UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

REDUZINDO O CUSTO DO TESTE DE MUTAÇÃO COM BASE EM INFORMAÇÕES DE ANÁLISE ESTÁTICA

VINÍCIUS BARCELOS SILVA

ORIENTADOR: PROF. DR. AURI MARCELO RIZZO VINCENZI

São Carlos – SP

Fevereiro/2019

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

REDUZINDO O CUSTO DO TESTE DE MUTAÇÃO COM BASE EM INFORMAÇÕES DE ANÁLISE ESTÁTICA

VINÍCIUS BARCELOS SILVA

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. Auri Marcelo Rizzo Vincenzi

São Carlos – SP

Fevereiro/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 Vinícius Barcelos Silva, realizada em 26/02/2019:

Prof. Dr. Auri Marcelo Rizzo Vincenzi
UFSCar

Prof. Dr. Delano Medeiros Beder
UFSCar

Prof. Dr. Marcio Eduardo Delamaro
USP

A todos que sempre estiveram ao meu lado.

AGRADECIMENTOS

A Deus por estar sempre comigo em todos momentos me mantendo firme em todas batalhas e desafios.

A minha família que sempre esteve comigo em cada momento, que sempre me apoiaram até nos momentos mais difíceis que passei.

A todas as pessoas que fazem parte da minha vida e se fizeram presentes cada um a sua maneira, mas que foram importantes também.

Ao meu orientador Prof. Dr. Auri Vincenzi pela atenção, orientação e dedicação acadêmica, por compartilhar seu conhecimento dentro do que fosse necessário para que os objetivos fossem atingidos da melhor maneira possível.

Ao DC-UFSCar pela estrutura e ensino de qualidade, e a Capes pela bolsa de estudos de Mestrado.

Agradeço do fundo do meu coração a todos.

Cada pessoa é aquilo que crê; fala o que gosta; retém o que procura; ensina o que aprende; tem o que dá e vale o que faz.

Chico Xavier

RESUMO

Para garantir a qualidade do software, podem-se usar técnicas de análise estática e dinâmica. Ambas têm vantagens e desvantagens e devem ser usadas em conjunto para melhorar a qualidade dos resultados obtidos. Neste trabalho, é apresentada uma estratégia para a aplicação de um conjunto de operadores de mutação no teste de software, o que representa uma técnica dinâmica, com base na dificuldade que analisadores estáticos automatizado têm na detecção de defeitos modelados. Em outras palavras, são investigadas quais categorias de defeitos, representados por operadores de mutação, os analisadores estáticos automatizados são capazes de reconhecer e, assim, priorizar o teste de mutação considerando apenas o conjunto de operadores de mutação que modelam defeitos difíceis de serem identificados pelo analisador estático utilizado. O subconjunto obtido é comparado com outros subconjuntos de operadores de mutação já propostos e, com os dados coletados, a análise estatística demonstrou que existem diferenças estatísticas entre o escore de mutação e a redução de custo da estratégia proposta e as estratégias presentes na literatura. Consideram-se os resultados obtidos promissores uma vez que o subconjunto de operadores de mutação identificados por meio da estratégia proposta apresenta custo e escore de mutação semelhantes aos subconjuntos de operadores de mutação comparados, mas ainda agregam informações não capturadas previamente por esses subconjuntos, ou seja, a intersecção com defeitos que podem ser identificados por meio de análise estática a um custo mais baixo.

Palavras-chave: Teste de Software, Análise Estática, Teste de Mutação, Mutação Seletiva, Estratégia de Teste Incremental

ABSTRACT

To guarantee the quality of the software, static and dynamic analysis techniques can be used. Both have advantages and disadvantages and should be used together to improve the quality of the results obtained. In this work, we present a strategy for the application of a set of mutation operators in the software test, which represents a dynamic technique, based on the difficulty that automated static analyzers have in the detection of model defects. In other words, we investigate which defect categories, represented by mutation operators; an automated static analyzer can recognize and thus prioritize the mutation test considering only the set of mutation operators that model defects that are difficult to identify by the static analyzer used. We compare our subset with other subsets of already proposed mutation operators and. With the data collected, the statistical analysis showed that there are statistical differences between the mutation score and the cost reduction of the proposed strategy and the strategies present in the literature. The results obtained are promising since the subset of mutation operators identified by the proposed approach presents similar cost and mutation score to the subsets of mutation operators compared but still aggregate information not previously captured by these subsets, i.e., the intersection with defects that can be identified employing static analysis at a lower cost.

Keywords: Software Testing, Static Analysis, Mutation Testing, Selective Mutation, Incremental Testing Strategy

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

Capítulo 1

INTRODUÇÃO

1.1 Contexto e Motivação

Nas atividades de desenvolvimento de software é comum a utilização de técnicas e ferramentas para obtenção de um produto de qualidade e de baixo custo. Entre as atividades que podem contribuir para um produto de qualidade tem-se a utilização de analisadores estáticos e a aplicação de técnicas e critérios de testes.

As ferramentas de análise estática automatizadas podem ser empregadas para auxiliar no reconhecimento de certos tipos de defeitos e vulnerabilidades. Exemplos de ocorrências que podem ser detectadas por essas ferramentas são: prováveis laços infinitos, estouro de *buffer*, erro no gerenciamento de memória, dentre outros.

Essas ferramentas emitem uma grande quantidade de avisos, alertando o testador sobre possíveis problemas que podem estar presentes no código fonte, sem a necessidade de execução do mesmo. A alta taxa de avisos falsos positivos relatados por ferramentas de análise estática é apontada como sua principal desvantagem, ou seja, avisos relatados que não correspondem a defeitos reais, mas que vão demandar tempo de investigação por parte da equipe para sua análise (FILHO et al., 2010; COUTO et al., 2013; JOHNSON et al., 2013). Os verdadeiros positivos são avisos emitidos que realmente indicam a presença de um defeito real no código fonte. Há ainda os avisos classificados como falsos negativos, que correspondem a defeitos existentes no código fonte, mas que devido à ausência de regras sintáticas implementadas, tais ferramentas de análise estáticas ainda não são capazes de detectá-los.

Existem diversas ferramentas de análise estática automatizadas disponíveis para diversas linguagens de programação, alguns exemplos são: a StyleCop (MICROSOFT, 2014) e a FxCop (MICROSOFT, 2011) para sistemas em .NET, a *FindBugs* (HOVEMEYER; PUGH, 2004) e

a PMD (COPELAND, 2005) para Java, a Lint (POHL, 2001), *Splint* (EVANS; LAROCHELLE, 2002) e Frama-C (CORRENSON et al., 2014a) para C/C++.

Apesar do interesse acadêmico e industrial no uso das ferramentas de análise estática automatizadas (HOVEMEYER; PUGH, 2004; LOURIDAS, 2006; AYEWAH et al., 2007b), principalmente devido ao seu baixo custo de utilização, ainda existe uma resistência devido à falta de evidências das contribuições reais que tais ferramentas agregam na produção de um software de qualidade (AYEWAH et al., 2007a), principalmente devido ao grande número de avisos emitidos e que são falsos positivos.

Uma técnica de teste bastante eficaz na detecção e eliminação dos defeitos presentes no software é o Teste de Mutação (DEMILLO et al., 1978). Sendo considerado um excelente modelo de defeitos, o Teste de Mutação é, em geral, empregado para avaliar a qualidade de conjuntos de teste ou critérios de teste, tornado-o muito utilizado em experimentação (ANDREWS et al., 2005). Em geral, defeitos são introduzidos no software baseado em enganos cometidos por pessoas ao interpretar erroneamente informações sobre o que deve ser feito ou ao utilizar determinada linguagem de programação. Um comando ou uma instrução incorreta presente no código fonte é um exemplo de defeito (ISO/IEC/IEEE, 2017).

O Teste de Mutação gera, por meio dos operadores de mutação, versões do programa em teste contendo possíveis defeitos. Assumindo que o programa original está correto, é necessário construir casos de testes que mostrem que o mutante e o programa original se comportem de maneira diferente e, nesse caso, descartando a possibilidade do programa possuir o defeito representado pelo mutante.

Um dos problemas do Teste de Mutação é o alto custo computacional devido ao grande número de mutantes gerados, que demandam tempo para a execução, e posterior análise de mutantes vivos para identificação de mutantes equivalentes. Desse modo, diferentes alternativas são empregadas para reduzir esses custos. Uma delas é diminuir o número de operadores de mutação que são utilizados, gerando-se apenas um subconjunto de todos os possíveis mutantes.

Araújo et al. (2015, 2016) fez um uso alternativo do teste de mutação para a avaliação da qualidade de analisadores estáticos automatizados para a linguagem Java. Nesse estudo, foi medida a taxa de correspondência entre avisos emitidos pelos analisadores estáticos e defeitos representados por meio de mutantes. Araújo et al. (2015, 2016) teve seus mutantes gerados por meio da ferramenta de mutação μJava e a ferramenta de análise estática utilizada foi a *FindBugs*. Observou-se que existem algumas categorias de defeitos, representadas pelos operadores de mutação, que são mais facilmente detectáveis pelos analisadores estáticos automatizados,

enquanto há outras categorias de defeitos que são quase imperceptíveis por tais analisadores, evidenciando o aspecto complementar da análise estática e dinâmica.

Barbosa et al. (2001) definiu dois subconjuntos essenciais a partir da execução de dois experimentos, sendo o Experimento-I que gerou um conjunto essencial, denominado E27, a partir de um grupo de 27 programas que compõem um editor de texto simplificado e, o Experimento-II que gerou um conjunto essencial, denominado E5, a partir do grupo de 5 programas utilitários Unix. Em seu trabalho o escore de mutação acima de 0,995 foi considerado um bom ponto de corte, proporcionando em média reduções de custo superiores a 65%. Tendo o escore de mutação definido por Barbosa et al. (2001) como referência para o ponto de corte, a definição de um critério de mutação seletiva no presente trabalho se baseou nessa informação.

O objetivo deste trabalho é realizar uma análise sob a perspectiva dos tipos de operadores de mutação que são mais difíceis de serem identificados por analisadores estáticos automatizados e definir uma ordem de aplicação incremental dos operadores de mutação com base em informações de análise estática, definir uma estratégia de mutação seletiva com base nos operadores priorizados e realizar uma análise de eficácia e custo em termos do número de mutantes gerados e o número de equivalentes da estratégia proposta.

Os resultados obtidos são animadores uma vez que o subconjunto de operadores de mutação identificado por meio da estratégia proposta apresenta custo e escore de mutação semelhantes ao conjunto essencial de operadores de mutação mas, ainda agregam informações não capturadas previamente pelo conjunto essencial.

1.2 Solução Proposta

No trabalho de Araújo et al. (2015, 2016) foi avaliada a correspondência entre operadores de mutação e tipos de avisos emitidos por analisador estático no contexto da linguagem Java. Como resultado, duas estratégias de priorização foram definidas. A primeira visava a priorização de tipos de aviso que mais detectam defeitos modelados pelos operadores de mutação. A segunda visava a priorização de operadores de mutação que geravam tipos de defeitos dificilmente detectáveis por analisadores estáticos.

Entretanto, apesar de propostas as estratégias de priorização, Araújo et al. (2015, 2016) não conduziu uma avaliação experimental para confirmar se as estratégias propostas proporcionavam uma melhor redução de custo e maior eficácia na detecção de defeitos, principalmente considerando o teste de mutação.

Nesse sentido, o presente trabalho pretende estender a análise da estratégia de priorização proposta em outro contexto, realizando uma avaliação dinâmica da ordem de priorização da aplicação dos operadores de mutação que geram mutantes difíceis de serem detectados pelo analisador estático. Tal ordem de prioridade pode ser entendida como um tipo de mutação seletiva, no qual apenas um subconjunto de operadores de mutação é utilizado, reduzindo o custo do teste de mutação mantendo-se praticamente a mesma eficácia.

A Ferramenta *Splint* foi escolhida como analisador estático e a Ferramenta *Proteum/IM* como a ferramenta de mutação. No caso da *Splint* foram utilizadas todas as possíveis regras de análise estática. Na *Proteum/IM* os 78 operadores de unidade implementados.

A escolha da *Splint* como ferramenta de análise estática é devido ao fato da mesma ser uma das ferramentas mais conhecidas para Linguagem C, e ser a evolução da Ferramenta Lint (POHL, 2001), que é uma das ferramentas referência para Linguagem C. A *Splint* também foi usada no trabalho de Evans e Larochelle (EVANS; LAROCHELLE, 2002), para detectar falhas de segurança. A escolha da Ferramenta *Proteum/IM* é devido ao fato dessa ser a mais popular ferramenta de apoio ao teste de mutação para Programas C e ser utilizada extensamente outros estudos experimentais (DELAMARO et al., 2000), já contando com uma vasata base de dados de programas com casos de testes e mutantes equivalentes determinados, sendo ideal para experimentação.

1.2.1 Objetivos

Considerando o contexto e a motivação apresentados, os objetivos do presente trabalho são:

1. Realizar uma análise sob a perspectiva dos tipos de operadores de mutação que são mais difíceis de serem identificados por analisadores estáticos automatizados;
   (a) Definir uma ordem de aplicação incremental dos operadores de mutação com base em informações de análise estática;
   (b) Definir uma estratégia de mutação seletiva baseada nos operadores priorizados, que será denominada como STAT; e
   (c) Realizar uma análise de custo e eficácia em termos do número de mutantes gerados e número de equivalentes da estratégia proposta.

1.2.2 Metodologia

A metodologia empregada no presente trabalho é baseada em experimentação. Para executar o experimento de modo a atingir os objetivos e coletar os dados necessários devem ser seguidos os seguintes passos:

1. Dado um conjunto S de sistemas;
2. Para cada sistema s_i do conjunto S;
   (a) A *Splint* é executada sobre cada arquivo original de s_i;
   (b) A saída gerada pela *Splint* é preparada para ser processada;
   (c) A *Proteum/IM* é executada sobre cada arquivo de s_i, gerando assim os mutantes;
   (d) A *Splint* é executada sobre cada mutante gerado;
   (e) A saída gerada pela *Splint* é preparada para ser processada;
   (f) É realizada a coleta dos dados gerados pela *Splint* e as informações relativas a cada mutante;
   (g) Os dados coletados são armazenados no banco de dados;
3. A correspondência entre avisos estáticos e mutações é calculada a partir dos dados armazenados no banco de dados.

Os dados armazenados agora podem ser combinados e assim pode-se obter as informações mencionadas nos objetivos. Pode-se contabilizar o total de avisos relatados em todos mutantes, verificar a variação da quantidade de avisos relatados no arquivo original e nos seus arquivos mutantes respectivos e verificar a quantidade de mutantes gerados.

Foram utilizados no experimento um total de quarenta e dois programas, dentre os quais estão programas utilitários do UNIX, programas da Siemens do repositório Sir (DO et al., 2005) e alguns outros do livro Introdução ao Teste de Software (AMMANN; OFFUTT, 2008). Estes programas foram escolhidos por já apresentarem conjuntos de casos de testes adequados ao teste de mutação, com mutantes equivalentes já determinados. Além disso tais programas já foram utilizados em uma vasta gama de estudos envolvendo teste de mutantes (WONG, 1993; DELAMARO, 1997b; WONG et al., 1997; BARBOSA et al., 2001; VINCENZI, 1998). Além disso, devido ao seu uso intenso, tais programas possuem uma baixa probabilidade de apresentarem defeitos reais em seu código.

1.2.3 Contribuições

As principais contribuições desse trabalho são:

1. Identificação de um subconjunto de operadores de mutação da *Proteum/IM* que produzem mutações difíceis de serem detectadas pelo analisador estático;
2. Definição de um critério de mutação seletiva com base no subconjunto de operadores identificado no item anteior;
3. Condução de estudos experimentais visando avaliar o custo e a eficácia do critério seletivo proposto em relação a outras abordagens (critérios seletivos) propostas na literatura;
4. Identificação das categorias de defeitos (representadas pelos operadores de mutação) que o analisador estático tem maior capacidade de detecção;
5. Identificação de um conjunto dos tipos de avisos do analisador estático que mais e menos detectaram defeitos modelados pelos operadores de mutação; e
6. Geração de dados quantitativos e qualitativos que permitam a evolução das ferramentas de mutação e análise estática utilizadas.

1.3 Organização do Texto

O restante deste trabalho encontra-se organizado da seguinte forma: no Capítulo 2 são apresentadas as terminologias e conceitos básicos para compreensão deste trabalho; no Capítulo 3 os passos empregados para coleta dos dados, os programas, as ferramentas e a hipótese estatística que será utilizada mais adiante para comparação das estratégias; no Capítulo 4 foi realizada a uma análise sobre os dados utilizados no presente trabalho; no Capítulo 5 são apresentados trabalhos relacionados e no Capítulo 6 são apresentadas a conclusão, perspectivas de trabalhos futuros e publicações.

Capítulo 2

FUNDAMENTEÇÃO TEÓRICA

Neste capítulo são apresentados alguns conceitos e terminologias relacionadas às atividades de Verificação e Validação ($V\&V$). São apresentados os conceitos sobre análise estática automatizada e seus tipos de analisadores, e a *Splint*. Finalmente são apresentados conceitos sobre o teste de mutação e a ferramenta *Proteum/IM*.

