UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

CLÁUDIO ANTÔNIO DE ARAÚJO

Uma Investigação da Correspondência entre Mutações e Avisos Relatados por Ferramenta de Análise Estática

Goiânia – Goiás
2015

CLÁUDIO ANTÔNIO DE ARAÚJO

Uma Investigação da Correspondência entre Mutações e Avisos Relatados por Ferramenta de Análise Estática

Dissertação apresentada ao Programa de Pós–Graduação do Instituto de Informática da Universidade Federal de Goiás, como requisito parcial para obtenção do título de Mestre em Computação.

Área de concentração: Ciência da Computação.

Orientador: Prof. Dr. Auri Marcelo Rizzo Vincenzi

Goiânia – Goiás
2015

CLÁUDIO ANTÔNIO DE ARAÚJO

Uma Investigação da Correspondência entre Mutações e Avisos Relatados por Ferramenta de Análise Estática

Dissertação defendida no Programa de Pós–Graduação do Instituto de Informática da Universidade Federal de Goiás como requisito parcial para obtenção do título de Mestre em Computação, aprovada em 04 de Dezembro de 2015, pela Banca Examinadora constituída pelos professores:

Prof. Dr. Auri Marcelo Rizzo Vincenzi
Instituto de Informática – UFG
Presidente da Banca

Prof. Dr. Marco Túlio de Oliveira Valente
Departamento de Ciência da Computação – UFMG

Prof. Dr. Fábio Nogueira de Lucena
Instituto de Informática – UFG

Cláudio Antônio de Araújo

Bacharel em Ciências da Computação pelo Instituto de Informática (INF) da Universidade Federal de Goiás (UFG) em 2006. Em 2008, concluiu pós graduação *lato sensu* em Desenvolvimento de Aplicações Web com Interfaces Ricas (INF/UFG). É servidor público na área de Tecnologia da Informação no Tribunal Regional do Trabalho da 18^a Região (TRT 18).

A todos que me apoiaram e me deram suporte para a realização deste trabalho e à Deus pela sabedoria.

Agradecimentos

A Deus, pelo dom da vida e por me acompanhar sempre.

Ao amigo e orientador, Prof. Dr. Auri Vincenzi, pelo apoio, sugestões e profissionalismo na orientação deste trabalho. Aos professores Dr. José Maldonado e Dr. Márcio Delamaro pelas sugestões e propostas de melhorias.

Aos meus familiares, que sempre estão ao meu lado.

Aos amigos do Tribunal Regional do Trabalho da 18^a Região, pelo apoio e incentivo.

Aos amigos Guilherme Maranhão e Bianca, pela atenção, amizade e pelos momentos de estudos de Teoria da Computação e Análise Projeto de Algoritmo, etc.

Ao amigo Marllos Paiva, pelas dicas, apoio e torcida.

Aos professores e funcionários do Instituto de Informática que contribuíram para a realização do presente trabalho.

Aos colegas e amigos das disciplinas Projeto e Análise de Algoritmos (Turma 2012/1), Teoria da Computação (Turma 2012/2), Engenharia de Software (Turma 2013/1) e Metodologia Científica (Turma 2014/1) pelos momentos de estudo, aprendizado e descontração.

Sou muito grato a todos vocês.

À UFG e FAPEG pela ajuda financeira na participação de eventos.

“Nós somos o que fazemos repetidas vezes, repetidamente
A excelência, portanto, não é um feito, mas um hábito.”

Aristóteles,

.

.

Resumo

de Araújo, Cláudio Antônio. **Uma Investigação da Correspondência entre Mutações e Avisos Relatados por Ferramenta de Análise Estática**. Goiânia – Goiás, 2015. 89p. Dissertação de Mestrado. Instituto de Informática, Universidade Federal de Goiás.

Contexto: Considerando que: 1) analisadores estáticos automatizados são ferramentas que emitem avisos, sem que seja necessário a execução do produto de software correspondente, alertando sobre a presença de possíveis defeitos no código. Uma das críticas a tais ferramentas é a grande quantidade de avisos falsos positivos emitidos, isto é, avisos relatados que não correspondem a defeitos reais, mas demandam tempo de análise por parte do desenvolvedor; 2) tradicionalmente, o Teste de Mutação tem sido utilizado para avaliar (e melhorar) a qualidade de conjuntos de casos de teste e/ou de critérios de teste, uma vez que é considerado um bom gerador de defeitos de software.

Objetivo: O objetivo do presente trabalho é investigar a correspondência entre avisos estáticos e mutações e, com isso, verificar quais avisos estão mais relacionados a esses possíveis defeitos (mutações) e, assim, possivelmente, serem avisos verdadeiros positivos.

Método: Os operadores de mutação são utilizados neste trabalho como um modelo de defeitos para avaliar a correspondência entre mutações e avisos estáticos. A principal vantagem da utilização de operadores de mutação é que eles geram um grande número de programas com defeitos de diferentes tipos. Esses tipos de defeitos são usados em estudos experimentais para investigar a capacidade dos analisadores estáticos em detectá-los.

Resultados: Os resultados obtidos com estudos experimentais para um conjunto de sistemas de código aberto indicam que existe correspondência quando são considerados alguns operadores de mutação da μJava e alguns tipos de avisos da FindBugs.

