes resultados. Limitações similares já sido descritas por Offutt et al. (1996) e Barbosa et al. (2001) e foram reforçadas mais recentemente, por exemplo, por Papadakis e Malevris (2010), Omar e Ghosh (2012) e Lacerda e Ferrari (2014). Em um trabalho mais recente, Kurtz et al. (2016) afirmaram que cada programa deve ser analisado individualmente, ou seja, operadores (e seus mutantes) que podem ser eficientes para um programa, podem não ser eficientes para outros. Sendo assim, a questão que perdura após geradas as evidências para grupos particulares de programas refere-se à utilidade dos resultados das técnicas de redução de custo para serem aplicados em outros programas que ainda não foram testados com o critério Análise de Mutantes.

Nesse contexto, algumas pesquisas em teste de mutação envolvem a utilização de heurísticas para calcular a similaridade entre programas, auxiliando de forma secundária as propostas. Por exemplo, no estudo de Bashir e Nadeen (2013) sobre teste de mutação OO evolucionário, utilizou-se a informação do fluxo de controle e estado dos objetos para comparar a saída dos mutantes com o programa original. Schuler e Zeller (2010), por sua vez, apresentaram um estudo sobre a cobertura do código e a probabilidade de um mutante não ser equivalente baseada na análise do fluxo de dados e nas mudanças de informações passadas por meio dos métodos. Em outra iniciativa, Anbalagan e Xie (2008) desenvolveram um *framework* para teste de mutação AO que classifica e ordena os mutantes utilizando uma medida de similaridade léxica para ajudar o desenvolvedor a escolher mutantes parecidos com o programa original.

As heurísticas para similaridade entre programas podem ser usadas de muitas formas, contudo elas não são o tema central dos trabalhos encontrados. Essa conclusão motivou o tema deste trabalho, o qual focou na utilização de heurísticas de similaridade para suporte ao teste de mutação de programas OO. Na seção seguinte são apresentados os objetivos buscados por este trabalho de forma mais detalhada.

1.2 Objetivos

Como apresentado na seção anterior, a análise de similaridade entre programas pode ser útil para compor estratégias de redução de custo ou na identificação de mutantes equivalentes, dentre outras necessidades do teste de mutação.

Neste projeto, investigou-se a similaridade entre programas como um subsídio primário para apoiar a definição de abordagens de teste de mutação, com foco inicial em programas OO. Nesse contexto, o desenvolvimento prático das técnicas escolhidas para o suporte ao teste de mutação também compreende os objetivos do projeto. Em específico, buscou-se:

- Caracterizar formas de calcular a similaridade entre programas, tendo como base um conjunto de estudos primários que abordaram o teste de mutação em programas orientados a objetos e orientados a aspectos;
- Definir uma abordagem de aplicação do cálculo de similaridade ao longo do processo de teste de mutação; e
- Prover recursos automatizados para apoiar a abordagem definida.

Como um exemplo de aplicação da abordagem, considerando-se técnicas de redução de custo do teste de mutação baseadas em subconjuntos de operadores de mutação, os cálculos de similaridade realizados podem ajudar a filtrar os resultados (que nesse caso são os operadores mais relevantes) para serem aplicados em programas similares aos já testados. Outros problemas inerentes ao teste de mutação que podem se beneficiar com os resultados de cálculo de similaridade são a identificação automática de mutantes equivalentes e a identificação de mutantes não triviais (ou seja, mutantes que não são mortos facilmente).

1.3 Organização do trabalho

Os próximos capítulos estão organizados da seguinte forma: no Capítulo 2 apresenta-se a fundamentação teórica para a compreensão da proposta; no Capítulo 3 apresenta-se a definição de um *framework* conceitual que estabelece os passos necessários para se introduzir o cálculo de similaridade como apoio à redução do custo do teste de mutação; no Capítulo 4 descreve-se a ferramenta desenvolvida para dar suporte à abordagem proposta; no Capítulo 5 relatam-se resultados de um estudo experimental de natureza exploratória que envolveu o uso dos conceitos e ferramental desenvolvidos neste trabalho.; e por fim,

no Capítulo 6 conclui-se este trabalho e apresentam-se as limitações e possibilidades de trabalhos futuros.

Capítulo

2

Fundamentação Teórica

2.1 Considerações Iniciais

Neste capítulo são descritos os principais conceitos que permeiam a atividade de teste, com enfoque no critério de teste de mutação, pertencente à técnica de teste baseada em defeitos. Especial atenção é dada à aplicação desse critério em software desenvolvido com o paradigma orientados a objetos (OO). Também são apresentadas técnicas de redução de custo do teste de mutação, conceito de similaridade de programas, métricas internas de software, algoritmo de agrupamento e os mais importantes trabalhos encontrados em um Mapeamento sistemático realizado pelo autor.

Na Seção 2.2 os conceitos básicos do teste de software, como seu objetivo e elementos, além de Técnicas e Critérios como o teste de mutação e estratégias de redução de custos associadas a este critério. A Seção 2.3 conceitua, para este trabalho, a similaridade de programas e apresenta um Mapeamento Sistemático com alguns trabalhos de aplicação pontual da similaridade. A Seção 2.4 apresenta métricas internas de software, dentre elas as métricas CK utilizadas neste trabalho como heurísticas para calcular a similaridade entre programas. Por fim, a Seção 2.5 apresenta algoritmos de agrupamento que foram utilizados com as métricas internas de software.

2.2 Conceitos de Teste de Software

Objetivo do teste

O teste de software é uma das importantes atividades para garantia da qualidade no desenvolvimento de sistemas. Muitos programadores acreditam que a finalidade do teste é a prova da isenção de defeitos do software; contudo, seu propósito é revelar a presença de defeitos no software (Myers et al., 2011). Os defeitos ocorrem, não apenas, devido à complexidade do software (em conjunto com as centenas ou milhares de entradas possíveis para ele), que pode gerar um grande número de saídas imprevistas do software desenvolvido; eles ocorrem também devido à complexidade do desenvolvimento em si, com a consequente inserção de defeitos no software.

Defeito, erro, falha

De acordo com o *Institute of Electrical and Electronics Engineers (IEEE)* (IEEE, 1990), defeitos representam inconformidades em relação ao produto esperado e aquele que foi desenvolvido ou planejado. Já os erros representam estados inconsistentes de um programa, resultante da execução de um ou mais defeitos presentes no software. Por fim, uma falha representa uma extrapolação do erro em relação às fronteiras do sistema, ou seja, é um erro que se torna observável externamente ao software.

Casos de teste, oráculos, conjuntos de testes

Durante a atividade de teste, tarefas indispensáveis são o projeto e a execução de *caso de teste*. De um modo mais formal, um caso de teste é uma tupla composta pelas entradas necessárias – também chamadas de dados de teste – e a saída esperada para se testar uma determinada operação (Myers et al., 2011). O *conjunto de testes* são todos os casos de teste projetados e, eventualmente, executados.

Em conjunto com os casos de testes, os *oráculos de teste* verificam se as saídas dos casos de teste correspondem com os valores esperados (Mathur, 2007). Dependendo do tipo de software e do tipo de funcionalidade, o oráculo pode ser automatizado, embora é comum que o papel de oráculo seja exercido pelo próprio testador.

Critérios de teste, requisitos de teste

O domínio de entrada representa todas as possíveis entradas de um determinado programa. Se um programa P qualquer aceitar números inteiros como entrada, então todos os números inteiros representam o domínio de entrada. Seria ideal que todas as combinações de valores possíveis do domínio pudessem ser testadas, possibilitando a execução

exaustiva do software, e garantindo a ausência de defeitos caso todos os testes passassem. Contudo, a execução exaustiva tornaria a atividade de teste inviável na maioria dos casos. Para resolver o problema, são encontrados subconjuntos do domínio com alta probabilidade de revelar defeitos no programa, no caso os subdomínios de teste. Nesse contexto, os *critérios de teste* representam regras que norteiam tanto a definição dos elementos do software – chamados *requisitos de teste* – que devem ser executados, quanto dos elementos dos subdomínios, para que se obtenha um conjunto de testes de alta qualidade (Frankl e Weyuker, 2000).

Processo de teste

O processo de teste possui um ciclo de vida composto por etapas básicas. Essas etapas são dependentes do software, recursos e requisitos de confiabilidade e segurança. Genericamente elas são divididas em: planejamento, projeto, execução, finalização e acompanhamento (realizado paralelamente às outras etapas) (Crespo et al., 2011).

O ciclo se inicia no *Planejamento*, etapa na qual é decidido o material utilizado, com qual abordagem, quando, os riscos, em quanto tempo e quem irá realizar o teste. No *Projeto* são criados os casos de teste em relação às técnicas e critérios escolhidos. Ajustes na abordagem, especificação e planejamento em geral podem ser realizados. Em seguida, ocorre a *Execução* dos casos de teste e o registro dos efeitos da execução. A *Finalização* termina o ciclo de teste com a análise, medição e consequente coleta de resultados não apenas do software escolhido, mas também da atividade e de seus envolvidos. Paralelamente às etapas anteriores, o *Acompanhamento* encarrega-se de relatar os casos de teste executados, defeitos em geral, cobertura obtida e a necessidade de se aplicarem casos de testes novamente.

Níveis de teste

Cada *nível* de aplicação do teste possui um objetivo diferente. No *teste de unidade*, as menores estruturas do sistema, como classes e métodos, são testadas isoladamente. O teste pode ser realizado pelo próprio desenvolvedor. O *teste de integração* considera a comunicação entre as unidades do sistema para encontrar defeitos, normalmente sendo executado pela equipe de desenvolvimento. Quando o sistema está completo, é necessário averiguar se ele atende a todos os requisitos funcionais e aos requisitos não funcionais, como desempenho e segurança. Nesse momento, realiza-se o *teste de sistema*, que pode ser executado pela mesma ou outra equipe. Ressalta-se que o ciclo de vida do teste e suas respectivas etapas, variando discretamente, está presente em cada nível do teste.

Na próxima subseção serão abordadas as principais técnicas de teste de software, em conjunto com os critérios de teste mais relevantes, de cada técnica, que são utilizados nos diferentes níveis do teste.

2.2.1 Técnicas de Teste e Critérios Associados

As técnicas (genericamente, o tipo de informação a ser utilizada) e os critérios (regras associadas à técnica para delimitar os subdomínios) geram modelos diferentes para representar o software. O modelo pode ser adquirido ou criado por meio de uma observação externa do software (especificação de requisitos), representações formais da estrutura estática ou em execução do código (grafos), ou por meio da descrição de defeitos comuns (modelos ou taxonomias de defeitos). Os requisitos de teste são derivados de tais artefatos, devendo ser executados pelos casos de teste. A seguir, o teste funcional, estrutural e o baseado em defeitos são apresentados, sendo eles as técnicas mais investigadas e empregadas pela comunidade de teste.

Teste Funcional

Também conhecido como teste caixa preta (Myers et al., 2011), essa técnica deriva os casos de teste com base na especificação do software, sem considerar a estrutura interna do mesmo. Em suma, procura-se identificar as funções do software e criar casos de teste para elas.

De acordo com Myers et al. (2011), os principais critérios associados a essa técnica são o *Particionamento de Equivalência*, a *Análise do Valor Limite* e o *Grafo de Causa-Efeito*. O primeiro critério gera classes de equivalência do domínio de entrada (baseadas na especificação), as quais definem um número mínimo de casos de teste para satisfazê-las. O segundo critério foca nos limites (superior e inferior) das classes de equivalência, motivado pela alta frequência de defeitos relacionados a esses limites. Por fim, o critério Grafo de Causa-Efeito baseia-se em um grafo que representa as possíveis condições de entrada e efeitos esperados e, a partir desse grafo, cria-se uma tabela de decisão para a derivação dos casos de teste.

Teste Estrutural

A técnica estrutural – também chamada de teste caixa branca (Myers et al., 2011) – baseia-se no conhecimento da estrutura interna do programa para gerar os casos de teste. Normalmente, os critérios associados a essa técnica utilizam um grafo do programa (*grafo de fluxo de controle* – GFC) para representar os caminhos lógicos (arcos) entre os blocos de código (nós).

Os nós do grafo compreendem comandos que sempre são executados, a partir do primeiro, em sequência. Já os arcos são todos os desvios entre os nós do programa. Os casos de teste, na técnica estrutural, verificam se os arcos estipulados são apropriadamente percorridos. Caso isso ocorra, o conjunto de casos de teste é considerado adequado ao critério (C-adequado).

Dentre os principais critérios para teste estrutural, encontram-se os baseados em fluxo de controle e os baseados em fluxo de dados. Os primeiros são baseados nas estruturas de controle do programa, como os comandos condicionais, estruturas de repetição e chamadas de métodos ou funções. Exemplos mais comuns de critérios exigem que se percorram todos os nós ou arcos pelo menos uma vez – *Todos-nós* e *Todos-Arcos*, respectivamente (Myers et al., 2011; Rapps e Weyuker, 1982) – ou percorrer todos os caminhos, o que na maioria dos casos é inviável por resultar em um número muito grande de requisitos de teste.

Os critérios baseados em fluxo de dados requerem a execução de caminhos que são gerados com base nas definições e usos das variáveis. Esses critérios utilizam uma especialização do grafo de fluxo de controle incluindo as definições e usos das variáveis, chamado *grafo definição-uso* – Grafo Def-Uso (Rapps e Weyuker, 1982). Nesse grafo, em relação ao uso, esse pode ser predicativo (causa um desvio no fluxo do programa, chamado *p-uso*) ou computacional (quando não causa um desvio, chamado *c-uso*). Esses critérios requerem, dentre outras possibilidades, a execução dos caminhos que percorrem nós entre a definição e algum uso de uma variável (*Todas-Defs*), os caminhos que percorrem a definição e todos os usos dessa variável (*Todos-Usos).* Ressalta-se que diversos outros critérios baseados em fluxo de dados podem ser encontrados na literatura, sendo alguns deles específicos para programas desenvolvidos nos paradigmas procedimentais, OO e OA (Harrold e Rothermel, 1994; Lemos et al., 2009; Maldonado, 1991; Rapps e Weyuker, 1982; Vincenzi, 2004).

Teste Baseado em Defeitos

Essa técnica procura gerar os casos de teste com base no conhecimento dos defeitos comuns dos projetistas ou desenvolvedores durante o desenvolvimento dos programas. Um dos critérios do teste baseado em defeitos é a Semeadura de Defeitos (Budd, 1981), no qual um número especificado de defeitos é semeado para verificar se os casos de teste conseguem encontrá-los; observa-se também se os mesmos casos de teste conseguem encontrar defeitos reais no programa.

O critério de Análise de Mutação (ou *teste de mutação*) (DeMillo et al., 1978; Hamlet, 1977), fundamental para o desenvolvimento deste trabalho, cria varias versões do programa original, com modificações sintáticas simples. Os casos de teste são aplicados

nessas versões (chamadas *mutantes*) na tentativa de distingui-las da versão original. O critério será explicado com mais detalhes a seguir.

2.2.2 Teste de Mutação

O objetivo do critério *Análise de Mutantes* – ou *teste de mutação* – é averiguar a qualidade dos casos de teste e também do programa testado (Mathur, 2007; Offutt e Pan, 1997). É baseado na *Hipótese do Programador Competente* e na *Hipótese do Efeito de Acoplamento* (DeMillo et al., 1978). Na primeira hipótese, os programas testados estão corretos ou próximos disso, graças à experiência do desenvolvedor. Na segunda hipótese, assume-se que os defeitos complexos são normalmente compostos ou influenciados por defeitos simples, portanto os casos de teste que revelam pequenos defeitos também revelam os mais complexos.

Com base nessas hipóteses, o critério consiste em criar varias cópias ligeiramente modificadas do programa original. Tais programas modificados são chamados *mutantes* e normalmente são criados de forma automática. Os casos de teste são aplicados aos mutantes com o intuito de distingui-los da versão original. De acordo com número de mutantes distinguidos, um resultado chamado *escore de mutação* é calculado, obtendo-se assim uma medida de qualidade dos casos de teste criados.

Durante a comparação, os mutantes que geram resultados distintos são identificados automaticamente (sendo considerados mortos). Alguns mutantes testados geram resultado igual ao da versão original do programa. Isso significa que a distinção, se existir, não foi detectada; logo, os casos de teste devem ser melhorados ou se trata de um mutante equivalente. Nesse último caso, é impossível criar casos de teste que consigam distinguir os comportamentos do mutante e do programa original.

A aplicação do teste de mutação ocorre em quatro etapas (DeMillo et al., 1978).

1. *Execução dos casos de teste no programa original:* Normalmente, o próprio testador verifica as saídas dos casos de teste no programa original, cujos resultados serão a base da comparação com os mutantes.

2. *Geração dos mutantes:* Operadores de mutação são utilizados para gerar os mutantes com o tipo de alteração desejada. Os operadores são escolhidos e variam de acordo com a linguagem, especificação do software, objetivo da mutação, entre outros. A qualidade do teste de mutação depende dos operadores escolhidos.

3. *Execução dos casos de teste nos mutantes:* Cada caso de teste é executado sobre cada um dos mutantes e suas saídas são comparadas às do programa original. Um

escore pode ser calculado, considerando os mutantes mortos, remanescentes (vivos ou equivalentes) e o total de mutantes gerados.

Dados um Programa P, casos de teste T, número total de mutantes M, número de mutantes mortos DM e número de mutantes equivalentes EM, o escore de mutação, denotado por *MS(P,T)*, é calculado com a seguinte fórmula:

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

O valor de MS(P,T) está sempre no intervalo $[0, 1]$. Quanto mais próximo de 1, mais adequado são os casos de teste.

4. *Análise dos mutantes vivos:* Primeiro é decidido se o teste continuará, baseado na proximidade do escore a 1. Decidindo-se por continuar, os mutantes vivos devem ser analisados para determinar se são equivalentes ao programa original. Para matar os mutantes não equivalentes, novos casos de teste devem ser criados. Ressalta-se que determinar a equivalência entre dois programas, no caso um mutante e o programa original, é indecidível. Portanto, em geral, a automatização de todo o processo não é possível (Offutt e Pan, 1997).

Ressalta-se que não apenas as características internas do teste influenciam em sua qualidade. A atividade de teste e suas respectivas técnicas e critérios interagem e são parcialmente dependentes das estratégias de desenvolvimento adotadas para o software em questão. O teste de mutação foi adaptado, no decorrer dos anos, para realizar o teste apropriadamente em relação às novas formas de desenvolvimento. Uma das formas para a qual o teste foi adaptado é o paradigma orientado a objetos (OO), já que este paradigma é amplamente empregado pela comunidade de desenvolvimento de software. O teste de mutação nesse contexto é abordado na próxima subseção.

Teste de Mutação de Software OO

Embora as características do paradigma OO tenham trazido benefícios para o desenvolvimento de software, elas passaram a representar um novo contexto para a atividade de teste. Neste contexto, alguns defeitos do paradigma procedimental se tornam menos prováveis enquanto outros se tornam mais presentes.

Para criar os casos de teste adequados, uma estratégia de teste deveria ser guiada por modelos de defeitos. Tais modelos servem de diretrizes para apoiar o teste baseado em defeitos. Como exemplo, o modelo defeitos de Offutt et al. (2001) está relacionado à herança e ao polimorfismo, restringindo o modelo às "heranças de subclasse" ou relação

"é um" das subclasses para as superclasses. Os defeitos do modelo podem ser distribuídos em categorias como, por exemplo, o uso de tipos polimórficos inconsistentes, anomalia de definição de estado, entre outras. Nesse sentido, os estudos a respeito do teste de mutação para programas OO começaram em torno duas décadas atrás. Alguns trabalhos têm foco principal na definição de operadores de mutação com base em modelos de defeitos, enquanto outros focam na implementação de ferramentas. Em linhas gerais, o teste de mutação para programas OO pode ser dividido em três categorias:

1. Focado em operadores para características OO como herança, encapsulamento e polimorfismo (Chevalley, 2001; Kim et al., 2000; Ma et al., 2002);

2. Focado em operadores para APIs especificas da linguagem Java (Bieman et al., 2001; Bradbury et al., 2006; Delamaro et al., 2001); e

3. Focado na avaliação de operadores tradicionais em software OO (Vincenzi, 2004).

Na primeira categoria, Kim et al. (2000) propuseram um conjunto de 13 operadores de mutação, chamados de operadores de mutação inter-classe, que abordam encapsulamento, herança e polimorfismo. Esse conjunto foi estendido por Chevalley (2001), com mais três operadores para a linguagem Java.

Ma et al. (2002), baseados no modelo de defeitos de Offutt et al. (2001) e nos conjuntos de operadores de Kim et al. (2000) e Chevalley (2001), definiram mais operadores de mutação inter-classe. Em um trabalho posterior, os autores apresentaram a ferramenta μJava (mujava), que implementa 23 dos operadores além de alguns intra-unidade (Ma et al., 2006).

Na segunda categoria, Delamaro et al. (2001) e Bradbury et al. (2006) abordaram operadores para teste de mutação de Programas Java concorrentes, enquanto Bieman et al. (2001) apresentaram uma técnica para mutação de objetos Java focado em coleções. O último, dentre outros trabalhos, seria usado para a especificação de um subconjunto de operadores para uma ferramenta de mutação Java que apoia a seleção de testes de regressão (Do e Rothermel, 2006).

Em relação à última categoria, observa-se inicialmente que os operadores são dependentes da linguagem alvo, apesar de ser possível o aproveitamento total ou parcial deles em linguagens semelhantes (Delamaro et al., 2001). Nessa linha, Vincenzi (2004) adaptou para as linguagens C++ e Java operadores de mutação originalmente projetados para a linguagem C (Agrawal et al., 1989; Delamaro et al., 2001) e Java (Ma et al., 2002).

2.2.3 Técnicas de Redução de Custo do Teste de Mutação

O critério de teste de mutação, apesar de eficaz, deixa a desejar em eficiência, graças à grande quantidade de mutantes gerados que devem ser executados e analisados. Ressalta-se que parte considerável desses mutantes demandam análise manual, seja para se decidir pela criação de novos casos de teste, ou pela classificação dos mutantes como equivalentes ao programa original. Essa característica do critério gerou um extenso número de trabalhos com novas técnicas de redução do custo.

Inicialmente, (Offutt e Untch, 2001) agruparam as técnicas de redução de custo do teste de mutação nas seguintes categorias:

1. *Do fewer*: Visa a executar um número menor de mutantes.
2. *Do smarter*: Visa a gerir o trabalho computacional por fora do teste, como exemplo, por meio de diversas máquinas, ou reter informação de estado para reutilizá-la posteriormente.
3. *Do faster*: Visa a gerir o trabalho computacional por dentro do teste por meio da geração e execução de mutantes da forma mais rápida possível.

Mais recentemente, Ferrari et al. (2018) relataram uma revisão sistemática da literatura (SLR, do inglês, *Systematic Literature Review*) sobre redução do custo do teste de mutação. Nesse trabalho, foram estimados 6 principais propósitos da redução de custo, encontradas 22 técnicas aplicadas e 18 métricas para medir os resultados. Os principais propósitos da redução de custos encontrados são apresentado em ordem de popularidade decrescente a seguir:

1. Reduzir o número de mutantes;
2. Acelerar a execução dos mutantes;
3. Detectar automaticamente mutantes equivalentes;
4. Gerar automaticamente casos de teste;
5. Reduzir o número de casos de teste ou o número de execuções de casos de teste; e
6. Evitar a criação de determinados tipos de mutantes.

Das 22 técnicas de redução de custo identificadas por Ferrari et al. (2018), a *Análise de fluxo de controle*, *Algoritmos evolucionários* e *Mutação seletiva* foram as mais populares.

A mutação seletiva realiza uma seleção de operadores que não criam um número alto de mutantes ou mutantes difíceis de matar. As mais antigas técnicas aplicadas são: *Mutação aleatória*, que consiste em escolher, aleatoriamente, uma montante do total de mutantes; *Mutação de maior ordem*, que combina duas ou mais mutações em um mesmo programa; e *Mutação fraca*, que verifica se o estado dos mutantes está infectado pouco depois de executar o ponto mutado.

São técnicas de interesse para este trabalho a *Mutação seletiva* (Offutt et al., 1993), *Operadores essenciais* (Barbosa et al., 2001) e, principalmente, *One-Op* (Untch, 2009) Em comum, todas essas técnicas visam a reduzir o conjunto de mutantes baseando-se em características e custos de aplicação dos operadores de mutação.

Também na SLR de Ferrari et al. (2018), identificaram-se métricas que são comumente utilizadas para se medir as reduções de custo obtidas com a aplicação de técnicas diversas. Dentre as métricas, as mais empregadas são: número de mutantes a serem executados; número de casos de teste requeridos para cobertura dos mutantes, aceleração (*speedup*) de execução dos mutantes ou de geração de casos de teste, número de mutantes equivalentes automaticamente detectados. Este trabalho pretende utilizar como métrica para avaliação das técnicas o *número de mutantes que serão executados*. A expectativa é que se consiga reduzir o número total de mutantes, assim diminuindo-se o esforço do teste, porém mantendo-se minimamente sua qualidade.

Na próxima seção será apresentada a perspectiva deste trabalho a respeito de similaridade entre programas. Neste trabalho a similaridade objetiva auxiliar na redução dos custos do teste de mutação. Esse auxílio baseia-se na extração de características dos programas e de sua quantificação com o intuito de comparar os valores resultantes de forma apropriada. As características utilizadas foram as métricas CK (Chidamber e Kemerer, 1994), as quais também serão apresentadas em seção subsequente. As métricas CK foram originalmente projetadas para tratar da coleta e quantificação de valores referentes a programas orientados a objetos.

2.3 Conceitos de Similaridade de Programas

Programas são comparados em muitos contextos diferentes. Alguns exemplos desses contextos, porém não limitados a esses, são: detecção de plagio em códigos, remoção de redundâncias para compressão, e diferenciação de programas – também conhecidos como *diff* – para mostrar as modificações feitas entre versões de sistemas cujo código sofreu manutenções ao longo do tempo. No teste de mutação, o *diff* entre programas é bastante

utilizado para que o testador identifique as diferenças entre o programa original e seus mutantes. Observa-se também que alguns trabalhos que propõem abordagens para realizar o teste de mutação empregam técnicas e/ou algoritmos para realizar a comparação de programas – neste projeto chamada de *cálculo de similaridade* – não apenas para fazer a análise convencional de mutantes, mas também nos passos de geração e execução.

No contexto deste trabalho, foi realizado um Mapeamento Sistemático (MS) sobre o estado da arte do teste de mutação nos paradigmas OO e OA (orientação a aspectos). No MS realizado, foram encontrados alguns estudos que empregaram, de forma pontual, a comparação entre programas, no caso em pequenos trechos de programas, para dar suporte às suas próprias propostas. Ressalta-se, de antemão, que nenhum dos estudos selecionados no MS apresentavam a comparação de programas como o meio para atingir os objetivos propostos neste trabalho. Uma visão geral dos estudos mencionados é apresentada mais adiante, na Seção 2.3.1. Antes disso, apresentam-se algumas perspectivas de definição de similaridade entre programas, de acordo com Walenstein et al. (2007).

Similaridade representacional e similaridade comportamental

Walenstein et al. (2007) apresentaram um trabalho conceito a respeito do que seria a similaridade, apesar da imprecisão natural do termo, dentro do escopo de comparação entre programas, dividindo-a em dois tipos, ambas relacionadas com uma visão tácita do que seria um "Programa": uma visão sintática (representacional), e uma visão semântica (comportamental). Além disso, os autores propuseram algumas categorias de métodos que poderiam ser usados para medir a similaridade baseando-se na ideia de que o nível em que dois programas são distintos está relacionado com o *como* precisamente eles são parecidos. Fica exposto uma posição ambígua em relação à natureza do *medir* por meio de uma similaridade *feature-based* (*feature* remete ao seu uso em *machine learning*).

A similaridade *feature-based* consistiria na procura pela correspondência entre listas desorganizadas de aspectos ou propriedades pré estipuladas entre os programas, e pesos poderiam ser aplicados durante o processo. Métricas poderiam ser consideradas *features* segundo o raciocínio, apesar de, por fim, reconhecê-las como sintáticas. A coleta de informações baseadas em *features* poderia ser considerada uma modalidade de similaridade diferente já que ela não "valora" diretamente a estrutura ou mesmo o significado final do programa ou trecho de código, mas caracteriza por meio de um ponto de referência, até certo ponto, subjetivo. Com o ponto de referência, realiza-se a comparação.

A seguir são apresentadas, resumidamente, as conclusões dos autores sobre os tipos e métodos de obtenção da similaridade entre programas.

- ***Representacional (sintática):*** Genericamente, consiste em uma comparação de estrutura e não de significado. Lida com muitos níveis de abstração, desde a declaração e blocos de código até níveis arquiteturais. Algumas perspectivas possíveis são:

    1. Por métricas: Comparação de valores de métricas de dois fragmentos de código. As métricas de código mapeiam características do código ou sua estrutura para o domínio dos números reais. Ressalta-se que as métricas de programas foram utilizadas neste trabalho para calcular a similaridade. Elas serão abordadas em um próximo subtópico.
    2. *Feature-based*: Consiste na procura de correspondência entre listas desorganizadas de aspectos ou propriedades estipuladas previamente.
    3. *Shared Information*: Programas podem ser comparados através do uso de método de *Shannon's information theory*, considerando-se os dois programas como mensagens de texto. Se os programas forem independentes é possível concatená-los e compará-los com o valor resultado da concatenação através da distância de compressão normalizada.

- ***Comportamental (semântica):*** Refere-se a uma comparação de significado e não de estrutura. Exemplos: uma similaridade funcional usando entradas e saídas; ou a execução procurando correspondências em suas declarações (em relação à sequência de execução delas). Algumas representações possíveis: autômatos, redes de Petri e teoria das funções. Algumas perspectivas possíveis são:

    1. *Execution Curve Similarity*: Os trajetos de execução podem ser pensados como uma curva mapeando a localização do programa em uma dimensão separada. Então os trajetos de execução podem ser comparados com a curva resultado.
    2. *Input-Output Relation Similarity*: Medindo a quantidade de entradas para dois programas que geram uma mesma saída.
    3. *Semantic Distance*: Medir o custo de alterar um programa para se tornar o outro por meio da distância de Levenshtein explicada por Ristad e Yianilos (1998).
    4. *Abstraction Equivalence Distance*: O nível de abstração que um dos programas precisa atingir para se tornar o outro (retirando-se detalhes não pertinentes).
    5. *Program-Dependence Graph Similarity*: Compara-se a estrutura dos programas, tendo como base medidas dos grafos.