2.1 Conceitos Básicos de Verificação e Validação

Verificação e Validação ($V\&V$) abrangem muitas atividades de garantia de qualidade de software (PRESSMAN, 2011). Podem ser divididas em duas categorias: análise dinâmica e análise estática.

- **Análise Dinâmica:** é necessária a execução do código analisado, podendo assim ter um custo mais elevado do que a análise estática de acordo com as técnicas e critérios de testes adotados.
- **Análise Estática:** não é necessária a execução do código, são construídas representações abstratas dos comportamentos do código e assim a ferramenta analisa seus dados.

Partindo da perspectiva de custo, seguindo o raciocínio da pesquisa de Boehm e Basili (2001), quanto mais se demora para corrigir um defeito, mais caro se torna a sua correção. O uso da análise estática tem o lado positivo por não ser necessária a execução do código, tornado assim um processo de baixo custo. O lado negativo está nas representações abstratas construídas que acabam ocasionando avisos sobre defeitos inexistentes.

A ISO/IEC/IEEE (2017) define o engano, defeito, erro e falha da seguinte maneira:

- **Engano (*Mistake*)**: ação humana que produz um resultado incorreto.
- **Defeito (*Fault*)**: passo, processo ou definição de dados incorreto em um programa de computador.
- **Erro (*Error*)**: diferença entre valor obtido e o valor esperado.
- **Falha (*Failure*)**: inabilidade de um sistema ou componente em desempenhar as suas funções de acordo com os requisitos especificados.

2.2 Análise Estática Automatizada

Analisadores estáticos automatizados constroem uma representação abstrata do comportamento do programa analisado e examina seus dados e assim emitem avisos com base em suas regras. Segundo Sommerville (2007), os analisadores estáticos são ferramentas de software que vasculham o código fonte e detectam possíveis defeitos. Na Figura 2.1 é mostrado como é o processo de geração de avisos.

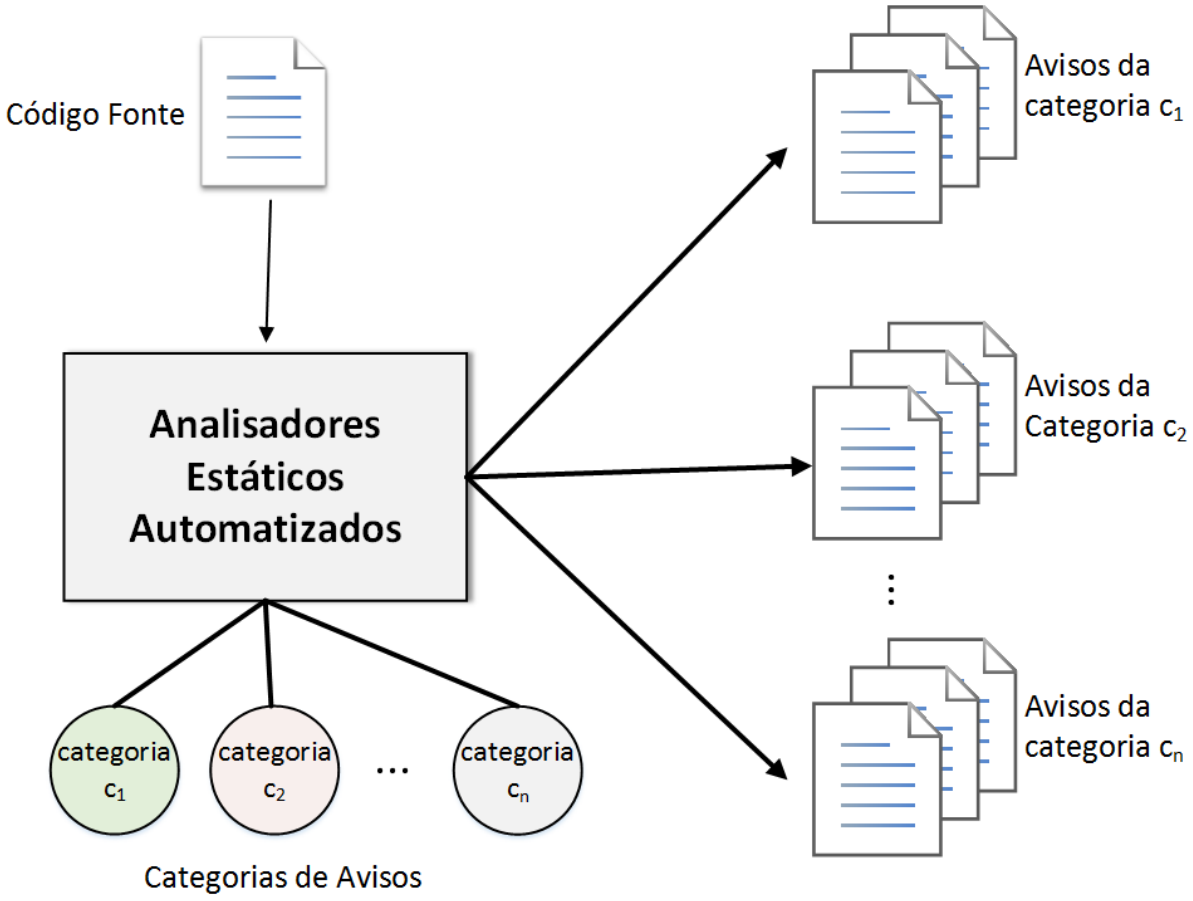

Figura 2.1: Processo de Geração de Avisos.

Os termos seguintes são utilizados por analisadores estáticos:

- **Falso positivo**: são avisos que são emitidos, mas não correspondem a defeitos reais;
- **Verdadeiro positivo**: são avisos emitidos e que realmente indicam a presença de um defeito real no código fonte;
- **Falso negativo**: ocorrem quando existem defeitos no código fonte, mas a ferramenta de análise estática não é capaz de detectar a presença do mesmo (ausência de regras de detecção).

Há várias ferramentas de análise estática disponíveis para diversas linguagens de programação, alguns exemplos dessas ferramentas são: a StyleCop (MICROSOFT, 2014) e a FxCop (KRESOWATY, 2008) para sistemas em .NET, a *FindBugs* (HOVEMEYER; PUGH, 2004) e a PMD (COPELAND, 2005) para Java, a Lint (POHL, 2001), *Splint* (EVANS; LAROCHELLE, 2002) e a Frama-C (CORRENSON et al., 2014b) para sistemas em C/C++.

A escolha da *Splint* como ferramenta de análise foi baseada no fato dela ser uma evolução da Ferramenta Lint (POHL, 2001), que é uma das ferramentas referência para Linguagem C, e no seu modo de operação, que tem como objetivo de reduzir o número de falsos positivos. A *Splint* adota a abordagem de analisar as anotações (comentários) do código fonte, assim ela tenta extrair o máximo de informações possíveis.

Um código quando carregado pelo sistema para montagem de uma representação abstrata, tudo que compõe o seu código é carregado, inclusive os comentários. A *Splint* adiciona os comentários à sua representação na árvore de execução para tentar compreender o código de maneira mais clara (EVANS; LAROCHELLE, 2002) e reduzir a incidência de falsos positivos.

2.2.1 Tipos de Analisadores Estáticos

Segundo Hovemeyer e Pugh (2004), os verificadores estáticos são classificados como de defeitos e de estilo. Os verificadores de defeitos identificam os defeitos mais comuns cometidos por desenvolvedores e que não são visíveis para os compiladores (TERRA; BIGONHA, 2008). Alguns exemplos de regras de detecção, segundo Terra e Bigonha (2008):

- Conexão com banco de dados não encerrada;
- Valor de retorno ignorado;
- Perda de referência;

- Concatenação de strings em laços de repetição;
- Fluxo não encerrado;
- Dentre outros.

Os verificadores de estilo identificam se um determinado código fonte está de acordo com determinada regra adotada para aquele projeto. Hovemeyer e Pugh (2004) considera que esse processo ajuda a prevenir certos tipos de defeitos, e que violar alguma regra de estilo não seja necessariamente um defeito real. Alguns exemplos de regras de estilos segundo Terra e Bigonha (2008):

- Instrução vazia;
- Padrão de nomenclatura;
- Ordem de declaração;
- Uso de chaves;
- Uso de referência *this*;
- Dentre outros.

2.2.2 Ferramenta de Análise Estática *Splint*

A *Splint* é uma das ferramentas de análise estática mais populares e é amplamente utilizada na comunidade da linguagem C. Ela verifica estaticamente vulnerabilidades de segurança e erros de programação. A verificação vai desde defeitos mais tradicionais como declarações não utilizadas até defeitos mais complexos a partir de informações complementares que a ferramenta aceita para obter melhores resultados.

A Versão 3.1.1 da *Splint* tem 487 tipos diferentes de avisos. Como exemplos de avisos relatados pela *Splint* podem ser citados, conforme apresentado na Tabela 2.1: o aviso *bufferoverflowhigh* (linha 34) relata um provável estouro de buffer; o aviso *fcnuse* (linha 121) relata uma função declarada, mas não usada; o aviso *infloops* (linha 183) relata um provável loop infinito.

Tabela 2.1: Detalhes de Avisos Relatados pela *Splint*

#	Aviso	Categoria(s)
1	abstract	abstract
2	abstractcompare	abstract
3	accessall	abstract + names
4	accessczech	abstract + names
5	accessczechoslovak	abstract + names
6	accessfile	abstract + names
7	accessmodule	abstract + names
8	accessslovak	abstract + names
9	aliasunique	aliasing + memory
10	allblock	controlflow
...	...	...
30	boundsread	memorybounds + memory
31	boundswrite	memorybounds + memory
32	branchstate	memory
33	bufferoverflow	warnuse + security
34	bufferoverflowhigh	warnuse + security
35	bugslimit	debug
36	casebreak	controlflow
37	caseinsensitivefilenames	headers + files
...	...	...
120	fcnpost	memorybounds + display
121	fcnuse	alluse
122	fielduse	alluse
123	fileextensions	help
124	filestaticprefix	names + prefixes
...	...	...
180	incondefs	declarations
181	incondefslib	declarations + libraries
182	indentspaces	format + display
183	infloops	controlflow
184	infloopsuncon	controlflow
...	...	...
480	warnsysfiles	display + files
481	warnunixlib	libraries + ansi
482	warnuse	warnuse
483	whichlib	libraries + initializations
484	whileblock	controlflow
485	whileempty	controlflow
486	whileloopexec	controlflow + memory
487	zerobool	typeequivalence + booleans

2.3 Teste de Mutação

O teste de mutação foi concebido baseado na hipótese do programador competente e no efeito de acoplamento (DEMILLO et al., 1978). A hipótese do programador competente parte

da premissa de que programadores experientes escrevem programas bem próximos de estarem corretos. Já o efeito de acoplamento parte da premissa de que defeitos mais complexos estão relacionados a defeitos mais simples.

Com base nessas hipóteses, defeitos artificiais são inseridos no programa original gerando os chamados mutantes, como pode ser visto na Figura 2.2. No teste de mutação tradicional, cada mutante contém apenas um defeito. As alterações sintáticas realizadas no programa original para a geração dos mutantes são feitas por meio dos chamados operadores de mutação (*op*), que simulam os defeitos mais frequentes introduzidos pelos programadores.

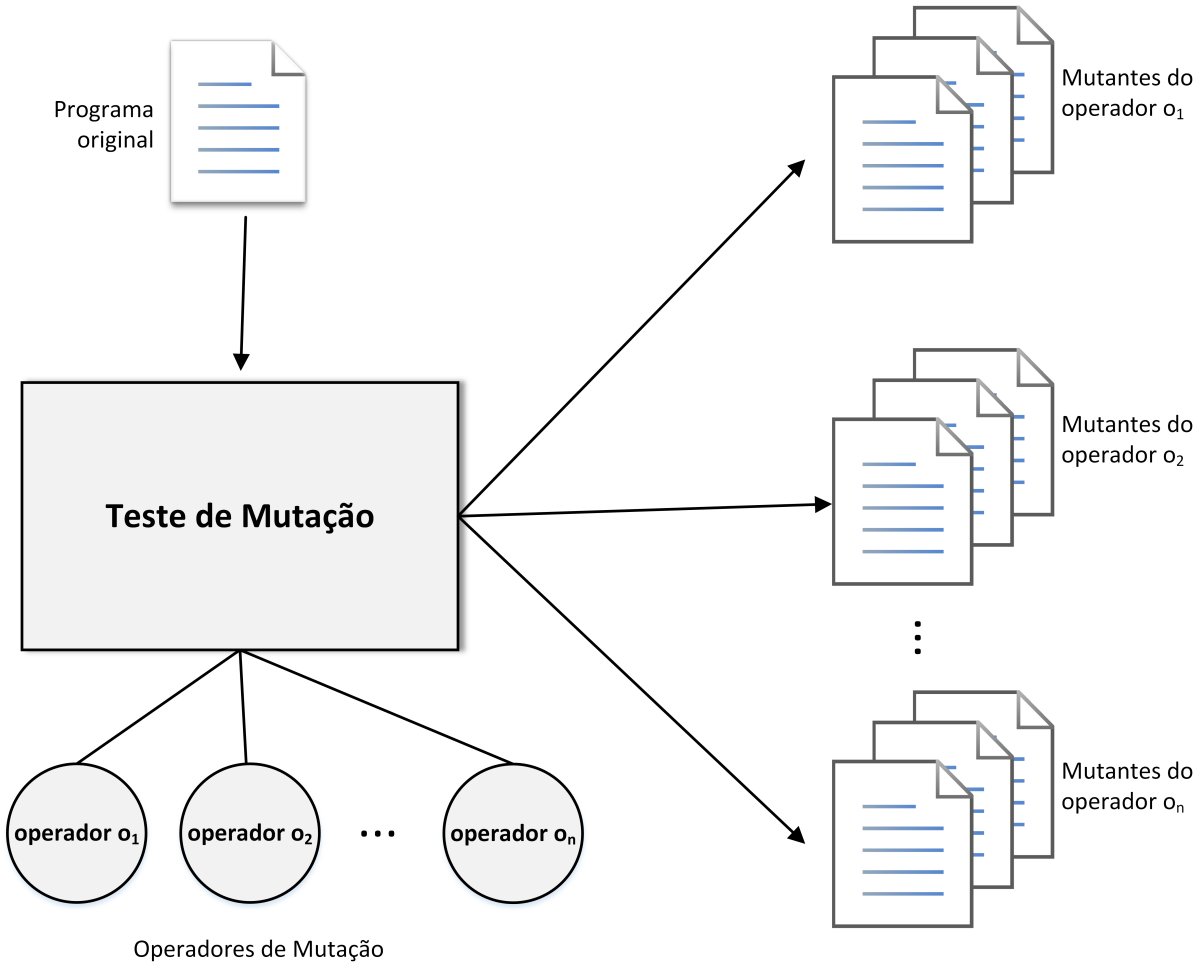

Figura 2.2: Processo de Geração de Mutantes.

Para que um *op* gere mutantes é necessário que o programa original contenha as estruturas sintáticas exigidas pelo operador para criar os mutantes. Por exemplo, o operador ORRN (Relational Operator Mutation) substitui a ocorrência de um operador relacional no programa original por todos os outros operadores relacionais existentes na linguagem C. Já o operador SSDL (Statement Deletion) remove um comando de cada vez do programa original para dar origem aos mutantes. Desse modo, o número de mutantes gerados com a aplicação de um ope-

rador varia em função da presença das estruturas sintáticas e do número de vezes que essas estruturas aparecem no programa em teste.

Inicialmente, considerando um conjunto de teste T e um programa P, cada caso de teste $t \in T$ é executado em P e observa-se se o resultado obtido condiz com o resultado esperado. Caso os resultados divirjam, uma falha foi exposta e P deve ser corrigida para a eliminação do defeito que levou à ocorrência da falha.

Entretanto, se nenhuma falha for observada não significa que, necessariamente, P não tenha defeitos. Pode ser que o conjunto de teste T seja de baixa qualidade. Para avaliar a qualidade de T pode ser utilizado o Teste de Mutação.

A execução do Teste de Mutação se inicia quando um ou mais operadores são aplicados à P, gerando o conjunto M de mutantes de P. Cada mutante $m \in M$ difere de P apenas em um defeito simples. Em seguida, cada caso de teste $t \in T$ é executado com cada mutante m e o resultado é comparado com o resultado obtido quando t foi executado com P. Se os resultados forem os mesmos, é dito que o mutante permanece vivo, do contrário é dito que o mutante é morto.

Um mutante ser morto significa simplesmente que o caso de teste em questão foi capaz de expor a diferença de comportamento entre ele e o programa original e, nesse caso, confirma que o programa original não contém o defeito modelado pelo mutante. Já os mutantes vivos precisam ser analisados. Pode ser que um mutante m esteja vivo porque T é de baixa qualidade e ainda não possui um teste capaz de matá-lo, ou então, m pode estar vivo por ser equivalente à P e, nesse caso, pode ser descartado.

Do ponto de vista de melhoria do conjunto de teste T o que mais interessa são os mutantes vivos não equivalentes pois são esses que contribuem para adicionar a T novos casos de teste.