Conclusão: Os resultados obtidos podem ser utilizados de duas maneiras distintas: Primeiro, é fornecida uma abordagem de análise incremental dos avisos, de acordo com o grau de correspondência com mutações. Segundo, com o objetivo de reduzir o custo do Teste de Mutação é fornecida uma abordagem de priorização incremental para análise dos mutantes dos operadores cujas mutações são menos "percebidas" pela FindBugs.

Palavras–chave

Teste de software, teste de mutação, warning, análise estática, analisadores estáticos

Abstract

de Araújo, Cláudio Antônio. **Investigating the Correspondence between Mutations and Static Warnings**. Goiânia – Goiás, 2015. 89p. MSc. Dissertation. Instituto de Informática, Universidade Federal de Goiás.

Context: Traditionally, mutation testing is used for test set and/or test criteria evaluation once it is considered a good fault model. Since static analyzers, in general, report a substantial number of false positive warnings,

Objective: This paper uses mutation testing for evaluating an automated static analyzer. The intention of this study is to define a prioritization approach of static warnings based on their correspondence with mutations.

Method: We used mutation operators as a fault model to evaluate the direct correspondence between mutations and static warnings. The main advantage of using mutation operators is that they generate a large number of programs containing faults of different types, which can be used to decide the ones most probable to be detected by static analyzers.

Results: The results obtained for a set of open-source programs indicate that: 1) correspondence exists when considering specific mutation operators such that static warnings may be prioritized based on their correspondence level with mutations; 2) correspondence exists when considering specific warning categories such that, assuming we perform static analysis considering these warning categories, mutation operators may be prioritized based on their correspondence level with warnings.

Conclusion: It is possible to provide an incremental testing strategy aiming at reducing the cost of both static analysis and mutation testing using the correspondence information. On the other hand, knowing that Mutation Test has a high application cost, we identified mutations of some specific mutation operators, which an automatic static analyzer is not able to detect. Therefore, this information can used to prioritize the order of applying mutation operators incrementally considering, firstly, those with no correspondence with static warnings.

Keywords

Software testing, mutants, warnings, static analysis, static analyzer evaluation.

Sumário

Lista de Figuras **11**

Lista de Tabelas **12**

1 Introdução **13**
- 1.1 Contexto e Motivação 13
- 1.2 Solução Proposta 16
  - 1.2.1 Objetivos 17
  - 1.2.2 Metodologia 18
  - 1.2.3 Contribuições 19
- 1.3 Organização da Dissertação 19

2 Fundamentação Teórica **21**
- 2.1 Conceitos Básicos de Verificação e Validação 21
- 2.2 Análise Estática Automatizada 22
- 2.3 Teste de Mutação 24
- 2.4 Considerações Finais do Capítulo 29

3 Arcabouço de Experimentação **31**
- 3.1 Visão Geral 31
- 3.2 Identificação dos Dados Necessários 37
- 3.3 Processo Utilizado 39
- 3.4 Primeiro Experimento Controlado 41
- 3.5 Considerações Finais do Capítulo 45

4 Estudo Experimental I – CDL por Aviso **46**
- 4.1 Sistemas Utilizados 46
- 4.2 CDL por Aviso 50
- 4.3 Cálculo da CDL por Aviso 51
- 4.4 Exemplo de Aplicação da CDL por Aviso 53
- 4.5 Considerações Finais do Capítulo 60

5 Estudo Experimental II – CDL por Operador **61**
- 5.1 CDL por Operador 61
- 5.2 Cálculo da CDL por Operador 62
- 5.3 Exemplo de Aplicação da CDL por Operador 65
- 5.4 Considerações Finais do Capítulo 70

6 Análise dos Resultados **71**
6.1 Análise Geral 71
6.2 Análise da CDL por Operador 72
6.3 Análise da CDL por Aviso 73
6.4 Lições Aprendidas 74
6.5 Ameaças à Validade do Estudo 76
6.5.1 Validade Externa 76
6.5.2 Validade Interna 77
6.5.3 Risco de Construção 77
6.6 Considerações Finais do Capítulo 78

7 Trabalhos Relacionados **79**

8 Conclusão, Trabalhos Futuros e Publicações **82**
8.1 Conclusão 82
8.2 Principais Contribuições 83
8.3 Trabalhos Futuros 84
8.4 Publicações 85

Referências Bibliográficas **87**

Lista de Figuras

2.1 Estrutura de Aviso da FindBugs. Adaptado de (SHEN; FANG; ZHAO, 2011) 23

3.1 Visão geral da estrutura comparativa entre avisos relatados em cada arquivo original f_j e em cada mutante $m_i o_k f_j$, nos níveis de linha, método, classe, arquivo e sistema. 33
3.2 Aviso w_1 relatado no mutante $m_i o_k f_j$ em virtude da mutação 34
3.3 Aviso w_2 relatado no mutante e no arquivo original, na mesma linha da mutação 35
3.4 Aviso w_3 relatado no arquivo original deixou de ser relatado no mutante 35
3.5 Exemplo do Caso 1 36
(a) Código original 1 36
(b) Código mutante 1 36
3.6 Exemplo do Caso 2 36
(a) Código original 2 36
(b) Código mutante 2 36
3.7 Exemplo do Caso 3 36
(a) Código original 3 36
(b) Código mutante 3 36
3.8 Modelo de Domínio Utilizado no Experimento. 38
3.9 Diagrama do Processo Utilizado no Experimento. 40