2.3.1 Mapeamento Sistemático

A organização do mapeamento sistemático (MS) realizado no contexto deste trabalho foi feita com auxilio da ferramenta *StArt (State of the Art through Systematic Review)* (Zamboni et al., 2010). O planejamento, a condução da pesquisa e a análise de resultados começou em *Setembro de 2016* e estendeu-se até Dezembro de 2017 O objetivo do MS foi *caracterizar o estado da arte em abordagens de teste de mutação para programas OO e OA*.

A aplicação da *string* de busca resultou em 1169 estudos retornados pelos motores de busca. Com a exclusão dos artigos duplicados, 389 artigos foram retirados e 780 artigos restaram. 600 artigos foram rejeitados durante a seleção inicial e 180 artigos foram aceitos. 96 artigos foram aceitos na seleção final e 84 artigos foram rejeitados. As informações extraídas dos artigos aceitos foram definidas no Protocolo da MS, que está descrito em trabalho anterior a este (Dallilo, 2017).

Durante a fase de extração, observou-se que alguns estudos selecionados utilizam heurísticas para calcular a similaridade entre programas. Este fato levou à definição de um novo objetivo para a MS: *Identificar, dentre os estudos aceitos na seleção inicial, aqueles que utilizam heurísticas para cálculo de similaridade entre programas, e descrevê-las*. Em suma, por meio do uso de heurísticas diversas, 9 estudos aceitos na seleção final comparam programas de forma pontual (apenas trechos e para auxiliar suas propostas). Uma breve descrição desses estudos é apresentada a seguir. De antemão, ressalta-se que todos os trabalhos descritos na sequência, diferentemente do trabalho apresentado nesta dissertação, exploram a similaridades entre programas que são sintaticamente muito similares; em particular, tais trabalhos exploram a similaridade entre um programa original e seus mutantes.

O estudo de Bashir e Nadeen (2013)

Os autores abordaram o teste de mutação OO evolucionário e propõem uma função de *fitness* com o objetivo de ajudar na geração de casos de teste efetivos. A função calcula a *branch distance* (distância percorrida no grafo do ponto inicial ao ponto esperado) das variáveis de estado separadamente, enquanto avalia um caso de teste. Também foi proposto uma comparação do fluxo de controle com as saídas (estados dos dados) para determinar se um mutante foi morto. A análise do fluxo de controle também encontra potenciais defeitos de software que estavam presentes, porém ocultos, em mutantes originalmente classificados como equivalentes.

O estudo de Schuler e Zeller (2010)

O estudo explorou a cobertura do código e a probabilidade de um mutante não ser equivalente. Os autores propuseram observar as mudanças de comportamento do mutante e e do programa original enquanto a computação da saída está sendo gerada. A suposição foi que um mutante que altera a estrutura interna é mais provável de alterar a funcionalidade do programa. Se mutações com impacto no decorrer do fluxo forem focadas, é mais provável encontrar alguns mutantes equivalentes. Para realizar as observações, uma análise do fluxo de dados e as mudanças de informações passadas por meio dos métodos, durante a computação, é utilizada. Aplicando-se algumas métricas com as informações anteriores, é gerado um número inteiro t que quantifica a probabilidade dos mutantes não serem equivalentes. O estudo também introduziu a ferramenta *Javalanche* para teste de mutação, e uma ferramenta que realiza a análise do fluxo de controle.

O estudo de Fraser e Zeller (2012)

Nesse estudo, os autores apresentaram uma abordagem automática para gerar testes de unidade que detectem mutantes que contém alterações que não são detectadas durante o teste de mutação. Na abordagem é realizada uma comparação dos grafos de fluxo de controle na tentativa de selecionar casos de teste mais apropriados para contribuir com a abordagem evolucionária proposta.

O algoritmo genético proposto evolui uma população de cromossomos que representam possíveis soluções do problema. A representação genética dos casos de teste é uma sequência de declarações (referentes aos parâmetros das chamadas de métodos). A população inicial de casos de teste é gerada aleatoriamente, porém a próxima geração é selecionada através de sua *fitness*.

A *fitness* dos casos de teste é medida em relação à proximidade do método que sofreu mutação, à proximidade da declaração dentro do método, e ao impacto do mutante no restante da execução. Se o método é executado, mas a declaração que sofreu mutação não for atingida, então a *fitness* especifica a distância do caso de teste para executar o ponto no qual ocorreu a mutação; o objetivo é minimizar esta distância. Para medir a distância entre o caso de teste e seu alvo é usado um grafo de fluxo de controle partindo do momento do desvio de curso em relação ao original. Para isso, é usado o número de dependências de controle não satisfeitas entre os pontos, sendo 0 (zero) se todas as dependências forem atingidas. Além disso, pode-se usar as condições de necessidade definíveis para diferentes operadores de mutação para estimar a distância da execução da mutação que infecta o estado (chamado de distância de necessidade). Nesse trabalho, foi criada a ferramenta μTEST, que é uma extensão da Javalanche.

O estudo de Moghadam e Babamir (2014)

A abordagem proposta pelos autores consiste em calcular o escore total de um programa levando-se em consideração as diferentes complexidades internas que as classes possuem da perspectiva dos tipos de operadores de mutação empregados (clássicos, classe ou integração). A abordagem utiliza algumas métricas CK para cálculo de complexidade. A abordagem utiliza uma média de métricas para os tipos de operadores clássicos, classe e integração respectivamente: WMC (*Weighted methods per class*), DIT (*Depth of inheritance tree*) e RFC (*Response for class*). Ressalta-se que as métricas CK são importantes para este trabalho e, assim sendo, apresentadas em detalhes na próxima seção. Após o cálculo individual, uma normalização foi realizada sobre os escores parciais para se obter, através de média, um escore final. No estudo, utilizou-se a a ferramenta MuJava, e os testes foram implementados em JUnit.

O estudo de Anbalagan e Xie (2008)

O estudo teve por objetivo reduzir o número de mutantes gerados pela mutação de designadores de pointcuts, que são construções típicas de programas orientados a aspectos (OA), e que definem padrões de casamento de código utilizados pelo compilador de programas OA durante a combinação de classes e aspectos (de forma simples, um pointcut define os pontos de junção entre os aspectos e classes de um programa). Foi proposto um *framework* para gerar mutantes relevantes e detectar automaticamente os equivalentes (nesse caso, a equivalência ocorre quando um pointcut mutante casa com o mesmo conjunto de pontos de junção quando comparado com o pointcut original). O *framework* classifica e ordena os mutantes utilizando uma medida de similaridade sobre *strings* para ajudar o desenvolvedor a escolher mutantes parecidos com o original. A distância léxica entre *strings* dos *pointcuts* original e mutante é definida por um número de caracteres que precisam ser inseridos, excluídos ou modificados em relação ao original para transformá-lo no mutante; essa é a chamada *distância de Levenshtein*. A classificação e ordenação dos mutantes são feitas sob essa perspectiva de distância.

O estudo de Kintis e Malevris (2013)

O estudo trabalha sobre o problema dos mutantes equivalentes e a necessidade da análise manual dos mutantes para identificá-los. Busca-se encontrar mutantes do mesmo local e operador que sejam parecidos de forma léxica. A ideia é descobrir se eles são "espelhados"; se um for equivalente, o outro provavelmente também será. Um estudo empírico foi realizado com programas empresariais. A ferramenta utilizada foi a MuJava com o CCFinderX (Uma ferramenta detectora de clonagem, baseada em *tokens*).

O estudo de Just et al. (2011)

O trabalho apresentou a ferramenta MAJOR (*Mutant analysis in a Java compiler*), que consiste em uma ferramenta de rápida semeadura de defeitos e teste de mutação integrada ao compilador padrão Java para ser usada como uma melhora não invasiva em qualquer ambiente de desenvolvimento Java. A MAJOR utiliza uma abordagem de mutação condicional que gera mutantes transformando a árvore sintática abstrata (AST). A ferramenta também utiliza as expressões condicionais e declarações para encapsular todos os mutantes e a versão original do programa em um mesmo bloco básico. Um mutante que não foi alcançado e executado não pode ser detectado sobre nenhuma circunstância; para não avaliar mutantes desse tipo, informações sobre a cobertura são coletadas em tempo de execução. Os casos de teste são executados com o programa original colhendo informações de cobertura e tempo de execução. Apenas o mais rápidos casos de teste são executados para cada mutante.

O estudo de Xu e Ding (2010)

Também no contexto de teste de mutação de programas OA, o estudo de Xu e Ding (2010) explicou a forma como programas podem ser testados de forma incremental; primeiro as classes bases e depois essas classes com os aspectos. A explicação foi apresentada para propor uma priorização do teste dos aspectos para revelar defeitos mais cedo. Isso é explorado por meio do teste dos programas OA contra seus modelos de estado. Os testes dos aspectos são gerados dos modelos combinados de classes e aspectos, Os caminhos e transições são escolhidos baseando-se no nível de alteração que o aspecto gera no código base. As comparações foram realizadas através de uma árvore de transição de estados e a composição léxica das transições em si. A ferramenta utilizada foi o MACT (*Model Based Aspect checking and testing*, criada pelos autores). Foram empregadas as técnicas de *Transition coverage* para gerar testes que cobrem pelo menos uma vez as transições de modelo de estados), e *round trip* para gerar testes que possibilitem que cada resultado do estado de um objeto, em cada sequência, apareça pelo menos uma vez em outra sequência).

O estudo de Namin et al. (2015)

Nesse estudo, os autores levaram em consideração o problema da equivalência de mutantes e propuseram criar um ranking baseado na dificuldade de expor o defeito simulado pelos mutantes. A estratégia consiste de começar a geração focando a criação de casos de teste para os mutantes mais e menos complexos, dependendo do objetivo. A abstração utilizada foi um grafo de fluxo de controle, uma representação da sintaxe do programa

Jimple e os rastro de execução dos teste. O cálculo de similaridade é uma distancia léxica das representações (Distância de Hamming, similar à de Levenshtein). As ferramentas utilizadas foram o MuRanker (que utiliza a MuJava), o Soot (jimple e grafo de fluxo de controle, o grafo feito pela linguagem intermediaria DOT), e o JaCoCo (código de cobertura, relatório da cobertura em XML).

Na seção seguinte é apresentada uma breve explicação sobre a natureza das métricas internas de software utilizadas neste trabalho como base para se calcular a similaridade entre programas.

2.4 Métricas Internas de Software

O objetivo das métricas de software é identificar e medir as principais características que incidem sobre o desenvolvimento de software (Putnam e Myers, 2003). Portanto, as métricas ajudam a decidir e encontrar formas de aperfeiçoar artefatos de software (Fentom e Bieman, 2014) e, por consequência, Assim elas contribuem no processo de desenvolvimento de software de qualidade.

Segundo Li (1999), as métricas podem ser divididas em dois tipos: Produto ou Processo. O primeiro tipo refere-se a métricas que são aplicáveis no produto final (por exemplo, no código fonte). Já o segundo tipo refere-se a métricas aplicáveis no processo de desenvolvimento como, por exemplo, o esforço para desenvolver em tempo hábil.

As questões que impactam a qualidade dos produtos software podem ser divididas, superficialmente, em questões relacionadas a métricas internas (linhas de código, número de classes entre outros), e questões relacionadas a atributos externos (manutenibilidade, extensibilidade e reusabilidade, dentre outros). Ressalta-se que as métricas relacionadas a atributos externos são definidas indiretamente com auxilio do primeiro grupo. Além disso, no escopo da avaliação de projetos OO, as métricas internas poderiam ser divididas em 3 categorias: de análise, de projeto e de construção. As duas últimas categorias têm forte relação com artefatos de software, portanto são propensas à extração por meio de análise automática (Jones, 2008). As métricas também podem ser divididas entre dinâmicas (extraídas de um programa em execução) e estáticas (extraídas através dos artefatos em geral).

Uma série de trabalhos abordam métricas internas relativas ao tamanho ou complexidade de software (Abdellatief et al., 2013). Dentre esses nichos de métricas, alguns exemplos são:

- Métricas de ***Tamanho:*** Têm por objetivo quantificar o tamanho do software. Um exemplo seria a métrica LOC (*lines of code*) que conta a quantidade de linhas de código (subtraindo linhas em branco e comentários) com o propósito de auxiliar na predição da complexidade de software e esforço de desenvolvimento entre outros.

- Métricas de ***Halstead:*** Quantificam a complexidade do software através de operadores e operandos em um módulo do sistema usando o código fonte. Avaliam a qualidade do código e a tendência de aumento de complexidade.

- Métrica de ***McCabe:*** A complexidade corresponde ao número máximo de percursos independentes (originados das condições presentes no código). A métrica pode ser representada pelo grafo de fluxo de controle (McCabe, 1976).

- Métricas de ***Software Orientado a Objetos:*** Aborda características do desenvolvimento de software OO como classe, métodos, herança, acoplamento entre outros. As métricas de software OO são pertinentes para o objetivo deste trabalho. Seguem alguns exemplos de estudos baseados em métricas no paradigma.

  - Métricas de ***Li e Henry (1993):*** Focadas em medir propriedades internas como acoplamento, complexidade e tamanho.
  - Métricas ***MOOD de Abreu e Carapuca (1994)***: compõem um conjunto de métricas relacionadas a herança, encapsulamento, acoplamento e polimorfismo. O propósito do conjunto é avaliar, em termos de produtividade, o desenvolvimento e qualidade de software.
  - Métricas ***CK de Chidamber e Kemerer (1994):*** medem a complexidade de um projeto em relação a atributos externos de qualidade. Elas foram utilizadas neste trabalho para quantificar valores de classes para futuras comparações. As métricas CK incluem as seguintes métricas:
    - ***CBO (Couplig between Object Classes):*** Acoplamento entre objetos de classe, refletido pelo número de interações com classes diferentes da classe escolhida tanto de métodos quanto de variáveis. Quanto maior o acoplamento, maior a dificuldade de manutenção, reuso e mais rigorosos devem ser os testes.
    - ***DIT (Depth of Inheritance Tree):*** Mensura a profundidade na árvore de herança. Quanto maior o valor, maior será a quantidade de código herdado e sua complexidade.

- ***LCOM (Lack of Cohesion of Methods):*** Falta de coesão dos métodos da classe. Todos os métodos são verificados em pares contando todos os pares que não possuem atributos de classe em comum e subtraindo no final de todos aqueles que possuem. Se o número de pares que não utilizam for menor, o valor da métrica será 0. Portanto, a métrica apenas avalia falta de coesão e não a coesão dos métodos. Um baixo LCOM promove utilização do encapsulamento na classe; caso contrário, indica-se que a classe poderia ser dividida, assim como um aumento do risco de erro na fase de desenvolvimento.
- ***NOC (Number of Children):*** Número de filhos da classe. A reutilização de código e um possível uso impróprio da herança são proporcionais ao aumento da métrica.
- ***RFC (Response for Class):*** Resposta para uma classe. Seu valor é igual à quantidade de métodos da classe somados à quantidade de métodos externos utilizados pela classe. A complexidade da classe aumenta proporcionalmente ao valor da métrica.
- ***WMC (Weighted Methods per Class):*** Complexidade dos métodos da classe. É possível utilizar complexidade ciclomática proposta por McCabe (1976), que consiste na soma de todos os caminhos distintos dentro do método. Depois basta somar o valor da complexidade dos métodos e obter o valor para a classe.

A subseção seguinte aborda algoritmos de clusterização. Esses algoritmos foram utilizados para calcular a proximidade dos programas através das métricas CK colhidas de suas classes.

2.5 Algoritmos de Agrupamento (Clusterização)

Agrupamento, ou clusterização (do inglês, *clustering*) é o processo de detecção de similaridades entre exemplares, descobrindo assim sua disposição em um grupo. Dentre os tipos de algoritmos que aplicam agrupamento, destacam-se, dentre outros: hierárquico, por partição, rede neural artificial, e K-Means.

O algoritmo de clusterização K-Means é a base do algoritmo X-Means que foi utilizado neste trabalho para agrupar as métricas CK das classes para, por meio dos *clusters* gerados, inferir classes ou programas similares. Na próxima seção apresentam-se o funcionamento e as características do algoritmo K-Means.

2.5.1 O Algoritmo K-Means

O K-Means é um dos algoritmos mais empregados para implementar o processo de agrupamento de exemplares. Ele é utilizado em trabalhos que envolvem o aprendizado de máquinas (do inglês, *machine learning*) como um aprendizado não supervisionado. No aprendizado não supervisionado, não se sabe o que será ensinado ao computador. Portanto, utiliza agrupamentos lógicos que segmentam os dados das amostras em busca de padrões que o computador poderá utilizar sempre que for solicitado.

Levando em consideração que se pode selecionar manualmente conjuntos diferentes de métricas e que seus valores podem divergir em relação aos diferentes tipos de programas, além de que o número de classes a serem analisadas na aplicação do framework conceitual definido no contexto deste trabalho é desconhecido, decidiu-se utilizar o K-Means, ou seja, um algoritmo de aprendizado não supervisionado. O *K-Means* encontra um número fixo K de grupos separados de exemplares em um conjunto de dados. Para isso os *K's* (centroides) são dispostos aleatoriamente em relação ao conjunto de dados, criando "setores" (os *clusters*) nesses dados. Os *clusters* são divididos na metade da distância entre os centroides. Então, os centroides são posicionados em relação ao seu *cluster* e o processo itera redimensionando os *clusters* dos centroides e os centralizando novamente. Para reposicionar o centroide em relação ao *cluster*, é utilizada a seguinte fórmula, apresentada por Boscarioli et al. (2016), na qual: $X \rightarrow_g$ é um exemplar do conjunto de dados associados ao centroide $c \rightarrow_p$, sendo que, p vai de 1 até k (número de *clusters*) e G é o número de exemplares associados ao centroide.

$$c \rightarrow_p = \frac{1}{G} \sum_{g=1}^{G} X \rightarrow_g$$

Para descobrir o centroide ao qual um dado exemplar pertence, é necessário comparar a distância do valor (numéricos) do exemplar com os dos centroides que podem estar divididos através de sua distância euclidiana (ou distancia métrica, comprovada por usos repetidos do teorema de Pitágoras, no caso uma linha imaginária que liga os dois pontos por uma linha).

Existem alguns problemas com o algoritmo como à aleatoriedade dos pontos iniciais dos centroides, que não garantirem uma distribuição final ideal. Outro problema refere-se ao número fixo e simultaneamente não ideal de *clusters* que devem ser estabelecidos em relação às informações oferecidas.

Na próxima subseção o algoritmo X-Means, uma versão do K-Means, é apresentado.

2.5.2 O Algoritmo X-Means

Criado por Pelleg e Moore (2000), o algoritmo de agrupamento X-Means tenta sanar algumas limitações do K-Means como, por exemplo, a baixa escalabilidade computacional, e a necessidade de inserir manualmente K. A técnica provou-se mais rápida do que o uso repetitivo de K-Means para encontrar o valor mais adequado de K.

O algoritmo começa com o menor valor K, em intervalo dado, e continua a adicionar centroides enquanto eles são necessários até o fim do intervalo. O melhor resultado é, então, utilizado. O algoritmo executa apenas duas operações:

- ***Melhorar os parâmetros:*** Executar K-Means até a convergência (o ponto onde poucas alterações acontecem ao reposicionar os centroides).

- ***Melhorar a estrutura:*** A operação encontra, se e onde, novos centroides devem aparecer. Isso é possível permitindo que alguns dos centroides se dividam. Para decidir se devem ou não se dividir, é utilizada a seguinte estratégia: Inicia-se dividindo cada centroide em dois que se movem para uma direção aleatória proporcional ao tamanho da região. Em seguida, em cada região (*cluster* do centroide pai) é executado um K-Means local com ($K = 2$) para cada um dos pares de filhos. Depois disso, uma *model selection test*[1] é realizada em todos os pares de filhos para identificar se exite alguma evidência de que os filhos estão modelando uma estrutura real, ou se o centroide pai modela a distribuição tão bem quanto; dependendo do resultado, ou o pai ou os filhos são apagados. Com isso, centroides pais bem distribuídos vão sobreviver, assim como novos centroides que aparecerem em regiões não tão bem representadas. A decisão de divisão é realizada através do cálculo de *Bayesian Information Criterion* (BIC). Uma *Bayesian inference* é relativa à montante da plausibilidade de um valor (ou proposição) randômico em relação a outro valor randômico. O BIC tenta evitar o *overfitting*, que é a produção de uma análise que corresponda, total ou parcialmente, a um conjunto especifico de dados, dessa forma não conseguindo ajustar dados adicionais ou prever observações futuras de forma confiável.

[1] É a tarefa de testar a seleção de um modelo estatístico de um grupo de modelos candidatos de uma determinada fonte de dados. Ou também a seleção de um baixo grupo de modelos representativos de um grande grupo de modelos computacionais com o propósito de auxiliar em decisões ou otimizar em relação à incerteza.

2.6 Considerações Finais

Neste capítulo foram descritos os principais conceitos que permeiam a atividade de teste. Tais conceitos incluem o critério de teste de mutação aplicado em software desenvolvido com o paradigma orientados a objetos (OO), e estratégias de redução de custo. Também abordou-se a similaridade de programas, métricas internas de software e algoritmos de agrupamento, os quais serão importantes para a compreensão deste trabalho.

No próximo capitulo uma abordagem para apoiar estratégias de redução de custo do teste de mutação é apresentada.

Capítulo

3

Abordagem para Apoiar a Redução de Custo do Teste de Mutação baseada em Similaridade de Programas

3.1 Considerações Iniciais

Neste capítulo é apresentada a abordagem para a redução de custo do teste de mutação com base na similaridade de programas. A abordagem é definida por meio de um *framework* conceitual, o qual é descrito em termos de passos a serem executados, e as respectivas entradas e saídas de dados e informações.

Na Seção 3.2 apresenta-se conceitos básicos da notação BPMN utilizada nas imagens que apresentam os processos do *framework* conceitual. São descritos, detalhadamente, os processos gerais do *framework* (Seção 3.3 e alguns cenários de uso Seção 3.3). Também são apresentados, detalhadamente, os subprocessos da composição do grupo de referência de programas, que é o primeiro processo do diagrama geral do *framework* (Seção 3.4). Por fim, na Seção 3.5 apresentam-se os subprocessos e tarefas que computam, pré-processam e armazenam novos programas na base de programas.

3.2 Linguagem BPMN

BPMN (Business Process Model and Notation) é uma linguagem de modelagem de processos disponibilizada atualmente pela OMG (Object Management Group)[1] que é uma organização internacional que aprova padrões abertos para aplicações orientadas a objetos. A linguagem BPMN possibilita a compreensão de procedimentos internos e estabelece uma comunicação padronizada entre eles. A BPMN fornece o diagrama BPD (Business Process Diagram) que é baseado no fluxo de trabalho semelhante ao padrão UML. A linguagem apresenta um grande conjunto de elementos para a representação dos processos. Neste trabalho utilizou-se notação baseada na BPD, os seguintes elementos foram utilizados:

- Evento: Corresponde às esferas nos diagramas. Os eventos influenciam no curso do fluxo do processo devido a uma causa ou impacto. Neste trabalho, eles delimitam o início (verde) e o fim (laranja) de um processo.
- Objetos de conexão: Corresponde à forma como os eventos, atividades e objetos de dados do processo se conectam. Neste trabalho existem dois tipos: o fluxo de sequência (seta preta, que corresponde a ordem em que o processo ocorre) e associação (usada para associar artefatos de diferentes tipos através de uma seta pontilhada. Utiliza-se a cor vermelha para representar entradas, e a cor azul para representar saídas.
- Atividade: Corresponde aos retângulos brancos nos diagramas. Referem-se ao que acontece durante o andamento do processo. Caso a atividade possua um quadrado contendo uma cruz ela pode ser dividida em um processo separado. Nesse caso, a atividade pode ser denominada como subprocesso. Caso contrário, ela pode ser chamada de tarefa.
- Objeto de dados: corresponde aos retângulos cinzas nos diagramas. São artefatos de tipos variados produzidos ou requeridos por uma atividade.
- *Gateways*: Corresponde ao losango nos diagramas. Controlam como um fluxo diverge ou converge ao longo de sua execução.
- Cabeçalho: Normalmente em um canto do diagrama, o cabeçalho apresenta o nome dos diferentes tipos de fluxo do diagrama.

[1] As informações referentes à linguagem foram coletadas e baseadas nas informações presentes no site oficial do OMG. Elas podem ser obtidas no site a seguir: https://www.omg.org/spec/BPMN/2.0/

3.3 Processo Geral do Framework conceitual

Na Figura 3.1 ilustra-se o processo geral[2] para apoiar a redução de custo do teste de mutação com base em dados históricos e na similaridade entre programas. A seguir apresenta-se a leitura geral do fluxo representado na figura, seguida por uma descrição de todos os elementos (Seção 3.3.2).

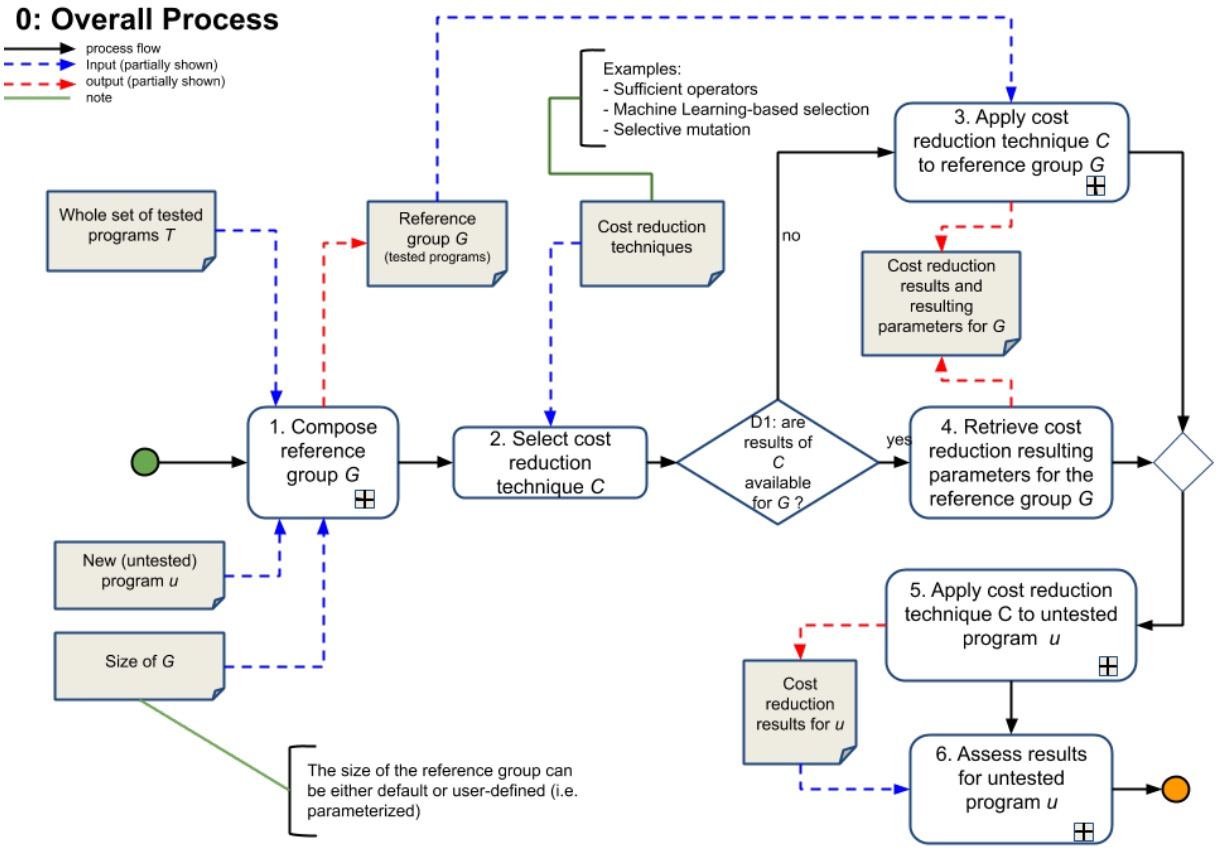

Figura 3.1: Processo geral.

O processo se inicia pela composição de um grupo de referência G (subprocesso 1), que incluirá os programas que mais se assemelham a um programa ainda não testado u. Em seguida, seleciona-se uma técnica de redução do custo do teste de mutação C (atividade 2), que será inicialmente aplicada nos programas do grupo G (subprocesso 3) e, em seguida, aplicada no programa u (subprocesso 5). Ressalta-se que os resultados obtidos da aplicação da técnica C nos programas de G servirão como base para a aplicação de C

[2]De antemão, ressalta-se por questões de consistência entre documentos de trabalho que estão em elaboração pelo grupo de pesquisa no laboratório LaPES, os rótulos de todos os elementos que compõem o processo foram mantidos em língua inglesa.

no programa u. Por fim, os resultados do teste com custo reduzido, obtidos para o programa u, serão avaliados (subprocesso 6). Ressalta-se que o subprocesso 6 está incluído no processo geral para fins de experimentação apenas; em um uso prático do framework conceitual, o processo se encerraria com a execução do subprocesso 5.

Como fluxo alternativo, caso a técnica C já tenha sido aplicada nos programas do grupo G (resposta afirmativa para questão que rotula o nó de decisão $D1$), os resultados serão recuperados (atividade 4) para embasar a aplicação de C no programa u.

Outros dois processos têm grande importância para que o processo geral se concretize. Ambos são descritos nas Seções 3.4 e 3.5. Antes disso, apresentam-se alguns cenários de uso do framework conceitual (Seção 3.3.1), e detalham-se os subprocessos e atividades do processo geral (Seção 3.3.2).

3.3.1 Exemplos de Cenários de Uso

Nesta seção apresentam-se dois cenários de uso do framework conceitual, dentre vários outros possíveis (por exemplo, um terceiro cenário foi implementado na avaliação apresentada no Capítulo 5 desta dissertação). O primeiro cenário considera a aplicação da técnica Operadores Essenciais (Barbosa et al., 2001), e o segundo cenário considera a aplicação de técnicas de otimização baseada em aprendizagem de máquina.