Por meio do cálculo do escore de mutação, que é a razão entre o número de mutantes mortos e o número de mutantes não-equivalentes (do total de mutantes criados desconsidera-se o número de mutantes equivalentes), avalia-se qual a qualidade de T para o teste de P considerando o conjunto de mutantes gerados. O valor obtido do escore de mutação varia entre 0 e 1, sendo que quanto mais próximo de 1, melhor a qualidade de T no teste de P. Em outras palavras, se T obtiver um escore de 1, significa que ele possui casos de teste capaz de expor todas as possíveis falhas modeladas pelos operadores de mutação utilizados. O escore de mutação é calculado da seguinte forma:

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

Sendo:

$DM(P,T)$: total de mutantes mortos pelo conjunto de casos de teste T;

$M(P)$ total de mutantes gerados a partir do programa P;

$EM(P)$: total de mutantes equivalentes ao programa P.

Do ponto de vista teórico, cada mutante representa um possível defeito que pode estar presente no programa original (DELAMARO et al., 2016). Em estudo realizado por (ANDREWS et al., 2005) demonstrou-se que o teste de mutação pode ser considerado um excelente modelo de falhas e é empregado frequentemente na avaliação da qualidade de conjuntos e critérios de teste.

Na Tabela 2.2 são apresentados exemplos de mutantes de acordo com o seu respectivo trecho de código original. Pode-se notar que na Tabela 2.2, diferentes mutações podem ser realizadas simulando diferentes tipos de defeitos. O ponto de mutação aparece sublinhado na Tabela 2.2. Por exemplo, no item 2 o operador de mutação troca um incremento/decremento prefixo ao invés do incremento/decremento posfixo, para cada alteração que ele realiza é criado um mutante, nesse exemplo mostramos a troca do incremento posfixo pelo prefixo: (no original) `j++;` por (no mutante) `++j;`.

Tabela 2.2: Exemplos de Mutantes (adaptado de (DELAMARO et al., 2000); (DELAMARO et al., 2014))

id	Operador	Trecho de Código Original	Trecho de Código Mutante	Função

1	u-ORRN	`if(i <u>≤</u> lp){`
		`  goto teste;`
		`}`
			`if(i <u>≥</u> lp){`
			`  goto teste;`
			`}`
				Substitui um operador relacional por outro operador relacional.

2	u-OPPO	`if(nfiles <= eargv){`
		`  <u>j++</u>;`
		`}`
			`if(nfiles <= eargv){`
			`  <u>++j</u>;`
			`}`
				Usa o incremento/decremento prefixo ao invés de incremento/decremento posfixo.

3	u-OCNG	`if(argc == 2){`
		`  goto xlong;`
		`}`
			`if(<u>!</u>(argc == 2)){`
			`  goto xlong;`
			`}`
				Inverte condicionais.

2.3.1 Ferramenta de Mutação *Proteum/IM*

Existem várias ferramentas de mutação disponíveis para sistemas em C (JIA; HARMAN, 2008); (DAN; HIERONS, 2012). A ferramenta utilizada nesse experimento foi a *Proteum/IM*, que suporta Teste de Mutação para sistemas em C (DELAMARO et al., 2000).

A ferramenta *Proteum/IM* analisa o programa original e aplica os operadores de mutação escolhidos pelo testador de acordo com os seus 78 operadores de mutação especializados em representarem defeitos de unidade que são apresentados na Tabela 2.3 (DELAMARO et al., 2000, 2014).

Os operadores de mutação de unidade são divididos em 4 grupos: Mutação de Constante, Mutação de Operador, Mutação de Comandos e Mutação de Variáveis, tendo 4, 47, 15 e 12 operadores cada um respectivamente, os itens podem ser observados na Tabela 2.3.

Para mais informações sobre o conjunto de operadores de mutação da *Proteum/IM* e exemplos de mutações que eles realizam, pode-se consultar os trabalhos de Delamaro et al. (2000, 2014).

2.4 Considerações Finais

Nesse capítulo foram apresentados os principais conceitos relacionados de Verificação e Validação ($V\&V$), focando nos aspectos do teste de mutação de análise estática automatizada e apresentando a ferramenta de mutação *Proteum/IM* e a ferramenta de análise estática *Splint* que serão utilizadas no presente trabalho. No Capítulo 3 são apresentados os programas, ferramentas e passos empregados para coleta de dados e o resultado gerado.

Tabela 2.3: Operadores de Mutação de Unidade da *Proteum/IM* (adaptado de (DELAMARO et al., 2014))

Característica da Linguagem	Operador de Mutação	Descrição
Mutação de Constante (4)	u-Cccr	Constant for Constant Replacement
	u-CCDL	Constant Deletion
	u-Ccsr	Constant for Scalar Replacement
	u-CRCR	Required Constant Replacement
Mutação de Operador (47)	u-OAAA	Arithmetic Assignment Mutation
	u-OAAN	Arithmetic Operator Mutation
	u-OABA	Arithmetic Assignment by Bitwise Assignment
	u-OABN	Arithmetic by Bitwise Operator
	u-OAEA	Arithmetic Assignment by Plain Assignment
	u-OALN	Arithmetic Operator by Logical Operator
	u-OARN	Arithmetic Operator by Relational Operator
	u-OASA	Arithmetic Assignment by Shift Assignment
	u-OASN	Arithmetic Operator by Shift Operator
	u-OBAA	Bitwise Assignment by Arithmetic Assignment
	u-OBAN	Bitwise Operator by Arithmetic Assignment
	u-OBBA	Bitwise Assignment Mutation
	u-OBBN	Bitwise Operator Mutation
	u-OBEA	Bitwise Assignment by Plain Assignment
	u-OBLN	Bitwise Operator by Logical Operator
	u-OBNG	Bitwise Negation
	u-OBRN	Bitwise Operator by Relational Operator
	u-OBSA	Bitwise Assignment by Shift Assignment
	u-OBSN	Bitwise Operator by Shift Operator
	u-OCNG	Logical Context Negation
	u-OCOR	Cast Operator by Cast Operator
	u-OEAA	Plain assignment by Arithmetic Assignment
	u-OEBA	Plain assignment by Bitwise Assignment
	u-OESA	Plain assignment by Shift Assignment
	u-Oido	Increment/Decrement Mutation
	u-OIPM	Indirection Operator Precedence Mutation
	u-OLAN	Logical Operator by Arithmetic Operator
	u-OLBN	Logical Operator by Bitwise Operator
	u-OLLN	Logical Operator Mutation
	u-OLNG	Logical Negation
	u-OLRN	Logical Operator by Relational Operator
	u-OLSN	Logical Operator by Shift Operator
	u-OODL	Operator Deletion
	u-ORAN	Relational Operator by Arithmetic Operator
	u-ORBN	Relational Operator by Bitwise Operator
	u-ORLN	Relational Operator by Logical Operator
	u-ORRN	Relational Operator Mutation
	u-ORSN	Relational Operator by Shift Operator
	u-OSAA	Shift Assignment by Arithmetic Assignment
	u-OSAN	Shift Operator by Arithmetic Operator
	u-OSBA	Shift Assignment by Bitwise Assignment
	u-OSBN	Shift Operator by Bitwise Operator
	u-OSEA	Shift Assignment by Plain Assignment
	u-OSLN	Shift Operator by Logical Operator
	u-OSRN	Shift Operator by Relational Operator
	u-OSSA	Shift Assignment Mutation
	u-OSSN	Shift Operator Mutation
Mutação de Comandos(15)	u-SBRC	break Replacement by continue
	u-SBRn	break Out to Nth Level
	u-SCRB	continue Replacement by break
	u-SCRn	continue Out to Nth Level
	u-SDWD	do-while Replacement by while
	u-SGLR	goto Label Replacement
	u-SMTC	n-trip continue
	u-SMTT	n-trip trap
	u-SMVB	Move Brace Up and Down
	u-SRSR	return Replacement
	u-SSDL	Statement Deletion
	u-SSWM	switch Statement Mutation
	u-STRI	Trap on if Condition
	u-STRP	Trap on Statement Execution
	u-SWDD	while Replacement by do-while
Mutação de Variáveis(12)	u-VDTR	Domain Traps
	u-VGAR	Mutate Global Array References
	u-VGPR	Mutate Global Pointer References
	u-VGSR	Mutate Global Scalar References
	u-VGTR	Mutate Global Structure References
	u-VLAR	Mutate Local Array References
	u-VLPR	Mutate Local Pointer References
	u-VLSR	Mutate Local Scalar References
	u-VLTR	Mutate Local Structure References
	u-VSCR	Stucture Component Replacement
	u-VTWD	Twiddle Mutations
	u-VVDL	Variable Deletion

Capítulo 3

ESTUDO EXPERIMENTAL - CDL POR AVISO E POR OPERADOR

Neste capítulo são apresentados os passos empregados para coleta dos dados, os programas, as ferramentas e a hipótese estatística que será utilizada mais adiante para comparação das estratégias.

3.1 Estrutura da Experimentação

Nessa seção são definidos os conceitos apresentados por (ARAúJO, 2015) adaptados para o contexto da ferramenta de análise estática *Splint* e da ferramenta de mutação *Proteum/IM*.

As definições apresentadas a seguir foram definidas no trabalho de (ARAúJO, 2015).

Um conjunto S é composto por t sistemas, tal que $S = \{s_1, s_2, \ldots, s_t\}$ e $s_x \in S$ é um sistema sob análise. Cada sistema s_x é composto por um conjunto de arquivos f, tal que $F = \{f_{1_x}, f_{2_x}, \ldots, f_{n_x}\}$. A execução da *Splint* sobre cada arquivo f_j permite definir os conjuntos a seguir:

- Um conjunto de avisos de um arquivo f_j é representado por Wf_j.
- Um sistema s_x possui um conjunto $Ws_x = \{Wf_1 \cup Wf_2 \cup \ldots \cup Wf_n\}$, onde Ws_x é a união de todos conjuntos de avisos Wf_j.

A ferramenta *Proteum/IM* gera um conjunto de mutantes, cada mutante é definido da seguinte maneira: $m_i o_k f_j$, sendo o i-ésimo mutante gerado pelo operador o_k sobre o arquivo f_j.

Considerando $Wm_io_kf_j$ sendo o conjunto de avisos relatados no mutante $m_io_kf_j$, pode-se estabelecer uma relação de correspondência entre tipo de aviso e tipo de operador de mutação.

O processo utilizado neste experimento é definido da seguinte maneira:

1. Dado um conjunto S de sistemas, onde $S = \{s_1, s_2, \ldots, s_t\}$;
2. Para cada sistema $s_x \in S, 1 \leq x \leq t$;
   (a) A *Splint* é executada no arquivo $f_j \in s_x$, gerando assim o arquivo Wf_j contendo os avisos relatados no arquivo original f_j.
   (b) O arquivo Wf_j é preprocessado para que cada aviso ocupe apenas uma linha.
   (c) É feita a execução do *ParserSplint* para analisar e coletar os dados do arquivo Wf_j preprocessado, é gerado um arquivo *XML* contendo tais informações.
   (d) A *Proteum/IM* é executada em cada f_j e gera os mutantes $m_io_kf_j$.
   (e) Para cada mutante $m_io_kf_j \in Mf_j$:
      i. A *Splint* é executada no mutante $m_io_kf_j$ e gerando assim o arquivo $Wm_io_kf_j$ contendo os avisos relatados no arquivo mutante correspondente.
      ii. O arquivo $Wm_io_kf_j$ é preprocessado para que cada aviso ocupe apenas uma linha.
      iii. É gerado um arquivo contendo a diferença entre o arquivo original f_j e o arquivo de mutação $m_io_kf_j$ (*diff*).
      iv. É feita a execução do *ParserSplint* para analisar e coletar os dados dos arquivos $Wm_io_kf_j$ preprocessado e o arquivo de diferença gerado, é gerado um arquivo *XML* contendo tais informações.
3. Os dados de todos *XMLs* são salvos no banco de dados.
4. A informação sobre correspondência direta entre avisos estáticos e mutações é realizado por meio de *scripts SQL*[1].

Na Figura 3.1 é ilustrado o processo de coleta dos dados.

[1]Por meio de *scripts SQL* outras informações, tais como, quantidade de avisos, quantidade de mutantes, linhas de ocorrências de avisos e mutantes, dentre outras, também podem ser obtidas.

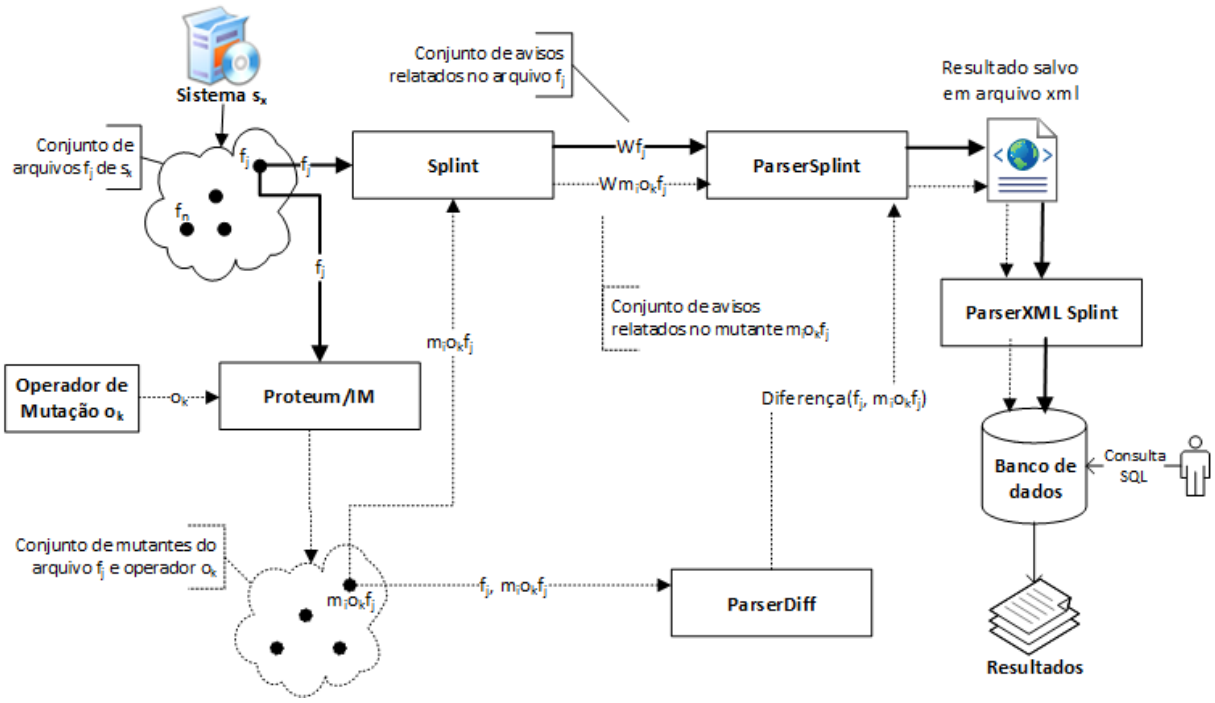

Figura 3.1: Processo de Coleta dos dados adaptado de (ARAúJO, 2015).

3.2 Sistemas Utilizados

Selecionaram-se para a condução do trabalho quarenta e dois programas, sendo que alguns fazem parte do conjunto de programas utilitários do UNIX, alguns outros da Siemens disponíveis no repositório Sir (DO et al., 2005), e alguns outros estão disponíveis no livro Introdução ao Teste de Software (AMMANN; OFFUTT, 2008) e tais programas já foram utilizados por outros pesquisadores (WONG, 1993; DELAMARO, 1997b; WONG et al., 1997; BARBOSA et al., 2001; VINCENZI, 1998). Além disso, devido ao seu uso intenso, tais programas possuem uma baixa probabilidade de apresentarem defeitos reais em seu código. Na Tabela 3.1 são apresentadas mais informações sobre os mesmos.

Por exemplo, o Programa Cal é o utilitário do UNIX responsável pela geração de calendário via linha de comando. Este programa possui 202 linhas de código (LOC), a Ferramenta *Splint* configurada para utilizar todos os tipos de avisos possíveis, emitiu 129 avisos durante sua execução, que dá uma taxa de 0,639 avisos por linha de código (W/LOC). Já a Ferramenta *Proteum/IM* configurada para utilizar todos os seus operadores de mutação para o teste de unidade, gerou 4.881 mutantes, que corresponde a uma taxa de 24,163 mutantes por linha de código (M/LOC).

O programa TotInfo é um programa da Siemens do repositório Sir, ele realiza calculo estatístico dado um conjunto de entrada. Este programa possui 409 linhas de código (LOC), a

Ferramenta *Splint* configurada para utilizar todos os tipos de avisos possíveis, emitiu 90 avisos durante sua execução, que dá uma taxa de 0,22 avisos por linha de código (W/LOC). Já a Ferramenta *Proteum/IM* configurada para utilizar todos os seus operadores de mutação para o teste de unidade, gerou 6.698 mutantes, que corresponde a uma taxa de 16,377 mutantes por linha de código (M/LOC).