4.1 Custo acumulado ($CC(w_i)$) versus redução de custo ($CR(w_i)$) na abordagem de priorização para análise dos avisos, de acordo com a $CDL_R(w)$. 57

5.1 Custo acumulado ($CC(o_k)$) versus redução de custo ($CR(o_k)$) na abordagem de priorização para análise de mutantes, de acordo com a $CDL_R(o_k)$ 69

Lista de Tabelas

2.1 Categorias de Avisos da FindBugs 24
2.2 Detalhes de Avisos Relatados pela FindBugs 25
2.3 Exemplos de Mutantes 26
2.4 Operadores de Mutação de Método da μJava (adaptado de (MA; OFFUTT, 2005)) 29
2.5 Operadores de Mutação de Classe da μJava (adaptado de (OFFUTT; MA; KWON, 2006)) 30

3.1 Programas Utilizados no Primeiro Experimento 41
3.2 Correspondência por Tipo de Operador no Primeiro Experimento 42
3.3 Correspondência por Tipo de Aviso no Primeiro Experimento 43

4.1 Sistemas Utilizados no Estudo Experimental Estendido 47
4.2 Avisos Relatados nos Mutantes por Operador 49
4.3 Correspondência Direta por Linha por Aviso 52
4.4 Sistemas para Exemplificar a Aplicação da $CDL_R(w)$ 54
4.5 Priorização de Avisos de acordo com a $CDL_R(w)$ 56
4.6 Avisos Relatados nos Mutantes dos Sistemas Adicionais 58
4.7 Avisos Analisados pela Categoria e Prioridade 59
4.8 Avisos Analisados pela CDL 60

5.1 Correspondência Direta por Linha por Operador 63
5.2 Sistemas para Exemplificar a Aplicação da $CDL_R(o_k)$ 66
5.3 Priorização de Operadores de Acordo com a $CDL_R(o_k)$ 68

6.1 Operadores com Alta Correspondência 72
6.2 Operadores de Método com Baixa Correspondência 72
6.3 Operadores de Classe com Baixa Correspondência 74
6.4 Detalhes dos Avisos com Alta Correspondência 75
6.5 Detalhes dos Avisos com Baixa Correspondência 76

CAPÍTULO 1

Introdução

1.1 Contexto e Motivação

No desenvolvimento de software é comum o emprego de diversas técnicas com a finalidade de obter um produto com qualidade e de baixo custo. Verificação e Validação (V & V) são técnicas que permitem melhorar a qualidade e a confiança em produtos de software. As atividades relacionadas à verificação e validação podem ser subdivididas em duas categorias: 1) atividades de análise dinâmica e 2) atividades de análise estática. As atividades de análise dinâmica demandam a execução do produto em análise. As diversas técnicas e critérios de teste – Caixa Preta, Caixa Branca e Baseada em Defeitos – podem ser classificadas como atividades de análise dinâmica, pois requerem a execução de casos de testes sobre o produto em análise. Já as atividades de análise estática avaliam o produto em questão sem a necessidade de execução do mesmo. Revisão e inspeção são exemplos de atividades de análise estática (MALDONADO; DELAMARO; JINO, 2007).

Durante o processo de desenvolvimento de software, ferramentas de análise estática automatizadas – também chamadas de *bug findings tools* – podem ser utilizadas para apoiar a verificação de determinados padrões de código. Em vez de verificar o atendimento do sistema à sua respectiva especificação, ferramentas de análise estática funcionam procurando violações de código. Em tais ferramentas são implementados algoritmos e regras que permitem detectar estas violações. São exemplos de violações detectadas por estas ferramentas: a) o acesso a objetos inválidos (isto é, antes de serem inicializados); b) o uso de métodos obsoletos (*deprecated*); c) a codificação em desacordo com determinado padrão estabelecido; d) trecho de código desnecessário (exemplo: variável definida e não utilizada); uso incorreto de API (*Application Programming Interface*) tais como o não fechamento de conexão com banco de dados ou comparação de strings por meio de operador lógico de igualdade, entre outros. As ocorrências destas violações são relatadas pelas ferramentas de análise estática por meio de avisos (*warnings*).

No desenvolvimento de software também é comum a existência de atividades de desenvolvimento e manutenção para encontrar (e corrigir) defeitos em produtos de software. Defeitos de software são introduzidos no código fonte por causa de enganos

cometidos por pessoas quando elas tentam compreender determinadas informações. Um comando ou uma instrução errada presente no código fonte é um exemplo de defeito de software (ISO, 2010).

Existem várias ferramentas de análise estática disponíveis para diferentes linguagens de programação, como exemplos podem ser citadas: a StyleCop (MICROSOFT, 2014) e a FxCop (KRESOWATY, 2008) para sistemas em .NET, a Lint (POHL, 2001) e a SPlint (EVANS; LAROCHELLE, 2002) para sistemas em C/C++. Para a linguagem Java podem ser utilizadas a FindBugs (HOVEMEYER; PUGH, 2004), a PMD (COPELAND, 2005), a CheckStyle (BURN, 2014), entre outras (DAIMI; BANITAAN; LISZKA, 2013).