Cenário 1 – Reutilizando Operadores Essenciais: Neste cenário, considera-se que operadores essenciais de mutação serão calculados para um determinado grupo de programas, e em seguida serão aplicados em um programa ainda não testado, com a expectativa de que os resultados sejam similares. Passo-a-passo, tem-se:

- (Subprocesso 1) O grupo de programas de referência G é formado
- (Atividade 2) A técnica de redução de custo C (Operadores Essenciais) é selecionada
- (Decisão D1) A resposta "não" é obtida, ou seja, operadores essenciais ainda não foram calculados para G
- (Subprocesso 3) A técnica C é aplicada no grupo G
- (Subprocesso 5) Os operadores essenciais são aplicados no programa não testado u
- (Subprocesso 6) Os resultados para u são avaliados, por exemplo, com respeito ao escore de mutação obtido, e com respeito aos resultados obtidos para o grupo G

Ao final do processo, o programa u pode passar a integrar a base de programas testados (grupo T), desde que um conjunto de testes C-adequado seja inicialmente obtido, ou se o conjunto inicialmente obtido for evoluído para se tornar C-adequado.

Cenário 2 – Reutilizando um Algoritmo de Aprendizagem de Máquina Treinado: Neste cenário, considera-se que um algoritmo de aprendizagem de máquina, capaz de classificar mutantes com relação à sua utilidade (por exemplo, dominador, trivial ou equivalente (Ammann et al., 2014; Kurtz et al., 2016)), já tenha sido treinado com programas que compõem o grupo G. A expectativa é que o algoritmo consiga classificar de forma satisfatória mutantes de um programa ainda não testado. Passo-a-passo, tem-se:

- (Subprocesso 1) O grupo de programas de referência G é formado
- (Atividade 2) A técnica de redução de custo C (classificação baseada em aprendizagem de máquina) é selecionada
- (Decisão D1) A Resposta "sim" é obtida, ou seja, o algoritmo de classificação já está treinado com programas do grupo G
- (Atividade 4) Os parâmetros de configuração do algoritmo são recuperados
- (Subprocesso 5) O classificador é aplicado a mutantes do programa não testado u
- (Subprocesso 6) Os resultados para u são avaliados, por exemplo, com respeito à precisão do classificador, e com respeito ao escore de mutação obtido com os mutantes selecionados

De forma análoga ao cenário 1, ao final do processo, o programa u pode passar a integrar a base de programas testados (grupo T), desde que se obtenha um conjunto de testes C-adequado para ele.

3.3.2 Detalhamento do Processo Geral

Nesta seção são descritos os subprocessos e atividades que compõem o processo geral do framework conceitual. Para cada elemento (subprocesso ou atividade), apresenta-se a descrição geral, as entradas e as saídas. Para efeitos de melhor visualização, ressalta-se que somente parte das entradas e saídas são representadas nas figuras dos processos apresentadas ao longo deste capítulo.

1 - Compose reference group G (Compor o grupo de referência G): Este processo consiste em se compor o grupo de programas testados G que são similares ao novo programa ainda não testado u. Um programa é considerado testado quando para o mesmo se tem um conjunto de testes C-adequado com respeito ao teste de mutação. Ademais, o tamanho do grupo G pode ser tanto pré-definido quanto definido pelo usuário.

- Entradas:
  - O conjunto completo de programas testados T
  - O programa não testado u
  - (opcional) O tamanho do grupo G
- Saída:
  - O grupo de referência G
- Notas adicionais:
  - Para mais detalhes, consultar o processo *Composition of the Reference Group (G)* na Seção 3.4

2 - Select cost reduction technique C (Selecionar a técnica de redução de custo C): Esta atividade consiste na seleção de uma técnica de redução de custo do teste de mutação C para ser aplicada nos programas do grupo G e também no programa u. Exemplos[3] de técnicas de redução de custo são *mutação seletiva*, *operadores essenciais*, *algoritmos evolucionários*, e técnicas otimizadas baseadas em aprendizagem de máquina. Se resultados da aplicação de C em G já estiverem disponíveis, os mesmos são recuperados (processo 4); caso contrário, eles serão calculados (processo 3).

- Entrada:
  - Um conjunto de técnicas de redução de custo disponíveis
- Saída:
  - A técnica de redução de custo C selecionada

[3]Uma lista extensa de técnicas pode ser encontrada no trabalho de Ferrari et al. (2018).

3. Apply cost reduction technique C to reference group G (Aplicar a técnica de redução de custo C no grupo de referência G): Este processo consiste em se aplicar a técnica de redução de custo C nos programas do grupo G. Os resultados e os parâmetros de execução de C para G são armazenados. Exemplos de parâmetros são um grupo de operadores essenciais de mutação, ou um algoritmo treinado de aprendizagem de máquina. Tais parâmetros serão reutilizados para a aplicação de C em u.

- Entradas:
  - O grupo de referência G
  - A técnica de redução de custo C selecionada
- Saídas:
  - Os resultados R_G obtidos com a aplicação de C em G
  - Os parâmetros S de configuração de C

4 - Retrieve cost reduction resulting parameters for the reference group G (Recuperar parâmetros de redução de custo para o grupo de referência G): Esta atividade consiste em recuperar parâmetros S resultantes a aplicação da técnica de redução de custo C no grupo de referência G.

- Entradas:
  - O grupo de referência G
  - A técnica de redução de custo C selecionada
- Saída:
  - Os parâmetros S de configuração de C

5. Apply cost reduction technique C to untested program u (Aplicar a técnicas de redução de custo C no programa não testado u): Este processo consiste em aplicar a técnica de redução de custo C no programa não testado u, tendo como base os parâmetros de configuração S obtidos com a aplicação de C nos programas do grupo de referência G.

- Entradas:
  - O programa não testado u

  - A técnica de redução de custo C selecionada
  - Os parâmetros S de configuração de C

- Saída:
  - Os resultados R_u da aplicação de C, configurada com os parâmetros S, em u

6 - Assess results for untested program u (Avaliar os resultados para o programa não testado u): Este processo consiste na avaliação dos resultados R_u obtidos com a aplicação dá técnica C no programa u. Dentre diversas opções de avaliação, pode-se levar em consideração o escore de mutação produzido pelos testes adequados ao conjunto enxuto de mutantes quando aplicados ao conjunto completo de mutantes. Outra opção é a comparação do ganhos em termos de redução de custo quando o grupo de referência G e o programa u são considerados.

- Entradas:
  - Os resultados de redução de custo R_u
  - O conjunto completo de mutantes
  - (opcional) Os resultados R_G obtidos com a aplicação de C em G

- Saída:
  - Variadas (dependendo do objetivo da avaliação que está sendo realizada)

3.4 Processo de Composição do Grupo de Referência de Programas

O processo representado na Figura 3.2 visa a compor o grupo de referência G. Inicialmente, são computadas as medidas (também chamadas de *abstrações*) para o programa não testado u (subprocesso 1.1). Em seguida, seleciona-se e aplica-se a abordagem de cálculo de similaridade desejada (atividade 1.2 e subprocesso 1.3, respectivamente), e conclui-se o processo com a geração do grupo G (atividade 1.4). Detalhes são apresentados a seguir.

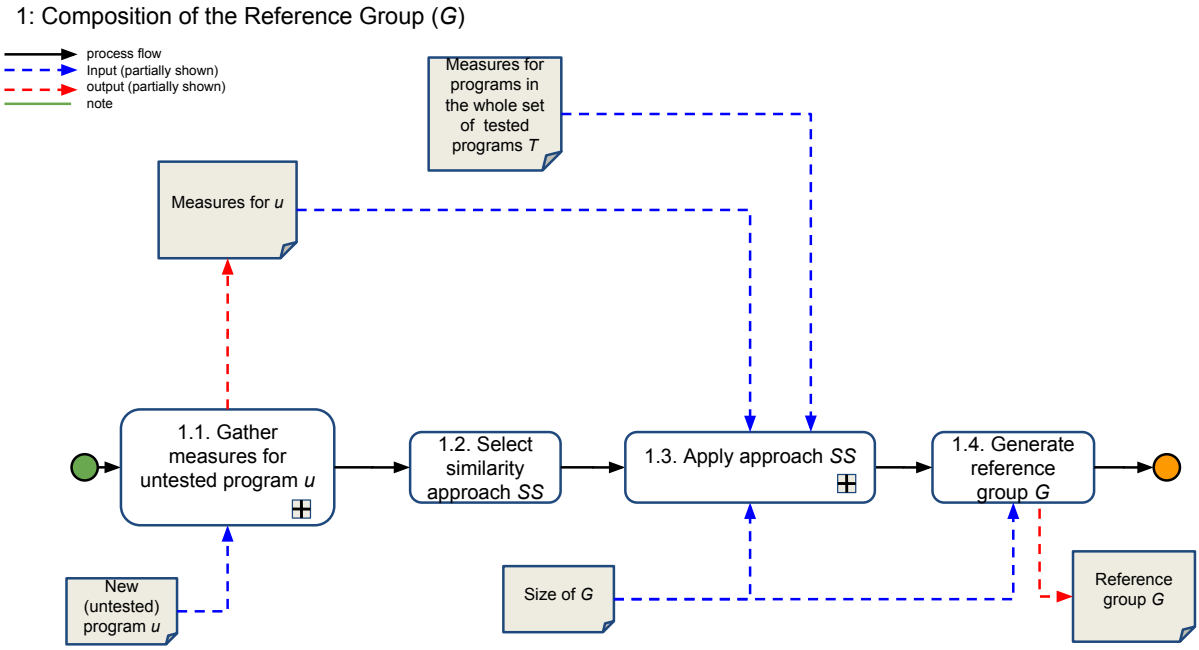

Figura 3.2: Composição do grupo de referência G.

3.4.1 Detalhamento do Processo de Composição do Grupo de Programas de Referência

1.1 - Gather measures for untested program u (Obter medidas para o programa não testado u): Este processo consiste em se extrair as medidas do programa não testado u que serão utilizadas para se computar a similaridade entre u e programas já testados. As medidas (também chamadas de *abstrações* neste trabalho) representam os programas sob uma determinada perspectiva. Exemplos de medidas seriam conjuntos de valores de métricas internas, grafos de fluxo de controle, e código-fonte ofuscado. Mais detalhes sobre a obtenção de medidas podem ser obtidos na Seção 3.5, na qual se descreve o processo para composição do grupo completo de programas.

- Entrada:
  - O programa não testado u
- Saída:
  - Um conjunto de medidas A_u referentes a u

- Notas adicionais:
  - Para mais detalhes, consultar o processo *Composition of the Whole Group (T)* na Seção 3.5

1.2 - Select similarity approach SS (Selecionar a abordagem de cálculo de similaridade SS): Esta atividade requer a seleção de uma abordagem SS que calcula a similaridade entre programas sob perspectivas pré-definidas (por exemplo, as perspectivas citadas no processo 1.1 descrito nesta seção). Opções já identificadas que podem ser inseridas na implementação do framework conceitual são o agrupamento de programas (clusterização) e o cálculo do nível de diversidade de informação presente em um grupo de programas, conforme definido por Feldt et al. (2016) em trabalho prévio que envolveu conjuntos de casos de teste. De fato, a abordagem de agrupamento de programas foi inserida no framework, conforme descrito no Capítulo 4 desta dissertação.

- Entrada:
  - Um conjunto de abordagens (ou estratégias) de cálculo de similaridade de programas
- Output:
  - A abordagem de cálculo de similaridade SS selecionada

1.3 - Apply approach SS (Aplicar a abordagem SS): Este processo consiste em aplicar a abordagem de cálculo de similaridade SS selecionada, de forma a obter o grupo de programas de referência G que são mais similares ao programa não testado u.

- Entradas:
  - O conjunto de medidas A_u referentes a u
  - O conjunto de medidas medidas A_T referentes aos programas que compõem o grupo completo de programas testados T
  - (opcional) O tamanho do grupo G
- Saída:
  - Versão preliminar do grupo de referência G, o qual pode incluir o programa não testado u (por exemplo, no caso de formação de grupos – ou *clusters* – de programas a um dos quais u esteja associado)

1.4 - Generate reference group G (Gerar o grupo de referência G): Esta atividade consiste em gerar a versão final do grupo de referência G, que será utilizado como base para se aplicar a técnica pretendida de redução de custo do teste de mutação.

- Entradas:
  - Versão preliminar do grupo de referência G
  - (opcional) O tamanho do grupo G
- Saída:
  - Versão final do grupo de referência G

3.5 Processo de Composição do Grupo Completo de Programas

O processo representado na Figura 3.3 tem duas finalidades: computar as medidas (ou abstrações) de programas (subprocessos 2.1 e 2.2) e compor a base histórica de programas testados T (atividade 2.3). Mais especificamente, as abstrações computadas servirão de entrada para se calcular a similaridade entre programas e gerar o grupo de referência G, conforme especificado no processo da Figura 3.2. Ademais, a execução do processo de forma iterativa resulta na criação de um grupo de programas testados T, o qual servirá como base para se compor o grupo de referência G. Detalhes são apresentados a seguir.

3.5.1 Detalhamento do Processo de Composição do Grupo Completo de Programas

2.1 - Compute abstractions for new program p (Computar abstrações para um novo programa p): Este processo visa a computar as medidas (também chamadas de abstrações) para um novo programa p. Entende-se por *novo programa* como sendo um programa que ainda não faz parte do conjunto completo de programas testados T (por exemplo, o programa não testado u mencionado nas seções anteriores).

- Entradas:
  - O novo programa t

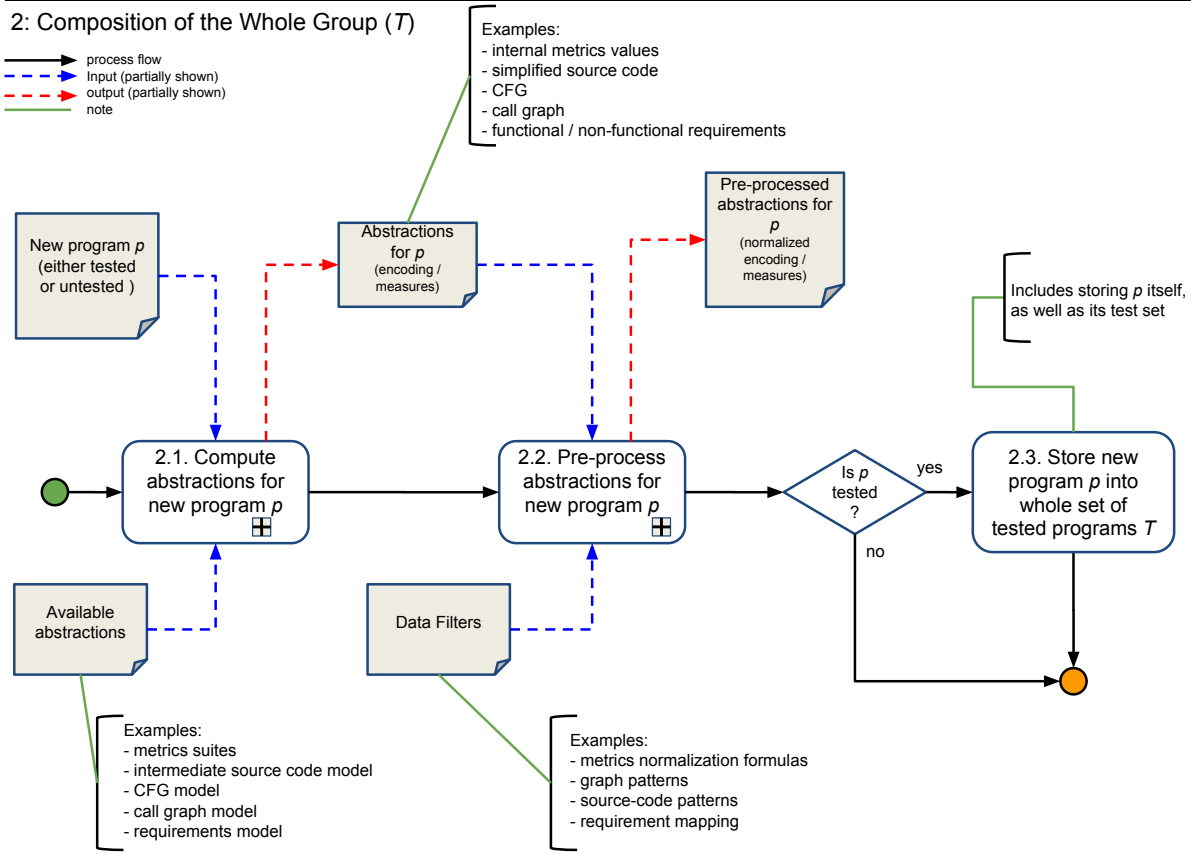

Figura 3.3: Composição do grupo completo de programas testados T.

  - Um conjunto de abstrações que podem ser computadas para o programa t

- Saída:
  - As medidas (ou abstrações) específicas do programa p

- Notas adicionais:
  - Informações relacionadas encontram-se na descrição do processo 1.1 (*Gather measures for untested program u*), na Seção 3.4

2.2 - Pre-process abstractions for new program p (Pré-processar abstrações para o novo programa p): Este processo consiste em normalizar (ou padronizar) as medidas (ou, abstrações) computadas para o programa p. Os resultados desse pré-processamento são armazenados para uso posterior. Exemplos de pré-processamento seriam a normalização de valores para um intervalo $[0, 1]$ ou a identificação de padrões de estruturas de grafos de fluxo de controle ou de código-fonte.

- Entradas:
  - As medidas (ou abstrações) específicas do programa p
  - Filtros de dados que podem ser aplicados sobre as medidas computadas para o programa p
- Saída:
  - As medidas (ou abstrações) pré-processadas para o programa p

2.3 - Store new program p into whole set of tested programs T (Armazenar novo programa t no conjunto completo de programas T): Esta atividade consiste em armazenar no grupo completo de programas testados T o novo programa p juntamente com seu conjunto de testes. Nota-se que p somente será armazenado em T caso se disponha de um conjunto de testes C-adequado para ele, especificamente para o critério de teste de mutação.

- Entrada:
  - O novo programa p e seu conjunto de testes
- Saída:
  - Grupo completo de programas testados T atualizado (incluindo p)

3.6 Considerações Finais

Neste capítulo foi apresentada uma abordagem, representada como um *framework* conceitual, para a redução de custo do teste de mutação apoiada por informações sobre a similaridade entre programas. Subprocessos do *framework* conceitual contemplam a criação do grupo de referência e do grupo total de programas, sendo ambos fundamentais para a abordagem proposta.

No próximo capítulo apresenta-se uma iniciativa de implementação do *framework* proposto, na qual foram contemplados passos fundamentais como, por exemplo, a obtenção de medidas de programa, o cálculo de similaridade baseado na técnica de agrupamento (do inglês, *clustering*) de programas, e a aplicação de uma técnica de redução de custo do teste de mutação inspirada na técnica *One-Op* Untch (2009). Resultados de um experimento realizado com os mecanismos automatizados são apresentados no Capítulo 5.

Capítulo

4

Aspectos de Automatização

4.1 Considerações Iniciais

Neste capítulo, aspectos da automatização do framework conceitual são descritos. A descrição contempla as principais dependências e processos do protótipo implementado, assim como um mapeamento entre os módulos do protótipo e o framework conceitual descrito no capítulo anterior. Ressalta-se que deste ponto em diante, o protótipo será mencionado como *ferramenta de similaridade*.

Em específico, na Seção 4.2 apresentam-se as principais dependências da ferramenta de similaridade, sendo elas as ferramentas Analizo e Weka. Na Seção 4.3, uma visão geral da ferramenta é apresentada; para tal, são apresentados alguns diagramas, são descritos os processos principais, e são discutidos alguns detalhes de implementação como, por exemplo, o processo de normalização de valores extraídos de programas analisados pela ferramenta. Por fim, na Seção 4.4 apresenta-se o mapeamento da ferramenta de similaridade ao framework conceitual.

4.2 Principais Dependências

Nesta Seção descrevem-se duas ferramentas (Analizo e Weka) que foram utilizadas dentro da ferramenta de similaridade que implementa parcialmente o framework conceitual descrito no capítulo anterior.

4.2.1 A Ferramenta Analizo

Criada por Terceiro et al. (2010), a Analizo é um conjunto de ferramentas de código aberto (do inglês, *open source*) disponibilizado sob a Licença GNU (*General Public License Version 3*). O conjunto de ferramentas é multi-linguagem, e realiza extensa análise de código. A seguir, listam-se as análises realizadas pela versão 1.21.0 da Analizo, que é a versão utilizada neste trabalho.

- ***DSM:*** Desenha uma matriz de estrutura de projeto de um grafo de chamadas.
- ***Evolution-Matrix:*** Gera uma matriz evolucionária dos arquivos de métricas *.yml* da Analizo.
- ***Graph:*** Gerador de grafo de dependência.
- ***Metrics, Metrics-Batch e Metrics-History:*** Respectivamente, é a ferramenta de métricas da Analizo, a capacidade de processar vários diretórios de código fonte em lote, e a capacidade de processar um repositório *Git* de coleção de métricas.
- ***Tree-Evolution:*** Permite se observar a evolução do código fonte.

Neste trabalho, a ferramenta Analizo foi utilizada para extração de métricas em projetos de programas Java. A ferramenta extrai 10 métricas em nível de projeto e 16 em nível de classe (ou, em nível de módulo). Em nível de projeto, a ferramenta calcula estatísticas de 13 formas diferentes sobre todas as métricas de nível de classe. Também calcula as métricas das classes individualmente, extraindo 16 tipos diferentes.

A seguir, apresenta-se um exemplo do comando executado no terminal para extração de métricas, sendo que neste trabalho tal comando foi invocado via uma classe utilitária implementada em Java. O exemplo tem como alvo o projeto de nome *Projeto*, no diretório de nome *Diretório-Usuário*. Os resutados são armazenados no arquivo *metrics* no formato *yaml*. Nota-se que os termos que aparecem em itálico (*analizo metrics*) representam os comandos da ferramenta Analizo que são invocados.

- `Diretório-Usuário` *`analizo metrics`* `Projeto > metrics.yaml`

A ferramenta apresenta as 16 métricas, em nível de classe, sendo 6 delas as métricas CK apresentadas no Capítulo 2 e mais 10 apresentadas na Tabela 4.1.

Tabela 4.1: Métricas além das CK, a nível de classe, apresentadas pela Analizo.

Siglas	Métricas
acc:	Conexões aferentes por classe (usada para calcular COF - Fator de acoplamento).
amloc:	Número médio de linhas de código por método.
anpm:	Número médio de parâmetros por método.
loc:	Linhas de código.
mmloc:	Máximo de linhas de código por método.
noa:	Número de atributos.
nom:	Número de métodos.
npa:	Número de atributos públicos.
npm:	Número de métodos públicos.
sc:	Complexidade estrutural.

4.2.2 A Ferramenta Weka

A Weka (*Waikato Environment for Knowledge Analysis*) é uma coleção de algoritmos de aprendizado de máquina para tarefas de mineração de dados (do inglês, *data mining*). Possui código aberto sob a Licença GNU (*General Public License, version 3.0*). A Weka contém ferramentas para preparação de dados, classificação, regressão, clusterização, visualização entre outros. Na versão 3.8.3, utilizada neste trabalho, requer o Java JRE versão 1.8 para funcionar (Frank et al., 2016).

Todos os algoritmos recebem suas entradas na forma de uma tabela relacional que pode ser lida de um arquivo ou de uma consulta em uma base de dados. Com a Weka, pode-se aplicar um método de aprendizado em uma base de dados e analisar seus resultados, usar modelos de aprendizado para gerar predições em novas instâncias, ou aplicar diferentes *learners* e comparar seu desempenho para escolher um para predição.

Os arquivos ARFF (*Attribute-Relation File Format*) foram desenvolvidos para serem usados com a Weka. Um ARFF é um arquivo de texto padrão no formato ASCII que descreve uma lista de instâncias com um conjunto de atributos.

O arquivo pode ser dividido em duas seções distintas, a de Cabeçalho e a dos Dados. O Cabeçalho contém o nome da relação, uma lista de atributos (relativa a colunas em tabelas) e seus tipos. Os Dados seriam as linhas das tabelas relativas as colunas predefinidas no Cabeçalho. Um exemplo de conteúdo de um arquivo ARFF é apresentado na Figura 4.1.

A Weka pode ser instalada com sua IDE ou utilizada em um projeto através de seu arquivo executável Java (do inglês, *Java ARchive*, ou simplesmente arquivo ".jar"). Na ferramenta de similaridade descrita neste capítulo, os principais pacotes da Weka uti-

```
1 @RELATION Exemplo
2
3 @ATTRIBUTE codigo NUMERIC
4 @ATTRIBUTE nome STRING
5
6 @DATA
7
8 5.1,'Ana'
9 4,'Joao'
```

Figura 4.1: Exemplo de arquivo ARFF.

lizados foram *weka.clusterers* e *weka.core*. As principais classes utilizadas, do pacote *weka.clusterers*, são apresentadas a seguir.

- ***ClusterEvaluation:*** Como o próprio nome apresenta, essa classe avalia os *clusters* gerados através do objeto X-Means que os criou. Para criar os *clusters* o objeto X-Means faz uso do objeto *Instances* cujas caracteristicas são apresentas logo a seguir.

- ***X-Means:*** Como mencionado no capitulo 2, a biblioteca X-Means foi adicionada para a aplicação do algoritmo de agrupamento de mesmo nome. Portanto, a biblioteca não pertence à Weka, mas utiliza os recursos de seu pacote *clusterers* e de outros pacotes. A instância desta classe é responsável por configurar e criar o *cluster* com a aplicação do algoritmo e parâmetros necessários (objeto *Instances*).

As principais classes utilizadas, do pacote *weka.core*, são apresentadas a seguir.

- ***Instances:*** O objeto é a representação do arquivo ARFF (as informações do objeto ajudam a compor o Cabeçalho). Para criá-lo, é preciso passar o nome da relação, o tamanho estimado da relação, e uma lista de objetos *Attribute*. Para populá-lo, basta adicionar objetos do tipo *DenseInstance*. Ele é o objeto utilizado como parâmetro para gerar os *clusters*.

- ***Attribute:*** O objeto contém o nome de um atributo especifico (neste trabalho, uma métrica). Em um arquivo ARFF, um objeto *Attribute* ajudaria a compor o Cabeçalho, com os nomes e tipos dos atributos.

- ***DenseInstance:*** Cria um objeto de um subtipo de *Instance*. Ele representa uma instância (conjunto de valores dos atributos de uma linha na relação; no caso deste trabalho, tais valores são referente às métricas de um projeto ou classe especifica). Para criá-lo, é preciso passar a quantidade de atributos do objeto *Instances*. Popula-se com valores reais e normalizados (preferencialmente). Em um arquivo ARFF, uma instância de *DenseDistance* corresponderia aos Dados.

4.3 Visão Geral da Ferramenta de Similaridade

O objetivo da ferramenta desenvolvida neste trabalho é dividir em subgrupos uma massa de programas baseado em sua similaridade. Desenvolvida na forma de um protótipo para apoiar estudos exploratórios, a ferramenta realiza quatro funções principais: a coleta de métricas (pela ferramenta Analizo); a normalização dos valores; a conversão dos dados em objetos para serem enviados como dados de entrada para a ferramenta Weka; e a criação e apresentação dos *clusters* gerados (com ajuda da biblioteca X-Means). O diagrama de classes da ferramenta é apresentado na Figura 4.2. Na Tabela 4.2 caracteriza-se o esforço de implementação da ferramenta em termos de linhas de código (LOC) implementadas. No total, aproximadamente 3 mil linhas de código foram implementadas neste trabalho. Nota-se que os dados mostrados na tabela não consideram código reutilizado (por exemplo, código das bibliotecas da Weka ou da ferramenta Analizo).

A seguir descreve-se o diagrama em partes, de acordo com as processos implementados na ferramenta. Inicia-se apresentando detalhes da fórmula de normalização utilizada.