O programa MergeSort é um programa que está disponível no livro Introdução ao Teste de Software (AMMANN; OFFUTT, 2008) e que realiza a ordenação baseado no conceito de dividir e conquistar através da recursividade. Este programa possui 76 linhas de código (LOC), a Ferramenta *Splint* configurada para utilizar todos os tipos de avisos possíveis, emitiu 37 avisos durante sua execução, que dá uma taxa de 0,487 avisos por linha de código (W/LOC). Já a Ferramenta *Proteum/IM* configurada para utilizar todos os seus operadores de mutação para o teste de unidade, gerou 991 mutantes, que corresponde a uma taxa de 13.039 mutantes por linha de código (M/LOC).

No total, os quarenta e dois programas somam 6.160 linhas de código, 2.164 avisos e 80.124 mutantes, conforme a Tabela 3.1, representando uma taxa de avisos por linha de código de 0,351 e taxa de mutante por linha de código de 13,007. Destaca-se ainda que, do total de mutantes gerados, 80.124, 2.654 mutantes (3,312%) são equivalentes e foram identificados de forma manual.

Tanto os conjuntos de teste adequados quanto os equivalentes foram obtidos a partir de trabalhos anteriores, que realizaram experimentos com os mesmos programas (VINCENZI, 1998; DO et al., 2005; DELAMARO et al., 2018). Nesses trabalhos, a análise dos equivalentes foi feita de forma manual. Ressalta-se, entretanto que, para a estratégia em questão, não é necessária a determinação dos equivalentes. Tais dados foram utilizados neste trabalho para a comparação de custo não apenas em termos do número de mutantes gerados, mas também em termos do número de equivalentes.

3.3 CDL por Aviso

Para se determinar a correspondência entre mutantes e avisos empregou-se a Correspondência Direta por Linha (*CDL*) por aviso definida em (ARAúJO et al., 2015, 2016). O objetivo da CDL é quantificar o número de avisos emitidos em virtude do ponto de mutação criado, ou seja, a *CDL* representa a quantidade de avisos emitidos no ponto de mutação e que não estava presente no programa original. O cálculo do *CDL* é realizado da seguinte maneira segundo (Araújo et al., 2015):

Tabela 3.1: Sistemas Utilizados no Estudo Experimental

Sistema	LOC	Avisos (W)	Mutantes (M)	W/LOC	M/LOC
cal	202	129	4.881	0,64	24,16
checkeq	102	108	4.492	1,06	44,04
comm	169	134	1.966	0,79	11,63
look	171	33	2.077	0,19	12,15
uniq	143	104	1.723	0,73	12,05
print_tokens	731	0	4.320	0,00	5,91
print_tokens2	482	2	4.719	0,00	9,79
replace	513	117	11.101	0,23	21,64
schedule	386	0	2.109	0,00	5,46
schedule2	297	211	2.627	0,71	8,85
tcas	164	149	2.384	0,91	14,54
totinfo	409	90	6.698	0,22	16,38
boundedQueue	85	43	1.121	0,51	13,19
Calculation	120	37	1.118	0,31	9,32
checkIt	18	9	104	0,50	5,78
CheckPalindrome	30	14	166	0,47	5,53
countPositive	31	8	151	0,26	4,87
date-plus	201	194	2.421	0,97	12,04
DigitReverser	32	12	496	0,38	15,50
findLast	40	11	198	0,28	4,95
findVal	26	12	190	0,46	7,31
Gaussian	92	35	1.086	0,38	11,80
Heap	95	61	1.079	0,64	11,36
InversePermutation	36	25	576	0,69	16,00
jday-jdate	87	23	2.821	0,26	32,43
lastZero	33	5	173	0,15	5,24
LRS	102	41	1.132	0,40	11,10
MergeSort	76	37	991	0,49	13,04
numZero	47	9	151	0,19	3,21
oddOrPos	38	5	361	0,13	9,50
pcal	405	249	6.419	0,61	15,85
power	33	4	268	0,12	8,12
printPrimes	69	15	715	0,22	10,36
Queue	96	6	469	0,06	4,89
quicksort	57	20	1.026	0,35	18,00
RecursiveSelectionSort	46	22	555	0,48	12,07
Stack	101	0	461	0,00	4,56
stats	48	22	884	0,46	18,42
sum	31	7	165	0,23	5,32
testPad	69	20	629	0,29	9,12
twoPred	25	10	246	0,40	9,84
UnixCal	222	131	4.855	0,59	21,87
Total	6.160	2.164	80.124	0,35	13,01

- $TW(w)$ = quantidade total de avisos do tipo w relatados em todos os mutantes (independemente do tipo de operador)

- $CDL_A(w)$ = quantidade absoluta de avisos do tipo w que são relatados exatamente no ponto de mutação, mas que tal w não exista na mesma linha no arquivo original correspondente.

- $CDL_R(w) = CDL_A(w)/TW(w)$. O valor de $CDL_R(w)$, que varia entre 0 e 1 (100%), representa a efetividade do tipo de aviso w em perceber os pontos de mutações. O valor final da taxa $CDL_R(w)$ se baseia nas informações geradas em conjunto pelo teste de mutação que é uma técnica bastante eficaz na detecção e eliminação de defeitos e no analisador estático, com isso, quanto mais próximo de 1, maior a chance do aviso w ser um aviso verdadeiro positivo. De forma simétrica, quanto mais próximo de zero, menor a chance do aviso w ser um aviso verdadeiro positivo.

A definição de tais valores para $TW(w)$, $CDL_A(w)$ e $CDL_R(w)$ visa determinar uma classificação para os avisos w de acordo com a capacidade de cada um em relatar os defeitos gerados pelos mutantes. A classificação de priorização é determinada da seguinte maneira: quanto maior for o $CDL_R(w)$, maior a chance dele ser um verdadeiro positivo.

Tabela 3.2: Correspondência Direta por Linha por Aviso

id	**Aviso** w	**Operadores de Mutação**					$TW(w)$	$CDL_A(w)$	$CDL_R(w)(\approx$ **%)**
		u-Oido	**u-OBSA**	**u-Cccr**	**u-VDTR**	⋯			
1º	bitwisesigned	0	0	0	0	⋯	3.007	3.007	100,00
2º	shiftimplementation	0	0	0	0	⋯	1.982	1.982	100,00
3º	strictops	0	0	46	0	⋯	1.230	1.230	100,00
4º	shiftnegative	0	0	0	0	⋯	1.160	1.160	100,00
5º	charindex	0	0	112	0	⋯	308	308	100,00
6º	ifempty	0	0	0	0	⋯	179	179	100,00
7º	boolcompare	0	0	0	0	⋯	12	12	100,00
8º	noeffect	0	0	0	0	⋯	7	7	100,00
9º	ptrcompare	0	0	0	0	⋯	6	6	100,00
10º	looploopcontinue	0	0	0	0	⋯	3	3	100,00
11º	temptrans	0	0	0	0	⋯	1	1	100,00
12º	onlytrans	0	0	0	0	⋯	1	1	100,00
13º	likelyboundswrite	0	0	79	0	⋯	108	98	90,74
14º	infloops	0	0	0	0	⋯	187	113	60,43
15º	likelyboundsread	0	0	0	0	⋯	28	16	57,14
16º	unreachable	0	0	0	0	⋯	931	323	34,69
17º	formattype	3	0	14	87	⋯	1.226	386	31,48
18º	sysunrecog	48	8	514	597	⋯	12.856	2.366	18,40
19º	usereleased	0	0	0	0	⋯	33	5	15,15
20º	evalorder	0	0	31	117	⋯	1.308	154	11,77
21º	usedef	126	6	1.998	2.625	⋯	46.382	3.490	7,52
22º	boolops	186	8	3.416	2.865	⋯	50.814	3.094	6,09
23º	unrecog	558	40	14.655	11.487	⋯	153.986	9.175	5,96
24º	emptyret	94	4	1.159	342	⋯	9.883	565	5,72
25º	predboolint	239	4	7.519	3.189	⋯	68.430	2.646	3,87
…	…	…	…	…	…	⋱	…	…	…
66º	dependenttrans	27	0	1.076	267	⋯	6.459	3	0,05
67º	retvalint	186	8	3.100	2.034	⋯	37.759	3	0,01
68º	branchstate	22	0	130	540	⋯	6.199	2	0,03
69º	stringliteralnoroom	18	0	279	72	⋯	1.980	0	0,00
70º	exportheader	907	10	21.469	14.646	⋯	251.981	0	0,00
71º	declundef	60	0	880	372	⋯	9.740	0	0,00
72º	cppnames	0	0	278	111	⋯	2.975	0	0,00
73º	casebreak	0	0	0	0	⋯	92	0	0,00
74º	bufferoverflowhigh	24	0	465	396	⋯	6.566	0	0,00
75º	formatconst	68	2	1.091	855	⋯	14.843	0	0,00
76º	nestedextern	18	8	222	204	⋯	4.126	0	0,00
77º	mustfreeonly	0	0	0	0	⋯	3	0	0,00
78º	oldstyle	858	50	30.033	20.280	⋯	349.518	0	0,00
79º	maintype	92	0	875	1.374	⋯	18.462	0	0,00
80º	paramuse	13	0	120	171	⋯	4.543	0	0,00
81º	predassign	7	0	100	177	⋯	2.661	0	0,00
82º	isoreservedinternal	80	12	1.866	1.518	⋯	25.301	0	0,00
83º	protoparamname	17	0	26	102	⋯	1.132	0	0,00
84º	incondefs	15	0	502	264	⋯	5.544	0	0,00
85º	imptype	18	8	333	306	⋯	4.634	0	0,00
86º	readonlytrans	336	4	10.492	4.458	⋯	85.944	0	0,00
87º	noparams	185	8	5.689	4.845	⋯	84.629	0	0,00
88º	shadow	48	0	930	792	⋯	13.150	0	0,00
89º	fcnuse	390	8	11.656	5.274	⋯	101.973	0	0,00
90º	exportlocal	1.169	20	33.301	17.037	⋯	325.135	0	0,00
91º	exportheadervar	503	10	18.025	6.333	⋯	143.195	0	0,00
Total		25.351	604	779.559	459.693	⋯	8.132.121	53.407	0,66

A Tabela 3.2 apresenta os dados em ordem decrescente definida pelas colunas $CDL_R(w)$ e $CDL_A(w)$, ou seja, as primeiras linhas dessa tabela exibem os avisos da *Splint* que mais detectaram as mutações geradas pela *Proteum/IM*; análogo a isso, as últimas linhas exibem os avisos que menos detectaram mutações.

Temos que dos 91 tipos de avisos relatados pela *Splint* para esse conjunto de sistemas, 68 tipos avisos emitiram pelo menos um aviso na mesma linha de mutação como mostrado na Tabela 3.2.

Na Tabela 3.2 o total de avisos relatados nos mutantes de acordo com cada operador foi de 8.132.121, sendo que 53.407 avisos foram relatados exatamente no ponto de mutação, ou seja, a taxa $CDL_R(w)$ foi de $0,66\%$.

Mesmo com o $CDL_R(w)$ geral apresentado sendo baixo, temos que 15 tipos de avisos tiveram o $CDL_R(w)$ com mais de 40% de correspondência. Sendo que 12 possuem taxa de $CDL_R(w)$ igual a 100%. Ressalta-se que os avisos emitidos pelos 12 primeiros tipos de avisos estão sempre relacionados a uma mutação, minimizando a chance do avisos corresponder a um falso positivo.

Obtivemos um conjunto de 23 avisos em que o $CDL_R(w)$ foi igual a 0%, ou seja, tais avisos não tiveram sucesso na detecção dos mutantes. Esses resultados não determinam o grau de correção de um aviso ou se ele é bom ou ruim em identificar defeitos, mas que ela possui certa dificuldade na detecção de determinados tipos de defeitos.

Um estudo posterior pode ser feito aproveitando tal fato, podendo assim aprimorar a ferramenta de análise estática com o objetivo de torná-la mais eficiente na detecção de diferentes tipos de defeitos, sejam eles mutantes ou não.

Os dados apresentados na Tabela 3.2 tem como um dos seus objetivos determinar uma abordagem incremental para a análise de avisos onde seria seguida uma ordem decrescente da taxa $CDL_R(w)$ obtida de cada aviso w, assim seriam analisados primeiro os avisos com as maiores taxas, isto é, aqueles que representariam verdadeiros positivos.

3.4 CDL por Operador

Para se determinar a correspondência entre avisos e mutantes empregou-se a Correspondência Direta por Linha (CDL) por operador definida em (ARAúJO et al., 2015, 2016). O objetivo da CDL é quantificar o número de mutantes identificados pela *Splint*, ou seja, CDL é a quantidade

de avisos emitida no mutante, mas que não estão presentes no programa original. Uma vez computada a CDL de cada mutante é possível determinar a CDL de cada operador de mutação.

A CDL é obtida da seguinte maneira, segundo Araújo et al. (2015, 2016):

- $TM(o_k)$ – quantidade total de mutantes gerados pelo operador o_k;
- $CDL_A(o_k)$ – quantidade absoluta de mutantes gerados pelo operador o_k com pelo menos um aviso relatado no ponto de mutação, mas que o mesmo aviso não exista, na mesma linha, no arquivo original;
- $CDL_R(o_k) = CDL_A(o_k)/TM(o_k)$ – quantidade relativa da $CDL_A(o_k)$ sobre $TM(o_k)$. O valor de $CDL_R(o_k)$, que varia entre 0 e 1 (100%), representa a probabilidade das mutações produzidas pelo operador o_k de serem detectadas pela ferramenta de análise estática.

A definição dos valores de $TM(o_k)$, $CDL_A(o_k)$ e $CDL_R(o_k)$ permitem classificar os operadores de mutação de acordo com a capacidade que o analisador estático tem em detectá-lo. A classificação representa, teoricamente, que os operadores de mutação com maiores taxas $CDL_R(o_k)$, modelam defeitos mais fáceis de serem detectados pela *Splint*. Por outro lado, operadores de mutação com menores taxas $CDL_R(o_k)$ modelam os defeitos mais difíceis de serem detectados pelo analisador estático.

Na Tabela 3.3 são apresentados os dados dos operadores da *Proteum/IM* em ordem crescente pelo valor CDL_R, ou seja, dos operadores que geram os mutantes mais difíceis de serem percebidos pela *Splint* para os mais fáceis.

Dos 78 operadores de unidade da *Proteum/IM*, 15 não geraram mutantes para os programas utilizados. São eles, u-OBAA, u-OBBA, u-OBEA, u-OSAA, u-OSAN, u-OSBA, u-OSBN, u-OSEA, u-OSLN, u-OSRN, u-OSSA, u-OSSN, u-SCRn, u-VGTR, u-VLTR. Nesse caso, a geração de mutantes não ocorreu pois os programas não possuem as estruturas sintáticas exigidas pelos operadores. Para mais informações sobre os operadores de mutação da *Proteum/IM*, o leitor interessado pode consultar (DELAMARO et al., 2000).

Dos 63 operadores de mutação que geraram ao menos um mutante para os programas utilizados, pode-se observar na Tabela 3.3 que os operadores 62º e o 63º obtiveram uma taxa CDL_R de 100%, ou seja, todos os defeitos modelados por tais operadores foram identificados pelo analisador estático automatizado. O u-OIPM (62º operador) tem como objetivo de revelar defeitos no uso incorreto de expressões que envolvam ++, --, e operadores de indireção *, e o u-SBRn (63º operador) tem como objetivo trocar comandos break ou continue, alterando o comportamento do laço.