Contudo, apesar do interesse, tanto acadêmico como industrial, pela utilização das ferramentas de análise estática (LOURIDAS, 2006; AYEWAH et al., 2007b; HOVEMEYER; PUGH, 2004), ainda não há consenso sobre os reais benefícios e qualidade que elas proporcionam ao produto de software para indicar a presença de defeitos (AYEWAH et al., 2007a).

Alguns trabalhos avaliam a qualidade e precisão das ferramentas de análise estática para indicar defeitos em produtos de software (FILHO et al., 2010; COUTO et al., 2013; JOHNSON et al., 2013). Os resultados dos trabalhos de Filho et al. (2010), Couto et al. (2013) indicam que não há correlação entre avisos e defeitos reais (defeitos de campo). Supõe-se, contudo, que tais resultados podem estar relacionados aos seguintes motivos: a) pequena quantidade de defeitos reais observados durante os experimentos; b) dificuldade em classificar os tipos de defeitos (defeitos reais ou funcionalidade ausente); c) os tipos de defeitos são tratados de forma geral, não explorando o fato de que tais defeitos podem ser agrupados em categorias de acordo com algum critério (aritmético, lógico, etc); entre outros.

Do ponto de vista de custo, sabe-se que quanto antes um defeito é detectado, mais barato é a sua respectiva correção (BOEHM; BASILI, 2001). Além disso, o fato de não exigir a execução do produto final tornam atrativas as ferramentas que realizam análise estática. Outra vantagem do emprego destas ferramentas é o fato de serem consideradas de baixo custo, pois a aplicação de tais ferramentas sobre um produto em análise é feita de forma automatizada. Entretanto, essas ferramentas costumam relatar uma grande quantidade de avisos chamados **falsos positivos**, ou seja, avisos que não correspondem a defeitos reais, mas que vão demandar tempo de investigação por parte do testador para sua análise Filho et al. (2010), Couto et al. (2013), Johnson et al. (2013). Devido aos avisos falsos positivos, as ferramentas de análise estática acabam sendo subutilizadas pelos desenvolvedores. Idealmente, tais ferramentas deveriam relatar apenas avisos verdadeiros positivos, ou seja, ao corrigir/remover o problema indicado pelo aviso, um defeito é eliminado e o aviso em questão deixaria de ser relatado.

Filho et al. (2010) e Couto et al. (2013), investigando a correlação entre defeitos e avisos, concluíram que não há correlação entre defeitos reais e avisos estáticos da FindBugs.

Diante disso, para contornar as possíveis limitações dos resultados nos trabalhos de Filho et al. (2010) e Couto et al. (2013), é proposta, nesta dissertação, a utilização de defeitos semeados artificialmente para investigar a existência da correspondência entre defeitos e avisos estáticos. Esses defeitos artificiais são obtidos por meio da utilização da Técnica de Teste Baseada em Defeitos, em especial o critério de Análise de Mutantes (Teste de Mutação).

O Teste de Mutação é um critério de teste muito eficaz, uma vez que os operadores de mutação – responsáveis por efetuar as alterações sintáticas sobre o programa original para gerar o mutante – representam um modelo de defeitos. Tradicionalmente, o Teste de Mutação é empregado para avaliar conjuntos de casos de teste ou critérios de teste, o que o torna uma boa ferramenta para experimentação (ANDREWS; BRIAND; LABICHE, 2005).

Como ferramenta de análise estática é utilizada a FindBugs porque, além de ser uma das mais utilizadas para sistemas na linguagem Java, foi também empregada nos trabalhos de Filho et al. (2010) e Couto et al. (2013), e isso permitirá a comparação dos resultados obtidos nesta dissertação com esses trabalhos.

Os resultados obtidos nesta dissertação podem ser utilizados para avaliar a correspondência de cada tipo de aviso com cada categoria de defeito (reprentada pelos operadores de mutação). Essa correspondência pode servir como subsídio para classificar os avisos em uma ordem de prioridade, de acordo com seu grau de correspondência com mutações. De maneira prática, ao aplicar a FindBugs em um produto de software, os avisos com taxas de correspondências maiores – isto é, com maior probabilidade de ser um aviso verdadeiro positivo – deveriam ser analisados antes dos avisos com taxas de correspondências menores.

Diante do contexto apresentado, as questões de interesse desta pesquisa são:

- **Questão** Q_1: Existe alguma correspondência entre o local dos avisos estáticos e o local de mutação?
- **Questão** Q_2: Quais tipos de avisos conseguem perceber um maior/menor número de mutações, ou seja, as alterações no código produzidas nos mutantes?
- **Questão** Q_3: Quais operadores geram mutações que são mais/menos detectadas por analisadores estáticos?