4.3.1 Normalização linear (Max-Min) ou normalização por interpolação linear

A normalização é um processo de transformação linear dos dados para prepará-los, por exemplo, para a aplicação de algum algoritmo de mineração, como redes neurais artificiais ou métodos baseados em distância (Goldschmidt e Passos, 2005). Para isso, cria-se um domínio em relação aos valores máximo e mínimo de um determinado atributo. Isso possibilita colocar todos os valores deste tipo de atributo em um mesmo intervalo de valores, neste caso [0,1]. Neste trabalho, os projetos ou classes foram agrupadas através de diferentes métricas extraídas deles. Como o intervalo numérico entre as métricas pode variar muito, graças à sua natureza, uma métrica pode causar um impacto muito maior que as outras. A normalização foi aplicada para garantir que as métricas possuiriam o mesmo peso no momento da geração dos *clusters*. A seguir, apresenta-se a fórmula de

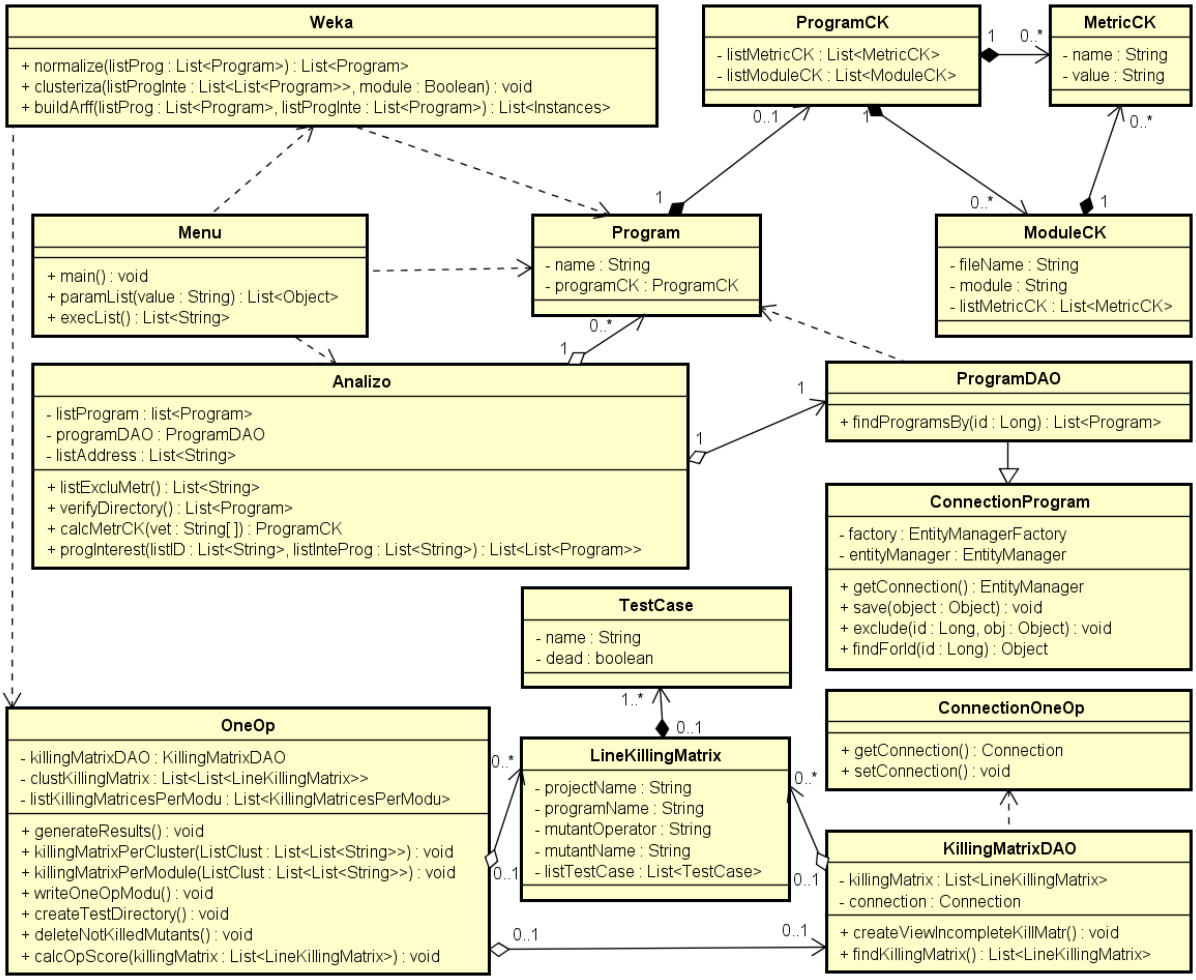

Figura 4.2: Diagrama de classes da Ferramenta.

normalização utilizada neste trabalho, na qual A' é o valor normalizado, A é o valor a ser normalizado, Max e Min são o maior e o menor valores encontrados para o tipo de atributo, respectivamente.

$$A' = \frac{A - Min}{Max - Min}$$

4.3.2 Processos da ferramenta de similaridade

A ferramenta inicia sua atividade lendo um arquivo em formato textual no diretório *paramList* dentro do diretório do projeto. Neste arquivo, o usuário deve especificar os processos que deseja executar usando a ferramenta. São dois processos diferentes que podem ser usados ao mesmo tempo ou separadamente: de inclusão (-i); e de agrupamento (-c). A classe `Menu` e seus métodos, na Figura 4.2, são responsáveis por essas tarefas.

Tabela 4.2: Esforço de implementação da ferramenta desenvolvida neste trabalho (classes indicadas com "*" são classes internas não representadas no diagrama da Figura 4.2.)

Classe	loc
TestCase	12
MetricCK	18
ModuleCK	28
ProgramCK	18
Program	18
LineKillingMatrix	30
ProgramDAO	48
KillingMatrixDAO	209
ConnectionProgram	72
ConnectionOneOp	36
Menu	367
Analizo	768
Weka	562
OneOp	525
*InstanceProgramInterest	26
*MinMax	17
*KillingMatricesPerModu	24
*LineOpScore	17
*InstanceModuleInterest	38
*ModuleParam	14
Total	**2867**

Na Figura 4.3 apresenta-se o processo de inclusão. Esse processo é realizado pela classe `Analizo`. Segue, resumidamente, uma descrição das tarefas que constituem o processo, assim como suas entradas e saídas.

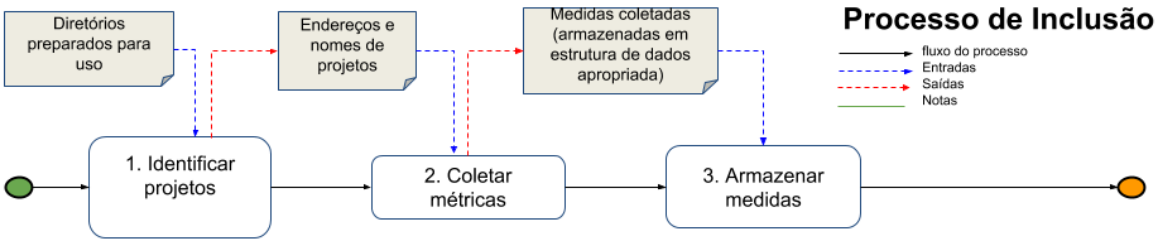

Figura 4.3: Processo de Inclusão implementado na ferramenta.

1. ***Identificar projetos:*** O comando de inclusão não recebe parâmetros. O código inicia verificando os "diretórios preparados para uso" (que contém os projetos a serem cadastrados), através do método `verifyDirectory`. Para realizar essa tarefa, o método verifica o conteúdo de *CompareDir*, o qual é um subdiretório do diretório de instalação da ferramenta. Em *CompareDir*, podem existir arquivos textuais e dentro deles, por linha, caminhos completos para diretórios no sistema de arquivos,

nos quais se encontram os projetos a serem incluídos. É possível criar nos diretórios apontados um arquivo textual de nome *banList*, o qual contém os nomes, por linha, de todos arquivos a serem desconsiderados da inclusão no respectivo diretório. Cada um dos endereços e nomes de projetos são adicionados em uma lista.

- Entrada:
  - Diretórios preparados para uso.
- Saída:
  - Endereços e nomes de projetos.

2. ***Coletar métricas:*** Separadamente, os endereços e nomes obtidos na tarefa *Identificar projetos* são organizados em uma *string* que será passada para a execução por linha de comando da ferramenta Analizo. A Analizo calcula um conjunto de métricas em nível de projeto e de classes, e disponibiliza o resultado dessas métricas no próprio console. Então, respeitando as regras sintáticas da apresentação das métricas, um *parsing* é realizado, através do método `calcMetrCK` da classe `Analiso`, para mapear as medidas coletadas e que se encontram armazenadas em estrutura de dados apropriada (no caso, em objetos Java).

   - Entrada:
     - Endereços e nomes de projetos.
   - Saída:
     - Medidas coletadas (armazenadas em estrutura de dados apropriada).

3. ***Armazenar medidas:*** As medidas coletadas na tarefa *Coletar métricas* são cadastrados na base de dados pelo do método `save`, o qual é implementado na classe `ConnectionProgram` e herdado pela classe `ProgramDAO`.

   - Entrada:
     - Medidas coletadas (armazenadas em estrutura de dados apropriada). T

Na Figura 4.4 apresenta-se o processo de agrupamento, realizada pelas classes `Analizo` e `Weka`. Segue, resumidamente, uma descrição das tarefas que constituem o processo, assim como suas entradas e saídas.

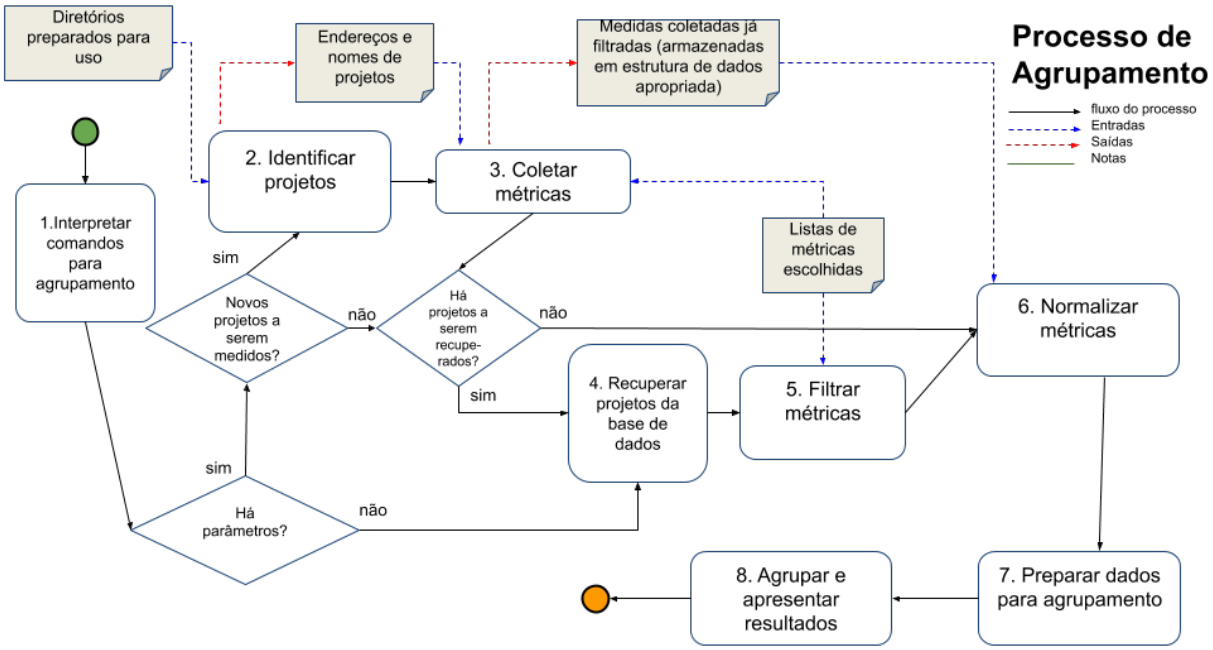

Figura 4.4: Processo de agrupamento da ferramenta de similaridade

1. ***Interpretar comandos para agrupamento:*** A clusterização possui dois tipos de parâmetros. O primeiro tipo se refere aos parâmetros referentes à massa de projetos a serem utilizados na geração dos *clusters*. O segundo tipo se refere aos projetos de interesse que também estarão contidos na massa. A interpretação dos comandos é realizada pelos mesmos métodos apresentados no processo de inclusão. Tanto programas em diretórios especificados por *CompareDir* (o que levaria o processo à tarefa 2) quanto programas de dentro da base de dados (o que eventualmente levaria o processo a executar a tarefa 4), sendo esses informados com parâmetros de ID(s) da base de dados ou intervalos de ID(s), podem ser adicionados à massa de programas, sendo ou não de interesse. Também é possível executar a clusterização sem informar projetos específicos para a massa de projetos (o que levaria à tarefa 4 diretamente). Nesse caso, todos os programas disponíveis da base seriam selecionados para a massa de projetos.

    - Entrada:
        - Comando de agrupamento, com ou sem parâmetros.

2. ***Identificar projetos:*** A atividade é equivalente à tarefa 1 do processo de Inclusão.

    - Entrada:
        - Diretórios preparados para uso.

- Saída:
  - Endereços e nomes de projetos.

3. ***Coletar métricas:*** Executa de forma similar à tarefa 2 do processo de Inclusão. Contudo, durante a atividade, apenas as métricas desejadas são armazenadas nos objetos. Este processo de filtragem funciona sobre as mesmas regras que a tarefa 5 deste processo. A classe `Analizo` coleta apenas as métricas desejadas dos projetos nos diretórios através do método `progInterest`.

   - Entrada:
     - Endereços e nomes de projetos.
   - Saída:
     - Medidas coletadas já filtradas (armazenadas em estrutura de dados apropriada)

4. ***Recuperar projetos da base de dados:*** Esta tarefa recupera os projetos e suas métricas referentes aos código e intervalos de código especificados nos parâmetros da tarefa 1. Caso nenhum parâmetro seja especificado, esta tarefa retornará todos os programas previamente cadastrados. Na ferramenta, esta tarefa atividade é realizada pelo método `progInterest` da classe `Analizo` e resgatada da base de dados pelo método `findProgramsBy` da classe `ProgramDAO`).

   - Entrada:
     - Parâmetros do comando de agrupamento.
   - Saída:
     - Medidas recuperadas da base de dados (armazenadas em estrutura de dados apropriada).

5. ***Filtrar métricas:*** Para filtrar as métricas não desejadas, são utilizados dois subdiretórios presentes no diretório de instalação da ferramenta. O primeiro, de nome $metrBanDirProj$, contém um arquivo textual com as métricas, por linha, que devem ser adicionadas (isso porque as estatísticas geradas pela ferramenta *Analizo* são apresentadas juntas com as métricas de projeto, criando um grande número de valores). O segundo subdiretório, de nome $metrBanDir$, contém um arquivo textual com as métricas, por linha, que devem ser excluídas (devido ao pequeno número de

métricas por classe disponibilizadas). A lista de métricas é obtida através do método `listExcluMetr)` da classe `Analizo` durante a execução do método `progInterest`.

- Entrada:
  - Listas de métricas escolhidas.
  - Medidas recuperadas da base de dados (armazenadas em estrutura de dados apropriada).
- Saída:
  - Estrutura de dados filtrada por métricas.

6. ***Normalizar métricas:*** Após obter o conjunto de objetos referentes aos projetos e suas métricas, os valores das métricas desses objetos são normalizados. Na ferramenta, a normalização é realizada pelo método `normalize` da classe `Weka`.

   - Entrada:
     - Estrutura de dados filtrada por métricas.
   - Saída:
     - Estrutura de dados normalizada.

7. ***Preparar dados para agrupamento:*** Os objetos cujos valores de métricas foram normalizados são alterados para um tipo de objeto (*Instances*) similar a um arquivo ARFF (*Attribute-Relation File Format*) com auxilio da biblioteca *Weka*. Essa tarefa é realizado pelo método `buildArff` na classe `Weka`.

   - Entrada:
     - Estrutura de dados normalizada.
   - Saída:
     - Objeto *Instances*.

8. ***Agrupar e apresentar resultados:*** Com o auxilio da biblioteca X-Means (uma extensão da biblioteca Weka), os *clusters* são gerados dos objetos de tipo equivalente ao formato de arquivos *ARFF* e posteriormente exibidos no console em ordem de *cluster* e programa, respectivamente. Essa tarefa é realizada pelo método `clusteriza` presente na classe `Weka`.

- Entrada:
  - Objeto *Instances*.
- Saída:
  - *Clusters* e informações gerais.

Na Figura 4.5 apresenta-se o processo de identificação dos melhores operadores de mutação (do inglês, *One-Op*) para uma determinada classe, o qual é realizado pela classe `OneOp`. Segue, resumidamente, uma descrição das tarefas que constituem o processo, assim como suas entradas e saídas.

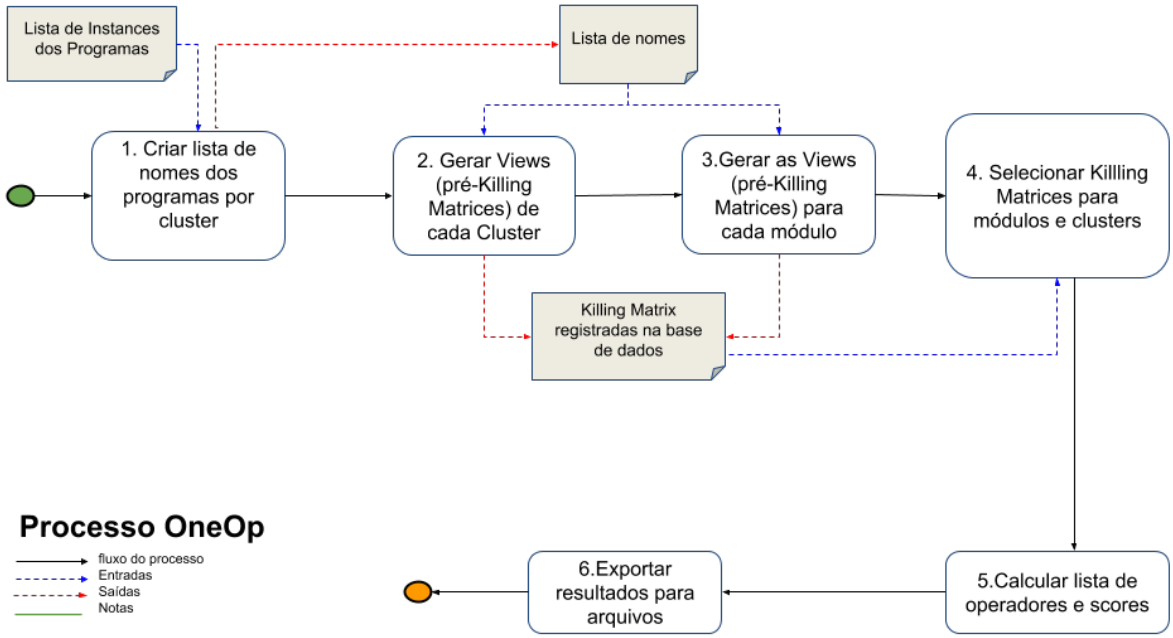

Figura 4.5: Processo de identificação do One Operator

1. ***Criar lista de nomes dos programas por* cluster*:*** Inicialmente, recebe-se a lista de objetos *Instances* gerados pela classe `Weka`. Os nomes dos módulos (classes) dos projetos são extraídos e organizados por *cluster*. Também se cria um diretório para conter os resultados da rodada do experimento (representada pela conclusão dos processos passados por parâmetro nos documentos de texto antes da execução da ferramenta). Dentro desse diretório cria-se um arquivo para cada *cluster* através do método `createTestDirectory` na classe `OneOp`. A tarefa é realizada pelo método `generateResults` presente na classe `OneOp`.

- Entrada:
  - Lista de Objetos *Instances*.
- Saída:
  - Lista de nomes de programas organizados por *cluster*.

2. ***Gerar* Views (pré-Killing Matrices) *de cada* cluster*:*** Com a lista de nomes por *cluster*, é gerada uma *view* (neste trabalho chamada de pré-*killing matrix*) na base de dados para cada *cluster* com as informações de todos os programas (previamente testados com o teste de mutação) que a constituem. As informações que constituem a *View* são os nomes por linha: do projeto, módulo (ou, classe) operador de mutação, mutante, e caso de teste. A tarefa é realizada pelo método `killingMatrixPerCluster` presente na classe `OneOp` e pelo método `createViewIncompleteKillMatr` na classe `KillingMatrixDAO`.

   - Entrada:
     - Lista de nomes por *cluster*.
   - Saída:
     - *Views* (pré-*killing matrix*) dos *clusters* na base de dados.

3. ***Gerar as* Views (pré-Killing Matrices) *para cada módulo:*** Para cada módulo pertencente a cada *cluster*, é gerada uma *view* com as mesmas características apresentadas na tarefa 2. Também é gerada uma *view* para todos os programas restantes do mesmo *cluster*. A tarefa é realizada pelo método `killingMatrixPerModule` presente da classe `OneOp` e pelo método `createViewIncompleteKillMatr` da classe `KillingMatrixDAO`.

   - Entrada:
     - Lista de nomes por *cluster*.
   - Saída:
     - *Views* (pré-Killing Matrix) dos módulos e do restante dos módulos dos *clusters* na base de dados.

4. ***Selecionar Killing Matrices para módulos e clusters:*** Seleciona, da base de dados, de cada *View* criada nas últimos tarefas: o nome do projeto, o programa, o operador de mutação, o mutante, e os caso de teste e seus resultados para estes. A tarefa é realizada pelo método da classe `OneOp` apresentado nas últimas duas tarefas (`killingMatrixPer-Module`) e pelo método `findKillingMatrix` da classe `KillingMatrixDAO`. As informações são armazenadas em objetos do tipo `LineKillingMatrix`, enquanto a lista de casos de teste é armazenada em objetos do tipo `TestCase`.

   - Entrada:
     - *Views* previamente cadastradas na base de dados.
   - Saída:
     - Objetos do tipo `LineKillingMatrix`.

5. ***Calcular lista de operadores e escores:*** Para cada *killing matrix* é calculado o escore de mutação por operador. Para calcular o escore são encontrados os casos de teste que matam os mutantes gerados pelo operadores e se verifica o quanto eles matam do total de mutantes. Também é encontrado o número de mutantes vivos (não são mortos por nenhum caso de teste) através do método `deleteNotKilled-Mutants()` da classe `OneOp`. Então, o número total de mutantes mortos é dividido pelo total de mutantes desconsiderando-se os ainda vivos. Neste trabalho é uma pré-condição que os programas que compõem a bases de dados possuam conjuntos de teste C-adequados para o teste de mutação; dessa forma, qualquer mutante vivo é automaticamente considerado equivalente e, assim sendo, são desconsiderados para o cálculo do escore de mutação. O processo é realizado pelo método `calcOpScore` da classe `OneOp`. Os resultados são armazenados em objetos do tipo `LineOpScore` que contém um operador e seu escore.

   - Entrada:
     - Objetos do tipo `LineKillingMatrix`.
   - Saída:
     - Objetos do tipo `LineOpScore`.

6. ***Exportar resultados para arquivos:*** Nos arquivos criados na primeira tarefa (*Criar lista de nomes dos programas por cluster*) são colocadas as informações provenientes dos módulos e seus respectivos objetos `LineOpScore`. Apenas o operador

com o maior escore é adicionado nos arquivos. O processo é realizado pelo método `writeOneOpModu()` da classe `OneOp`.

- Entrada:
  - Objetos do tipo `LineOpScore`.
- Saída:
  - Arquivos textuais por cluster dos módulos e seus operadores e respectivos escores.

4.4 Mapeando a Ferramenta de Similaridade ao Framework conceitual

Como apresentado no Capítulo 3, em relação ao processo completo do *framework* conceitual representado na Figura 3.1 (*Overall Process*), é necessário compor um grupo de referência para posteriormente aplicar as técnicas de redução de custo escolhidas.

A Figura 3.2, no Capítulo 3, apresenta um processo que corresponde à primeira atividade (*Compose reference group G*) do processo apresentado na Figura 3.1 (*Overall Process*). Essa atividade gera um grupo de referência criado a partir da aplicação de uma estratégia de similaridade. A primeira atividade (*Gather measures for untested program u*) da Figura 3.2 foi representado no processo contido na Figura 3.3 (*Composition of the Whole Group (T)*), que tem por objetivo computar, pré-processar e cadastrar as abstrações de um novo programa *p*.

O mapeamento dos processos implementados na ferramenta com os processos do *framework* conceitual seguirá a ordem de execução da ferramenta, contemplando-se, nessa ordem, os processos conceituais representados nas Figuras 3.3 (*Composition of the Whole Group (T)*), 3.2 (*Composition of the Reference Group (G)*), e 3.1 (*Overall Process*).

O mapeamento, baseado nas atividades dos processos ilustrados nas Figuras 4.3 (Processo de Inclusão), 4.4 (Processo de Agrupamento), e 4.5 (Processo One-Op) , é apresentado a seguir. Nota-se que a numeração utilizada a seguir é a mesma utilizada nas atividades que compõem os processos apresentados nas Figuras 3.3, 3.2, e 3.1.

- *Composition of the Whole Group (T)* (Figura 3.3): Este processo do *framework* pode ser parcialmente mapeado para o primeiro processo da ferramenta, sendo ele o Processo de Inclusão.

  - *2.1. Compute abstraction for new program p*: Como próprio nome sugere, o processo computa medidas (abstrações) para um novo programa. Considerando-se o Processo de Inclusão (Figura 4.3), esta atividade pode ser mapeada nas atividades 1 (Identificar projetos) e 2 (Coletar métricas). Na atividade 1, os programas de interesse são encontrados nos diretórios, e na atividade 2, com ajuda da ferramenta Analizo, as métricas dos programas são coletadas. As atividades 1 e 2 também podem ser executadas no Processo de Agrupamento apresentado na Figura 4.4.
  - *2.2. Pre-process abstractions for new program p*: Esse processo consiste em padronizar (neste trabalho foi realizada a normalização) as medidas computadas e colhidas pela atividade anterior. Esta atividade não pode ser mapeada no Processo de Inclusão da ferramenta. Isso ocorre porque as métricas coletadas são cadastradas diretamente na base de dados, sem normalização. A normalização dessas métricas ocorre no Processo de Agrupamento apresentado na Figura 4.4, na atividade 6 (Normalizar métricas).
  - *2.3. Store new program p into whole set of tested programs T*: O objetivo desta atividade é armazenar o programa coletado nas últimas atividades junto com o seu conjunto de testes C-adequado em uma base de dados. Esta atividade não pode ser mapeada na ferramenta implementada. A ferramenta de similaridade não contempla o cadastro das informações dos testes dos programas. As informações provenientes dos testes dos programas adicionados neste experimento foram cadastradas com ajuda de *scripts* criados por membros da equipe de pesquisa.

- *Composition of the Reference Group (G)* (Figura 3.2): Este processo do *framework* inclui o processo (*Composition of the Whole Group (T)*). Essa inclusão ocorre em sua primeira atividade, que já foi abordada nos parágrafos anteriores. Suas outras atividades, apresentadas a seguir, podem ser mapeadas para o segundo processo da ferramenta, o Processo de Agrupamento.

  - *1.2. Select similarity approach SS*: Após coletar as métricas do novo programa (atividade omitida neste processo, que corresponde ao processo *Composition of the Whole Group (T)*), esta atividade promove a seleção de uma abordagem que calcula a similaridade sob perspectivas pré-definidas. Como apenas uma abordagem de cálculo de similaridade foi implementada na ferramenta (clusterização), ela ocorre por padrão no Processo de Agrupamento apresentado

pela Figura 4.4. As atividades 2 até 7 podem ser executadas em consequência dos parâmetros escolhidos na atividade 1 (Interpretar comandos para agrupamento). Dessas atividades, as de número 1, 6 (Normalização) e 7 (Preparar dados para agrupamento) são obrigatórias em combinação com as outras.

  - *1.3. Apply approach SS*: Esta atividade (que constitui um processo não detalhado no *framework* conceitual,) aplica a abordagem selecionada na atividade anterior. Esta atividade pode ser mapeada na atividade 8 (Agrupar e apresentar resultados) do Processo de Agrupamento automatizado pela ferramenta. Ressalta-se que na atividade 7 as métricas recém normalizadas são convertidas para objetos do tipo *Instances*, e na atividade 8, com ajuda da biblioteca X-Means, os *clusters* são gerados.
  - *1.4. Generate reference group G*: O objetivo desta atividade é gerar a versão final do grupo de referência G para a aplicação da técnica de redução de custo. Esta atividade pode ser parcialmente mapeada à atividade 8 do Processo de Agrupamento, e à atividade 1 (Criar lista de nomes dos programas por *cluster*) do Processo One-Op (Figura 4.5. Na atividade 8 do Processo de Agrupamento, os programas são re-padronizados em variáveis Java para que sejam exibidos os resultados da clusterização no console. Em seguida, os objetos *Instances* são enviados para classe `OneOp`. Na atividade 1 do Processo One-Op, os objetos *Instances* são recebidos e os nomes das classes são extraídos e organizados por *cluster*; também são criados diretórios para conter os arquivos de texto com resultados da aplicação da técnica de redução de custo.

- *Overall Process* (Figura 3.1): Este processo inclui o subprocesso 1 (*Compose reference group G*) do framework conceitual, o qual já foi abordado. Suas outras atividades, apresentadas a seguir, podem ser mapeadas para o terceiro processo da ferramenta, o Processo One-Op.

  - *2. Select cost reduction technique C*: Após compor o grupo de programas testados G que são similares ao novo programa não testado u, como próprio nome sugere, essa atividade seleciona uma técnica de redução de custo do teste de mutação como, por exemplo, Mutação Seletiva, Operadores Essenciais, ou One-Op. Neste trabalho, a técnica One-Op para redução de custo foi pré-definida.
  - *3. Apply cost reduction technique C to reference group G*: Essa atividade (que constitui um subprocesso não detalhado no *framework* conceitual) consiste na

aplicação da técnica de redução de custo C nos programas do grupo G com posterior armazenamento de resultados.

Esta atividade pode ser mapeada parcialmente nos processos One-Op da ferramenta, mais especificamente às atividades 2 (Gerar *Views (pré-Killing Matrices)* de cada *Cluster*), 4 (Selecionar *Killing Matrices* para módulos e *clusters*), e 5 (Calcular lista de operadores e escores).

- *4. Retrieve cost reduction resulting parameters for the reference group G*: Esta atividade consiste em recuperar resultados da aplicação da técnica de redução de custo C no grupo de referencia G. Esta atividade não pode ser mapeada para o Processo One-Op, pois o cálculo dos melhores operadores (candidatos a One-Op) são realizados em tempo de execução, não sendo recuperados de uma base de dados.

- *5. Apply cost reduction technique C to untested program u*: Esta atividade (que constitui um processo não especificado) tem por objetivo aplicar a técnica de redução de custo C no programa não testado u, utilizando a mesma configuração utilizada no grupo G.

  Esta atividade pode ser mapeada parcialmente na ferramenta, no Processo One-Op apresentado na Figura 4.5; em específico, pode ser mapeada nas atividades 3 (Gerar *Views(pré-Killing Matrices)* de cada módulo), 4 (Selecionar *Killing Matrices* para módulos e *clusters*), e 5 (Calcular lista de operadores e escores). Na atividade 3 são geradas duas *Views* na base de dados, uma para o módulo e outra para todos os outros programas restantes do mesmo *cluster*. As informações da *View* são as mesmas da atividade 2 (Gerar Views (pré-Killing Matrices) de cada cluster). A atividade 4 seleciona, de cada *View* criada para os módulos, o restante do *cluster* corrente, e demais *clusters*, as informações necessárias para formar uma *Killing Matrix* para cada *View*. A atividade 5 calcula com as informações da *Killing Matrix* o escore de mutação por operador. Este mapeamento é parcial porque o grupo G não é previamente obtido. Ele é gerado a partir de módulo e existe apenas durante a execução do experimento.

- *6. Assess results for untested program u*: Esta atividade (que constitui um subprocesso não detalhado no *framework* conceitual) consiste na avaliação dos resultados obtidos com a aplicação da técnica C no programa u. Esta atividade pode ser mapeada parcialmente à atividade 6 (Exportar resultados para arquivos) do Processo One-Op da ferramenta implementada. Nos arquivos criados

na atividade 1 (Criar lista de nomes dos programas por *cluster*) são colocadas todas as informações por módulos e seus respectivos operadores com seus respectivos escores de mutação (apenas o(s) operador(es) com maior escore(s) é (são) adicionado(s)). Esta atividade é parcialmente mapeada já que a análise dos valores dos escores foi realizada manualmente.