Tabela 3.3: Dados agregados da correspondência dos avisos da *Splint* com operadores unitários da *Proteum/IM*

E5	E27	STAT	Operador	CDL_R (≈%)	#Mut	#Eq	Escore	CC	CCEq	RC(≈%)	RCEq(≈%)
		1º	u-SGLR	0,0	10	0	0,08645	10	0	100,0	100,0
		2º	u-OAAA	0,9	224	1	0,44957	234	1	99,7	100,0
	1º	3º	u-SWDD	3,4	61	8	0,79564	295	9	99,6	99,7
		4º	u-Oido	6,6	257	2	0,91266	552	11	99,3	99,6
		5º	u-OBAN	6,7	75	5	0,91363	627	16	99,2	99,4
		6º	u-OBBN	6,7	30	2	0,91363	657	18	99,2	99,3
		7º	u-OBNG	6,7	45	2	0,91456	702	20	99,1	99,3
		8º	u-OBSN	6,7	30	2	0,91456	732	22	99,1	99,2
		9º	u-OBSA	7,1	28	28	0,91456	760	50	99,1	98,2
4º	6º	10º	u-ORRN	7,5	2.585	132	0,98256	3.345	182	95,8	93,3
		11º	u-OCNG	9,4	529	1	0,98323	3.874	183	95,2	93,2
		12º	u-OAAN	10,4	1.319	22	0,98568	5.193	205	93,5	92,4
		13º	u-OLLN	12,2	135	0	0,98900	5.328	205	93,4	92,4
		14º	u-OLNG	12,5	405	4	0,98923	5.733	209	92,8	92,3
		15º	u-OBLN	13,3	30	2	0,98923	5.763	211	92,8	92,2
		16º	u-OBRN	13,3	90	3	0,98923	5.853	214	92,7	92,1
		17º	u-OCOR	13,5	148	20	0,98923	6.001	234	92,5	91,3
2º	3º	18º	u-SSDL	14,6	2.494	45	0,99330	8.495	279	89,4	89,7
		19º	u-OAEA	15,0	60	0	0,99330	8.555	279	89,3	89,7
		20º	u-VLAR	17,3	106	5	0,99344	8.661	284	89,2	89,5
		21º	u-SBRC	20,0	15	3	0,99344	8.676	287	89,2	89,4
		22º	u-VSCR	20,1	366	0	0,99345	9.042	287	88,7	89,4
		23º	u-OLRN	22,3	810	33	0,99597	9.852	320	87,7	88,2
	9º	24º	u-Cccr	23,3	6.938	275	0,99615	16.790	595	79,0	78,0
		25º	u-SMVB	24,3	84	6	0,99615	16.874	601	78,9	77,8
		26º	u-VLPR	24,7	323	1	0,99615	17.197	602	78,5	77,7
	10º	27º	u-Ccsr	25,4	8.110	23	0,99735	25.307	625	68,4	76,9
...	...	...	...	...	...	...	...	...	...	...	...
1º	2º	36º	u-SMTC	49,5	149	1	0,99872	51.492	1.135	35,7	58,0
		37º	u-SMTT	49,5	149	0	0,99872	51.641	1.135	35,5	58,0
		38º	u-VVDL	49,5	917	15	0,99872	52.558	1.150	34,4	57,5
		39º	u-SDWD	50,0	2	0	0,99872	52.560	1.150	34,4	57,5
...	...	...	...	...	...	...	...	...	...	...	...
		50º	u-ORSN	67,7	930	59	0,99898	64.837	1.599	19,1	40,8
3º		51º	u-OEBA	68,8	1.488	172	0,99899	66.325	1.771	17,2	34,5
		52º	u-OESA	68,8	992	26	0,99899	67.317	1.797	16,0	33,5
	5º	53º	u-OASN	73,4	538	9	0,99910	67.855	1.806	15,3	33,2
6º	8º	54º	u-VDTR	75,7	4.818	633	0,99988	72.673	2.439	9,3	9,8
5º	7º	55º	u-VTWD	75,7	3.212	119	1,00000	75.885	2.558	5,3	5,4
		56º	u-OARN	76,1	2.094	3	1,00000	77.979	2.561	2,7	5,3
		57º	u-OLAN	78,6	656	68	1,00000	78.635	2.629	1,9	2,7
	4º	58º	u-OLBN	79,9	390	62	1,00000	79.025	2.691	1,4	0,4
...	...	...	...	...	...	...	...	...	...	...	...
		62º	u-OIPM	100,0	10	0	1,00000	80.121	2.703	0,0	0,0
		63º	u-SBRn	100,0	3	0	1,00000	80.124	2.703	0,0	0,0

O operador u-SGLR que tem como objetivo substituir os rótulos dos comandos do tipo goto por todos os outros rótulos que aparecem na mesma função teve um total de 10 mutantes gerados. A *Splint* não foi capaz de identificar nenhum dos mutantes gerados por esse operador. Desse modo, o uso da CDL_R permite determinar uma ordem incremental de aplicação dos operadores de mutação. As demais colunas da Tabela 3.3 serão explicadas no próximo capítulo.

O subconjunto proposto, denominado STAT, composto por 23 operadores de mutação, gerou um total de 9.852 mutantes, dos quais, apenas 320 mutantes são equivalentes, ou seja, apenas 3,25% do total, com uma taxa de redução de custo de 87,7% e taxa de redução de custo equivalente de 88,2% em relação ao conjunto completo de mutantes (80.124 mutantes).

Barbosa et al. (2001) definiu dois subconjuntos essenciais a partir da execução de dois experimentos, sendo o Experimento-I que gerou um conjunto essencial, denominado E27, a partir de um grupo de 27 programas que compõem um editor de texto simplificado e, o Experimento-II que gerou um conjunto essencial, denominado E5, a partir do grupo de 5 programas utilitários Unix. Para mais informações sobre como os conjuntos essenciais foram determinados, o leitor interessado pode consultar (BARBOSA et al., 2001).

Para se obter resultados em termos do escore de mutação semelhantes ao obtido pelos conjuntos essenciais E5 e E27 deveriam ser utilizados os 23 primeiros operadores da estratégia STAT. Nessa situação, o escore de mutação determinado seria superior a 0,995 com redução de custo em termos do número de mutantes gerados e equivalentes de 87,7% e 88,2%, respectivamente.

3.5 Hipóteses Avaliadas

Visando avaliar a qualidade da estratégia incremental de aplicação dos operadores de mutação sugerida, foi realizada uma comparação dessa estratégia, denominada STAT, com os conjuntos de operadores de mutação essenciais definidos para a Linguagem C (BARBOSA et al., 2001).

O resultado da aplicação incremental do conjuntos essenciais obtidos por Barbosa et al. (2001) são apresentados nas Tabelas 3.4 e 3.5. A ordem de aplicação dos operadores dos conjuntos essenciais E5 e E27 prioriza, basicamente, o incremento no escore de mutação (BARBOSA et al., 2001), sem levar em consideração a representatividade dos defeitos modelados pelos operadores.

A título de ilustração, nas Tabelas 3.4 e 3.5 também é apresentada a taxa de correspondência entre avisos e os mutantes de cada um dos operadores dos conjuntos essenciais E5 e E27.

Tabela 3.4: Dados agregados da correspondência dos avisos da *Splint* com operadores essenciais do conjunto E5

E5	Op	$CDL_R(\approx\%)$	#Mut	#Eq	Escore	CC	CCEq	RC($\approx$%)	RCEq($\approx$%)
1º	u-SMTC	49,5	124	0	0,89570	124	0	99,8	100,0
2º	u-SSDL	14,6	2.141	25	0,98575	2.265	25	96,8	98,6
3º	u-OEBA	68,8	1.317	126	0,98835	3.582	151	94,9	91,7
4º	u-ORRN	7,5	2.205	83	0,99417	5.787	234	91,7	87,1
5º	u-VTWD	75,7	2.916	96	0,99797	8.703	330	87,5	81,8
6º	u-VDTR	75,7	4.374	424	0,99927	13.077	754	81,3	58,4

Tabela 3.5: Dados agregados da correspondência dos avisos da *Splint* com operadores essenciais do conjunto E27

E27	Op	$CDL_R(\approx\%)$	#Mut	#Eq	Escore	CC	CCEq	RC($\approx$%)	RCEq($\approx$%)
1º	u-SWDD	3,4	45	6	0,62907	45	6	99,9	99,7
2º	u-SMTC	49,5	124	0	0,89627	169	6	99,8	99,7
3º	u-SSDL	14,6	2.141	25	0,98575	2.310	31	96,7	98,3
4º	u-OLBN	79,9	321	39	0,98836	2.631	70	96,2	96,1
5º	u-OASN	73,4	516	9	0,98985	3.147	79	95,5	95,6
6º	u-ORRN	7,5	2.205	83	0,99458	5.352	162	92,3	91,1
7º	u-VTWD	75,7	2.916	96	0,99806	8.268	258	88,2	85,8
8º	u-VDTR	75,7	4.374	424	0,99950	12.642	682	81,9	62,4
9º	u-Cccr	23,3	5.590	156	0,99959	18.232	838	73,9	53,8
10º	u-Ccsr	25,4	6.554	18	0,99966	24.786	856	64,5	52,8

De maneira diferente da estratégia de aplicação incremental dos operadores essenciais, no contexto deste trabalho, a sequência de aplicação dos operadores obtida é baseada na taxa de $CDL_R(o_k)$, priorizando aqueles com as menores taxas, ou seja, aqueles operadores cujos defeitos modelados tiveram uma menor taxa de identificação por parte da *Splint*. Determinada essa ordem de aplicação, a questão que se coloca é se tal sequência gera resultados tão bons quanto aqueles determinados pelos conjuntos essenciais considerando os aspectos de escore de mutação, total de mutantes gerados e total de mutantes equivalentes gerados (vide Tabela 4.1, p. 44).

Conforme mencionado na introdução, o objetivo principal do nosso experimento é definir uma estratégia incremental de aplicação de operadores de mutação para C, com base nas informações de análise estática (ARAúJO et al., 2015). E realizar uma avaliação da qualidade da estratégia proposta em relação aos conjuntos essenciais (BARBOSA et al., 2001), segundo os aspectos de escore de mutação, custo em termos de mutantes gerados, e custo em termos do número de mutantes equivalentes gerados pelas estratégias.

Nesse sentido, a Tabela 3.6 apresenta as hipóteses definidas visando a avaliação dos conjuntos de operadores sob esses aspectos.

Para avaliação das hipóteses, foram computados o escore de mutação, a redução de custo e a redução de custo equivalente dos três conjuntos em análise sob as mesmas condições, possibilitando a comparação estatística dos resultados para determinar se existe ou não uma diferença entre os conjuntos. Todas estas medidas podem ser influenciadas pelas ferramentas de teste adotadas; e pelo tamanho e complexidade dos programas sob análise.

Um resumo dos dados apresentados nas Tabelas 3.3, 3.4 e 3.5 são apresentados na Tabela 4.1. O detalhamento da análise desses dados é realizado no próximo capítulo.

Tabela 3.6: Formalização das Hipóteses Investigadas

Hipótese Nula	Hipótese Alternativa
Não há diferença no Escore de Mutação entre o conjunto STAT(S) e o ESS(ES). $H1_0$: Escore de Mutação(S) = Escore de Mutação(ES)	Existe uma diferença no Escore de Mutação entre o conjunto STAT(S) e o ESS(ES). $H1_1$: Escore de Mutação(S) = Escore de Mutação(ES)
Não há diferença na Redução de Custo entre o conjunto STAT(S) e o ESS(ES). $H2_0$: Redução de Custo(S) = Redução de Custo(ES)	Existe uma diferença na Redução de Custo entre o conjunto STAT(S) e o ESS(ES). $H2_1$: Redução de Custo(S) = Redução de Custo(ES)
Não há diferença na Redução de Custo Equivalente entre o conjunto STAT(S) e o ESS(ES). $H3_0$: Redução de Custo Equivalente(S) = Redução de Custo Equivalente(ES)	Existe uma diferença na Redução de Custo Equivalente entre o conjunto STAT(S) e o ESS(ES). $H3_1$: Redução de Custo Equivalente(S) = Redução de Custo Equivalente(ES)

3.6 Considerações Finais

Como pode ser observado, existe uma correlação entre os avisos emitidos pelo analisador estático e os operadores de mutação. Adiante faremos a comparação estatística entre entre as estratégias, realizando a comparação par a par de alguns elementos presentes nas Tabelas 3.3, 3.4 e 3.5.

Capítulo 4

DISCUSSÃO

Neste capítulo são discutidas as contribuições desta dissertação, considerando os seis itens apresentados no Capítulo 1 e, no item 3, são analisadas as hipóteses levantadas na Seção 3.5 e as ameaças a validade do estudo.

4.1 Item 1 - Identificação de um subconjunto de operadores de mutação da *Proteum/IM*

Como pode ser observado na Tabela 3.3, apenas um operador obteve taxa de CDL_R igual a zero, ou seja, a *Splint* não foi capaz de relatar qualquer aviso no local que ocorreu a mutação desse operador. Isso demonstra que a *Splint* não é capaz de detectar um defeito modelado por tal operador. O operador em questão é o SGLR (Goto Label Replacement), que tem como objetivo substituir os rótulos dos comandos do tipo goto por outros rótulos que aparecem na mesma função (DELAMARO, 1997a).

Na terminologia dos analisadores estáticos, tais tipos de avisos são chamados de avisos falsos negativos, ou seja, um defeito existe no código fonte, mas o analisador estático não é capaz de detectá-lo. Mas é importante ressaltar também que para alguns tipos de defeitos inseridos o comportamento do possível defeito seja considerado dentro do normal e não um possível problema e acaba passando pelo analisador estático se emitir nenhum aviso, como no caso do u-SGLR ou o u-OAAA por exemplo.

Dos 63 operadores de mutação de utilizados, 23 obtiveram taxa CDL_R abaixo dos 23%. Este resultado pode ser considerado promissor para melhoria do analisador estático devido a dificuldade na detecção de tais defeitos. Com o tipo de defeito modelado pelos operadores é

possível se basear nas suas características para elaboração de novas regras para o analisador estático.

Um exemplo seria o operador u-OAAA (CDL_R 0,9%) que realiza a substituição de um operador aritmético com atribuição por outro operador aritmético com atribuição e o operador u-OAEA (CDL_R 15%) que realiza a substituição de um operador aritmético com atribuição por um operador plano. Tem-se que salientar que isso não significa que a *Splint* não seja capaz de relatar avisos para tais defeitos, mas que ela possui certa dificuldade na detecção de tais defeitos.

Dos 23 operadores com taxa CDL_R abaixo dos 23%, 17 são operadores projetados para modelar defeitos de erro no uso de operadores na Linguagem C, o que mostra a dificuldade da *Splint* na detecção de tais tipos de defeitos. Tem-se ainda quatro operadores projetados para modelar defeitos de comandos e dois operadores projetados para modelar defeitos de variáveis. Cada operador modela um tipo diferente defeito:

- Os 17 operadores relacionados aos defeitos no uso de operadores na Linguagem C são apresentados a seguir: os operadores u-OAAA e u-OAEA substituem um operador aritmético com atribuição por outro operador aritmético com atribuição e por um operador plano respectivamente. O operador u-Oido modela defeitos que surgem com o uso incorreto dos operadores ++ e --. Os operadores u-OBNG, u-OCNG e u-OLNG modelam erros em comandos de seleção e repetição, revertendo essas condições. O operador u-OCOR substitui operadores de cast apenas de tipos primitivos por outros operadores de cast de tipos primitivos da linguagem. O operador u-OAAN substitui um operador aritmético sem atribuição por outro operador aritmético sem atribuição. O operador u-OBSA substitui um operador bitwise com atribuição por um operador swift com atribuição. Os operadores u-OBSN, u-OBLN, u-OBRN, u-OBAN e u-OBBN substituem um operador bitwise sem atribuição, por um operador de deslocamento sem atribuição, por um operador lógico, por um operador relacional, por um outro operador aritmético sem atribuição e por um outro operador bitwise sem atribuição respectivamente. Os operadores u-OLLN e u-OLRN, substituem um operador lógico por outro operador lógico e por outro operador relacional respectivamente. O operador u-ORRN substitui um operador relacional por outro operador relacional.

- Os 4 operadores relacionados aos defeitos no uso de comandos na Linguagem C são apresentados a seguir: o operador u-SGLR substitui rótulos dos comandos do tipo goto por todos os outros rótulos que aparecem na mesma função. O operador u-SWDD substitui um comando while por um comando do-while. O operador u-SSDL foi projetado para mostrar que cada comando do programa tem um efeito sobre a saída, quando executado,

este operador apaga os comandos sistematicamente, mas mantem o ponto-e-vírgula no final do comando para manter a validade sintática do programa. O operador u-SBRC substitui comandos break por comandos continue, exceto quando o break faz parte de um case.

- Os 2 operadores relacionados aos defeitos no uso de variáveis na Linguagem C são apresentados a seguir: o operador u-VLAR substitui referências a vetores por variáveis escalares locais, o operador u-VSCR é responsável pela mutação de referências a componentes de uma estrutura que são substituídas por referências aos demais componentes da mesma estrutura, respeitando os tipos dos componentes.

A partir do subconjunto de operadores obtido, tais informações podem ser utilizadas como subsídio para melhoria da *Splint*, definindo-se novas regras que possam tentar identificar esses possíveis tipos de defeitos.

4.2 Item 2 - Critério de mutação seletiva com base em CDL_R

No trabalho feito por Barbosa et al. (2001) o escore de mutação acima de 0,995 foi considerado um bom ponto de corte proporcionando em média reduções de custo superiores a 65%. Tendo o escore de mutação definido por Barbosa et al. (2001) como referência para o ponto de corte da STAT, foi possível obter uma quantidade de equivalentes baixa e uma alta taxa de redução de custo, que fica acima de 88%.

O subconjunto de operadores da STAT gerou um total de 9.852 mutantes, dos quais, apenas 320 mutantes são equivalentes, ou seja, apenas 3,25% do total. No estudo exploratório Silva et al. (2017), que utilizou um subconjunto de programas do presente trabalho, o subconjunto de operadores proposto possui 23 operadores de mutação que gerou um total de 2.435 mutantes, com um total de 145 mutantes equivalentes, ou seja, 5,95% do total.

No presente estudo, foram utilizados quarenta e dois programas, aumentando assim a variedade de estruturas sintáticas e possibilitando uma maior cobertura de código por parte dos mutantes.

O subconjunto de operadores obtido possibilita o uso da estratégia de mutação seletiva, que tem por objetivo utilizar apenas uma parte dos operadores de mutação, em vez de empregar todo o conjunto de operadores disponível.

4.3 Item 3 - Comparação com outros critérios de mutação seletiva propostas na literatura

O primeiro passo foi reavaliar os conjunto essenciais E5 e E27 no contexto dos quarenta e dois programas selecionados nesse trabalho, considerando a versão atual da Ferramenta *Proteum/IM*. Os dados obtidos são apresentados nas Tabelas 3.4 e 3.5, páginas 38 e 39, respectivamente.