Observe que uma resposta positiva para a Questão Q_1 indica que um defeito inserido artificialmente deveria ser detectado por ferramentas de análise estática e que essas ferramentas são capazes de detectar certos tipos específicos de mutações. As

respostas para a Questão Q_2 podem ser utilizadas para classificar os avisos de acordo com sua capacidade de detecção de defeitos representados pelas mutações, evitando que os avisos com baixa correspondência – ou seja, tipos de avisos que menos detectaram mutações – sejam analisados primeiros. Além disso, também podem ser identificados os tipos de mutações que são ou não detectados por ferramentas de análise estática (Questão Q_3), o que pode ser utilizado para melhorar tanto as ferramentas de análise estática, bem como o Teste de Mutação. Exemplos de melhorias que podem ser sugeridas a partir das Questões Q_2 e Q_3: a) evolução dos analisadores estáticos por meio da criação de regras de verificação estática adicionais para detectarem certas mutações (tipos de defeitos) ainda não cobertos; b) no sentido de evitar a geração (ou análise) de mutantes dos operadores cujas mutações são mais percebidas por analisadores estáticos – permitindo assim, diminuir o custo do Teste de Mutação.

Considerando o contexto apresentado, esta dissertação descreve um estudo que propõe a utilização de mutantes como modelo de defeitos para avaliar a correspondência entre mutações e avisos relatados pela ferramenta de análise estática FindBugs.

1.2 Solução Proposta

O Teste de Mutação (TM) permite a geração de um maior número de versões de um determinado sistema por meio de alterações automatizadas produzidas a partir do código fonte original. Esta geração é realizada devido às pequenas alterações sintáticas que são feitas no programa original por meio dos operadores de mutação, que simulam os defeitos mais comuns cometidos pelos desenvolvedores (DEMILLO; LIPTON; SAYWARD, 1978). Para cada alteração feita por um operador uma nova versão do sistema, chamada mutante, é gerada. Do ponto de vista teórico, cada mutante representa um possível defeito que pode estar presente no programa original (MALDONADO; DELAMARO; JINO, 2007). Os mutantes utilizados neste trabalho foram gerados por meio do conjunto de operadores implementados na ferramenta de mutação μJava, a qual suporta TM para programas na linguagem Java (MA; OFFUTT; KWON, 2005). A ferramenta μJava cria mutantes orientados a objetos para classes Java de acordo com os seus 47 operadores de mutação, que são especializados para representarem defeitos no escopo de método (MA; KWON; OFFUTT, 2002) (19 operadores) e defeitos no escopo de classe (OFFUTT; MA; KWON, 2006) (28 operadores).

O Teste de Mutação (TM) foi escolhido pelos seguintes motivos:

1. O TM é considerado um modelo de defeitos para a experimentação e é utilizado com sucesso na avaliação (e melhoria) da qualidade de conjuntos de testes (ANDREWS; BRIAND; LABICHE, 2005);

2. Os mutantes são gerados por meio dos chamados operadores de mutação, que podem ser vistos como categorias de defeitos, as quais podem ser utilizadas para que seja verificada a existência da correspondência entre tipos de avisos estáticos e mutações produzidas por tipos de operadores específicos;
3. O TM permite aumentar a taxa de defeitos por linha, ou seja, o número de defeitos gerados por linha de código fonte é maior do que a taxa de defeitos reais utilizada em trabalhos semelhantes (FILHO et al., 2010; COUTO et al., 2013);
4. O TM permite a experimentação para uma quantidade maior de produtos de software.

Esses motivos ajudam a contornar as possíveis limitações encontradas em trabalhos anteriores, relacionadas à baixa taxa de defeitos e a dificuldade de categorizar os defeitos (defeitos lógicos, aritméticos, etc) de acordo com algum critério de semelhança entre eles.

Diante do exposto, nesta dissertação é relatada uma investigação sobre a capacidade da ferramenta de análise estática FindBugs em detectar defeitos de software semeados artificialmente por meio de mutações produzidas por operadores de mutação da ferramenta μJava.

1.2.1 Objetivos

Os objetivos definidos para este trabalho são apresentados a seguir:

Objetivo Geral: O objetivo geral é estabelecer estratégias de priorização de avisos e de operadores de mutação, a partir do grau de correspondência entre avisos estáticos e mutações.

Objetivos Específicos: O objetivo geral será alcançado por meio dos seguintes objetivos específicos:

- OE_1: Identificar a existência de correspondência entre alguns tipos de avisos e mutações;
- OE_2: Identificar os avisos da FindBugs que mais detectam as mutações produzidas nos mutantes, isto é, possivelmente, os avisos verdadeiros positivos;
- OE_3: Identificar os operadores de mutação específicos cujas mutações a FindBugs é mais capaz de detectar;
- OE_4: Identificar os operadores de mutação específicos cujas mutações a FindBugs é menos capaz de detectar.

1.2.2 Metodologia

A metodologia utilizada neste trabalho é baseada em experimentação. Para alcançar o objetivo geral e os objetivos específicos, foi utilizado o processo a seguir:

1. Selecionar um conjunto S de sistemas;
2. Para cada sistema s_i do conjunto S;
   (a) Gerar os mutantes do sistema s_i;
   (b) Executar a ferramenta FindBugs sobre cada arquivo do sistema original s_i;
   (c) Executar a ferramenta FindBugs sobre cada mutante gerado;
   (d) Coletar os dados da execução da FindBugs dos itens (b) e (c), e armazenar tais dados em banco de dados;
   (e) Coletar os dados de cada mutante (operador, mutação produzida, aviso relatado, etc) e armazenar tais dados em banco de dados.
3. Calcular a correspondência entre avisos estáticos e mutações.