4.5 Considerações Finais

Neste capítulo, aspectos da automatização do *framework* conceitual foram descritos. Dentre os aspectos, incluem-se o diagrama de classes, a normalização realizada, detalhes da ferramenta Weka e Analizo. Também foi apresentado um mapeamento dos processos da ferramenta de similaridade em relação ao *framework*. O mapeamento apresentou a cobertura parcial da ferramenta em relação ao *framework*, a qual pode ser verificada na Seção 4.4.

No próximo capítulo, descreve-se a definição e a execução de estudo exploratório que coloca em prática tanto os conceitos explorados no *framework* conceitual quanto os mecanismos automatizados descritos neste capítulo.

Capítulo

5

Estudo Exploratório

5.1 Considerações Iniciais

No capítulo anterior descreveram-se as características e processos da ferramenta de similaridade desenvolvida com base no *framework* conceitual apresentado no Capítulo 3. Neste capítulo são abordadas as informações que permeiam o experimento realizado com auxílio da ferramenta desenvolvida.

Nas Seções 5.3, 5.4 e 5.5 descrevem-se, respectivamente, a técnica One-Op, a base de programas utilizada no experimento, e algumas observações iniciais sobre métricas extraídas dos programas considerados para o experimento realizado. O desenho do experimento é apresentado na Seção 5.6, e os resultados são analisados e discutidos nas seções subsequentes. Ressalta-se que conceitos sobre o teste de mutação, assim como similaridade de programas, foram apresentados no Capítulo 2.

5.2 Objetivo do Estudo

O objetivo geral do experimento descrito neste capítulo é *avaliar se informações sobre a similaridade de programas podem levar a estimativas mais precisas sobre a qualidade dos testes quando uma técnica de redução de custo de um critério de teste é aplicada.* Em

particular: a similaridade entre programas será calculada com base em métricas internas de código, considerando-se grupos de programas já testados e programas não testados; o critério de teste considerado é o teste de mutação (DeMillo et al., 1978; Hamlet, 1977); e a técnica de redução de custo é a técnica One-Op (Untch, 2009).

5.3 Técnica One-Op

As mudanças sintáticas que geram os mutantes são definidas por operadores de mutação. Portanto, a qualidade do operador impacta a qualidade do teste. A abordagem de redução de custo One-Op, originalmente empregada por Untch (2009), foi derivada de trabalhos que utilizavam um conjunto seleto de operadores, ou seja, a *mutação seletiva*. Recapitulando: a mutação seletiva surgiu como uma alternativa para solucionar o problema relacionado ao grande número de mutantes redundantes (isto é, um grupo de mutantes mortos pelo mesmo caso de teste), o que levou ao seu propósito de redução do número de mutantes através da criação deles por operadores eficientes.

A técnica One-op visa reduzir custo com diferentes propósitos, utilizando apenas o operador mais relevante. A técnica teoriza que com o uso de um operador único e poderoso, produzem testes que são quase tão efetivos quanto a de um conjunto completo. O escore de mutação é utilizado comumente para avaliar a qualidade dos casos de teste mas, no contexto de One-op, ele pode ser usado para cada operador com o propósito de avaliá-los. Segue abaixo alguns exemplos de trabalhos a respeito da técnica de redução de custo.

- Untch (2009): Foi o trabalho a motivar a técnica One-op. Resultados experimentais com o uso do operador SSDL (do inglês, *statement deletion*, e que consiste na remoção de instruções de programas, uma a uma) produziu resultados comparáveis a resultados produzidos com base em grupos maiores, ainda assim, seletivos, de operadores. O experimento relatado por Untch (2009) envolveu 7 programas implementados em linguagem C.

- Deng et al. (2013): Também investigou a relevância do operador SSDL (em sua abreviação do nome, SDL) para gerar bons casos de teste. Uma versão do operador para Java foi implementada na ferramenta MuJava e aplicada a 40 classes escritas em linguagem Java.

- Delamaro et al. (2014): trata-se de outro trabalho que avaliou o operador SDL aplicado a programas escritos em linguagem C, porém outros operadores de mutação também foram considerados, todos implementados na ferramenta Proteum. Um

experimento controlado foi realizado, envolvendo 39 programas. Foi utilizado o escore do teste de mutação para avaliar os operadores.

- Delamaro et al. (2014): Também exploraram sobre o operador SDL, porém concentraram em descobrir se remover elementos dos programas é uma forma custo-efetiva de gerar casos de teste. Para isso, os autores analisaram o operador SDL e propuseram novos operadores baseados em suas características. Três novos operadores, variações do SDL, foram concebidos.

- Derezinska (2016): Baseou-se em trabalhos anteriores que investigaram o operador SDL, e avaliou tal operador no contexto estrutural e orientado a objetos usados em mutantes de primeira e segunda ordem (com uma ou mais alterações simultâneas, por mutante, respectivamente) em programas C#.

5.4 Artefatos-Alvos do Experimento

Os projetos de programas Java utilizados no experimento foram coletadas de duas bases de dados previamente exploradas em trabalhos que aplicaram o critério de teste de mutação[1]. As bases originais foram inicialmente enviadas pelos autores originais (Deng et al., 2013; Oliveira et al., 2013) para outro aluno, membro do mesmo grupo de pesquisa do autor desta dissertação. Esse aluno desenvolveu *scripts* para importar esses dados para uma base de dados relacional, que pudesse ser utilizada na condução dos experimentos.

Ressalta-se que a base de programas fornecida por Oliveira et al. (2013) havia sido previamente gerada e utilizada por Polo et al. (2009). Oliveira et al. reutilizaram e estenderam a base de Polo et al., acrescentado novos programas e respectivos artefatos produzidos no teste de mutação.

Ressalta-se também que os dados enviados pelos autores originais encontravam-se distribuídos em diferentes tipos de arquivos como, por exemplo, arquivos de código-fonte, planilhas eletrônicas, arquivos textuais etc. Os *scripts* de importação se encarregaram de fazer a leitura desses dados e padronizá-los para serem inseridos na base relacional.

O esquema da base relacional criada é mostrado na Figura 5.1. É possível observar na figura a relação da tabela de projetos com a de programas, dos programas com os mutantes e, dos mutantes com casos de teste (por onde descobre-se se estão mortos) e operadores.

[1]Ambas as bases, com todas as informações disponíveis, foram compartilhadas pelos pesquisadores originais. De acordo com os relatos originais, para ambas as bases foram criados conjuntos de testes C-adequados para o teste de mutação. A despeito disso, ressalta-se que os trabalhos originais são relativamente antigos (datados de 2013), e os autores que compartilharam suas bases não puderam fazer uma checagem completa dos dados que compartilharam.

Ressalta-se que no contexto da base relacional em questão, um projeto se refere a uma coleção de programas que podem ser formados por uma ou mais classes cada. A despeito dessa característica de poder ser formado por múltiplas classes, cada um dos programas que compõem as bases de dados utilizadas no estudo descrito neste capítulo é formado por uma única classe (Deng et al., 2013; Oliveira et al., 2013; Polo et al., 2009).

5.4.1 Programas que Compõem as Bases de Dados

Tabela 5.1: Informações dos programas de Oliveira et al. (2013) e Polo et al. (2009).

	Num métodos	Atributos	Média de linhas de métodos	LOC (Analizo)	Métodos com mais linhas	Métodos com menos linhas
Bisec	2	3	9	24	16	2
BubCorrecto	7	2	3,5	35	8	2
Find	4	3	8	44	25	2
Fourballs	2	5	12,5	28	20	5
Mid	8	4	3	41	6	2
Triangulo	4	9	9	54	29	3
PluginTokenizer	16	4	3,5	103	10	2
Ciudad	22	4	8	262	38	2
IgnoreList	3	2	8	36	17	2

A base de programas de Polo et al. (2009) e Oliveira et al. (2013): Oliveira et al. (2013) abordam a dificuldade de encontrar um conjunto de teste adequado para encontrar os defeitos de um programa. Os autores propõem o uso de uma abordagem de teste de software baseado em busca (do inglês, *Search-Based Software Testing* – SBST) utilizando o critério de teste de mutação para auxiliar na seleção de mutantes e casos de teste. O SBST usa meta heurísticas para encontrar um conjunto de teste de baixo custo com alta efetividade.

Oliveira et al. (2013) exploram algoritmos genéticos co-evolucionários e os aplicam em cinco programas oriundos do trabalho de Polo et al. (2009), cujos programas e resultados do teste de mutação aplicado já estavam disponíveis. O resultado do CGA é comparado com outros cinco métodos.

O trabalho de Polo et al. (2009), por sua vez, abordou a redução de custos no teste de mutação através da geração de mutantes de segunda ordem (mutantes gerados a partir de dois defeitos inseridos). Os autores relataram dois experimentos que envolveram dois pequenos grupos de programas Java. O primeiro grupo consistia em programas implementados como classes Java isoladas (sendo eles, *Bisect*, *Bub* (*bubcorrecto*), *Fourballs*, *Mid*, *TriTyp* (triangulo) e *Find*. O segundo grupo consistia dos programas *Ciudad*, *IgnoreList* e *PluginTokenizer*). Na Tabela 5.1 apresentam-se informações gerais sobre os 9 projetos

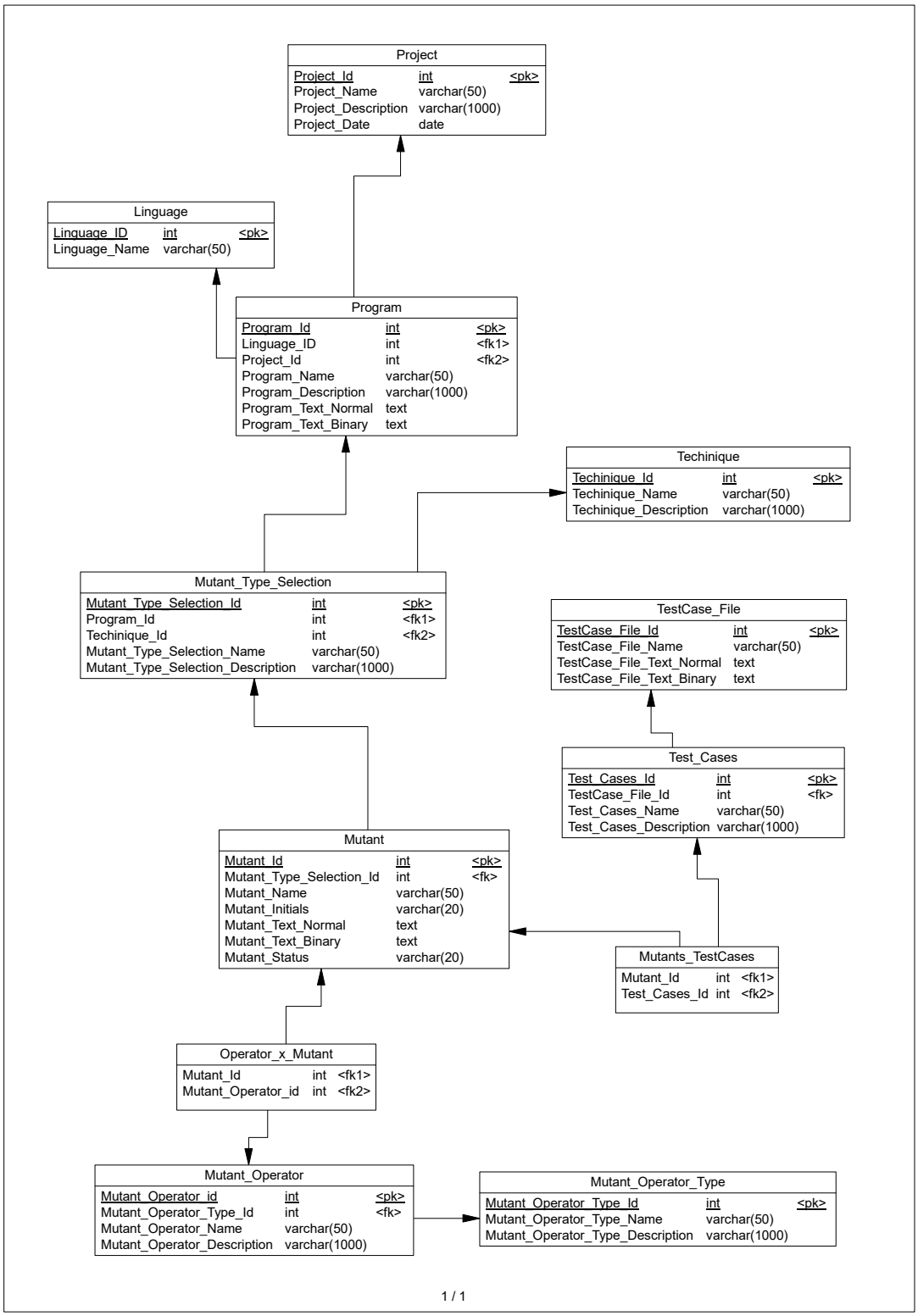

Figura 5.1: Modelagem da base de dados dos projetos testados com o critério de teste de mutação.

Tabela 5.2: Informações dos programas de Deng et al. (2013).

	Num métodos	Atributos	Média de linhas de métodos	LOC (Analizo)	Métodos com mais linhas	Métodos com menos linhas
BoundedQueue	5	5	6	78	8	2
countPositive	2	0	9	63	12	6
DigitReverser	2	0	8,5	28	14	3
Gaussian	7	0	4	36	8	2
Heap	8	0	4	44	7	2
numZero	2	0	9	39	12	6
oddOrPos	2	0	9	40	12	6
power	2	0	12	50	15	9
printPrimes	3	0	11	67	19	5
Queue	6	5	5	62	8	2
QuickSort	4	0	8	51	21	2
TestPat	1	0	17	35	17	17
trashAndTakeOut	2	0	9,5	25	11	8
cal	3	0	22	119	36	12
Calculation	9	0	6	76	23	2
checkIt	2	0	6,5	28	8	5
CheckPalindrome	2	0	8,5	30	9	8
findLast	3	0	10	66	14	5
findVal	3	0	11	61	15	6
InversePermutation	2	3	9	34	13	5
lastZero	2	0	8,5	39	12	5
LRS	3	0	7	34	12	4
MergeSort	3	0	9	42	13	5
RecursiveSelectionSort	3	0	6	29	11	2
Stack	7	4	4	29	8	2
stats	2	0	17,5	59	18	17
sum	2	0	8,5	35	12	5
Triangle	9	6	3,5	44	7	2
twoPred	2	0	11	39	12	10

Tabela 5.3: Quantidade de mutantes por operador dos programas de Oliveira et al. (2013) e Polo et al. (2009).

	AODS	AODU	AOIS	AOIU	AORB	AORS	ASRS	CDL	COD	COI	COR	LOI	ODL	ROR	SDL	VDL	Total
Bisec			44	10	16									4			74
BubCorrecto			34	9	12	2						13		10			80
Find	7		97	13	4	7						40		10			174
Fourballs			128	17	48							4		15			212
Mid			110	25							4	37		5			181
Triangulo			168	22	12						10	57		40			309
PluginTokenizer		1	48	10	8						2	15		10			94
Ciudad			108	16	4	14			5	4	6	30		16			203
IgnoreList			10	1		1						5		10			27
Total																	1354

testados e a Tabela 5.3 apresenta a quantidade de mutantes por operador e projeto. É importante salientar que, a respeito da Tabela 5.1 e da Tabela 5.2 (que ainda será descrita), as informações da coluna que representa o total de linhas (LOC) foram coletadas

Tabela 5.4: Quantidade de mutantes por operador dos programas de Deng et al. (2013).

	AODS	AODU	AOIS	AOIU	AORB	AORS	ASRS	CDL	COD	COI	COR	LOI	ODL	ROR	SDL	VDL	Total
BoundedQueue			82	6	32	3		3		8		24	19	29	33	13	253
countPositive			10	1		2				2		4		7	6	2	34
DigitReverser			36	6	24			6		2		9	16	10	9	7	125
Gaussian		1	124	23	80		4	4		5		1	44	33	21	22	360
Heap	1		72	12	40	4		10	1	8	2	29	24	43	29	13	299
numZero			10	1		2				2		4		12	6	2	39
oddOrPos			14	1	4	2		1		4	2	5	4	19	6	3	65
power			26	3	4	1				2		7	2	13	9	3	70
printPrimes			32	3	8	3		1		4		10	4	18	20	6	109
Queue			52	5	32	3		3		7		22	19	41	29	13	226
QuickSort			126	17	16	3		4		13	6	41	14	68	21	6	335
TestPat		1	38	6	12	2		2		7	2	15	11	28	15	7	146
trashAndTakeOut			46	9	24			4		3		14	12	21	15	7	155
cal			70	7	32	1		4		7	4	18	20	34	13	12	222
Calculation		1	116	17	4	2	16			16	6	34	13	85	32	3	346
checkIt										5	4		4		3	3	19
CheckPalindrome			16		4	2		1		2		4	2	7	7	2	47
findLast		1	14	1		1				2		4	1	12	6	1	43
findVal		1	18	2		1				2		6	1	12	6	1	50
InversePermutation			48	1		3		1		8	4	14	8	36	15	4	142
lastZero		1	10	1		1				2		4	1	12	6	1	39
LRS			52	7	8	3		2		5		16	4	29	16	5	148
MergeSort	8		28	6	20	12		4		7	2	26	20	7	17		157
RecursiveSelectionSort			50	9	8	1		2		3		15	4	19	12	2	126
Stack			34	4		3				7		10		31	30	3	122
stats			82	12	32	2	4	5		2		11	23	10	15	14	212
sum			14	2	4	1				1		5	2	5	4	3	41
Triangle			26	6	8			3	6	7	10	9	31		46	5	157
twoPred			16		4					4	2	4	4	14	7	3	58
Total																	4445

com a ferramenta Analizo, a qual conta todas linhas desde a declaração da classe até seu encerramento, incluindo linhas em branco e comentários). Enquanto isso, as outras informações das colunas das tabelas foram contadas manualmente (apenas as linhas com código executável foram consideradas).

A base de programas de Deng et al. (2013): Os autores propuseram uma abordagem de One-Op. Esta abordagem verifica se a aplicação de um único operador gera mutantes tão eficientes e eficazes quanto conjuntos de operadores previamente testados.

Os autores criaram um novo operador para Java através de uma simplificação na exclusão de declarações, originalmente implementada pelo operador (SDL) para programas Fortran. Esse operador foi implementado na ferramenta MuJava.

Em experimentos foram utilizadas 40 classes Java coletadas de livros e código livre compartilhado na internet. Foi realizada uma comparação do novo operador SDL com os

disponíveis na ferramenta. Desses 40 programas, com seus respectivos dados de teste do critério de mutação, 29 foram utilizados neste trabalho.

Na Tabela 5.2 apresentam-se informações gerais sobre os 29 programas em questão. Na Tabela 5.4 a quantidade de mutantes por operador é projeto.

As bases de programas utilizadas neste trabalho: Ao todo, 3 bases de programas (B_1, B_2, e B_3) foram organizadas. B_1 e B_2 possuem os 29 programas de Deng et al. (2013), e os 9 de Oliveira et al. (2013) e Polo et al. (2009). Todos os programas constam nos trabalhos citados mas, esses trabalhos possuem outros programas que não foram adicionados nas bases de dados deste experimento. A diferença entre as duas é que alguns casos de teste *excessivamente poderosos* foram desconsiderados em B_2. Considerou-se um caso de teste como sendo excessivamente *poderoso* quando o mesmo conseguia matar uma grande quantidade de mutantes; em alguns casos, até mesmo todos os mutantes. É importante salientar que não foi possível confirmar se os mutantes listados nas bases de dados fornecidas são todos os mutantes gerados.

Por fim, a base B_3 possui os 29 programas de Deng et al. (2013). Na próxima subseção são explicadas as funções dos operadores de mutação presentes nos testes dos programas escolhidos para este experimento.

5.4.2 Operadores de Mutação Aplicados nos Programas das Bases de Dados

Operadores estão implementados na ferramenta MuJava (Ma et al., 2005)

São apresentados nas Tabelas 5.3 e 5.4 os operadores de mutação presentes nos testes realizados nos programas adicionados nas bases utilizadas neste experimento. Segue uma breve descrição dos operadores divididos por seus tipos. As descrições foram extraídas de um relatório técnico (Ma e Offutt, 2005), e os operadores estão implementados na ferramenta MuJava (Ma et al., 2005).

- Operadores Aritméticos: São aplicados em operações de soma, subtração, multiplicação, divisão e modo.
  - AOR (*Arithmetic Operator Replacement*)
    - AORB: Substitui um operador aritmético binário por outro do mesmo tipo.
    - AORS: Substitui um operador aritmético de atalho por um unário.
  - AOI (*Arithmetic Operator Insertion*)

    - AOIU: Insere operadores aritméticos unários básicos.
    - AOIS: Insere operadores aritméticos de atalho.
  - AOD (*Arithmetic Operator Deletion*)
    - AODU: Remove operadores aritméticos unários básicos.
    - AODS: Remove operadores aritméticos de atalho.

- Operadores Relacionais: Realizam uma comparação entre dois elementos. Por exemplo maior, menor, igual, diferente, maior igual e menor igual. Como esse tipo de operador lida com dois operandos, apenas substituição é aplicável.
  - ROR (*Relational Operator Replacement*): Substitui operadores relacionais por outros, e altera o predicado todo com verdadeiro ou false.

- Operadores Condicionais: Em Java, podem ser divididos entre binários ('&&', '||') ou unário ('!').
  - COR (*Conditional Operator Replacement*): Substitui operador condicional binário por outro do mesmo tipo.
  - COI (*Conditional Operator Insertion*): Insere operador condicional unário.
  - COD (*Conditional Operator Deletion*): Remove operadores condicionais unários.

- Operadores Lógicos: Realizam funções bit a bit em seus operandos. Podem ser divididos em binários como ('&','|') e unário como ('∼').
  - LOI (*Logical Operator Insertion*): Insere um operador logico unário.

- Operadores de Atribuição: Atribui o valor de uma expressão a direita para uma a esquerda. Alguns exemplos são ('+=', '-=', '*=', '&=').
  - ASRS (*Short-Cut Assignment Operator Replacement*): Substitui operadores de atribuição de atalhos por outros do mesmo tipo.

- Operadores de Exclusão: Exclui declarações dos programas. Um mutante apenas pode ser morto caso ele percorra o ponto onde se encontra o defeito da declaração deletada e gere um erro.

- SDL (*Statement Deletion*): Remove cada declaração executável comentando-a. Quando aplicado a estruturas de controle como (*'while', 'if' e 'for'*) o bloco inteiro é comentado, assim como cada uma das declarações.
- VDL (*Variable Deletion*): As ocorrências de referencias de variáveis são removidas de todas as expressões. Quando necessário para preservar a compilação, operadores também podem ser removidos.
- CDL (*Constant Deletion*): As ocorrências de referências de constantes são removidas de todas as expressões. Quando necessário para preservar a compilação, operadores também podem ser removidos.
- ODL (*Operator Deletion*): Cada operador (aritmético, relacional, lógico entre outros) é removido de expressões e operadores de atribuição. Quando, removidos dos operadores de atribuição um atribuição plana é deixada ('var = 1'). Quando um operador binário é removido, um dos operandos também é removido (para preservar a compilação); isso cria dois mutantes, um para a remoção do operando à direita, e outro para o da esquerda.

Na próxima subseção são apresentados os resultados das métricas coletadas dos programas e divisões por clusterização.

5.5 Observações sobre Métricas CK e Clusterização

Nas Tabelas 5.5 e 5.6 são apresentados, respectivamente, os valores para as métricas coletadas dos 38 programas utilizados neste trabalho. Como é possível observar na Tabela 5.5 as métricas *CBO* e *NOC* apresentam valores 0 para todos os programas; o mesmo ocorre para *CBO, NOC* e *DIT* na Tabela 5.6. Isso significa que a similaridade foi baseada principalmente no comportamento das métricas *accm (WMC)*, *lcom4* e *rfc*. Portanto, os programas não possuíam características suficientes (eram programas muito simples) para uma observação completa da clusterização com base em métricas CK. A mesma tendencia pode ser observada nos operadores de mutação utilizados nos testes dos programas como apresentado anteriormente. Na Tabela 5.7 apresentam-se os valores para as métricas das classes da ferramenta de similaridade que foram agrupadas apenas para fins comparativos. Inicia-se a análise com uma discussão prévia sobre o trabalho de Cruz e Eler (2017), que também explorou a clusterização de programas com base em valores de métricas CK.

Observações sobre o trabalho de Cruz e Eler (2017): No trabalho, apresentam-se as classes de quatro programas Java que inicialmente foram agrupadas em *clusters* através de

suas métricas CK pelo algoritmo EM (Expectation Maximization)[2]. Em seguida, foram executados casos de teste para mostrar em que limites de valores a cobertura e o escore de mutação eram altos ou baixos, para fornecer evidências de alta ou baixa testabilidade. Depois de analisar todas as classes dos programas, foi encontrada uma correlação das métricas CK na cobertura estrutural obtida com os conjuntos de teste disponíveis, e também do escore de mutação produzido pelos testes. Em particular, as métricas CBO (Coupling Between Objects) e LCOM (Lack of Cohesion of Methods) têm uma fraca correlação com a cobertura de linhas(a porcentagem de declarações atingidas por um conjunto de teste como medida de adequação a um determinado critério) e escore; RFC (Response for Class) e WMC (Weighted Methods per Class) têm um negativa moderada correlação; DIT (Depth of Inheritance Tree) e NOC (Number of Children) foram inconclusivos.

Considerando todas as métricas juntas, o algoritmo de agrupamento utilizado no trabalho de Cruz e Eler (2017) dividiu todas as classes em dois *clusters*. Excluindo-se DIT e NOC, nove *clusters* foram gerados. Os *clusters* gerados pelo algoritmo EM foram validados utilizando o algoritmo KNN (*K Nearest Neighbors*)[3] e a correlação *rank-order* de *Spearman*[4].

Como resultado da classificação dos *clusters* realizada pelo algorítimo KNN foi obtido um valor de 81.53%, mostrando que é possível classificar com os algoritmos uma classe desconhecida através das métricas CBO, LCOM, RFC e WMC. A clusterização também foi aplicada pelas métricas individualmente.

Análise dos programas utilizados neste trabalho: Quatro análises foram realizadas com o objetivo de encontrar a melhor combinação de métricas (incluindo-se o uso conjunto de todas as métricas CK) para agrupar os programas deste experimento em *clusters*, sendo elas: (i) a clusterização de 20 programas de Deng et al. (2013) utilizados antes da coleta completa de programas; (ii) a clusterização dos 29 programas de Deng et al. (2013) (que compõem B_3); (iii) a clusterização dos 38 programas (que compõem B_1 e B_2) e; (iv) a

[2]O algoritmo EM gera uma sequência de soluções aproximadas que melhoram gradativamente. O algoritmo pode ser dividido em duas etapas. A primeira (*Expectation Step*) vincula cada objeto a um *cluster* (gerado por K-means) calculando sua probabilidade de vínculo. O segundo (*Maximization Step*) usa a probabilidade do passo anterior para re-estimar os parâmetros do modelo.

[3]O algoritmo KNN é um método para classificação ou regressão. Simplificadamente, define-se "K" elementos próximos para treinamento. Se usado para classificação, o elemento é classificado para o mesmo tipo da maioria de seus vizinhos. Caso o algoritmo seja usado para regressão, uma média dos valores dos vizinhos é atribuída ao elemento. No trabalho de Cruz e Eler (2017), o algoritmo é apresentado com uso para regressão.

[4]A correlação de *Spearman* analisa todos os valores por meio da intensidade da relação entre duas variáveis descrita por uma função monótona (quando preserva ou inverte a relação de ordem, referente à posição como anterior e posterior, de conjuntos ordenados)

Tabela 5.5: Valores das métricas das classes dos programas de Polo et al. (2009) (as métricas CK estão destacadas em negrito na primeira linha).

Projeto/Classe	acc	**accm**	amloc	anpm	**cbo**	**dit**	**lcom4**	loc	mmloc	noa	**noc**	nom	npa	npm	**rfc**	sc
Bisect	0	2	8	0.66	0	0	3	24	19	3	0	3	0	3	5	0
BubCorrecto	0	1.57	5	0.57	0	0	2	35	10	1	0	7	0	6	14	0
Find	0	3.25	11	0.75	0	0	1	44	32	3	0	4	0	4	11	0
Fourballs	0	2.5	14	2.5	0	0	1	28	22	5	0	2	0	2	11	0
Mid	0	1.44	4.55	1.44	0	0	4	41	8	4	0	9	0	5	18	0
Triangulo	0	3.6	10.8	0.6	0	1	2	54	39	9	0	5	3	5	16	0
PluginTokenizer	0	1.62	6.43	0.93	0	1	6	103	15	4	0	16	0	11	30	0
Ciudad	0	2.65	11.39	1	0	0	1	262	49	4	0	23	0	20	55	0
IgnoreList	0	2.66	12	0.33	0	0	1	36	24	2	0	3	0	3	7	0

Tabela 5.6: Valores das métricas das classes dos programas de Deng et al. (2013) (as métricas CK estão destacadas em negrito na primeira linha).