Como pode ser observado na Tabela 4.1, os conjuntos essenciais E5 e E27 determinam escores de mutação de 0,99927 e 0,99966, respectivamente. Utilizando-se apenas os operadores definidos nos conjuntos E5 e E27, a redução de custo em termos do número de mutantes gerados é de 81,3% e 64,5%, respectivamente. Em termos de número de mutantes equivalentes, a redução proporcionada pelo E5 é de 58,4% e do E27 é de 52,8%. STAT obteve escore de mutação de 0.99597, próximo aos escores de E5 e E27 mas com maior redução de custo (6,4% a mais que E5 e 23,2% a mais que E27), tanto em termos de mutantes gerados quanto de equivalentes (29,8% a mais que E5 e 35,4% a mais que E27).

Tabela 4.1: Escore, Redução de Custo (RC) e Redução de Custo de Equivalentes (RCE) dos Conjuntos Essenciais (BARBOSA et al., 2001)

Estratégia	Escore	RC(≈%)	RCEq(≈%)
STAT	0,99597	87,7	88,2
E5	0,99927	81,3	58,4
E27	0,99966	64,5	52,8

Nas Tabelas 3.3, 3.4 e 3.5 (vide Tabela 3.3, p. 37, Tabela 3.4, p. 38 e Tabela 3.5, p. 39) são apresentados os dados sobre a ordem de aplicação dos operadores obtidos, a taxa $CDL_R(o_k)$ de cada operador, a quantidade de mutantes gerados por cada operador, a quantidade de mutantes equivalentes em cada operador, o escore de mutação, o custo cumulativo em termos do número de mutantes gerados da estratégia (CC) e o custo cumulativo em termos do número de mutantes equivalente da estratégia (CCEq).

É possível observar que a STAT mesmo tendo 23 operadores de mutação, gerou 24,7% a menos de mutantes que a E5, que possui 6 operadores de mutação, e 60,3% menos mutantes que a E27, que possui 10 operadores de mutação. A quantidade de mutantes gerados por cada operador se dá em função da presença ou não das estruturas sintáticas exigidas para a sua aplicação. A STAT gerou um total de 9.852 mutantes, a E5 gerou um total de 13.077 mutantes e a E27 gerou um total de 24.786 mutantes.

Na terceira coluna da Tabela 3.3 é apresentada a ordem dos operadores de mutação da estratégia STAT. O primeiro operador sugerido pela STAT é o u-SGLR, que corresponde ao operador com menor CDL_R (0,0%). O u-SGLR gera 10 mutantes (coluna #Mut), considerando o conjunto de programas utilizados e o fato de não ter ocorrência de mutante equivalente desse operador (coluna #Eq). Ao encontrar um conjunto de teste que mate todos os mutantes do u-SGLR, esse conjunto de teste determina um escore de mutação de 0,08645 em relação ao conjunto completo de mutantes (80.124 mutantes).

Considerando, portanto, uma estratégia que utilize apenas esse operador, o custo da estratégia (CC) seria de apenas 10 mutantes, representando uma redução (RC) de 100,0% em relação ao total de mutantes gerados. O custo de equivalentes da estratégia (CCEq) seria 0, que equivale a 100% de redução no número de equivalentes (RCEq).

Ao se utilizar os cinco primeiros operadores da STAT (u-SGLR, u-OAAA, u-SWDD, u-Oido e u-OBAN) o escore de mutação em relação ao total supera os 0,91 e o custo da estratégia em termos do número total de mutantes gerados e equivalentes é de 627 e 16, respectivamente, o que corresponde a uma redução de custo em temos do total gerado e de equivalentes de 99,2% e 99,4%, respectivamente.

Considerando as hipóteses estabelecidas, aplicou-se o teste de normalidade de Shapiro-Wilk baseado no Escore de Mutação, Redução de Custo e Redução de Custo Equivalente, considerando os dados individuais de cada estratégia em cada programa. Por exemplo, a Tabela 4.2 apresenta os dados referentes ao escore de mutação de cada estratégia em relação a cada programa analisado.

Os resultados dos testes estatísticos indicam que os dados não possuem uma distribuição normal com nível de confiança de 95% (p-*value* $\leq$ 0,05). Isso sugere o uso de teste não paramétrico para verificar se existe diferença entre os grupos, para tal foi utilizado o teste de Wilcoxon (BOX et al., 2005) considerando nível de confiança de 95% (α = 0,05).

As colunas nas Tabelas 4.3, 4.5 e 4.7, apresentam os resultados do teste para cada par de conjuntos de operadores. Como são realizados testes múltiplas hipóteses, deve-se aplicar o método de correção de Holm-Bonferroni, que resultou nos valores apresentados (BOX et al., 2005).

Como pode ser observado na Tabela 4.3, o teste estatístico mostra que existe diferença entre as estratégias quando comparadas par a par em termos de escore de mutação, redução de custo e redução de custo equivalente. Desse modo, devem ser aceitas as hipóteses alternativas $H1_1$, $H2_1$ e $H3_1$.

Tabela 4.2: Escore de Mutação das Estratégias para os Programas Unix

Programas	STAT	E5	E27
cal	0,998670	0,999335	0,999778
checkeq	0,995318	0,997893	0,999766
comm	0,994179	0,993597	0,996508
look	0,997253	0,994505	0,995055
uniq	0,998070	1,000000	1,000000
print_tokens	0,999768	1,000000	1,000000
print_tokens2	0,997026	1,000000	1,000000
replace	0,996304	1,000000	1,000000
schedule	0,983871	1,000000	1,000000
schedule2	0,916603	1,000000	1,000000
tcas	0,997955	0,994376	0,996933
totinfo	1,000000	1,000000	1,000000
boundedQueue	0,997324	1,000000	1,000000
Calculation	1,000000	1,000000	1,000000
checkIt	0,960396	0,970297	1,000000
CheckPalindrome	0,993151	1,000000	1,000000
countPositive	0,676056	1,000000	1,000000
date-plus	0,995043	1,000000	1,000000
DigitReverser	0,995575	1,000000	1,000000
findLast	0,994652	0,951872	1,000000
findVal	0,994253	1,000000	1,000000
Gaussian	1,000000	1,000000	1,000000
Heap	0,995935	0,998984	1,000000
InversePermutation	0,998062	0,984496	0,990310
jday-jdate	0,994891	0,999635	0,999635
lastZero	0,945122	1,000000	1,000000
LRS	0,994407	0,996644	1,000000
MergeSort	1,000000	1,000000	1,000000
numZero	0,925373	1,000000	1,000000
oddOrPos	0,996552	0,993103	0,993103
pcal	0,994859	1,000000	1,000000
power	0,992188	0,996094	1,000000
printPrimes	0,993865	1,000000	1,000000
Queue	1,000000	1,000000	1,000000
quicksort	0,994726	1,000000	1,000000
RecursiveSelectionSort	0,990196	1,000000	1,000000
Stack	1,000000	1,000000	1,000000
stats	1,000000	1,000000	1,000000
sum	0,993506	1,000000	1,000000
testPad	0,998252	0,998252	0,998252
twoPred	0,972973	1,000000	1,000000
UnixCal	0,998556	0,998556	0,998556

Tabela 4.3: Escore de Mutação: *p-value* ajustado

Estratégias	E5	E27
E27	0,004	-
STAT	0,001	54,019

A diferença estatística que existe em termos da redução de custo do número de mutantes equivalentes é entre as estratégias E5 e E27, que não tem como foco de análise do presente trabalho.

A diferença estatística em termos do escore de mutação é entre as estratégias STAT e E27 e a diferença estatística em termos da redução de custo é entre as estratégias STAT e E5, o que nos permite realizar uma análise sob a perspectiva da redução de custo e redução de custo equivalente da STAT em relação as duas estratátegias. Como mencionado anteriormente a STAT utiliza uma quantidade maior de operadores e possui também uma taxa de intersecção baixa com os operadores utilizados nas demais estratégias.

Apenas 2 dos 6 operadores da E5 fazem intersecção com os 23 operadores propostos pela STAT, e os demais operadores da estratégia E5 possuem taxa CDL_R acima dos 49,5%, ou seja, são operadores que modelam defeitos com maior probabilidade de serem identificados pela *Splint*. Apenas 3 dos 10 operadores da E27 fazem intersecção com os 23 operadores propostos pela STAT, e gera quase três vezes mais mutantes que a STAT também, tornando assim seu custo mais elevado.

A STAT teve um custo computacional menor que as demais estratégias e conseguiu gerar uma quantidade menor de mutantes equivalentes proporcionalmente, sendo que dos mutantes gerados, a STAT gerou apenas 3,25% de mutantes equivalentes dos totais, a E5 gerou 5,76% de mutantes equivalentes dos totais e a E27 gerou apenas 3,45% de mutantes equivalentes dos totais.

4.4 Item 4 - Defeitos fáceis de serem detectados pelo analisador estático

Alguns operadores foram mais facilmente identificados pelo analisador estático, dois operadores se destacaram obtendo taxas de CDL_R de 100%, ou seja, os defeitos modelados por eles possuem taxa de correspondência máxima, todos os defeitos modelados por eles geraram defeitos que não existiam antes e foram identificados pelo analisador estático.

Tabela 4.4: Porcentagem de Redução de Custo das Estratégias para os Programas Unix

Programas	STAT(≈%)	E5(≈%)	E27(≈%)
cal	68,920	87,216	51,301
checkeq	46,505	87,177	42,253
comm	69,176	81,129	62,767
look	61,338	81,416	70,438
uniq	91,526	80,499	66,976
print_tokens	99,931	78,657	64,699
print_tokens2	99,894	80,949	59,271
replace	99,910	80,326	64,652
schedule	99,526	77,351	72,309
schedule2	99,886	75,107	70,156
tcas	10,487	85,067	78,733
totinfo	99,851	83,473	72,425
boundedQueue	99,911	76,182	72,881
Calculation	98,569	74,776	69,499
checkIt	74,038	79,808	74,038
CheckPalindrome	77,108	70,482	61,446
countPositive	98,675	69,536	68,212
date-plus	99,091	82,693	59,397
DigitReverser	14,919	81,855	68,347
findLast	47,475	74,242	67,172
findVal	50,526	70,000	69,474
Gaussian	99,632	78,085	71,915
Heap	90,547	76,552	68,489
InversePermutation	79,688	72,049	65,451
jday-jdate	92,556	91,741	53,456
lastZero	93,642	68,786	68,786
LRS	95,583	72,968	67,491
MergeSort	99,596	76,589	72,149
numZero	98,675	69,536	68,212
oddOrPos	65,374	78,947	64,789
pcal	99,564	83,268	64,013
power	40,672	75,000	68,657
printPrimes	45,594	78,042	70,490
Queue	98,934	69,936	67,377
quicksort	92,203	73,977	38,462
RecursiveSelectionSort	29,189	74,054	37,778
Stack	99,783	72,668	70,282
stats	99,661	81,335	73,982
sum	74,545	71,515	70,303
testPad	40,223	78,060	72,655
twoPred	62,602	80,488	62,602
UnixCal	99,156	87,436	51,329

Tabela 4.5: Redução de Custo: *p-value* ajustado

Estratégia	E5	E27
E27	0,000	-
STAT	0,476	0,004

Tabela 4.6: Porcentagem de Redução de Custo Equivalente das Estratégias para os Programas Unix

Programas	E5(≈%)	E27(≈%)	STAT(≈%)
cal	67,209	54,570	33,871
checkeq	92,727	41,818	42,273
comm	66,532	72,984	52,016
look	59,144	61,089	48,638
uniq	89,941	68,639	56,213
print_tokens	100,000	100,000	100,000
print_tokens2	100,000	100,000	100,000
replace	100,000	87,500	75,000
schedule	100,000	100,000	100,000
schedule2	100,000	100,000	100,000
tcas	16,589	77,804	76,168
totinfo	100,000	100,000	100,000
boundedQueue	0,000	0,000	0,000
Calculation	0,000	0,000	0,000
checkIt	100,000	100,000	33,333
CheckPalindrome	85,000	40,000	30,000
countPositive	100,000	33,333	44,444
date-plus	0,000	0,000	0,000
DigitReverser	0,000	63,636	52,273
findLast	81,818	36,364	54,545
findVal	93,750	43,750	68,750
Gaussian	0,000	0,000	0,000
Heap	92,632	32,632	29,474
InversePermutation	90,000	48,333	51,667
jday-jdate	93,827	71,605	37,037
lastZero	88,889	33,333	44,444
LRS	94,538	54,202	56,303
MergeSort	0,000	0,000	0,000
numZero	100,000	47,059	64,706
oddOrPos	76,056	77,465	66,205
pcal	0,000	0,000	0,000
power	75,000	16,667	50,000
printPrimes	84,127	33,333	41,270
Queue	0,000	0,000	0,000
quicksort	91,026	38,462	72,027
RecursiveSelectionSort	62,222	33,333	71,892
Stack	100,000	100,000	100,000
stats	0,000	0,000	0,000
sum	81,818	27,273	36,364
testPad	66,667	42,105	43,860
twoPred	75,000	62,500	66,667
UnixCal	100,000	83,333	83,333

Tabela 4.7: Redução de Custo Equivalente: *p-value* ajustado

Estratégia	E5	E27
E27	0,933	-
STAT	0,003	0,002

Os operadores que obteveram taxa CDL_R de 100% foram o u-OIPM que é um operador projetado para modelar defeitos de erro no uso de operadores na Linguagem C, e o operador u-SBRn que é um operador projetado para modelar defeitos de comandos.

Do ponto de vista do analisador estático, considerou-se que, quando o mesmo fosse capaz de detectar no mínimo 50% dos defeitos de um determinado operador de mutação, tais defeitos seriam considerados de fácil detecção. Tais defeitos correspondem aos operadores com taxa de CDL_R de 50%. Assim foi possível determinar um subconjunto de 25 operadores geram defeitos que *Splint* obteve mais sucesso na detecção.

Tais operadores, são divididos em três tipos de modeladores de defeitos, os de operadores, de comandos e de variávies:

- Os 16 operadores relacionados aos defeitos no uso de operadores na Linguagem C são apresentados a seguir: tem o u-OABA e u-OASA que substituem um operador aritmético com atribuição por um operador bitwise e um operador de deslocamento com atrituição respectivamente. Tem o u-ORAN, o u-ORLN, o u-ORBN e o u-ORSN que substituem um operador relacional por um operador artimético sem atribuição, um operador lógico, um operador bitwise sem atribuição e um operador de deslocamento sem atribuição respectivamente. Tem o u-OABN, o u-OASN, o u-OARN e u-OALN que substituem um operador aritmético sem atribuição por um operador bitwise sem atribuição, um operador de deslocamento sem atribuição, um operador operador relacional e um operador lógico respectivamente. Tem o u-OEBA e o u-OESA que substituem um operador plano por um operador bitwise com atribuição e um operador de deslocamento com atribuição respectivamente. Tem o u-OLAN, o u-OLBN e u-OLSN que substituem um operador lógico por um operador aritmético sem atribuição, um operador bitwise sem atribuição e um operador de deslocamento sem atribuição respectivamente. Tem o u-OIPM que tem como objetivo de revelar defeitos no uso incorreto de expressões que envolvam ++, --, e operadores de indireção $*$.

- Os 4 operadores relacionados aos defeitos no uso de comandos na Linguagem C são apresentados a seguir: o operador u-SDWD substitui comandos do-while por comandos while. O operador u-STRI foi projetado para garantir que a condição de cada comando

if seja executada pelo menos uma vez pelo ramo verdadeiro e pelo ramo falso. O operador u-STRP foi projetado para revelar códigos inalcançáveis do programa a ser testado. O operador u-SBRn foi projetado para que quando os comandos break ou continue forém usados dentro de vários laços(X), o operador vai trocar esses comandos pela função BREAK-OUT-TO-N-LEVEL(X), fazendo com que os outros laços sejam interrompidos imediatamente.

- Os 5 operadores relacionados aos defeitos no uso de variáveis na Linguagem C são apresentados a seguir: o operador u-VGAR substitui referências a vetores por variáveis escalares globais. O operador u-VGSR substitui uma referência escalar por uma varivável escalar global. O operador u-VDTR substitui cada referência escalar x pelas chamadas de funções TRAP-ON-ZERO(x), TRAP-ON-POSITIVE(x) e TRAP-ON-NEGATIVE(x), estas funções abortam a execução dos mutantes caso a expressão tenha valor zero, positivo e negativo, respectivamente. O operador u-VTWD substitui a referência escalar pelo seu valor sucessor e predecessor. O operador u-VGPR substitui uma referência a apontador por variáveis escalares globais.

Dos 25 operadores, 16 são projetados para modelar defeitos de erro no uso de operadores na Linguagem C, 4 são projetados para modelar defeitos de comandos e 5 para modelar defeitos de variáveis.

A partir do subconjunto de operadores obtido, tais informações podem ser utilizadas como subsídio para melhoria da *Proteum/IM*, eliminando operadores que modelam defeitos de fácil detecção por meio de análise estática.