Desse modo foi criado um banco de dados no qual foram armazenadas informações coletadas sobre os avisos e os mutantes. Com base nos dados armazenados é possível analisar questões como: informações referentes aos operadores de mutação como, por exemplo, o ponto de mutação (trecho de código que foi alterado); informações referentes aos avisos relatados (a linha de ocorrência do aviso, o tipo de aviso específico, etc) pela FindBugs nos arquivos fontes do programa original e/ou em cada um dos mutantes; entre outras.

O passo seguinte foi a realização de dois estudos experimentais: o primeiro trata-se de um estudo exploratório, que tem por finalidade validar o arcabouço (modelo de dados e processo) por meio de alguns sistemas simples; já no segundo experimento foram utilizados sistemas mais complexos disponibilizados pela Fundação Apache.

Primeiro experimento controlado: neste experimento foram utilizados 7 pequenos programas, os quais somam 446 linhas de código, para validar o emprego de mutantes, o modelo da base de dados e o desenvolvimento de diversos *scripts* para auxiliar na coleta dos dados. Sobre o conjunto dos 7 programas foi executado o processo (geração de mutantes, execução da FindBugs sobre cada mutante e no programa original, etc.). Esses programas foram selecionados por já terem sido utilizados em experimentos anteriores (POLO; PIATTINI; GARCÍA-RODRÍGUEZ, 2009). Motivados pelos resultados obtidos a partir desses programas, o experimento foi estendido para o segundo experimento controlado.

Segundo experimento controlado: para este experimento foram selecionados 19 novos programas, os quais somam quase 750 mil linhas de código. Esses programas foram selecionados pelos seguintes motivos: a) são programas cujos códigos fontes estão

publicamente disponíveis no site e repositório da Fundação Apache[1]; b) foram utilizados em trabalhos anteriores (FILHO et al., 2010; COUTO et al., 2013), cujos resultados serviram como motivação para a realização do estudo relatado nesta dissertação. Os resultados obtidos nesse segundo experimento indicam que existe correspondência entre mutações e avisos estáticos quando são considerados alguns tipos de avisos e alguns operadores específicos.

1.2.3 Contribuições

As contribuições deste trabalho são apresentadas nos itens abaixo:

- C_1: Identificação de um conjunto de tipos de avisos da FindBugs que mais percebem a alteração produzida nos mutantes;
- C_2: Identificação de um conjunto de tipos de avisos da FindBugs que menos percebem a alteração produzida nos mutantes;
- C_3: Identificação de um conjunto de operadores de mutação da μJava cujas mutações são mais percebidas por avisos da FindBugs;
- C_4: Identificação de um conjunto de operadores de mutação da μJava cujas mutações são menos percebidas por avisos da FindBugs;
- C_5: Apresentação de uma estratégia incremental de priorização de avisos da FindBugs a partir da correspondência entre avisos e mutações;
- C_6: Apresentação de uma estratégia incremental de priorização dos operadores de mutação da μJava, com o objetivo de reduzir o custo do Teste de Mutação a partir da correspondência entre avisos e mutações.
- C_7: Criação de um arcabouço de experimentação para ser utilizado por diferentes linguagens, ferramentas de análise estática e ferramentas de mutação.

1.3 Organização da Dissertação

Neste capítulo foram apresentados o contexto, a motivação, a solução proposta, a metodologia, os objetivos gerais e específicos a serem atingidos e as contribuições do presente trabalho. O restante dessa dissertação está organizado conforme descrito a seguir:

- Capítulo 2: Apresentação dos principais conceitos e terminologia relacionados às atividades de Verificação e Validação (V & V). São apresentados os analisadores

[1] http://www.apache.org

estáticos automatizados , ferramenta de análise estática FindBugs, o Teste de Mutação e a ferramenta μJava). A finalidade do Capítulo 2 é fornecer subsídio para melhor compreensão do trabalho;

- Capítulo 3: Definição do arcabouço utilizado durante a experimentação para coletar os dados necessários para a realização do estudo proposto, além de apresentar o primeiro estudo experimental;
- Capítulo 4: Apresentação do estudo experimental I – CDL (Correspondência direta por linha) por Aviso – para analisar a correspondência entre mutações e avisos do ponto de vista dos avisos estáticos. Nesse capítulo são apresentados os tipos de avisos que mais/menos detectaram as mutações. Também apresenta um exemplo de aplicação da correspondência obtida para o tipo de aviso.
- Capítulo 5: Apresentação do estudo experimental II – CDL (Correspondência direta por linha) por Operador – para analisar a correspondência entre mutações e avisos do ponto de vista dos operadores de mutação. Nesse capítulo são apresentados os operadores cujas mutações mais/menos foram "percebidas" por avisos estáticos da FindBugs. Também apresenta um exemplo de aplicação da correspondência obtida para o operador.
- Capítulo 6: É feita uma análise dos resultados obtidos nos Capítulos 4 e 5, registros das lições aprendidas e identificação das ameaças de validação do presente estudo;
- Capítulo 7: São apresentados os principais trabalhos relacionados;
- Capítulo 8: São apresentadas as conclusões, as perspectivas de trabalhos futuros e as publicações decorrentes dos resultados obtidos do presente trabalho.