Projeto/Classe	acc	**accm**	amloc	anpm	**cbo**	**dit**	**lcom4**	loc	mmloc	noa	**noc**	nom	npa	npm	**rfc**	sc
BoundedQueue	0	2	10.5	0.33	0	0	1	63	17	5	0	6	0	6	26	0
DigitReverser	0	2	16	1	0	0	1	32	28	0	0	2	0	2	3	0
Gaussian	0	1.71	5.14	2	0	0	1	36	10	0	0	7	0	6	14	0
Heap	0	2	5.5	1.87	0	0	2	44	9	0	0	8	0	8	15	0
Calculation	0	2.66	8.44	1.55	0	0	1	76	30	0	0	9	0	9	22	0
CheckPalindrome	0	2.5	15	1	0	0	1	30	17	0	0	2	0	2	3	0
InversePermutation	0	3	17	1.5	0	0	1	34	24	3	0	2	1	2	6	0
LRS	0	2.66	11.33	1.33	0	0	1	34	21	0	0	3	0	3	5	0
MergeSort	0	3	14	1.66	0	0	1	42	18	0	0	3	0	3	5	0
RecursiveSelectionSort	0	2.33	9.66	1.66	0	0	1	29	20	0	0	3	0	3	5	0
Stack	0	1.71	4.14	0.57	0	0	1	29	9	4	0	7	3	7	20	0
Triangle	0	1.88	4.88	0.44	0	0	1	44	8	6	0	9	0	8	44	0
cal	0	4.33	39.66	2	0	0	1	119	59	0	0	3	0	2	5	0
checkIt	0	2.5	14	2	0	0	1	28	17	0	0	2	0	2	3	0
findLast	0	3	22	1	0	0	2	66	28	0	0	3	0	2	4	0
findVal	0	3	20.33	1	0	0	2	61	29	0	0	3	0	2	4	0
lastZero	0	3.5	19.5	1	0	0	2	39	25	0	0	2	0	2	2	0
stats	0	2.5	29.5	1	0	0	1	59	33	0	0	2	0	2	3	0
sum	0	3	17.5	1	0	0	2	35	25	0	0	2	0	2	2	0
twoPred	0	3.5	19.5	1.5	0	0	2	39	25	0	0	2	0	2	2	0
Queue	0	1.83	10.33	0.16	0	0	1	62	17	5	0	6	0	6	26	0
QuickSort	0	3	12.75	2	0	0	1	51	35	0	0	4	0	2	7	0
TestPat	0	3	17.5	1	0	0	2	35	32	0	0	2	0	2	2	0
countPositive	0	3.5	20	1	0	0	2	40	25	0	0	2	0	2	2	0
numZero	0	3.5	19.5	1	0	0	1	39	24	0	0	2	0	2	3	0
oddOrPos	0	3.5	20	1	0	0	2	40	25	0	0	2	0	2	2	0
power	0	4	25	1.5	0	0	2	50	30	0	0	2	0	2	2	0
printPrimes	0	3	22.33	1.33	0	0	1	67	38	0	0	3	0	3	5	0
trashAndTakeOut	0	2.5	12.5	1.5	0	0	1	25	14	0	0	2	0	2	3	0

clusterização das classes pertencentes a ferramenta de similaridade implementada neste trabalho.

No primeiro agrupamento (com base em 20 programas que constituem projetos simples, de classe única) as métricas que tratam da interação de diferentes classes e objetos

Tabela 5.7: Valores das métricas da ferramenta de similaridade (as métricas CK estão destacadas em negrito na primeira linha).

Classe	acc	**accm**	amloc	anpm	**cbo**	**dit**	**lcom4**	loc	mmloc	noa	**noc**	nom	npa	npm	**rfc**	sc
TestCase	1	1	3	0.5	0	0	2	12	3	2	0	4	0	4	8	0
MetricCK	5	1	3	0.5	0	1	3	18	3	3	0	6	0	6	12	0
ModuleCK	7	1.12	3.5	0.5	0	1	4	28	7	4	0	8	0	8	16	0
ProgramCK	6	1	3	0.5	0	1	3	18	3	3	0	6	0	6	12	0
Program	9	1	3	0.5	0	1	3	18	3	3	0	6	0	6	12	0
LineKillingMatrix	3	1	3	0.5	0	0	5	30	3	5	0	10	0	10	20	0
ProgramDAO	1	2.33	16	1	1	1	3	48	21	0	0	3	0	3	6	3
KillingMatrixDAO	4	4.71	29.85	1	1	0	1	209	76	7	0	7	4	7	31	1
ConnectionProgram	5	1.62	9	0.75	0	0	1	72	16	2	1	8	0	8	20	0
ConnectionOneOp	0	2	9	0	0	0	1	36	18	4	0	4	0	4	11	0
Menu	0	6.36	33.36	0.81	7	1	2	367	102	2	0	11	0	9	38	14
Analizo	1	7.26	40.4	0.8	6	0	2	768	215	7	0	19	0	15	87	12
Weka	1	15.33	93.66	2.16	7	0	1	562	233	2	0	6	2	6	58	7
OneOp	2	5.7	37.5	1.2	5	0	2	525	111	11	0	14	6	14	67	10
*InstanceProgramInterest	2	1	2.88	0.4	0	0	5	26	3	4	0	9	0	8	17	0
*MinMax	0	1	3.4	0.8	0	0	3	17	5	2	0	5	0	4	9	0
*KillingMatricesPerModu	1	1	3.4	0.8	0	0	5	24	6	3	0	7	0	7	12	0
*LineOpScore	1	1	3.4	0.8	0	0	3	17	5	2	0	5	0	4	8	0
*InstanceModuleInterest	2	1	2.92	0.46	0	0	7	38	3	6	0	13	0	12	25	0
*ModuleParam	1	1	2.8	0.4	0	0	3	14	3	2	0	5	0	4	9	0

retornaram valores nulos ou ruins. Das métricas CK, as únicas que apresentaram bons valores (uma quantidade acima de dois *clusters* com os programas bem distribuídos entre eles) foram ACCM (calcula complexidade ciclomática por método, de forma similar à WMC) e RFC (resposta por classe). Notou-se também que LCOM4 possui um valor razoável por verificar a presença de atributos da classe nos métodos; contudo, devido à simplicidade dos programas, LCOM4 gerou uma baixa quantidade de *clusters*.

Na geração de *clusters* para os 38 programas do experimento (referente à segunda análise aqui descrita), dois programas (`Plugin Tokenizer` e `Triangulo`) possuíam alguma profundidade na árvore de herança (valor representado pela métrica DIT), enquanto todos os outros não apresentavam valor. Isso alterou a influência dessa métrica na geração dos *clusters*. Sempre que DIT era combinada a outras métricas CK, o agrupamento gerava um *cluster* com dois programas apenas. Algo similar aconteceu com o agrupamento para as métricas da ferramenta de similaridade: sempre que a métrica NOC (número de filhos) era combinada, um *cluster* exclusivo para a classe `ConnectionProgram` era gerado; isso também ocorreu par a classe `Menu` quando CBO (acoplamento entre objetos) e DIT eram combinadas.

Apesar dos valores das métricas CBO e NOC serem zero para todos os 38 programas de B_1 e B_2 (graças à simplicidade dos programas nelas presentes), ambas possuem relevância no agrupamento dos 38 programas. Coincidentemente, a melhor combinação de *clusters*

foi obtida quando ambas foram consideradas. Contudo, quando aplicados apena nos 29 programas de Deng et al. (2013), essas métricas não possuem qualquer impacto e poderiam ser omitidas.

Em conclusão, a combinação de métricas definida para ser utilizada neste experimento é CBO, RFC, NOC e ACCM, pois geraram a melhor combinação de *clusters* para o conjunto completo de programas (três *clusters* contendo seis, treze e dezenove programas cada) e a segunda melhor para a base de Deng et al. (2013) (três *clusters* contendo onze, treze e cinco programas cada).

Como observação adicional, nota-se que a geração de *clusters* é bastante dependente das características dos programas utilizados. Quanto mais complexos os programas, maior o impacto de métricas como DIT, CBO e NOC. Contudo, elas sempre possuem valores menos abrangentes que ACCM, LCOM4 e RFC. Isso acontece porque os programas são desorganizados em relação à sua distribuição, possuindo classes mais complexas e maiores em relação a um grande número de classes simples e pequenas. Um estudo mais aprofundado da geração de *clusters* através da combinação de métricas não é o objetivo deste trabalho.

5.6 Desenho do Experimento

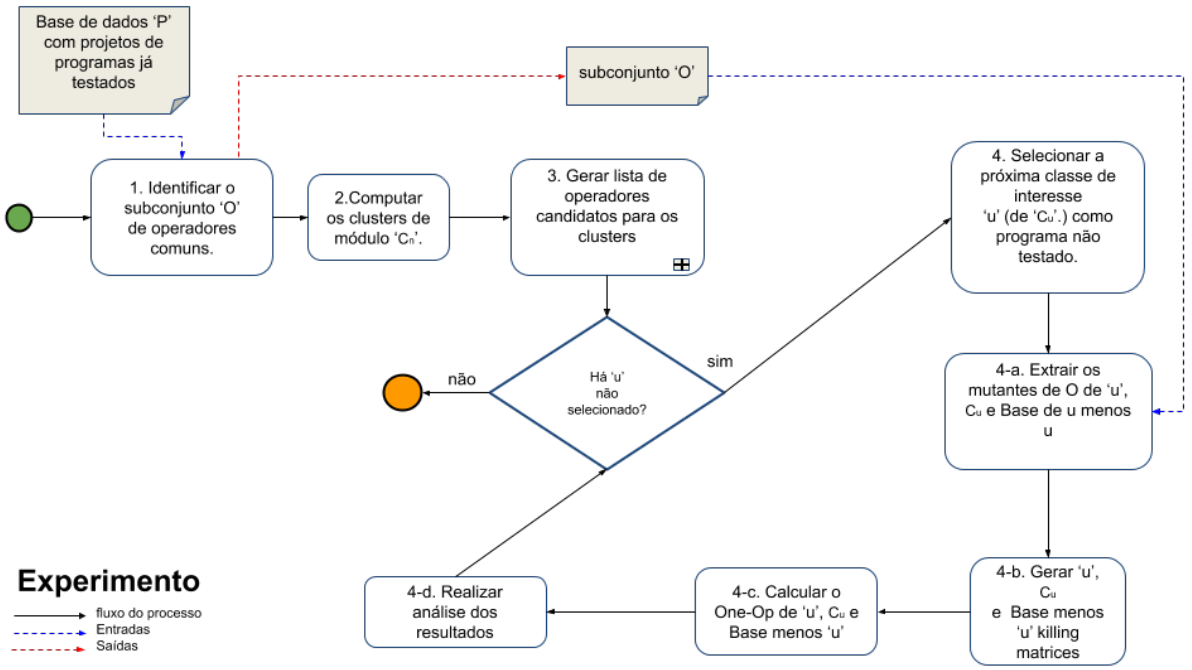

Figura 5.2: Diagrama do Experimento.

A ideia geral do experimento é avaliar se a similaridade de programas, calculada através da clusterização das métricas, auxilia na estimativa de escore de mutação, para um programa ainda não testado, dos operadores disponíveis e aplicados em programas já testados. Para alcançar este objetivo, o experimento agrupa programas por suas métricas (ou seja, criam-se *clusters*) e verifica se programas pertencentes ao mesmo *cluster* possuem os resultados dos cálculos de operadores candidatos a One-Op mais parecidos entre si do que os programas pertencentes a *clusters* diferentes. Esses resultados também são comparados com resultados obtidos para o conjunto completo de programas. O objetivo é averiguar o quão relevante é a similaridade através da clusterização de métricas como método.

Dado que se tem uma base de dados B com diversos programas já testados (isto é, com testes c-adequados para o critério de mutação), os passos a seguir foram realizados (a Figura 5.2 ilustra esse passo-a-passo):

1. Identificar o subconjunto O de operadores de mutação que são comuns a todos os programas de B.

2. Computar os clusters $(C_1, C_2, ...C_n)$ de módulos (isto é, classes de um programa Java) pertencentes aos programas de B.

3. Para todos os módulos de interesse u selecionados sequencialmente (no caso, u é uma classe presente em algum *cluster*, e u será considerada uma classe não testada para efeitos experimentais):

   (a) Para cada cluster gerado no passo anterior, selecionar os mutantes gerados pelos operadores de O. Os mutantes de u também são selecionados, porém colocados em um conjunto separado para posterior processamento. Também para a base B completa de programas, selecionar os mutantes gerados pelos operadores de O, exceto os mutantes gerados para u.

   (b) Gerar as *killing matrices*[5] considerando os mutantes selecionados no passo anterior. Neste passo, serão geradas $n+2$ matrizes, sendo n o número de clusters, e as duas matrizes adicionais referentes a B e u (nota-se que u não é considerado nos *clusters*, nem em B).

[5]Uma *killing matrix* é uma matriz bi-dimensional que mostra os mutantes mortos por cada caso de teste. Essa matriz permite o cálculo dos escores de mutação por operador. Tal escore é calculado selecionando-se os casos de teste que matam mutantes de um determinado operador, e checando-se a cobertura total de mutantes que esses testes produzem.

(c) Calcular o One-Op para cada cluster, para B, e para u. Neste passo, os escores de mutação (MS) para todos os operadores de O serão calculados.

(d) Realizar a análise dos resultados nos seguintes pontos de vista:

i. Checar se o One-Op de u é consistente com o One-Op calculado para o *cluster* ao qual u se associa.

ii. Avaliar se os resultados combinados de u e seu *cluster* são mais consistentes do que os resultados de u combinados com os outros *clusters*, ou com a base toda.

4. Fim para.

5.7 Modelo de Representação e Interpretação dos Resultados

As Tabelas 5.9, 5.10 e 5.11, incluídas no final deste capítulo, apresentam os operadores com o melhor Escore das três bases agrupadas do experimento divididos pelos seus clusters. Nas tabelas: "u" representa um programa, "C" representa um cluster, e "P" representa toda a base de programas. Ademais, "O" equivale à lista de operadores, "I" representa a contagem da intersecção de um grupo de operadores com o grupo de operadores de u. "C_u" são todos os melhores operadores do cluster de u sem considerar u. "P_u" segue a mesma lógica de C_u, mas em relação a toda a base de programas. "m" representa a função que obtém o maior valor dentre um conjunto de valores; serve para indicar que apenas o maior dos valores é considerado para a comparação. Os valores numéricos entre parênteses são os resultados das contagens (ou seja, são as cardinalidades dos conjuntos) das intersecções à sua esquerda. R1 e R2, definidos a seguir, podem ter um resultado positivo (V), negativo (X) ou neutro (N).

Exemplo de interpretação das informações apresentadas nas Tabelas 5.9, 5.10 e 5.11: Considerando-se a Tabela 5.9, na segunda coluna e terceira linha são descritas as informações coletadas para o programa `PluginTokenizer`. São cinco linhas que apresentam esses resultados:

- Ou: Lista os operadores com o maior escore de `PluginTokenizer`. São eles: AOIS, AOIU, AORB, COR, LOI, ROR
- OC_u: Lista os operadores com o maior escore dos programas do cluster de `PluginTokenizer`, sem considerar o programa em questão. São eles: AOIS, COI, LOI

- OP_u: Lista os operadores com o maior escore dos programas da base completa, sem considerar o programa `PluginTokenizer`. São eles: AOIS, LOI

- R1: Avalia a intersecção entre os melhores operadores para o programa `PluginTokenizer` e os melhores operadores para o restante de seu cluster (IC_u). Além disso, compara tal resultado com o mesmo cálculo de intersecção para cada um dos outros clusters (IC_n). O resultado é positivo ("V") caso IC_u possua valor igual ou maior ao melhor resultado de IC_n; negativo ("X") se menor; ou neutro ("N") caso possuam o mesmo valor.

- R2: Avalia a intersecção entre os melhores operadores para o programa `PluginTokenizer` e os melhores operadores para toda a base de dados menos o programa `PluginTokenizer` (IP_u). Além disso, compara tal resultado com IC_u. O resultado é positivo ("V") caso IC_u seja maior que IP_u, negativo ("X") se menor, ou neutro ("N") se iguais.

O objetivo de R1 é atender a etapa 3.e.iv conforme descrito na Seção 5.6. R1 verifica se o cluster que inclui u apresenta mais operadores com melhor escore em comum com u do que os outros clusters, com o objetivo de avaliar a clusterização de métricas CK como medida de similaridade praticável em relação aos conjuntos de operadores One-Op. R2, por sua vez, procura validar a qualidade da clusterização de métricas como medida de similaridade comparando-a com o resultado conseguido quando se considera o conjunto completo de programas da base. Em outras palavras, se a quantidade de operadores candidatos a One-Op em comum entre u e todos os outros programas da base for maior do que a quantidade quando se considera u e o cluster ao qual u se associa, não haveria razão para realizar a clusterização.

5.8 Resultados do Experimento

Nesta subseção serão apresentados resultados coletados pelo experimento. É importante considerar que apenas os operadores com o maior escore de mutação foram considerados, seguindo-se o conceito da técnica One-Op de identificar o melhor – ou os melhores, em caso de empate – operadores para ser aplicado nos programas em teste. Isso significa que se o maior score entre os operadores for 1, apenas os operadores com esse valor foram selecionados. Portanto, vários operadores de OC_u, OP_u e u foram desconsiderados, mesmo possuindo escores muito próximos dos escores produzidos pelos operadores One-OP.

5.8.1 Observações Iniciais

Em uma análise inicial dos resultados obtidos, verificou-se a proporção de operadores aceitos como candidatos a One-Op em relação ao total de operadores para u e C_u nas três bases (B_1, B_2, e B_3). Essa informação é importante já que as bases se comportam de formas distintas e isso impacta nas análises gerais de R1 e R2 apresentadas na próxima subseção.

A respeito da proporção entre o conjunto total de operadores e o conjunto de operadores candidatos a One-Op, nas base de dados B_1 e B_2, u possui a totalidade ou maioria de seus operadores formando o conjunto candidato a One-Op , enquanto C_u possui a minoria. Já B_3 possui a mesma distribuição de valores de B_1 e B_2 com relação a u; mas, diferentemente de B_1 e B_2, para B_3 a maioria dos operadores de C_u são candidatos a One-Op. Em outras palavras, a diferença entre as bases é que C_u possui a minoria dos operadores como candidatos a One-Op em B_1 e B_2, porém possui a maioria em B_3. Uma possível razão para este fato pode ter sido os programas diferentes pertencentes as duas bases que reduzem os valores dos operadores candidatos do grupo similar que é o resultado do calculo da média dos escores dos operadores dos programas envolvidos. Isso pode ter sido a causa do menor número de operadores coincidentes de C_u e u , em B_1 e B_2.

Levando em consideração que um grupo similar de programas foi identificado para cada programa das três bases de dados, um ponto importante para auxiliar nas conclusões da próxima subseção é a quantidade de operadores em comum entre os u's (cada programa) e seus respectivos C_u's (grupo similar). A seguir apresenta-se, em média, a quantidade de operadores candidatos a One-Op em comum entre todos os u's e seus respectivos C_u's para cada uma das três bases para todos os programas, seguida pela distribuição desses programas em relação às médias encontradas:

- B_1: C_u's possuem metade dos operadores em comum com seus respectivos u's, onde: 14 dos programas geram C_u com menos da metade dos operadores em comum com u, 9 programas geram C_u com mais da metade dos operadores em comum com u 12 programas geram C_u com metade dos operadores em comum com u e apenas 3 programas possuem em C_u todos os operadores de u.

- B_2: C_u's possuem metade dos operadores em comum com seus respectivos u's, onde: 14 dos programas geram C_u com menos da metade dos operadores em comum com u, 9 programas geram C_u com mais da metade dos operadores em comum com u, 11 programas geram C_u com metade dos operadores em comum com u, e apenas 4 programas possuem em C_u todos os operadores de u.

- B_3: C_u's possuem quase todos os operadores em comum com seus respectivos u's onde: 11 dos programas geram C_u com todos menos 1 dos operadores em comum com u, e 17 geram C_u com todos os operadores de u.

5.8.2 Resultados

Na Tabela 5.8 apresenta-se a somatória de R1 e R2, e uma contagem por *cluster*, considerando-se as três bases. As informações mostradas na tabela devem ser interpretadas da seguinte forma:

- R_1: Apresenta a quantidade de vezes, para cada base de dados, em que C_u possui mais operadores candidatos a One-Op em comum com seu respectivo u, quando comparado com a quantidade de operadores comuns a u e os demais clusters. Tomando-se como exemplo os valores da coluna 2, linha 2 da tabela, V(21) indica que C_u possui valor maior 21 vezes, N(3) indica que C_u possui igual valor 3 vezes, e X(14) indica que C_u possui valor menor 14 vezes.

- R_2: Apresenta a quantidade de vezes para cada base de dados em que C_u possui mais operadores candidatos a One-Op em comum com seu respectivo u, quando comparado com a quantidade de operadores comuns a u e todos os programas da base (exceto u). Tomando-se como exemplo os valores da coluna 2, linha 3 da tabela, V(27) indica que C_u possui valor maior 27 vezes, N(11) indica que C_u possui igual valor 11 vezes, e X(0) indica que C_u possui valor menor em nenhum caso.

- C_1, C_2 e C_3: Apresentam na primeira e segunda posição entre parênteses, respectivamente, os valores de R1 e R2 para o respectivo cluster. Tomando-se como exemplo os valores da coluna 2, linha 4 da tabela, V(1;0) demonstra que, no primeiro cluster de B_1, existe 1 caso em que C_u tem mais operadores em comum com u do que cada um dos outros clusters, e que em nenhum caso C_u possui mais operadores em comum com u do que todos os outros programas da base (exceto u). Analogamente, N apresenta a situação na qual C_u possui operadores em comum com u em igual número, e X apresenta a situação na qual C_u possui operadores em comum com u em menor número, sempre comparando-se com os demais cluster ou com a base completa (exceto u).

Os valores gerais de R1 são positivos nas três bases, levando-se em conta que mais da metade de resultados são positivos, sendo especialmente melhor em B_3. Os resultados de B_1 e B_2 para R2 possuem mais de 66% de resultados positivos, enquanto para B_3 o

Tabela 5.8: Resultados gerais do experimento

Clusters e Resultados Bases	B_1	B_2	B_3
$R1$	V(21), N(3) e X(14)	V(23), N(1) e X(14)	V(17), N(5) e X(7)
$R2$	V(27), N(11) e X(0)	V(28), N(10) e X(0)	V(5), N(19) e X(5)
C_1	V(1;0), N(1;6) e X(4;0)	V(2;0), N(0;6) e X(4;0)	V(9;2), N(1;9) e X(1;0)
C_2	V(10;10), N(0;3) e X(3;0)	V(10;10), N(0;3) e X(3;0)	V(8;3), N(0;5) e X(5;5)
C_3	V(10;17), N(2;2) e X(7;0)	V(11;18), N(1;1) e X(7;0)	V(0;0), N(4;5) e X(1;0)

resultado foi majoritariamente neutro. B_3, possuindo um número menor de programas, possivelmente não contem os programas de características internas discrepantes contidos em B_1 e B_2 o que gerou a grande quantidade de operadores em comum de C_u ou P_u com u em B_3. O resultado majoritariamente neutro vindo de R2 em B_3 na comparação entre o IC_u e IP_u pode ser explicado pelo mesmo argumento, no qual o grupo completo possui um grupo de operadores candidatos a One-Op tão bom quanto o especifico. O mesmo não ocorre em B_1 e B_2, já que a diversidade dos programas gera um valor de IP_u reduzido.

Ressalta-se que o número de programas presentes em cada cluster também pode ser extraído da Tabela 5.8. Para obter essa informação, somam-se os valores à direita (ou os valores à esquerda) que aparecem entre parênteses nas células das linhas 4 a 6 da tabela. Por exemplo, o cluster C_1 na base B_1 possui 6 (*i.e.* 1+1+4) programas, enquanto o cluster C_1 na base B_3 possui 11 (*i.e.* 9+1+1) programas. Dito isso, de acordo com os resultados mostrados na Tabela 5.8, todos os *clusters* com menor número de programas para cada base de programas considerada (sendo eles, C_1 em B_1 e B_2, e C_3 em B_3) possuem balanços negativos na comparação de IC_u e IC_n (os R1's dos clusters) sem nenhum outro resultado negativo nos *clusters* maiores; e resultados neutros quando IC_u é comparado a IP_u (os R2's do clusters) nos menores *clusters*. Possivelmente um *cluster* pequeno torne menos provável que os operadores certos estejam distribuídos entre os membros do *cluster* e estejam presentes nos *clusters* maiores. Também é possível que os menores *clusters* sejam compostos pelos programas mais distintos, o que acarretaria nos piores resultados de comparação para a similaridade. Uma possível premissa para este argumento está no comportamento dos menores *clusters* gerados, apresentados na seção 5.5 (Métricas CK e Clusterização). Naquela seção foi observado o número de programas em *clusters* gerados pela clusterização de grupos específicos de métricas para os programas `PluginTokenizer` e `Triangulo` e como as combinações desses valores geravam *clusters* pequenos, com dois ou três programas, devido às combinações únicas de seus valores de métricas.

Todos os outros resultados dos *clusters* e respectivas bases foram positivos, exceto no *cluster* C_2 pertencente a $B3$ (isto é, o *cluster* com maior número de programas desta base), o qual possui um número ligeiramente maior de resultados negativos do que positivos para R2 do *cluster*. Uma comparação entre o número de elementos do *cluster* de C_2 em B_3 sobre

R2 do cluster com os valores dos outros *clusters* desta mesma base e em relação a esta mesma comparação nos *clusters* das outras bases, apenas demonstra que B_3 é indiferente ao número de elementos no *cluster* cujos valores positivos e negativos são muito próximos, enquanto em B_1 e B_2, quanto maior o número de elementos no *cluster*, maior a quantidade de resultados positivos em relação aos negativos.

Levando em consideração os argumentos apresentados, é possível que, quanto mais programas diferentes forem adicionados (e consequentemente mais grupos de operadores envolvidos no calculo da média do grupo similar) menos operadores candidatos a One-Op em comum C_u possuirá com u (o que não significa que os outros operadores de u não estejam contidos e bem posicionados na lista completa de operadores de C_u), portanto melhor será em comparação a P_u como apresenta os resultados de R_2 em B_1 e B_2 em relação a B_3.

5.8.3 Lições aprendidas

Em suma, com base nas discussões sobre os resultados obtidos, conclui-se que:

- Apesar do resultado geral positivo da similaridade, os menores *clusters* apresentaram resultado negativo, ou seja, o grupo similar apresenta menos operadores candidatos em comum com o programa a ele similar que os outros *clusters*.

- B_3 possui valores melhores de similaridade que B_1 e B_2 quando se trata de R_1 mas, em R_2, diferente do resultado positivo de B_1 e B_2, B_3 obteve um resultado neutro. Isso significa que os operadores candidatos a One-Op do grupo similar têm, em média, o mesmo número de operadores em comum com u que os operadores candidatos de toda a base de dados menos u. Isso pode significar que a similaridade passa a ser uma comparação positiva (melhor do que a comparação em relação ao conjunto completo de programas menos u) quando novos programas com características internas diferentes são adicionados no conjunto de programas. Em outras palavras, o resultado obtido no experimento sugere que quanto maior a quantidade de programas diferentes, melhor será o valor do grupo similar quando se trata da comparação R_2.

- B_1 e B_2 apresentam, em média, metade dos operadores em comum entre C_u e u. Em B_3, essa comparação possui resultado próximo do máximo. Isso significa que, possivelmente, quanto maior o número de programas diferentes compondo C_u, menor será a intersecção de operadores candidatos de C_u e u (já que os valores dos escores

de C_u são o resultado da média dos valores de cada operador de cada programa contido em C_u).

Na próxima seção são apresentadas as ameaças à validade do experimento.

5.9 Ameaças à Validade

Estudos experimentais possuem ameaças que podem invalidá-los. A categorização das ameaças é baseada no padrão recomendado por Cook e Campbell (1979), classificando como: validade de conclusão, validade interna, validade de construção e validade externa. A seguir, apresentam-se as ameaças à validade pela categorização utilizada.

- *De Conclusão*: Está relacionada a obter conclusões corretas a partir dos resultados. A confiabilidade das medidas, o ambiente de operação e irrelevâncias aleatórias nesse ambiente (*e.g.* ruído em experiências práticas com pessoas). No experimento descrito neste capítulo, ocorreu a execução de uma ferramenta de similaridade euma posterior contagem manual (baixo número de informações) e exposição de resultados. A natureza exploratória do experimento reduz os riscos de ameaça nesta etapa. Um especialista da área também foi consultado durante a interpretação dos dados.

- *Interna*: Consiste em retratar se as mudanças observadas podem ser atribuídas ao tratamento (causa) e não a outros fatores. Na natureza da clusterização dois programas podem ser de clusters diferentes e mais parecidos entre si do que a outros membros de seu cluster. Pequenas diferenças de valor de métricas podem distorcer os clusters gerados. O algoritmo também utiliza pontos iniciais aleatórios garantindo uma distribuição final satisfatória, mas não necessariamente a ideal.

  As informações referentes às bases de dados do teste de mutação foram coletadas de diferentes formatos de arquivos. Scripts automáticos foram criados por membros do grupo de pesquisa para converter e cadastrar as informações nas bases.

  As métricas utilizadas para a clusterização foram escolhidas em relação à quantidade e distribuição dos programas nos clusters por elas gerados (através da execução de todas as possíveis combinações de métricas CK). Mesmo que a melhor combinação tenha sido escolhida manualmente, de acordo com critérios pré-estabelecidos pelo grupo de pesquisa, e que o algoritmo gere a melhor distribuição possível para as informações a ele concedidas, não há garantia que a distribuição seja a melhor possível ou a mais fidedigna às características internas dos programas.

As medidas tomadas em relação a essas ameaças foram: a consulta e recomendação do algoritmo, dentre outros, de clusterização por um pesquisador especializado na área; o acompanhamento de especialistas sobre o teste de mutação e experiências previamente realizadas com clusterização de métricas referenciadas neste trabalho; a qualidade das informações contidas nas bases de dados está em função das garantias oferecidas pelos autores dos estudos que os disponibilizaram e ao grupo de pequisa que verificou os resultados parciais e finais coletados pelos scripts.

- *De Construção*: Está relacionada ao planejamento (*design*) do experimento. Em relação a isso, a forma como a ferramente de coleta de métricas Analizo funciona não é compatível com o formato e propósito do algoritmo de clusterização no que se refere a classes que são internas a outras classes. Portanto, apenas classes sem essa característica foram escolhidas para compor as bases de dados.

- *Externa*: Está associada às dificuldades de projetar os resultados que foram obtidos para o contexto real. Nesse sentido, os cálculos dos operadores de mutação em outros projetos, seguindo um mesmo padrão e nomenclatura, em ferramentas diferentes, não geram um resultado necessariamente igual, o que poderia distorcer a interpretação do resultado final.

  A natureza dos programas utilizados neste experimento é simples. O comportamento do agrupamento das métricas de programas da indústria pode alterar a natureza e consequente interpretação de resultados. Programas testados com conjuntos diferentes de operadores podem não possuir nenhum operador ou muito poucos em comum, o que impossibilitaria uma comparação. Uma massa maior ou menor de programas pode alterar os resultados e sua consequente interpretação.