4.5 Item 5 - Avisos de maior e menor eficácia em detectar defeitos

Como pode ser observado na Tabela 3.2, a taxa CDL_R total foi de 0,66%, sendo que foram relatados ao todo 8.132.121 avisos e destes, 53.407 foram exatamente no ponto de mutação. Apesar da taxa CDL_R total ser baixa, foi obtido um total de 17 tipos de avisos com taxa superior a 30%.

A taxa CDL_R por Aviso visa estabelecer uma classificação para determinar a capacidade de cada aviso em relatar os defeitos gerados pelos mutantes. Dos 17 tipos de aviso acima dos 30%, pode-se observar que 12 obtiveram taxa CDL_R de 100%, ou seja, todos os defeitos gerados pelos mutantes geraram pelo menos um aviso e a *Splint* foi capaz de detectar todos defeitos.

Os 5 tipos de avisos restantes acima dos 30%, também obtiveram valores satisfatórios e que contribuem na escolha de um subconjunto de tipos de avisos que podem ser utilizados em uma pré analise, pois além de serem efetivos na identificação de tipos de defeitos de manipulação de memória e manipulação de variáveis na Linguagem C, a quantidade de avisos gerados por eles foi de 2.480 avisos ou 0,03% da quantidade de avisos totais gerados, tornando assim a análise dos avisos e busca de falsos positivos posterior menos onerosa.

Dos 487 tipos de avisos da versão 3.1.1 da *Splint*, 91 foram relatados ao longo do presente trabalho como pode se observar na Tabela 3.2, este número pode ser considerado baixo, mas deve ser levado em consideração o escopo e quantidade de programas analisados, pois a tendência é que a medida que novos programas de diferentes escopos forem adicionados, o número de avisos deve subir.

Entre os 91 tipos de avisos relatados, 23 obtiveram uma taxa CDL_R por Aviso de 0.0%, tais resultados não determinam se um aviso é bom ou ruim em identificar determinados defeitos, apenas mostra que tais avisos não tiveram sucesso na detecção de alguns defeitos. Entre eles estão avisos que buscam reuso de uma variável de forma indevida, falha no armazenamento do buffer de memória, problemas com declarações de variáveis e funções.

4.6 Item 6 - Dados quantitativos e qualitativos para evolução das ferramentas utilizadas

Os resultados aqui apresentados mostram que é possível fazer uma correlação entre os avisos da *Splint* e os operadores de mutação da *Proteum/IM*, possibilitando combinar técnicas de análise estática e dinâmica de forma complementar.

Na combinação entre as técnicas, pode-se utilizar tipos de avisos que consigam identificar possíveis defeitos de forma mais eficiente para análise estática, visando a minimizar a ocorrência de falsos positivos. Por outro lado, no teste de mutação, é possível utilizar os operadores que simulam defeitos com menor chance de serem detectados pela análise estática. Tal combinação pode reduzir o tempo de identificação de defeitos em um código, além de reduzir a quantidade de dados a serem analisados para uma validação e posterior correção dos defeitos identificados.

As melhorias para a ferramenta *Splint* podem ser baseadas nos tipos de avisos que poderiam ser criados para identificar determinados tipos de defeitos, modelados pelos operadores de mutação com baixa ou nenhuma correspondência. Desse modo, é possível adicionar novos tipos de avisos ao analisador estático, além de melhorar a eficiência dos existentes com base na taxa CDL_R.

As melhorias para a ferramenta *Proteum/IM* vão desde aprimoramento dos operadores de mutação que obtiveram alta taxa de CDL_R, para que os mesmos produzam defeitos mais difíceis de serem detectados durante os testes, até a inclusão de novos tipos de operadores que modelem novos defeitos, relevantes para domínios de aplicação específicos.

4.7 Ameaças à Validade do Estudo

Nesta seção são apresentadas as ameaças observadas bem como as limitações relacionadas ao processo utilizado neste trabalho.

4.7.1 Validade Externa

A validade externa se deve ao risco da generalização dos resultados, pois se tratando de um estudo experimental, alguns fatores causam ameaças à validade do estudo e não permitem a generalização dos resultados obtidos.

Algumas limitações do presente trabalho estão relacionados com utilização da linguagem de programação C, da ferramenta de mutação e da ferramenta de análise estática.

Nesse trabalho foi empregada apenas a linguagem C e, assim, tais resultados não podem ser estendidos automaticamente para outras linguagens de programação.

Em relação à ferramentas de análise estática e de mutação, foram empregadas, respectivamente, apenas a *Splint* e *Proteum/IM*, e, deste modo, os resultados não podem ser automaticamente estendidos para outras ferramentas. De toda forma, os resultados demonstram que há correspondência entre os avisos emitidos por esse analisador e os tipos de defeitos modelados pelos operadores da *Proteum/IM*, resultado semelhante ao obtido no contexto de Java (ARAúJO et al., 2015, 2016).

Outra ameaça é o número de programas utilizados, bem como o domínio de aplicação destes programas. Entretanto, tais programas continuam sendo extensivamente empregados em outros estudos realizados por outros pesquisadores.

4.7.2 Validade Interna

A validade interna se deve ao controle do processo de experimentação para a coleta dos dados analisados. Inicialmente, foram gerados os mutantes, por meio da ferramenta *Proteum/IM*,

dos programas apresentados na Tabela 3.1. Todos os operadores de unidade da *Proteum/IM* foram selecionados para gerar a maior quantidade possível de mutantes.

Nem todos os tipos de defeitos modelados pelos operadores foram avaliados, uma vez que 15 operadores não geraram mutantes pois os programas não possuíam as estruturas sintáticas exigidas pelos mesmos. Entretanto, para a determinação dos conjuntos E5 e E27 essa mesma ameaça existiu e novos estudos envolvendo outros programas precisam ser identificados para minimizar essa ameaça.

A *Splint* foi utilizada no seu modo mais completo(*-strict*), para que fosse relatado o maior número de avisos possíveis. Para determinar o local onde ocorreu a mutação, durante a coleta dos dados foi realizada a diferença textual(*diff*) entre cada mutante e o programa original.

4.7.3 Validade de Conclusão

A validade de conclusão diz respeito à capacidade de se chegar a conclusão correta entre os relacionamentos entre o tratamento e o resultado.

Mesmo com um conjunto reduzido de dados, os testes realizados são condizentes e sustentam os resultados obtidos.

Com a expansão do trabalho para um grupo maior de programas espera-se aumentar ainda mais a confiança estatística nos resultados.

4.7.4 Validade de Construção

A validade de construção está relacionada com a teoria e a observação. Como a coleta dos dados foi realizada de forma automatizada por meio de scripts construídos por um dos autores e esses scripts foram revisados e validados por outro autor, problemas decorrentes da intervenção humana na geração e manipulação dos dados do experimento foram minimizados.

4.8 Considerações Finais

Neste capitulo foi possível abordar as contribuições e os benefícios que elas podem trazer tanto para a análise estática, quanto para análise dinâmica. Além da comparação estatística entre a STAT e as estratégias E5 e E27, em que foi constatada a diferença estatística entre as estratégias e informações pertinentes sobre dados coletados foram detalhados.

Capítulo 5

TRABALHOS RELACIONADOS

Offutt et al. (1996) realizou uma avaliação experimental para determinar um conjunto essencial de operadores de mutação para testes em Programas Fortran a partir de operadores utilizados pela Mothra. A Mothra é uma ferramenta de apoio ao critério de Análise de Mutantes para programas em Linguagem Fortran, ela possui 22 operadores de mutação. Os resultados obtidos demonstraram que utilizando apenas cinco operadores já seria possível aplicar o teste de mutação de forma eficiente.

Wong et al. (1997) realizou um estudo preliminar semelhante, no qual foi realizada a comparação entre Mutação Seletiva em Programas C e Fortran, resultando na seleção de um subconjunto de operadores de mutação da Ferramenta *Proteum/IM*. O subconjunto obtido possibilitou reduzir o número de mutantes gerados e manter a eficácia do critério em revelar a presença de defeitos.

A seleção de tais operadores por parte de Wong et al. (1997) se baseou na experiência dos autores, os resultados motivaram a condução do trabalho de Barbosa et al. (2001). Os experimentos foram conduzidos com o objetivo de investigar alternativas pragmáticas para a aplicação do Teste de Mutação e, nesse contexto, foi proposto o procedimento Essencial para determinação do conjunto essencial de operadores de mutação para Linguagem C, baseado nos operadores de mutação implementados na Ferramenta *Proteum/IM*. A validação do procedimento proposto foi feita a partir de dois estudos experimentais. No primeiro utilizou-se um grupo de 27 programas, os quais compõem um editor de texto simplificado; no segundo, utilizou-se um grupo de 5 programas utilitários do Unix. Os resultados obtidos a partir de ambos conjuntos mostraram um alto grau de adequação em relação ao Teste de Mutação, tendo escores de mutação acima de 0,995, proporcionando em média reduções de custo superiores a 65% (BARBOSA et al., 2001).

Delamaro et al. (2014) conduziu um estudo a fim de validar a seguinte questão: se a exclusão de declarações é uma maneira econômica de projetar casos de teste, a exclusão de outros elementos do programa também será eficaz? Foi testada a exclusão de variáveis, operadores e constantes, tal abordagem se mostrou eficiente e eficaz. A partir desse estudo foram criados mais três operadores de mutação para a Ferramenta *Proteum/IM* com o objetivo de reduzir o número de mutantes gerados que podem ser mortos com os mesmos casos de teste, tentado tornar o teste de mutação mais barato e mantendo sua eficácia.

O estudo conduzido por Araújo et al. (2015, 2016) teve como objetivo investigar a correspondência entre mutantes e avisos emitidos por ferramentas de análise estática automatizada. Os operadores de mutação foram utilizados como um modelo de defeitos para avaliar a correspondência entre mutantes e avisos estáticos. Foi constatada a correspondência entre alguns operadores de mutação da μJava (MA et al., 2005) e alguns tipos de avisos da *FindBugs* (AYEWAH et al., 2007b). Os autores sugerem que os resultados obtidos podem ser utilizados de duas maneiras:

- Caso decida-se por realizar análise estática com a *FindBugs*, os autores sugerem que, inicialmente, utilizasse aqueles tipos de aviso que possuem maior correspondência com os mutantes pois, desse modo, aumenta-se a chance de tais avisos corresponderem a verdadeiro positivos. A premissa é que se os avisos foram capazes de perceber as mutações possivelmente sejam realmente bons em detectar a presença de determinados tipos de defeitos no código fonte.
- Caso decida-se por realizar o Teste de Mutação a recomendação dada é a inversa, ou seja, deveria dar prioridade para aqueles operadores que geram mutantes que são praticamente imperceptíveis ao analisador estático. Tais mutantes corresponderiam aos falso negativos, ou seja, possíveis defeitos que o analisador estático é incapaz de detectar.

Observa-se que no estudo de Araújo et al. (2015, 2016) as estratégias propostas não foram avaliadas uma vez que o objetivo daquele trabalho era o de evidenciar a existência das correspondências e não a efetividade das estratégias de priorização definidas.

Petrović et al. (2018) conduziu um estudo em que uma aplicação industrial é submetida a testes de mutação, tal aplicação envolvendo mais de 30.000 desenvolvedores e 1,9 milhões de conjuntos de alterações, sendo escrito em 4 linguagens de programação (C++, Go, Java e Python). Utilizando a ferramenta de mutação do Google, que suporta até nove linguagens, foi mostrado que o teste de mutação com mutantes produtivos não adiciona uma sobrecarga significativa ao processo de desenvolvimento de software e relata alguns benefícios do teste de

mutação percebido pelos desenvolvedores, como depuração de código mais eficaz, prevenção de bugs e códigos com melhor qualidade. O mutante produtivo é um mutante que é facilmente morto e provoca um teste produtivo ou um mutante equivalente que sua análise melhora a qualidade do conhecimento e do código.

Just et al. (2017a) aborda o fato de que o trabalho para obtenção de um subconjunto de operadores e a seleção aleatória de tal subconjunto gera um resultado final semelhante, mostrando que o esforço para determinar um subconjunto ainda mantém problemas como a grande quantidade de equivalentes e alguns mutantes relevantes acabam ficando fora do subconjunto. Just et al. (2017a) mostra mostra que para cada contexto de programa, deve ser selecionado um subconjunto diferente e tal analise é feita a partir da Árvore Sintática Abstrata (AST) do programa em teste.

Just et al. (2017b) conduziu um estudo que hipotetizou que a utilidade de um mutante, em termos de equivalência, trivialidade e dominância, pode ser prevista a partir da incorporação de informações do contexto do programa no qual o mutante esta sendo aplicado. Just et al. (2017b) explicou a intuição por trás da hipótese com um exemplo motivacional, em que se propõe uma abordagem para modelar o contexto do programa usando sua Árvore Sintática Abstrata (AST) e propõe avaliar uma série de modelos de contexto de programas para prever a utilidade do mutante. O resultado final foi que realmente é importante considerar o contexto do programa para operadores de mutação individual em vez de grupes de operadores de mutação.

O estudo conduzido por Kurtz et al. (2016) teve como objetivo apresentar evidências de que o escore de mutação não possui precisão para determinar a integridade de um conjunto de casos de testes devido ao ruído introduzido pela redundância inerente a mutação, e que a pontuação da mutação dominadora é uma métrica superior para esta questão. Foi realizada uma avaliação do impacto de diferentes níveis de mutantes redundantes e equivalentes no escore de mutação e a capacidade de determinar a completude de um conjunto de teste de mutação adequada em programas da Linguagem C utilizando a ferramenta *Proteum/IM*.

Kurtz et al. (2016) conduziu um estudo em que foi feita uma analise de subconjuntos utilizados na mutação seletiva a partir do conceito de mutantes dominadores e análise de mutação miníma, podendo assim analisar a mutação seletiva sem o ruído introduzido pela redundância inerente a mutação tradicional utilizando a ferramenta *Proteum/IM*. Foram avaliados de forma exaustiva todos os pequenos conjuntos de operadores de mutação, os resultados mostraram que todas mutações seletivas possíveis tem o escore de mutação do mutante dominador baixo em alguns desses programas em Linguagem C. Isso mostrou que para alcançar um desempenho

melhor em relação a análise de mutação completa, as abordagens terão de se tornar mais sofisticadas.

Durelli et al. (2017) conduziu um estudo onde foi feita a comparação de custo (humano) para detecção de mutantes equivalentes gerados apenas por operadores de mutação de exclusão da ferramenta *Proteum/IM* em relação ao custo para analise de operadores tradicionais, pois segundo Durelli et al. (2017), operadores de exclusão são considerados mais eficazes e mais fáceis de se avaliar manualmente sua equivalência. Os mutantes equivalentes foram avaliados por um grupo de 12 alunos de pós-graduação, em que foi levado em consideração dois fatores, primeiro o número de erros cometidos pelos alunos e depois o tempo gasto nessa análise. Por fim, constataram que não existe uma diferença significativa no custo da avaliação manual, mas foi possível observar que o operador CCDL é o operador de exclusão mais dispendioso.

5.1 Considerações Finais

Neste capítulo foram apresentadas algumas abordagens cujo o objetivo era gerar um conjunto reduzido de operadores que obtivessem um resultado satisfatório para a aplicação do teste de mutação de forma eficiente, tais conjuntos foram obtidos baseado na experiência e conhecimento dos seus autores. Tem-se também abordagens sobre otimização e redução de custo do processo de teste, além de apresentar indicadores que podem auxiliar na redução de custo de forma significativa. Com base nessas recomendações, este trabalho visa a reproduzir e estender o estudo de Araújo et al. (2015, 2016) no contexto de Programas C e investigar se a estratégia de utilizar os operadores de mutação que possuem menor correspondência com os avisos do analisador estático permite obter bons resultados em termos do escore de mutação, número de mutantes gerados e número de mutantes equivalentes.

Capítulo 6

CONCLUSÃO

6.1 Conclusão

No presente trabalho foi feita uma análise sob a perspectiva dos tipos de operadores de mutação que são mais difíceis de serem identificados por analisadores estáticos automatizados e definir uma ordem de aplicação incremental dos operadores de mutação com base em informações de análise estática, definir uma estratégia de mutação seletiva com base nos operadores priorizados e realizar uma análise da eficácia e custo em termos do número de mutantes gerados e o número de equivalentes da estratégia proposta.

As evidências apresentadas neste trabalho podem colaborar para a criação de estratégias complementares de teste que permitam que atividades de análise estática e dinâmica sejam empregadas em conjunto e assim reduzir o custo para se identificar defeitos.

O subconjunto proposto, denominado STAT, composto por 23 operadores de mutação, gerou um total de 9.852 mutantes, dos quais, apenas 320 mutantes são equivalentes, ou seja, apenas 3,25% do total, com uma taxa de redução de custo de 87,7% e taxa de redução de custo equivalente de 88,2% em relação ao conjunto completo de mutantes (80.124 mutantes).

O subconjunto E5 gerou um total de 13.077 mutantes, sendo que 5,76% são equivalentes. Obteve uma taxa de redução de custo de 81,3% e taxa de redução de custo equivalente de 58,4%. O Subconjunto E27 gerou um total de 24.786 mutantes, sendo que 3,45% são equivalentes. Obteve uma taxa de redução de custo de 64,5% e taxa de redução de custo equivalente de 52,8%.