.

CAPÍTULO 2

Fundamentação Teórica

Neste capítulo são apresentados alguns conceitos e terminologia relacionados às atividades de Verificação e Validação (V & V). São apresentados a análise estática automatizada, a ferramenta de análise estática FindBugs, o Teste de Mutação e a ferramenta de mutação μJava.

2.1 Conceitos Básicos de Verificação e Validação

As atividades relacionadas à verificação e validação de software podem ser divididas em dois tipos: 1) atividades de análise dinâmica e 2) atividades de análise estática. As atividades de análise dinâmica demandam a execução do produto sendo avaliado. As diversas técnicas e critérios de teste podem ser classificados como exemplos atividades de análise dinâmica. Já as atividades de análise estática avaliam o produto em questão sem a necessidade de execução do referido produto. Revisão e inspeção são exemplos de atividades de análise estática (MALDONADO; DELAMARO; JINO, 2007)

Do ponto de vista de custo, sabe-se que quanto antes um defeito é detectado, entre as fases de desenvolvimento, mais barato é a sua respectiva correção (BOEHM; BASILI, 2001). Além disso, o fato de não exigir a execução do produto tornam atrativas as ferramentas que realizam análise estática. Outra vantagem do emprego destas ferramentas é o fato de serem consideradas de baixo custo, pois podem ser utilizadas de forma automatizada. Entretanto, essas ferramentas costumam relatar uma grande quantidade de avisos chamados avisos falsos positivos, ou seja, avisos que não correspondem a defeitos reais, mas que vão demandar tempo de investigação por parte do testador para sua análise (SHEN; FANG; ZHAO, 2011; FILHO et al., 2010; COUTO et al., 2013; JOHNSON et al., 2013).

A ISO (2010) apresenta as definições dos seguintes termos:

- **Engano (*Mistake*)**: ação humana causadora de resultado incorreto. Um engano ocorre, por exemplo, quando ao ler uma informação em um documento de requisito

ou ao desenvolver em uma determinada linguagem de programação, uma ação incorreta é tomada pelos desenvolvedores.

- **Defeito (*Fault*)**: passo, processo, ou definição de dados incorreta em um produto de software. Um comando ou instrução incorreta é um exemplo de defeito.
- **Erro (*Error*)**: diferença entre o valor computado e o valor correto de acordo com a especificação.
- **Falha (*Failure*)**: é a inabilidade do sistema ou componente em realizar a função especificada.

2.2 Análise Estática Automatizada

Um analisador estático automatizado é uma ferramenta que lê o código fonte e/ou o código objeto (no caso de Java, *bytecode*) de um determinado sistema e emite um conjunto de avisos com base em regras que observam a presença de desvios em relação a um determinado padrão de código. Os avisos são emitidos em relação a uma determinada linha de código fonte, na qual a ferramenta detecta qualquer possível defeito no que diz respeito às suas regras consideradas.

Os termos seguintes são utilizados por analisadores estáticos:

- **Falso positivo**: um falso positivo ocorre quando um aviso da ferramenta de análise estática não condiz com a real presença de um defeito no código.
- **Verdadeiro positivo**: um verdadeiro positivo ocorre quando um aviso é relatado e indica a presença real de um defeito no sistema em análise;
- **Falso negativo**: um falso negativo ocorre quando sabe-se que existe um defeito no sistema em análise, mas esse defeito não é percebido pela analisador devido ao fato de que as ferramentas de análise estática não detectam todos os defeitos (ausência de regras de detecção).

Existem várias ferramentas de análise estática disponíveis para programas desenvolvidos nas mais diferentes linguagens de programação. Entre outras ferramentas, podem ser utilizadas a StyleCop e a FxCop para programas em .NET e Lint para programas em C/C++. Para a linguagem Java, podem ser citadas a FindBugs, a PMD, a CheckStyle, dentre outras (KRESOWATY, 2008; MICROSOFT, 2014; POHL, 2001; EVANS; LAROCHELLE, 2002; HOVEMEYER; PUGH, 2004; COPELAND, 2005; BURN, 2014; DAIMI; BANITAAN; LISZKA, 2013).

Neste trabalho é utilizada a ferramenta FindBugs (HOVEMEYER; PUGH, 2004) por duas razões. A primeira porque essa ferramenta foi utilizada nos trabalhos Filho et al. (2010) e Couto et al. (2013) que serviram de inspiração para a realização do estudo objeto desta dissertação. Filho et al. (2010) e Couto et al. (2013) utilizaram a FindBugs

para avaliar a correlação entre avisos e defeitos reais (defeitos de campo reclamados por usuários). No trabalho relatado nesta dissertação, os defeitos reais são substituídos por mutantes. Para fazer futuras comparações com aqueles trabalhos, optou-se por utilizar a mesma ferramenta de análise estática. Além disso, Terra e Bigonha (2008) classificam a FindBugs como um dos analisador estático de código aberto mais eficaz, ou seja, com a menor quantidade de falsos positivos.