5.10 Considerações Finais

Neste capítulo foi apresentado o experimento e as informações a ele relacionadas, conclusões e ameaças a validade. O experimento consistiu em um estudo exploratório que colocou em prática os conceitos definidos no *framework* conceitual, assim como os mecanismos automatizados que implementam parcialmente o *framework*.

A similaridade é constatada na maior parte dos *clusters* gerados, com exclusão daqueles contendo menor número de programas. Também a similaridade aparenta melhorar em relação ao aumento do número de programas em B_1 e B_2 em relação a B_3. Isso pode ser graças as características internas distintas dos programas adicionados em B_1 e B_2

coletados do trabalho de (Oliveira et al., 2013). Em geral, os resultados positivos podem significar que o conjunto de operadores candidatos do grupo similar poderia ser utilizado em um novo programa na aplicação do teste de mutação na estratégia de redução de custo One-Op em futuros experimentos. Por fim, o próximo capitulo apresenta as conclusões, sumariza as contribuições, e discute limitações gerais e possibilidades de trabalhos futuros.

Tabela 5.9: Resultados do experimento em B_1 (detalhes para a interpretação dos dados mostrados nesta tabela estão na Seção 5.7).

C\P			
C1	1- PluginTokenizer	2- Ciudad	3- Queue
Ou	AOIS,AOIU,AORB,COR,LOI,ROR	AOIS,COD,COI,COR,LOI	AOIS,AOIU,COI,LOI,ROR
OC_u	AOIS,COI,LOI	AOIS,AOIU,LOI,ROR	AOIS, LOI
OP_u	AOIS,LOI	AOIS,ROR,LOI	AOIS,LOI
R1	X: $IC_u(2) < m(IC_2(3), IC_3(4))$	V: $IC_u(2) \geq m(IC_2(1), IC_3(2))$	X: $IC_u(2) < m(IC_2(3), IC_3(3))$
R2	N: $IC_u(2) = IP_u(2)$	N: $IC_u(2) = IP_u(2)$	N: $IC_u(2) = IP_u(2)$
	4- Triangle	5- Stack	6- Calculation
Ou	AOIS	AOIS,AOIU,AORS,COI,LOI,ROR	AOIS,AOIU,COI,LOI,ROR
OC_u	AOIS,LOI	AOIS,LOI	AOIS, LOI
OP_u	AOIS,LOI	AOIS,LOI	AOIS,LOI
R1	N: $IC_u(1) = m(IC_2(1), IC_3(1))$	X: $IC_u(2) < m(IC_2(4), IC_3(3))$	X: $IC_u(2) < m(IC_2(3), IC_3(3))$
R2	N: $IC_u(1) = IP_u(1)$	N: $IC_u(2) = IP_u(2)$	N: $IC_u(2) = IP_u(2)$
C2	7- printPrimes	8- Heap	9- QuickSort
Ou	AOIS,AORS,COI,ROR	AODS,AOIS,AORB,COI,ROR	AOIS,AOIU,AORB,AORS,COI,COR,LOI,ROR
OC_u	AOIS,AOIU,AORB,ROR	AOIS,AOIU,AORB,ROR	AOIS,AOIU,AORB,ROR
OP_u	AOIS,LOI	AOIS,LOI	AOIS,LOI
R1	V: $IC_u(2) \geq m(IC_1(1), IC_3(2))$	V: $IC_u(3) > m(IC_1(1), IC_3(2))$	V: $IC_u(4) > m(IC_1(2), IC_3(3))$
R2	V: $IC_u(2) > IP_u(1)$	V: $IC_u(3) > IP_u(1)$	V: $IC_u(4) < IP_u(2)$
	10- Bisect	11- DigitReverser	12- power
Ou	AOIS,AOIU,AORB,ROR	AOIS,AOIU,AORB,COI,LOI,ROR	AOIS,AOIU,AORS,COI,LOI,ROR
OC_u	AOIS,AOIU,AORB,AORS,COI,LOI,ROR,COR	AOIS,AOIU,AORB,ROR	AOIS,AOIU,AORB,ROR
OP_u	AOIS,LOI	AOIS,LOI	AOIS,LOI
R1	V: $IC_u(4) > m(IC_1(1), IC_3(2))$	V: $IC_u(4) > m(IC_1(2), IC_3(3))$	V: $IC_u(3) \geq m(IC_1(2), IC_3(3))$
R2	V: $IC_u(4) > IP_u(1)$	V: $IC_u(4) > IP_u(2)$	V: $IC_u(3) > IP_u(2)$
	13- TestPat	14- BoundedQueue	15- sum
Ou	AOIS,AOIU,AORB,COI,COR,LOI,ROR	AOIS,AOIU,COI,LOI,ROR	AOIS,AOIU,AORB,AORS,COI,LOI,ROR
OC_u	AOIS,AOIU,AORB,ROR	AOIS,AOIU,AORB,ROR	AOIS,AOIU,AORB,ROR
OP_u	AOIS,LOI	AOIS,LOI	AOIS,LOI
R1	V: $IC_u(4) > m(IC_1(2), IC_3(3))$	V: $IC_u(3) \geq m(IC_1(2), IC_3(3))$	V: $IC_u(4) > m(IC_1(2), IC_3(3))$
R2	V: $IC_u(4) > IP_u(2)$	V: $IC_u(3) > IP_u(2)$	V: $IC_u(4) > IP_u(2)$
	16- InversePermutation	17- findVal	18- findLast
Ou	AOIS,COI,LOI,ROR	AOIS,AORS,COI,LOI,ROR	AOIS,AORS,COI,LOI,ROR
OC_u	AOIS,AOIU,AORB,ROR	AOIS,AOIU,AORB,ROR	AOIS,AOIU,AORB,ROR
OP_u	AOIS,LOI	AOIS,LOI	AOIS,LOI
R1	X: $IC_u(2) \leq m(IC_1(2), IC_3(3))$	X: $IC_u(2) \leq m(IC_1(2), IC_3(3))$	X: $IC_u(2) \leq m(IC_1(2), IC_3(3))$

84

Tabela 5.9

C\P				
R2	N: $IC_u(2) = IP_u(2)$	N: $IC_u(2) = IP_u(2)$	N: $IC_u(2) = IP_u(2)$	
	19- MergeSort			
Ou	AODS,AOIS,AOIU,AORB,AORS,COI,COR,LOI,ROR			
OC_u	AOIS,AOIU,AORB,ROR			
OP_u	AOIS,LOI			
R1	V: $IC_u(4) > (IC_1(2), IC_3(3))$			
R2	V: $IC_u(4) > IP_u(2)$			
C3	20- oddOrPos	21- Mid	22- numZero	
Ou	AOIS,AOIU,AORS,COI,COR,LOI,ROR	AOIS,AOIU,COR,LOI	AOIS,AOIU,AORS,COI,LOI,ROR	
OC_u	AOIS,LOI,ROR	AOIS,LOI,ROR	AOIS,LOI,ROR	
OP_u	AOIS,LOI	AOIS, LOI	AOIS,LOI	
R1	V: $IC_u(4) > m(IC_1(2), IC_2(3))$	N: $IC_u(2) = m(IC_1(2), IC_2(2))$	V: $IC_u(3) \geq m(IC_1(2), IC_2(3))$	
R2	V: $IC_u(3) > IP_u(2)$	N: $IC_u(2) = IP_u(2)$	V: $IC_u(3) > IP_u(2)$	
	23- BubCorrecto	24- Gaussian	25- trashAndTakeOut	
Ou	AOIS,AOIU,AORB,AORS,LOI,ROR	AOIS,AOIU,AORB,COI,ROR	AOIS,AORB,COI,LOI,ROR	
OC_u	AOIS,LOI,ROR	AOIS,LOI,ROR	AOIS,LOI,ROR	
OP_u	AOIS,LOI	AOIS,LOI	AOIS,LOI	
R1	X: $IC_u(3) < m(IC_1(2), IC_2(4))$	X: $IC_u(2) < m(IC_1(1), IC_2(4))$	V: $IC_u(3) \geq m(IC_1(2), IC_2(3))$	
R2	V: $IC_u(3) > IP_u(2)$	V: $IC_u(2) > IP_u(1)$	V: $IC_u(3) > IP_u(2)$	
	26- IgnoreList	27- countPositive	28- Fourballs	
Ou	AOIS,LOI,ROR	AOIS,AOIU,AORS,COI,LOI,ROR	AOIS,AOIU,AORB,ROR	
OC_u	AOIS,AOIU,LOI,ROR	AOIS,LOI,ROR	AOIS,LOI,ROR	
OP_u	AOIS,LOI	AOIS,LOI	AOIS,LOI	
R1	V: $IC_u(3) > m(IC_1(2), IC_2(2))$	V: $IC_u(3) \geq m(IC_1(2), IC_2(3))$	X: $IC_u(2) < m(IC_1(1), IC_2(4))$	
R2	V: $IC_u(3) > IP_u(2)$	V: $IC_u(3) > IP_u(2)$	V: $IC_u(2) > IP_u(1)$	
	29- Triangulo	30- twoPred	31- RecursiveSelectionSort	
Ou	ROR	COI,COR,ROR	AOIS,AORB,COI,ROR	
OC_u	AOIS,LOI,ROR	AOIS,ROR	AOIS,LOI,ROR	
OP_u	AOIS,LOI	AOIS,LOI	AOIS,LOI	
R1	V: $IC_u(1) \geq m(IC_1(0), IC_2(1))$	V: $IC_u(1) \geq m(IC_1(0), IC_2(1))$	X: $IC_u(2) < m(IC_1(1), IC_2(3))$	
R2	V: $IC_u(1) > IP_u(0)$	V: $IC_u(1) > IP_u(0)$	V: $IC_u(2) > IP_u(1)$	
	32- LRS	33- checkIt	34- cal	
Ou	AOIS,AORB,COI,LOI,ROR	COI,COR	AOIS,AOIU,AORB,COI,LOI,ROR	
OC_u	AOIS,LOI,ROR	AOIS,LOI,ROR	AOIS,LOI,ROR	
OP_u	AOIS,LOI	AOIS,LOI	AOIS,LOI	
R1	V: $IC_u(3) \geq m(IC_1(2), IC_2(3))$	N: $IC_u(0) = m(IC_1(0), IC_2(0))$	X: $IC_u(3) < m(IC_1(2), IC_2(4))$	

Tabela 5.9

C\P			
R2	V: $IC_u(3) > IP_u(2)$	N: $IC_u(0) = IP_u(0)$	V: $IC_u(3) > IP_u(2)$
	35- stats	36- Find	37- lastZero
Ou	AOIS,AOIU,AORB,AORS,COI,LOI,ROR	AODS,AOIS,AOIU,AORB,AORS,LOI,ROR	AOIS,AORS,COI,LOI,ROR
OC_u	AOIS,LOI,ROR	AOIS,LOI,ROR	AOIS,LOI,ROR
OP_u	AOIS,LOI	AOIS,LOI	AOIS,LOI
R1	X: $IC_u(3) < m(IC_1(2), IC_2(4))$	X: $IC_u(3) < m(IC_1(2), IC_2(4))$	V: $IC_u(3) > m(IC_1(2), IC_2(2))$
R2	V: $IC_u(3) > IP_u(2)$	V: $IC_u(3) > IP_u(2)$	V: $IC_u(3) > IP_u(2)$
	38- CheckPalindrome		
Ou	AOIS,AORB,COI,LOI,ROR		
OC_u	AOIS,LOI,ROR		
OP_u	AOIS,LOI		
R1	V: $IC_u(3) \geq (IC_1(2), IC_2(3))$		
R2	V: $IC_u(3) > IP_u(2)$		

Tabela 5.10: Resultados do experimento em B_2 (detalhes para a interpretação dos dados mostrados nesta tabela estão na Seção 5.7).

C\P			
C1	1- PluginTokenizer	2- Ciudad	3- Queue
Ou	AOIS,AOIU,AORB,LOI,ROR	AOIS,COD,COI,COR,LOI	AOIS,AOIU,COI,LOI,ROR
OC_U	AOIS,COI,LOI	AOIS,AOIU,LOI,ROR	AOIS,LOI
OP_u	AOIS,LOI	AOIS,ROR,LOI	AOIS,LOI
R1	X: $IC_u(2) < m(IC_2(4), IC_3(4))$	V: $IC_u(2) \geq m(IC_2(1), IC_3(2))$	X: $IC_u(2) < m(IC_2(3), IC_3(3))$
R2	N: $IC_u(2) = IP_u(2)$	N: $IC_u(2) = IP_u(2)$	N: $IC_u(2) = IP_u(2)$
	4- Triangle	5- Stack	6- Calculation
Ou	AOIS	AOIS,AOIU,AORS,COI,LOI,ROR	AOIS,AOIU,COI,LOI,ROR
OC_u	AOIS,LOI	AOIS,LOI	AOIS,LOI
OP_u	AOIS,LOI	AOIS,LOI	AOIS,LOI
R1	V: $IC_u(1) < m(IC_2(1), IC_3(1))$	X: $IC_u(2) < m(IC_2(3), IC_3(3))$	X: $IC_u(2) < m(IC_2(3), IC_3(3))$
R2	N: $IC_u(1) = IP_u(1)$	N: $IC_u(2) = IP_u(2)$	N: $IC_u(2) = IP_u(2)$
C2	7- printPrimes	8- Heap	9- QuickSort
Ou	AOIS,AORS,COI,ROR	AODS,AOIS,AORB,COI,ROR	AOIS,AOIU,AORB,AORS,COI,COR,LOI,ROR
OC_u	AOIS,AOIU,AORB,ROR	AOIS,AOIU,AORB,ROR	AOIS,AOIU,AORB,ROR
OP_u	AOIS,LOI	AOIS,LOI	AOIS,LOI
R1	V: $IC_u(2) >= m(IC_1(1), IC_3(2))$	V: $IC_u(3) > m(IC_1(1), IC_3(2))$	V: $IC_u(4) > m(IC_1(2), IC_3(3))$
R2	V: $IC_u(2) > IP_u(1)$	V: $IC_u(3) > IP_u(1)$	V: $IC_u(4) > IP_u(2)$
	10- Bisect	11- DigitReverser	12- power
Ou	AOIS,AOIU,AORB,ROR	AOIS,AOIU,AORB,COI,LOI,ROR	AOIS,AOIU,AORS,COI,LOI,ROR
OC_u	AOIS,AOIU,AORB,AORS,COI,LOI,ROR,COR	AOIS,AOIU,AORB,ROR	AOIS,AOIU,AORB,ROR
OP_u	AOIS,LOI	AOIS,LOI	AOIS,LOI
R1	V: $IC_u(4) > m(IC_1(1), IC_3(2))$	V: $IC_u(4) > m(IC_1(2), IC_3(3))$	V: $IC_u(3) \geq m(IC_1(2), IC_3(3))$
R2	V: $IC_u(4) > IP_u(1)$	V: $IC_u(4) > IP_u(2)$	V: $IC_u(3) > IP_u(2)$
	13- TestPat	14- BoundedQueue	15- sum
Ou	AOIS,AOIU,AORB,COI,COR,LOI,ROR	AOIS,AOIU,COI,LOI,ROR	AOIS,AOIU,AORB,AORS,COI,LOI,ROR
OC_u	AOIS,AOIU,AORB,ROR	AOIS,AOIU,AORB,ROR	AOIS,AOIU,AORB,ROR
OP_u	AOIS,LOI	AOIS,LOI	AOIS,LOI
R1	V: $IC_u(4) > m(IC_1(2), IC_3(3))$	V: $IC_u(3) \geq m(IC_1(2), IC_3(3))$	V: $IC_u(4) > m(IC_1(1), IC_3(2))$
R2	V: $IC_u(4) > IP_u(2)$	V: $IC_u(3) > IP_u(2)$	V: $IC_u(4) > IP_u(2)$
	16- InversePermutation	17- findVal	18- findLast
Ou	AOIS,COI,LOI,ROR	AOIS,AORS,COI,LOI,ROR	AOIS,AORS,COI,LOI,ROR
OC_u	AOIS,AOIU,AORB,ROR	AOIS,AOIU,AORB,ROR	AOIS,AOIU,AORB,ROR
OP_u	AOIS,LOI	AOIS,LOI	AOIS,LOI
R1	X: $IC_u(2) < m(IC_1(2), IC_3(3))$	X: $IC_u(2) < m(IC_1(2), IC_3(3))$	X: $IC_u(2) < m(IC_1(2), IC_3(3))$

87

Tabela 5.10

C\P			
R2	N: $IC_u(2) = IP_u(2)$	N: $IC_u(2) = IP_u(2)$	N: $IC_u(2) = IP_u(2)$
	19- MergeSort		
Ou	AODS,AOIS,AOIU,AORB,AORS,COI,COR,LOI,ROR		
OC_u	AOIS,AOIU,AORB,ROR		
OP_u	AOIS,LOI		
R1	V: $IC_u(4) > m(IC_1(2), IC_3(3))$		
R2	V: $IC_u(4) > IP_u(2)$		
C3	20- oddOrPos	21- Mid	22- numZero
Ou	AOIS,AOIU,AORS,COI,COR,LOI,ROR	AOIS,AOIU,COR,LOI	AOIS,AOIU,AORS,COI,LOI,ROR
OC_u	AOIS,LOI,ROR	AOIS,LOI,ROR	AOIS,LOI,ROR
OP_u	AOIS,LOI	AOIS	AOIS,LOI
R1	V: $IC_u(3) \geq m(IC_1(2), IC_2(3))$	V: $IC_u(2) \geq m(IC_1(1), IC_2(2))$	V: $IC_uP(3) \geq m(IC_1(2), IC_2(3))$
R2	V: $IC_u(3) > IP_u(2)$	V: $IC_u(2) > IP_u(1)$	V: $IC_u(3) > IP_u(2)$
	23- BubCorrecto	24- Gaussian	25- trashAndTakeOut
Ou	AOIS,AOIU,AORB,AORS,LOI,ROR	AOIS,AOIU,AORB,COI,ROR	AOIS,AORB,COI,LOI,ROR
OC_u	AOIS,LOI,ROR	AOIS,LOI,ROR	AOIS,LOI,ROR
OP_u	AOIS,LOI	AOIS,LOI	AOIS,LOI
R1	X: $IC_u(3) < m(IC_1(2), IC_2(4))$	X: $IC_u(2) < m(IC_1(1), IC_2(4))$	V: $IC_u(3) \geq m(IC_1(2), IC_2(3))$
R2	V: $IC_u(3) > IP_u(2)$	V: $IC_u(2) > IP_u(1)$	V: $IC_u(3) > IP_u(2)$
	26- IgnoreList	27- countPositive	28- Fourballs
Ou	AOIS,LOI,ROR	AOIS,AOIU,AORS,COI,LOI,ROR	AOIS,AOIU,AORB,ROR
OC_u	AOIS,AOIU,LOI,ROR	AOIS,LOI,ROR	AOIS,LOI,ROR
OP_u	AOIS,LOI	AOIS,LOI	AOIS,LOI
R1	V: $IC_u(3) > m(IC_1(2), IC_2(2))$	V: $IC_u(3) \geq m(IC_1(2), IC_2(3))$	X: $IC_u(2) < m(IC_1(1), IC_2(4))$
R2	V: $IC_u(3) > IP_u(2)$	V: $IC_u(3) > IP_u(2)$	V: $IC_u(2) > IP_u(1)$
	29- Triangulo	30- twoPred	31- RecursiveSelectionSort
Ou	ROR	COI,COR,ROR	AOIS,AORB,COI,ROR
OC_u	AOIS,LOI,ROR	AOIS,ROR	AOIS,LOI,ROR
OP_u	AOIS,LOI	AOIS,LOI	AOIS,LOI
R1	V: $IC_u(1) \geq m(IC_1(0), IC_2(1))$	V: $IC_u(1) \geq m(IC_1(0), IC_2(1))$	X: $IC_u(2) < m(IC_1(1), IC_2(3))$
R2	V: $IC_u(1) > IP_u(0)$	V: $IC_u(1) > IP_u(0)$	V: $IC_u(2) > IP_u(1)$
	32- LRS	33- checkIt	34- cal
Ou	AOIS,AORB,COI,LOI,ROR	COI,COR	AOIS,AOIU,AORB,COI,LOI,ROR
OC_u	AOIS,LOI,ROR	AOIS,LOI,ROR	AOIS,LOI,ROR
OP_u	AOIS,LOI	AOIS,LOI	AOIS,LOI
R1	V: $IC_u(3) \geq m(IC_1(2), IC_2(3))$	N: $IC_u(0) \geq m(IC_1(0), IC_2(0))$	X: $IC_u(3) < m(IC_1(2), IC_2(4))$

Tabela 5.10

C\P			
R2	V: $IC_u(3) > IP_u(2)$	N: $IC_u(0) = IP_u(0)$	V: $IC_u(3) > IP_u(2)$
	35- stats	36- Find	37- lastZero
Ou	AOIS,AOIU,AORB,AORS,COI,LOI,ROR	AODS,AOIS,AOIU,AORB,AORS,LOI,ROR	AOIS,AORS,COI,LOI,ROR
OC_u	AOIS,LOI,ROR	AOIS,LOI,ROR	AOIS,LOI,ROR
OP_u	AOIS,LOI	AOIS,LOI	AOIS,LOI
R1	X: $IC_u(3) < m(IC_2(2), IC_3(4))$	X: $IC_u(3) < m(IC_2(2), IC_3(4))$	V: $IC_u(3) > m(IC_2(2), IC_3(2))$
R2	V: $IC_u(3) > IP_u(2)$	V: $IC_u(3) > IP_u(2)$	V: $IC_u(3) > IP_u(2)$
	38- CheckPalindrome		
Ou	AOIS,AORB,COI,LOI,ROR		
OC_u	AOIS,LOI,ROR		
OP_u	AOIS,LOI		
R1	V: $IC_u(3) \geq m(IC_2(2), IC_3(3))$		
R2	V: $IC_u(3) > IP_u(2)$		

Tabela 5.11: Resultados do experimento em B_3 (detalhes para a interpretação dos dados mostrados nesta tabela estão na Seção 5.7).

C\P		
C1	1- printPrimes	2- Heap
Ou	AOIS,AORS,COI,ROR,SDL,VDL	AODS,AOIS,AORB,COI,ODL,ROR,SDL,VDL
OC_u	AOIS,AOIU,AORB,CDL,COI,LOI,ODL,ROR,SDL,VDL,AORS,AODS,COR	AOIS,AOIU,AORB,CDL,COI,LOI,ODL,ROR,SDL,VDL,AORS,COR
OP_u	AOIS,AOIU,AORB,AORS,COI,LOI,ODL,ROR,SDL,VDL,COR	AOIS,AOIU,AORB,AORS,COI,LOI,ODL,ROR,SDL,VDL,COR
R1	V: $IC_u(6) > m(IC_2(5), IC_3(5))$	V: $IC_u(7) \geq m(IC_2(7), IC_3(6))$
R2	N: $IP_u(6) = IC_u(6)$	N: $IP_u(7) = IC_u(7)$
	3- QuickSort	4- DigitReverser
Ou	AOIS,AOIU,AORB,AORS,CDL,COI,COR,LOI,ODL,ROR,SDL,VDL	AOIS,AOIU,AORB,CDL,COI,LOI,ODL,ROR,SDL,VDL
OC_u	AOIS,AOIU,AORB,COI,LOI,ODL,ROR,SDL,VDL,AORS,AODS,COR	AOIS,AOIU,AORS,COI,LOI,ODL,ROR,SDL,VDL,AODS,AORB,CDL,COR
OP_u	AOIS,AOIU,AORB,AORS,COI,LOI,ODL,ROR,SDL,VDL,COR	AOIS,AOIU,AORB,AORS,COI,LOI,ODL,ROR,SDL,VDL,COR
R1	V: $IC_u(11) \geq m(IC_2(11), IC_3(8))$	V: $IC_u(10) \geq m(IC_2(10), IC_3(8))$
R2	N: $IC_u(11) = IP_u(11)$	V: $IC_u(10) > IP_u(9)$
	5- power	6- TestPat
Ou	AOIS,AOIU,AORS,COI,LOI,ROR,SDL,VDL	AOIS,AOIU,AORB,COI,COR,LOI,ODL,ROR,SDL,VDL
OC_u	AOIS,AOIU,AORB,CDL,COI,LOI,ODL,ROR,SDL,VDL,AORS,AODS,COR	AOIS,AORB,CDL,COI,LOI,ODL,ROR,SDL,VDL,AODS,COR
OP_u	AOIS,AOIU,AORB,AORS,COI,LOI,ODL,ROR,SDL,VDL,COR	AOIS,AOIU,AORB,AORS,COI,LOI,ODL,ROR,SDL,VDL
R1	V: $IC_u(8) > m(IC_2(7), IC_3(7))$	X: $IC_u(9) < m(IC_2(10), IC_3(8))$
R2	N: $IC_u(8) = IP_u(8)$	N: $IC_u(9) = IP_u(9)$
	7- sum	8- InversePermutation
Ou	AOIS,AOIU,AORB,AORS,COI,LOI,ODL,ROR,SDL,VDL	AOIS,COI,LOI,ODL,ROR,SDL
OC_u	AOIS,AOIU,AORB,CDL,COI,LOI,ODL,ROR,SDL,VDL,AORS,AODS,COR	AOIS,AOIU,AORB,CDL,COI,LOI,ODL,ROR,SDL,VDL,AORS,AODS,COR
OP_u	AOIS,AOIU,AORB,AORS,COI,LOI,ODL,ROR,SDL,VDL,COR	AOIS,AOIU,AORB,AORS,COI,LOI,ODL,ROR,SDL,VDL,COR
R1	V: $IC_u(10) > m(IC_2(9), IC_3(8))$	N: $IC_u(6) = m(IC_2(6), IC_3(6))$
R2	N: $IC_u(10) = IP_u(10)$	N: $IC_u(6) = IP_u(6)$
	9- findVal	10- findLast
Ou	AOIS,AORS,COI,LOI,ROR,SDL,VDL	AOIS,AORS,COI,LOI,ROR,SDL,VDL
OC_u	AOIS,AOIU,AORB,CDL,COI,LOI,ODL,ROR,SDL,VDL,AORS,AODS,COR	AOIS,AOIU,AORB,CDL,COI,LOI,ODL,ROR,SDL,VDL,AORS,AODS,COR
OP_u	AOIS,AOIU,AORB,AORS,COI,LOI,ODL,ROR,SDL,VDL,COR	AOIS,AOIU,AORB,AORS,COI,LOI,ODL,ROR,SDL,VDL,COR
R1	V: $IC_u(7) > m(IC_2(6), IC_3(6))$	V: $IC_u(7) > m(IC_2(6), IC_3(6))$
R2	N: $IC_u(7) = IP_u(7)$	N: $IC_u(7) = IP_u(7)$
	11- MergeSort	
Ou	AODS,AOIS,AOIU,AORB,AORS,CDL,COI,COR,LOI,ODL,ROR,SDL	
OC_u	AOIS,AOIU,AORB,CDL,COI,LOI,ODL,ROR,SDL,VDL,AORS,AODS,COR	
OP_u	AOIS,AOIU,AORB,AORS,COI,LOI,ODL,ROR,SDL,VDL,COR	
R1	V: $IC_u(12) > m(IC_2(10), IC_3(7))$	

Tabela 5.11

C\P		
R2	V: $IC_u(12) > IP_u(10)$	
C2	12- oddOrPos	13- numZero
Ou	AOIS,AOIU,AORS,COI,COR,LOI,ODL,ROR,SDL,VDL	AOIS,AOIU,AORS,COI,LOI,ROR,SDL,VDL
OC_u	AOIS,AOIU,AORB,CDL,COI,COR,LOI,ODL,ROR,SDL,VDL,ASRS	AOIS,AOIU,AORB,CDL,COI,COR,LOI,ODL,ROR,SDL,VDL,ASRS
OP_u	AOIS,AOIU,AORB,AORS,COI,LOI,ODL,ROR,SDL,VDL,COR	AOIS,AOIU,AORB,AORS,COI,LOI,ODL,ROR,SDL,VDL,COR
R1	X: $IC_u(9) < m(IC_1(10), IC_3(8))$	X: $IC_u(7) < m(IC_1(8), IC_3(7))$
R2	X: $IC_u(9) < IP_u(10)$	X: $IC_u(7) < IP_u(8)$
	14- Gaussian	15- trashAndTakeOut
Ou	AOIS,AOIU,AORB,ASRS,CDL,COI,ODL,ROR,SDL,VDL	AOIS,AORB,COI,LOI,ODL,ROR,SDL,VDL
OC_u	AOIS,AOIU,AORB,AORS,CDL,COI,COR,LOI,ODL,ROR,SDL,VDL	AOIS,AOIU,AORB,CDL,COI,COR,LOI,ODL,ROR,SDL,VDL,ASRS
OP_u	AOIS,AOIU,AORB,AORS,COI,LOI,ODL,ROR,SDL,VDL,COR	AOIS,AOIU,AORB,AORS,COI,LOI,ODL,ROR,SDL,VDL,COR
R1	V: $IC_u(10) > m(IC_1(9), IC_3(7))$	V: $IC_u(8) \geq m(IC_1(8), IC_3(7))$
R2	V: $IC_u(9) > IP_u(8)$	N: $IC_u(8) = IP_u(8)$
	16- countPositive	17- twoPred
Ou	AOIS,AOIU,AORS,COI,LOI,ROR,SDL,VDL	COI,COR,ODL,ROR,SDL
OC_u	AOIS,AOIU,AORB,CDL,COI,COR,LOI,ODL,ROR,SDL,VDL,ASRS	AOIS,AOIU,AORB,CDL,COI,ODL,ROR,SDL,VDL,ASRS
OP_u	AOIS,AOIU,AORB,AORS,COI,LOI,ODL,ROR,SDL,VDL,COR	AOIS,AOIU,AORB,AORS,COI,LOI,ODL,ROR,SDL,VDL,COR
R1	X: $IC_u(7) < m(IC_1(8), IC_3(7))$	X: $IC_u(4) < m(IC_1(5), IC_3(4))$
R2	X: $IC_u(7) < IP_u(8)$	X: $IC_u(4) < IP_u(5)$
	18- RecursiveSelectionSort	19- LRS
Ou	AOIS,AORB,COI,ODL,ROR,SDL,VDL	AOIS,AORB,COI,LOI,ODL,ROR,SDL,VDL
OC_u	AOIS,AOIU,AORB,CDL,COI,COR,LOI,ODL,ROR,SDL,VDL,ASRS	AOIS,AOIU,AORB,CDL,COI,COR,LOI,ODL,ROR,SDL,VDL,ASRS
OP_u	AOIS,AOIU,AORB,AORS,COI,LOI,ODL,ROR,SDL,VDL,COR	AOIS,AOIU,AORB,AORS,COI,LOI,ODL,ROR,SDL,VDL,COR
R1	V: $IC_u(7) \geq m(IC_1(7), IC_3(6))$	V: $IC_u(8) \geq m(IC_1(8), IC_3(7))$
R2	N: $IC_u(7) = IP_u(7)$	N: $IC_u(8) = IP_u(8)$
	20- checkIt	21- cal
Ou	COI,COR,SDL	AOIS,AOIU,AORB,COI,LOI,ODL,ROR,SDL,VDL
OC_u	AOIS,AOIU,AORB,CDL,COI,COR,LOI,ODL,ROR,SDL,VDL,ASRS	COI,ODL,SDL,VDL,AOIS,AORB,CDL,LOI,ROR,AOIU,ASRS
OP_u	AOIS,AOIU,AORB,AORS,COI,LOI,ODL,ROR,SDL,VDL,COR	AOIS,AOIU,AORB,AORS,COI,LOI,ODL,ROR,SDL,VDL,COR
R1	V: $IC_u(3) \geq m(IC_1(3), IC_3(2))$	V: $IC_u(9) \geq m(IC_1(9), IC_3(8))$
R2	N: $IC_u(3) = IP_u(3)$	N: $IC_u(8/8) = IP_u(9/9)$
	22- stats	23- lastZero
Ou	AOIS,AOIU,AORB,AORS,ASRS,CDL,COI,LOI,ODL,ROR,SDL,VDL	AOIS,AORS,COI,LOI,ROR,SDL,VDL
OC_u	AOIS,AOIU,AORB,CDL,COI,COR,LOI,ODL,ROR,SDL,VDL,ASRS	AOIS,AOIU,AORB,CDL,COI,COR,LOI,ODL,ROR,SDL,VDL,ASRS
OP_u	AOIS,AOIU,AORB,AORS,COI,LOI,ODL,ROR,SDL,VDL,COR	AOIS,AOIU,AORB,AORS,COI,LOI,ODL,ROR,SDL,VDL,COR
R1	V: $IC_u(11) \geq m(IC_1(11), IC_3(8))$	X: $IC_u(6) < m(IC_1(7), IC_3(6))$