Os testes estatísticos realizados demonstraram que segundo os aspectos investigados, existe diferença entre as estratégias STAT e E27 em termos do escore de mutação e diferença entre as estratégias STAT e E5 em termos da redução de custo, o que nos permite realizar uma análise

sob a perspectiva da redução de custo e redução de custo equivalente da STAT em relação as duas estratégias.

A STAT possui uma quantidade maior de operadores e possui também uma baixa taxa de intersecção com os operadores utilizados nas demais estratégias. Apenas dois dos seis operadores da E5 fazem intersecção com os 23 operadores propostos pela STAT, além dos seus operadores possuírem taxa CDL_R acima dos 49,5%, ou seja, são operadores que modelam defeitos com maior probabilidade de serem identificados pela *Splint*. Apenas três dos dez operadores da E27 fazem intersecção com os 23 operadores propostos pela STAT, além de gerar quase três vezes mais mutantes que a STAT também.

Além do custo computacional menor, considera-se uma vantagem o uso da STAT pois esta utiliza, inicialmente, os operadores que modelam defeitos mais difíceis de serem detectados por meio da análise estática.

6.2 Contribuições

As principais contribuições dessse trabalho foram:

1. Uso alternativo do teste de mutação com a finalidade de avaliar analisadores estáticos e fornecer subsídios para melhoria a *Splint*. Até o presente momento sabe-se que o teste de mutação é bastante utilizado para avaliar a qualidade de conjuntos de teste e comparar diferentes critérios de teste;

2. Identificação de correspondência entre determinados tipos de avisos e operadores de mutação. Os estudos permitiram identificar que existe uma correspondência direta (no nível de linha de código fonte) entre determinadas categorias de avisos e tipos específicos de mutações, modeladas por determinados operadores;

3. O subconjunto de operadores obtido possibilita o uso da estratégia de mutação seletiva, que tem por objetivo utilizar apenas uma parte dos operadores de mutação, em vez de empregar todo o conjunto de operadores disponível;

4. A partir do subconjunto de operadores obtido, tais informações podem ser utilizadas como subsídio para melhoria da *Proteum/IM*, eliminando operadores que modelam defeitos de fácil detecção por meio de análise estática; e

5. A combinação da análise estática e dinâmica de forma complementar, reduzindo o custo de tempo na análise manual de falsos positivos e equivalentes e o custo computacional que

exigiria o uso de todos tipos de avisos do analisador estático e o uso de todos operadores de mutação.

6.3 Trabalhos Futuros

O trabalho conduzido pode ser melhorado por meio dos seguintes trabalhos futuros:

1. Inclusão de outros analisadores estáticos, assim os programas podem ser analisados por diferentes analisadores estáticos e as informações geradas podem ser cruzadas com o objetivo de aumentar a base de conhecimento de defeitos relatados e reduzir a probabilidade de falsos positivos de acordo com a quantidade de ferramentas que relatam um aviso no mesmo ponto por exemplo.
2. Investigar quando um aviso é relatado na mesma linha em que a mutação ocorreu, mas tal aviso também foi relatado no arquivo original, ou seja, o aviso não dependeu da mutação.
3. Investigar quando um aviso é relatado em uma linha no arquivo original, e que não foi relatado no mutante considerando que a mutação foi nessa mesma linha, ou seja, a mutação "corrigiu"o aviso relatado no arquivo original.
4. Investigar quando um aviso não é relatado em um ponto que ocorreu uma mutação para determinar a utilidade daquele operador na estratégia proposta.
5. Incluir outros programas com diferentes estruturas sintáticas, afim de possibilitar um maior alcance por parte dos operadores e melhorar a base de conhecimento e a significância estatística das análises.
6. Sugerir melhorias para o analisador estático.
7. Sugerir melhorias para a ferramenta de mutação.
8. Estabelecer estratégias incrementas combinando análise estática e dinâmica.
9. Estudo do apoio da análise estática na identificação de mutantes equivalentes.
10. Comparar os resultados obtidos com o conjunto minimal de operadores de mutação.

6.4 Publicação

O trabalho desenvolvido permitiu conduzir um estudo exploratório, que além de gerar essa dissertação, gerou um artigo científico:

- Artigo: ***Incremental Strategy for Applying Mutation Operators Emphasizing Faults Difficult to be Detected by Automated Static Analyser***
  Evento: SBES 2017 - Proceedings of the 31st Brazilian Symposium on Software Engineering
  Situação: Aceito e Apresentado.

REFERÊNCIAS

AMMANN, P.; OFFUTT, J. *Introduction to Software Testing*. 1. ed. New York, NY, USA: Cambridge University Press, 2008. ISBN 0521880386, 9780521880381.

ANDREWS, J. H.; BRIAND, L. C.; LABICHE, Y. Is mutation an appropriate tool for testing experiments? In: *XXVII International Conference on Software Engineering – ICSE'05*. New York, NY, USA: ACM Press, 2005. p. 402–411. ISBN 1-59593-963-2.

ARAúJO, C. A.; DELAMARO, M. E.; MALDONADO, J. C.; VINCENZI, A. M. R. Investigating the corresponcende between mutations and static warnings. In: SBC. *XXIX Simpósio Brasileiro de Engenharia de Software – SBES'2015*. Belo Horizonte, MG: SBC, 2015. p. 1–10. Premiado como um dos melhores artigos do SBES'2015.

ARAúJO, C. A.; DELAMARO, M. E.; MALDONADO, J. C.; VINCENZI, A. M. R. Correlating automatic static analysis and mutation testing: towards incremental strategies. *Journal of Software Engineering Research and Development – JSERD*, v. 4, n. 1, p. 1–32, nov. 2016. Artigo em avaliação. Versão estendida do artigo "Investigating the Corresponcende between Mutations and Static Warnings" premiado entre os melhores artigos do SBES'2015.

Araújo, C. A. D.; Delamaro, M. E.; Maldonado, J. C.; Vincenzi, A. M. R. Investigating the correspondence between mutations and static warnings. In: *2015 29th Brazilian Symposium on Software Engineering*. [S.l.: s.n.], 2015. p. 1–10.

ARAúJO, C. A. de. *Uma Investigação da Correspondência entre Mutações e Avisos Relatados por Ferramenta de Análise Estática*. Dissertação (Dissertação de Mestrado) — Instituto de Informática – INF/UFG, Goiânia, GO, dez. 2015.

AYEWAH, N.; PUGH, W.; MORGENTHALER, J. D.; PENIX, J.; ZHOU, Y. Evaluating static analysis defect warnings on production software. In: *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. New York, NY, USA: ACM, 2007. (PASTE'07), p. 1–8. ISBN 978-1-59593-595-3.

AYEWAH, N.; PUGH, W.; MORGENTHALER, J. D.; PENIX, J.; ZHOU, Y. Using findbugs on production software. In: *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*. New York, NY, USA: ACM, 2007. (OOPSLA '07), p. 805–806. ISBN 978-1-59593-865-7.

BARBOSA, E. F.; MALDONADO, J. C.; VINCENZI, A. M. R. Towards the determination of sufficient mutant operators for C. *STVR – Software Testing, Verification and Reliability*, John Wiley & Sons, Inc., New York, NY, USA, v. 11, n. 2, p. 113–136, jun. 2001. ISSN 0960-0833.

BOEHM, B.; BASILI, V. R. Software defect reduction top 10 list. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 34, n. 1, p. 135–137, 2001. ISSN 0018-9162.

BOX, G. E. P.; HUNTER, J. S.; HUNTER, W. G. *Statistics for Experimenters: Design, Innovation, and Discovery*. 2. ed. [S.l.]: Wiley-Interscience, 2005.

COPELAND, T. *PMD Applied: An Easy-to-use Guide for Developers*. [S.l.]: Centennial Books, 2005. (An easy-to-use guide for developers). ISBN 9780976221418.

CORRENSON, L.; CUOQ, P.; KIRCHNER, F.; PREVOSTO, V.; PUCCETTI, A.; SIGNOLES, J.; YAKOBOWSKI, B. *Frama-C Software Analyzers*. mar. 2014. Página Web. Disponível em: `http://frama-c.com/`. Accesso em: 03/05/2017.

CORRENSON, L.; CUOQ, P.; KIRCHNER, F.; PREVOSTO, V.; PUCCETTI, A.; SIGNOLES, J.; YAKOBOWSKI, B. *Frama-C Software Analyzers*. mar. 2014. Tool's Dowload Page. Available at: `http://frama-c.com/`. Access on: 10/10/2015.

COUTO, C.; MONTANDON, J. a. E.; SILVA, C.; VALENTE, M. T. Static correspondence and correlation between field defects and warnings reported by a bug finding tool. *Software Quality Control*, Kluwer Academic Publishers, Hingham, MA, USA, v. 21, n. 2, p. 241–257, jun. 2013. ISSN 0963-9314.

DAN, H.; HIERONS, R. M. Smt-c: A semantic mutation testing tools for c. In: *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. Washington, DC, USA: IEEE Computer Society, 2012. (ICST '12), p. 654–663. ISBN 978-0-7695-4670-4. Disponível em: <http://dx.doi.org/10.1109/ICST.2012.155>.

DELAMARO, M.; CHAIM, M. L.; MALDONADO, J. C. Where are the minimal mutants? In: *Proceedings of the XXXII Brazilian Symposium on Software Engineering*. New York, NY, USA: ACM, 2018. (SBES '18), p. 190–195. ISBN 978-1-4503-6503-1. Disponível em: <http://doi.acm.org/10.1145/3266237.3266241>.

DELAMARO, M. E. *Interface Mutation: An Interprocedural Adequacy Criterion for Integration Testing*. Tese (Doctoral Dissertation) — Instituto de Física de São Carlos – Universidade de São Paulo, São Carlos, SP, jun. 1997. (in Portuguese).

DELAMARO, M. E. *Mutação de Interface: Um Critério de Adequação Inter-procedimental para o Teste de Integração*. Tese (Doutorado) — Instituto de Física de São Carlos – Universidade de São Paulo, São Carlos, SP, jun. 1997.

DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. *Introdução ao Teste de Software*. 2. ed. Rio de Janeiro, RJ: Elsevier, 2016.

DELAMARO, M. E.; MALDONADO, J. C.; VINCENZI, A. M. R. Proteum/IM 2.0: An integrated mutation testing environment. In: *Mutation 2000 Symposium*. San Jose, CA: Kluwer Academic Publishers, 2000. p. 91–101.

DELAMARO, M. E.; OFFUTT, J.; AMMANN, P. Designing deletion mutation operators. In: *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*. Washington, DC, USA: IEEE Computer Society, 2014. (ICST '14), p. 11–20. ISBN 978-1-4799-2255-0.

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, abr. 1978.

DO, H.; ELBAUM, S.; ROTHERMEL, G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, Kluwer Academic Publishers, Hingham, MA, USA, v. 10, n. 4, p. 405–435, out. 2005. ISSN 1382-3256. Disponível em: <http://dx.doi.org/10.1007/s10664-005-3861-2>.

Durelli, V. H. S.; De Souza, N. M.; Delamaro, M. E. Are deletion mutants easier to identify manually? In: *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. [S.l.: s.n.], 2017. p. 149–158.

EVANS, D.; LAROCHELLE, D. Improving security using extensible lightweight static analysis. *IEEE Software*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 19, n. 1, p. 42–51, jan. 2002. ISSN 0740-7459.

FILHO, J. E. de A.; COUTO, C. F. de M.; SOUZA, S. J. de; VALENTE, M. T. Um estudo sobre a correlação entre defeitos de campo e warnings reportados por uma ferramenta de análise estática. In: *IX Simpósio Brasileiro de Qualidade de Software – SBQS'2010*. Belém, PA: [s.n.], 2010. p. 9–23.

HOVEMEYER, D.; PUGH, W. Finding bugs is easy. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 39, n. 12, p. 92–106, dez. 2004. ISSN 0362-1340.

ISO/IEC/IEEE. *ISO/IEC/IEEE 24765:2017(E)*, p. 1–541, Aug 2017.

JIA, Y.; HARMAN, M. Milu: A customizable, runtime-optimized higher order mutation testing tool for the full c language. In: *Practice and Research Techniques, 2008. TAIC PART '08. Testing: Academic Industrial Conference*. [S.l.: s.n.], 2008. p. 94–98.

JOHNSON, B.; SONG, Y.; MURPHY-HILL, E.; BOWDIDGE, R. Why don't software developers use static analysis tools to find bugs? In: *Proceedings of the 2013 International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2013. (ICSE '13), p. 672–681. ISBN 978-1-4673-3076-3.

JUST, R.; KURTZ, B.; AMMANN, P. *Customized Program Mutation: Inferring Mutant Utility from Program Context*. Amherst, MA, USA, abr. 2017. 1–17 p.

JUST, R.; KURTZ, B.; AMMANN, P. Inferring mutant utility from program context. In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, 2017. (ISSTA 2017), p. 284–294. ISBN 978-1-4503-5076-1. Disponível em: <http://doi.acm.org/10.1145/3092703.3092732>.

KRESOWATY, J. *FxCop and Code Analysis: Writing Your Own Custom Rules*. [S.l.]: Citeseer, 2008.

KURTZ, B.; AMMANN, P.; OFFUTT, J.; DELAMARO, M. E.; KURTZ, M.; GöKçE, N. Analyzing the validity of selective mutation with dominator mutants. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: ACM, 2016. (FSE 2016), p. 571–582. ISBN 978-1-4503-4218-6. Disponível em: <http://doi.acm.org/10.1145/2950290.2950322>.

Kurtz, B.; Ammann, P.; Offutt, J.; Kurtz, M. Are we there yet? how redundant and equivalent mutants affect determination of test completeness. In: *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. [S.l.: s.n.], 2016. p. 142–151.

LOURIDAS, P. Static code analysis. *IEEE Software*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 23, n. 4, p. 58–61, jul. 2006. ISSN 0740-7459.

MA, Y.-S.; OFFUTT, J.; KWON, Y. R. Mujava: an automated class mutation system: Research articles. *STVR – Software Testing, Verification and Reliability*, John Wiley and Sons Ltd., Chichester, UK, UK, v. 15, n. 2, p. 97–133, 2005. ISSN 0960-0833.

MICROSOFT. *FxCop*. 2011. Página WEB. Disponível em: `http://msdn.microsoft.com/en-us/library/bb429476.aspx`. Acesso em: 01/08/2011.

MICROSOFT. *StyleCop*. mar. 2014. Página Web. Disponível em: `https://stylecop.codeplex.com/`. Accesso em: 07/02/2014.

OFFUTT, A. J.; LEE, A.; ROTHERMEL, G.; UNTCH, R. H.; ZAPF, C. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering Methodology*, v. 5, n. 2, p. 99–118, abr. 1996.

PETROVIĆ, G.; IVANKOVIĆ, M.; KURTZ, B.; AMMANN, P.; JUST, R. An industrial application of mutation testing: Lessons, challenges, and research directions. In: *Proceedings of the International Workshop on Mutation Analysis (Mutation)*. [S.l.: s.n.], 2018. p. 47–53.

POHL, J. *Lint – C program verifier*. maio 2001. Página Web. Disponível em: `http://www.unix.com/man-page/FreeBSD/1/lint`. Accesso em: 02/05/2017.

PRESSMAN, R. S. *Engenharia de Software - Uma Abordagem Profissional*. 7nd.. ed. [S.l.]: McGraw-Hill, 2011.

SILVA, V. B.; ARAUJO, C. A.; SPOTO, E. S.; VINCENZI, A. M. R. Incremental strategy for applying mutation operators emphasizing faults difficult to be detected by automated static analyser. In: *Proceedings of the 31st Brazilian Symposium on Software Engineering*. New York, NY, USA: ACM, 2017. (SBES'17), p. 24–33. ISBN 978-1-4503-5326-7. Disponível em: <http://doi.acm.org/10.1145/3131151.3131169>.

SOMMERVILLE, I. *Engenharia de Software*. 8. ed. São Paulo, SP: Addison Wesley, 2007. 568 p.

TERRA, R.; BIGONHA, R. S. Ferramentas para análise estática de códigos java. *Monografia, Universidade Federal de Minas Gerais (UFMG), Belo Horizonte*, 2008.

VINCENZI, A. M. R. *Subsídios para o Estabelecimento de Estratégias de Teste Baseadas na Técnica de Mutação*. Dissertação (Mestrado) — ICMC-USP, São Carlos, SP, Brasil, nov. 1998. Disponível em: `http://www.teses.usp.br/teses/disponiveis/55/55134/tde-06022001-182640`. Acesso em: 21/10/2004.

WONG, W.; MALDONADO, J.; DELAMARO, M.; SOUZA, S. A comparison of selective mutation in C and fortran. In: *Workshop do Projeto Validação e Teste de Sistemas de Operação*. Águas de Lindóia, SP: [s.n.], 1997. p. 71–80.

WONG, W. E. *On Mutation and Data Flow*. Tese (Doutorado) — Department of Computer Science, Purdue University, W. Lafayette, IN, dez. 1993.