Tabela 5.11

C\P		
R2	V: $IC_u(11) > IP_u(10)$	X: $IC_u(6) < IP_u(7)$
	24- CheckPalindrome	
Ou	AOIS,AORB,CDL,COI,LOI,ODL,ROR,SDL	
OC_u	AOIS,AOIU,AORB,CDL,COI,COR,LOI,ODL,ROR,SDL,VDL,ASRS	
OP_u	AOIS,AOIU,AORB,AORS,COI,LOI,ODL,ROR,SDL,VDL,COR	
R1	V: $IC_u(8) \geq m(IC_1(8), IC_3(5))$	
R2	V: $IC_u(8) > IP_u(7)$	
C3	25- Queue	26- BoundedQueue
Ou	AOIS,AOIU,LOI,SDL,VDL	AOIS,AOIU,COI,LOI,SDL,VDL
OC_u	AOIS,AOIU,COI,LOI,ODL,ROR,SDL,VDL	AOIS,AOIU,ASRS,COI,LOI,ODL,ROR,SDL,VDL
OP_u	AOIS,AOIU,AORB,AORS,COI,LOI,ODL,ROR,SDL,VDL,COR	AOIS,AOIU,AORB,AORS,CDL,COI,COR,LOI,ODL,ROR,SDL,VDL,ASRS
R1	N: $IC_u(5) = m(IC_1(5), IC_2(5))$	N: $IC_u(6) = m(IC_1(6), IC_2(6))$
R2	N: $IC_u(5) = IP_u(5)$	N: $IC_u(6) = IP_u(6)$
	27- Triangle	28- Stack
Ou	SDL	AOIS,COI,LOI,ROR,SDL
OC_u	AOIS,AOIU,COI,LOI,ODL,ROR,SDL,VDL	AOIS,AOIU,COI,LOI,ODL,ROR,SDL,VDL
OP_u	AOIS,AOIU,AORB,AORS,COI,LOI,ODL,ROR,SDL,VDL,COR	AOIS,AOIU,AORB,AORS,COI,LOI,ODL,ROR,SDL,VDL,COR
R1	N: $IC_u(1) = m(IC_1(1), IC_2(1))$	N: $IC_u(5) = m(IC_1(5), IC_2(5))$
R2	N: $IC_u(1) = IP_u(1)$	N: $IC_u(5) = IP_u(5)$
	29- Calculation	
Ou	AOIS,AOIU,ASRS,COI,LOI,ROR,SDL,VDL	
OC_u	AOIS,AOIU,COI,LOI,ROR,SDL,VDL	
OP_u	AOIS,AOIU,AORB,AORS,COI,LOI,ODL,ROR,SDL,VDL,COR	
R1	X: $IC_u(7) < m(IC_1(7), IC_2(8))$	
R2	N: $IC_u(7) = IP_u(7)$	

Capítulo

6

Conclusão

A atividade de teste é fundamental para revelar defeitos em busca do aumento da qualidade na produção de software. A atividade pode ser dividida em fases, em relação ao momento do desenvolvimento, e em técnicas, referentes às informações coletadas dos programas e a forma como são vistos. Tais técnicas podem seguir diferentes critérios de aplicação. Ademais, a atividade de teste é influenciada pelos paradigmas de programação e os novos defeitos deles provenientes, o que deu origem a diversas abordagem para as suas fases e nas diferentes técnicas.

Este trabalho abordou a técnica de teste baseada em defeitos; mais especificamente, o trabalho abordou o critério de teste de mutação (DeMillo et al., 1978; Hamlet, 1977), que é considerado muito efetivo para avaliar a qualidade dos casos de teste e a presença de defeitos nos programas.

Em pesquisas específicas sobre o teste de mutação no contexto dos novos defeitos introduzidos pelos paradigmas, o foco costuma estar na definição de operadores para simular apropriadamente defeitos comuns ao desenvolvimento nessas estratégias; além disso, direta e indiretamente, as pesquisas também focam a questão da eficiência. No decorrer das últimas décadas, diversas técnicas de redução de custo foram apresentadas. Porém, em geral, os resultados da aplicação dessas técnicas são válidos somente para os programas que foram alvos dos experimentos realizados, como constataram, por exemplo, Siami-Namin et al. (2008), Offutt et al. (1996), e, mais recentemente, Papadakis e Malevris

(2010), Omar e Ghosh (2012) e Lacerda e Ferrari (2014). Em um trabalho mais recente, Kurtz et al. (2016) afirmaram que cada programa deve ser analisado individualmente, ou seja, operadores (e seus mutantes) que podem ser eficientes para um programa, podem não ser eficientes para outros. Sendo assim, a questão que perdura após geradas as evidências para grupos reduzidos de programas refere-se à utilidade dos resultados das técnicas de redução de custo para serem aplicados em outros programas que ainda não foram testados com o critério Análise de Mutantes.

Nesse contexto, neste trabalho explorou-se a possibilidade da inferência da similaridade entre programas com o propósito futuro de reutilizar informação de um programa testado – portanto, com informação confiável – para um novo programa cujas características são desconhecidas, porém mensuráveis.

Em alguns trabalhos sobre teste de mutação aplicado a programas desenvolvidos sob os paradigmas orientado a objetos (OO) e orientado a aspectos (OA) foram utilizadas heurísticas para calcular similaridades pontuais entre programas para auxiliar de forma secundária suas propostas.

A função secundária das heurísticas, no contexto apresentado, motivou o tema deste trabalho, que foca na utilização das heurísticas de similaridade como meio principal no suporte ao teste de mutação em programas OO. Em resumo, buscou-se explorar a possibilidade da inferência da similaridade entre programas com o propósito futuro de reutilizar informação de um programa testado – e, consequentemente, informação confiável – para um novo programa cujas características são desconhecidas. Mais especificamente, o objetivo geral deste trabalho foi a investigação e o uso da similaridade de programas, explorada no contexto do critério de teste de mutação, para auxiliar em sua redução de custo com base em estimativas obtidas de base histórica de programas similares já testados com o critério. As contribuições obtidas e uma breve discussão dos resultados deste trabalho são apresentadas na próxima seção.

6.1 Contribuições

Os objetivos específicos e as contribuições a eles associadas incluíram:

(i) uma caracterização do uso de similaridade de programas no contexto do teste de mutação OO e OA através de uma Revisão Sistemática;

(ii) a definição e o desenvolvimento de um *framework* conceitual e consequente ferramental que implementa o *framework* conceitual proposto; e

(iii) a realização de um estudo exploratório utilizando a clusterização de métricas para observar se o agrupamento influenciava em valores similares dos escores associados a operadores de mutação aplicados a programas presentes em três bases de dados diferentes.

Em relação aos resultados coletados na Revisão Sistemática, alguns estudos primários selecionados apresentaram heurísticas para calcular similaridade pontual em programas, baseadas nas seguintes abstrações e/ou informações: grafo de fluxo de controle, grafo de chamadas, estados dos programas, modelo de fluxo de dados, métricas CK, tempo de execução de casos de teste, árvore de transição de estados, e distância léxica de *strings*.

Outro resultado deste trabalho foi a definição e apresentação de um *framework* conceitual com o objetivo de fornecer suporte através da informação histórica de testes previamente realizados. Uma ferramenta foi implementada para automatizar os processos incluídos no *framework*, e os processos do *framework* foram mapeados aos processos implementados na ferramenta.

Os resultados colhidos de um estudo experimental, realizado com apoio do ferramental desenvolvido, envolveram três configurações de bases de dados de programas, e o resultado geral foi positivo em relação à existência e influência da similaridade através do agrupamento de métricas. Isso ocorreu, em média, devido aos valores mais similares da intersecção do grupo similar com programa de interesse comparado à intersecção dos outros *clusters* ou restante dos programas da base quando realizam intersecção com o programa de interesse. Todavia, tal resultado não ocorreu para os menores *clusters* formados para cada configuração da base de programas. Além disso, a intersecção de operadores candidatos a One-Op dos $u's$ em relação aos grupos a eles similares correspondem a metade dos operadores candidatos dos $u's$ em duas das três bases de dados, provavelmente graças à presença de programas com características internas distintas provenientes das duas fontes que compõem as bases de dados.

6.2 Limitações e Trabalhos Futuros

As principais limitações deste trabalho são:

- A exploração de apenas uma técnica de redução de custo (One-Op), uma única técnica de cálculo de similaridade (agrupamento), e uma única abstração dos programas considerados (métricas internas).
- A não coleta de métricas de classes que possuem classes internas.

- O reduzido número de programas utilizados no experimento, oriundos de diferentes fontes (e consequente diferentes grupos de operadores utilizados nos testes), e o fato de em todas as fontes ter sido utilizada a mesma ferramenta de teste de mutação. No geral, obteve-se pouca diversidade de programas e pouca diversidade de formas de geração dos mutantes.

- A qualidade dos dados de teste disponibilizados na composição da base, e a importação de dados em diferentes formatos.

- A necessidade da execução, neste momento manual, da clusterização para as diferentes combinações de métricas CK com objetivo de selecionar a melhor combinação de métricas geradoras de *clusters* para os programas envolvidos.

Trabalhos futuros possíveis são:

- Um Mapeamento e posterior Revisão Sistemática para a caracterização e coleta da similaridade de programas e suas heurísticas e cálculos fora do contexto do teste de mutação, para a posterior aplicação no último.

- A implementação do cálculo sobre heurísticas já apresentadas como, por exemplo, o grafo de fluxo de chamadas calculado pela ferramenta Analizo.

- Ultrapassar as limitações deste trabalho com uma base de dados mais diversificada e com a presença de classes internas.

- A análise de diferentes cálculos para uma dada heurística.

- A análise das diferentes heurísticas para um dado cálculo.

- A observação de diferentes estratégias de redução de custo do teste de mutação com a assistência da similaridade diversamente calculada.

Referências

Abdellatief, M.; Sultan, A. B. M.; Ghani, A. A. A.; Jabar, M. A. A Mapping Study to Investigate Component-based Software System Metrics. *Journal of Systems and Software*, v. 86, n. 3, p. 587 – 603, 2013.

Abreu, F. B.; Carapuca, R. Object-Oriented Software Engineering: Measuring and Controlling the Development Process. In: *Proceedings of the 4th International Conference on Software Quality (ICSQ'94)*, McLean/VA -USA: ASQ – American Society for Quality, p. 1–8, 1994.

Agrawal, H.; DeMillo, R. A.; Hathaway, R.; Hsu, W.; Hsu, W.; Krauser, E. W.; Martin, R. J.; Mathur, A. P.; Spafford, E. H. *Design of Mutant Operators for the C Programming Language*. Technical Report SERC-TR41-P, Software Engineering Research Center, Purdue University, West Lafayette /IN-USA, disponível em `https://www.cs.purdue.edu/homes/apm/mutop-design-harness.pdf`, 1989.

Alexander, R. T.; Bieman, J. M.; Andrews, A. *Towards the Systematic Testing of Aspect-Oriented Programs*. Tech. Report CS-04-105, Dept. of Computer Science, Colorado State University, Fort Collins/CO-USA, 2004.

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, p. 21–30, 2014.

Anbalagan, P.; Xie, T. Automated Generation of Pointcut Mutants for Testing Pointcuts in AspectJ Programs. In: *Proceedings of the 19th International Symposium on Software Reliability Engineering (ISSRE)*, Seattle/WA-USA: IEEE Computer Society, p. 239–248, 2008.

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 Press, p. 402–411, 2005.

Andrews, J. H.; Briand, L. C.; Labiche, Y.; Namin, A. S. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. *IEEE Transactions on Software Engineering*, v. 32, n. 8, p. 608–624, 2006.

Barbosa, E. F.; Maldonado, J. C.; Vincenzi, A. M. R. Toward the Determination of Sufficient Mutant Operators for C. *stvr*, v. 11, n. 2, p. 113–136, 2001.

Bashir, M. B.; Nadeen, A. A Fitness Function for Evolutionary Mutation Testing of Object-Oriented Programs. In: *9th International Conference on Emerging Technologies (ICET)*, Islamabad, Pakistan: IEEE Computer Society, 2013.

Bieman, J. M.; Ghosh, S.; Alexander, R. T. A Technique for Mutation of Java Objects. In: *Proceedings of the 16th International Conference on Automated Software Engineering (ASE)*, San Diego/CA-USA: IEEE Computer Society, p. 23–26, 2001.

Boscarioli, C.; Peres, S. M.; Silva, L. A. *Introdução a Mineração de Dados: com Aplicações em R.* Rio de Janeiro/RJ - Brasil: Elsevier, 2016.

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.

Budd, T. A. *Mutation Analysis: Ideas, Example, Problems and Prospects.* Computer Program Testing North-Holand Publishing Company, 1981.

Chevalley, P. Applying Mutation Analysis for Object-Oriented Programs Using a Reflective Approach. In: *Proceedings of the 8th Asia-Pacific Software Engineering Conference (APSEC)*, MO-Chine: IEEE Computer Socitey Press, p. 267–272, 2001.

Chidamber, S. R.; Kemerer, C. F. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, v. 20, n. 6, p. 476–493, 1994.

Cook, T. D.; Campbell, D. T. *Quasi-Experimentation: Design & Analysis Issues for Field Settings.* Boston/MA-USA: Houghton Mifflin, 90-139 p., 1979.

Crespo, A. N.; Bueno, P.; Argollo, M.; Barros, C. P.; Jino, M. Software Testing in the Context of Brazilian Public Software. In: *Proceedings of the 2th Brazilian Program of Quality and Productivity in Software (PBQP), Campinas/SP-Brasil. Ministério da Ciência e Tecnologia (MCT)*, 2011.

Cruz, R. C.; Eler, M. M. Using a Cluster Analysis Method for Grouping Classes According to their inferred Testability: An Investigation of CK Metrics, Code Coverage and Mutation Score. In: *Proceedings of the 36th International Conference of the Chilean Computer Science Society (SCCC)*, Arica/Arica e Parinacota-Chile: Chilean Computer Science Society, p. 1–11, 2017.

Dallilo, L. D. Investigação de Similaridade entre Programas para Apoiar o Teste de Mutação. Exame Geral de Qualificação para Mestrado, DC/UFSCar, São Carlos/SP-Brasil (Mestrado em andamento), 2017.

Delamaro, M. E.; Deng, L.; Durelli, V.; Li, N.; Offutt, A. J. Experimental Evaluation of SDL and One-Op Mutation for C. In: *IEEE Seventh International Conference on Software Testing, Verification and Validation*, Cleveland/OH-USA, 2014.

Delamaro, M. E.; Maldonado, J. C.; Mathur, A. P. Interface Mutation: An Approach for Integration Testing. *IEEE Transactions on Software Engineering*, v. 27, n. 3, p. 228–247, 2001.

Delamaro, M. E.; Offutt, A. J.; Ammann, P. Designing Deletion Mutation Operators. In: *IEEE Seventh International Conference on Software Testing, Verification and Validation*, Cleveland/OH-USA, 2014.

Delamaro, M. E.; Pezzè, M.; Vincenzi, A. M. R.; Maldonado, J. C. Mutant Operator for Testing Concurrent Java Programs. In: *Anais do 15th Simpósio Brasileiro de Engenharia de Software (SBES)*, Rio de Janeiro/RJ-Brasil, p. 386–391, 2001.

DeMillo, R. A.; Lipton, R. J.; Sayward, F. G. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer*, v. 11, n. 4, p. 34–43, 1978.

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 Computer Society, p. 84–93, 2013.

Derezinska, A. *Evaluation of Deletion Mutation Operators in Mutation Testing of C# Programs*. Brunów/Lower Silesia Province-Poland: DepCoS-RELCOMEX, 2016.

Do, H.; Rothermel, G. On the Use of Mutation Faults in Empirical Assessments of Test Case Prioritization Techniques. *IEEE Transactions on Software Engineering*, v. 32, n. 9, p. 733–752, 2006.

Feldt, R.; Poulding, S.; Clark, D.; Yoo, S. Test Set Diameter: Quantifying the Diversity of Sets of Test Cases. In: *International Conference on Software Testing, Verification and Validation (ICST)*, Chicago/IL-USA: IEEE, p. 223–233, 2016.

Fentom, N.; Bieman, J. *Software Metrics: A Rigorous and Practical Approach.* 3^{rd} ed. Boca Raton/FL-USA: CRC Press, Inc, 2014.

Ferrari, F. C.; Burrows, R.; Lemos, O. A. L.; Garcia, A.; Maldonado, J. C. Characterising Faults in Aspect-Oriented Programs: Towards Filling the Gap between Theory and Practice. In: *Proceedings of the 24th Brazilian Symposium on Software Engineering (SBES)*, Salvador/BA-Brasil: IEEE Computer Society, p. 50–59, 2010.

Ferrari, F. C.; Pizzoleto, 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/Västmanland-Sweden: IEEE, p. 1–10, 2018.

Frank, E.; Hall, M. A.; Witten, I. H. *Data Mining: Edition Practical Machine Learning Tools and Techniques.* 4^{th} ed. Morgan Kaufmann, 2016.

Frankl, P. G.; Weyuker, E. J. Testing Software to Detect and Reduce Risk. *Journal of Systems and Software*, v. 53, n. 3, p. 275–286, 2000.

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

Goldschmidt, R.; Passos, E. *Data Mining: Um Guia Prático.* Rio de Janeiro/RJ-Brasil: Elsevier, 2005.

Hamlet, R. G. Testing Programs with the Aid of a Compiler. *IEEE Transactions on Software Engineering*, v. 3, n. 4, p. 279–290, 1977.

Harrold, M. J.; Rothermel, G. Performing Data Flow Testing on Classes. In: *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, New Orleans/LA-USA: ACM Press, p. 154–163, 1994.

IEEE *829 Standard for Software and System Test Documentation.* Standard 610.12, Institute of Electric and Electronic Engineers, New York/NY-USA, 1990.

Jones, C. *Applied Software Measurement: Global Analysis of Productivity and Quality.* 3^{rd} ed. New York/NY-USA: McGraw-Hill Education Group, 2008.

Just, R.; Schweiggert, F.; Kapfhammer, G. M. MAJOR: An Efficient and Extensible Tool for Mutation Analysis in a Java Compiler. In: *26^{th} IEEE/ACM International Conference on Automated Software Engineering*, Lawrence/KS-USA: IEEE, p. 612–615, 2011.

Kiczales, G.; Irwin, J.; Lamping, J.; Loingtier, J. M.; Lopes, C.; Maeda, C.; Menhdhekar, A. Aspect-Oriented Programming. In: *Proceedings of the 11^{th} European Conference on Object-Oriented Programming (ECOOP)*, Jyväskylä/Central Finland-Finland: Springer-Verlag, p. 220–242 (LNCS v.1241), 1997.

Kim, S.; Clark, J.; McDermid, J. Class Mutation: Mutation Testing for Object-Oriented Programs. In: *Proceedings of the Net.ObjectiveDays Conference*, 2000.

Kintis, M.; Malevris, N. Identifying More Equivalent Mutants via Code Similarity. In: *20^{th} Asia-Pacific Software Engineering Conference (APSEC)*, Bangkok/Central Thailand-Thailand: IEEE, p. 180–188, 2013.

Kurtz, B.; Ammann, P.; Offutt, A. J.; Delamaro, M. E.; Kurtz, M.; Gökçe, N. Analyzing the Validity of Selective Mutation with Dominator Mutants. In: *Proceedings of the 24^{th} ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Seattle/WA-USA: ACM, p. 571–582, 2016.

Lacerda, J. T. S.; Ferrari, F. C. Towards the Establishment of a Sufficient Set of Mutation Operators for AspectJ Programs. In: *Proceedings of the 8^{th} Brazilian Workshop on Systematic and Automated Software Testing (SAST)*, Maceio/AL-Brasil: Brazilian Computer Society, 2014.

Lemos, O. A. L.; Franchin, I. G.; Masiero, P. C. Integration Testing of Object-Oriented and Aspect-Oriented Programs: A Structural Pairwise Approach for Java. *Science of Computer Programming*, v. 74, n. 10, p. 861–878, 2009.

Li, W. Software Product Metrics. *IEEE Potentials*, v. 18, p. 24–27, 1999.

Li, W.; Henry, S. Maintenance Metrics for the Object Oriented Paradigm. In: *Proceedings First International Software Metrics Symposium*, Baltimore/MD-USA: IEEE, p. 52–60, 1993.

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, p. 352–366, 2002.

Ma, Y. S.; Offutt, A. J. *Description of muJava's Method-level Mutation Operators*. Technical report, Electronics and Telecommunications Research Institute, Korea, disponível em `https://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf`, 2005.

Ma, Y. S.; Offutt, A. J.; Kwon, Y. R. MuJava: An Automated Class Mutation System. *Software: Testing, Verification and Reliability*, v. 15, n. 2, p. 97–133, 2005.

Ma, Y. S.; Offutt, A. J.; Kwon, Y. R. *μJava*: A Mutation System for Java. In: *Proceedings of the 28th International Conference on Software engineering (ICSE) (Posters Section)*, Shanghai/China: ACM Press, p. 827–830, 2006.

Maldonado, J. C. *Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software*. PhD Thesis, DCA/FEE, Universidade Estadual de Campinas (UNICAMP), Campinas/SP-Brasil, 1991.

Mathur, A. P. *Foundations of Software Testing*. Toronto/ON-Canada: Addison-Wesley Professional, 2007.

McCabe, T. J. A Complexity Measure. In: *ICSE Proceedings of the 2nd international conference on Software engineering*, Los Alamitos/CA-USA: IEEE, p. 407, 1976.

Moghadam, M. H.; Babamir, S. M. Mutation Score Evaluation in Terms of Object-Oriented Metrics. In: *Proceedings of the 4th International Conference on Computer and Knowledge Engineering (ICCKE)*, Mashhad/Razavi Khorasan-Iran: IEEE, p. 775–780, 2014.

Myers, G. J.; Badgett, T.; Sandler, C. *The Art of Software Testing*. 3rd ed. Hoboken/NJ - USA: John Wiley & Sons, 2011.

Namin, A. S.; Xue, X.; Rosas, O.; Sharma, P. MuRanker: a Mutant Ranking Tool. *Software: Testing, Verification and Reliability*, v. 25, n. 5-7, p. 572–604, 2015.

Offutt, A. J.; Alexander, R.; Wu, Y.; Xiao, Q.; Hutchinson, C. A Fault Model for Subtype Inheritance and Polymorphism. In: *Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE)*, Hong Kong/China: IEEE Computer Society Press, p. 84–93, 2001.

Offutt, A. J.; Lee, A.; Rothermel, G.; Untch, R. H.; Zapf, C. An Experimental Determination of Sufficient Mutant Operators. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, v. 5, n. 2, p. 99–118, 1996.

Offutt, A. J.; Pan, J. Automatically Detecting Equivalent Mutants and Infeasible Paths. *Software: Testing, Verification, and Reliability*, v. 7, n. 3, p. 165–192, 1997.

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 Computer Society, p. 100–107, 1993.

Offutt, A. J.; Untch, R. H. Mutation 2000: Uniting the Orthogonal. In: *Wong W.E. (eds) Mutation Testing for the New Century.*, Boston/MA-EUA: Springer, p. The Springer International Series on Advances in Database Systems, vol 24, 2001.

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: *Congress on Evolutionary Computation, Cancún/ROO-México*, IEEE, p. 829–836, 2013.

Omar, E.; Ghosh, S. An Exploratory Study of Higher Order Mutation Testing in Aspect-Oriented Programming. In: *Proceedings of the 23rd International Symposium on Software Reliability Engineering (ISSRE)*, Dallas/TX-USA: IEEE, p. 1–10, 2012.

Papadakis, M.; Kintis, M.; Zhang, J.; Jia, Y.; Le Traon, Y.; Harman, M. Mutation Testing Advances: An Analysis and Survey. In: Memon, A. M., ed. *Advances in Computers*, v. 112, p. 275–378, 2019.

Papadakis, M.; Malevris, N. An Empirical Evaluation of the First and Second Order Mutation Testing Strategies. In: *Proceedings of the 5th International Workshop on Mutation Analysis (Mutation)*, Paris/Paris Region-France: IEEE Computer Society, p. 90–99, 2010.

Pelleg, D.; Moore, A. W. X-means: Extending K-means with Efficient Estimation of the Number of Clusters. In: *Proceedings of the 17th International Conference on Machine Learning*, San Francisco/CA-USA: Morgan Kaufmann Publishers, 2000.

Polo, M.; Piattini, M.; Rodríguez, I. G. Decreasing the Cost of Mutation Testing with Second-Order Mutants. *Software: Testing, Verification and Reliability*, v. 19, p. 111–131, 2009.

Pressman, R. S. *Software Engineering: A Practitioner's Approach.* 7^{th} ed. New York/NY-USA: McGraw-Hill, 2010.

Putnam, L. H.; Myers, W. *Five Core Metrics: Intelligence behind Successful Software Management.* New York/NY-USA: Dorset House Publishing Co., Inc, 2003.

Rapps, S.; Weyuker, E. J. Data Flow Analysis Techniques for Program Test Data Selection. In: *Proceedings of the 6^{th} International Conference on Software Engineering (ICSE)*, Tokio/Kantō-Japan: IEEE Computer Society, p. 272–278, 1982.

Ristad, E.; Yianilos, P. Learning String Edit Distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, p. 522–532, 1998.

Schuler, D.; Zeller, A. (Un-)Covering Equivalent Mutants. In: *3^{rd} International Conference on Software Testing, Verification and Validation*, Saarbrücken/Saarland-Germany: IEEE Computer Society, p. 45–54, 2010.

Siami-Namin, A.; ; Andrews, J. H.; Murdoch, D. J. Sufficient Mutation Operators for Measuring Test Effectiveness. In: *Proceedings of the 30^{th} International Conference on Software Engineering (ICSE)*, Leipzig/Saxony-Germany: ACM Press, p. 351–360, 2008.

Sommerville, I. *Software Engineering.* 9^{th} ed. Boston/MA-USA: Addison-Wesley, 2011.

Terceiro, A.; Costa, J.; Miranda, J.; Meirelles, P.; Rios, L. R.; Almeida, L.; Chavez, C.; Kon, F. Analizo: an Extensible Multi-Language Source Code Analysis and Visualization Toolkit. In: *Brazilian Conference on Software: Theory and Practice (CBSoft) – Tools*, Salvador/BA-Brasil, 2010.

Untch, R. H. On Reduced Neighborhood Mutation Analysis Using a Single Mutagenic Operator. In: *ACM-SE 47 Proceedings of the 47^{th} Annual Southeast Regional Conference*, Clemson/SC-USA: ACM, 2009.

Vincenzi, A. M. R. *Orientação a Objeto: Definição, Implementação e Análise de Recursos de Teste e Validação.* PhD Thesis, ICMC/USP, São Carlos/SP-Brasil, 2004.

Walenstein, A.; El-Ramly, M.; Cordy, J. R.; Evans, W. S.; Mahdavi, K.; Pizka, M.; Ramalingam, G.; Von-Gudenberg, J. W. Similarity in Programs. In: Koschke, R.; Merlo, E.; Walenstein, A., eds. *Duplication, Redundancy, and Similarity in Software*, Dagstuhl Seminar Proceedings, Wadern/Saarland-Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007 (*Dagstuhl Seminar Proceedings*,).
Disponível em `http://drops.dagstuhl.de/opus/volltexte/2007/968`

Xu, D.; Ding, J. Prioritizing State-Based Aspect Tests. In: *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST)*, Paris/Paris Region-France: IEEE Computer Society, p. 265–274, 2010.

Zamboni, A.; Thommazo, D. A.; Hernandes, E.; Fabbri, S. StArt: Uma Ferramenta Computacional de Apoio à Revisão Sistemática. In: *Salão de Ferramentas. Congresso Brasileiro de Software*, CBSoft, p. 91–96, 2010.