Ferramenta de Análise Estática FindBugs

A FindBugs é uma das ferramentas de análise estática mais populares e é amplamente utilizada na comunidade Java. Ela implementa um conjunto de detectores de defeitos para uma variedade de padrões de defeitos comuns. De acordo com a estrutura da FindBugs, cada categoria de defeito (*bug category*) possui vários tipos de defeitos (*bug kinds*) e cada tipo de defeito consiste de alguns padrões de defeitos (*bug patterns*). Os padrões de defeitos da FindBugs estão divididos nas categorias *Bad Practice, Correctness, Malicious code vulnerability, Multithreaded correctness, Internationalization, Performance, Security* e *Dodgy*.

Na Figura 2.1 são apresentados exemplos de tipos e padrões de defeitos da categoria *Correctness*. Os defeitos Eq (*equals method always returns false)*, BC (*Bad casts of object references*), NP (*Null pointer dereference*), DLS (*Dead local store*), e DMI (*Dubious method invocation*) são exemplos de tipos de defeitos da categoria *Correctness*. O tipo de defeito (*bug kind*) BC (*bad casts of object references*) contém 4 padrões de defeitos (*bug patterns*): *Impossible cast, Impossible downcast, Impossible downcast of toArray() result e instanceof will always return false*.

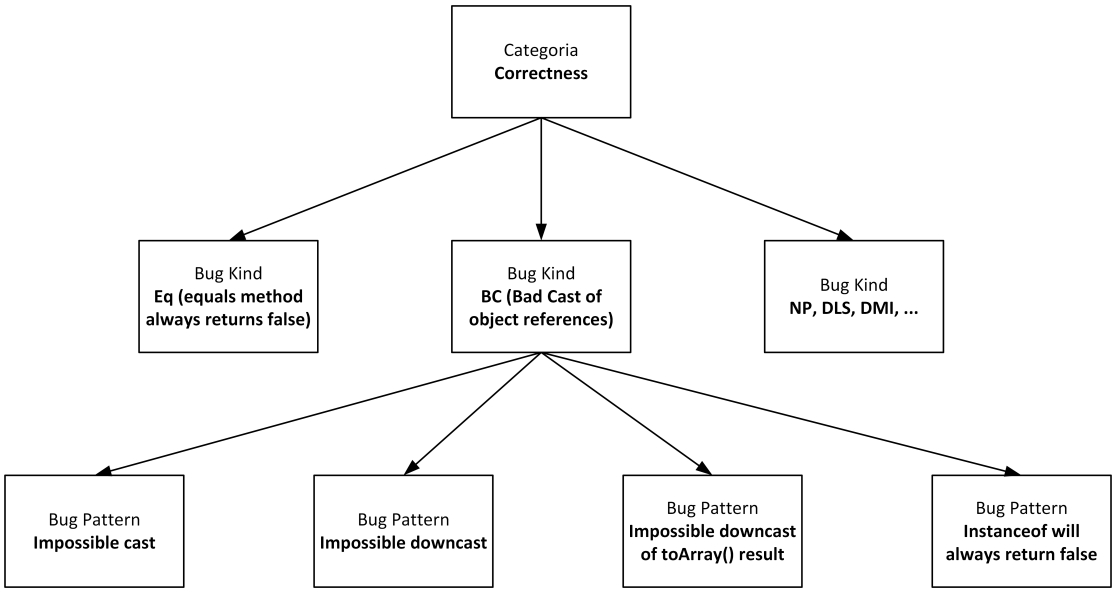

Figura 2.1: *Estrutura de Aviso da FindBugs. Adaptado de (SHEN; FANG; ZHAO, 2011)*

A versão 3.0 da FindBugs tem 9 categorias diferentes de defeitos. Cada categoria tem um conjunto de tipos de defeitos. Para cada tipo de defeito, há um conjunto de padrões de avisos. Existem cerca de 427 [1] diferentes padrões de avisos. A Tabela 2.1 apresenta a quantidade de padrões de defeitos por categoria.

Tabela 2.1: *Categorias de Avisos da FindBugs*

#	Categoria	Quant. de Avisos	Exemplo de Aviso
1	Bad practice	88	ES: Comparison of String parameteogur using == or ! =
2	Correctness	149	RV: Method ignores return value
3	Experimental	3	OBL: Method may fail to clean up stream or resource
4	Internationalization	2	Dm: Consider using Locale parameterized version of invoked method
5	Malicious code vulnerability	17	DP: Classloaders should only be created inside doPrivileged block
6	Multithreaded correctness	46	STCAL: Static DateFormat
7	Performance	32	UuF: Unused field
8	Security	11	Dm: Empty database password
9	Dodgy code	79	BC: Unchecked/unconfirmed cast
Total		427	–

Considere que os termos avisos e tipos de avisos utilizados neste trabalho referem-se ao termo padrão de defeito (*bug pattern*).

Cada aviso da FindBugs tem um nível de prioridade. Há três níveis de prioridade: 1 (mais alta), 2 (média) e 3 (baixa). Em geral, avisos de prioridade 1 significam avisos que devem ser corrigidos; os avisos de prioridade 2 representam os avisos não são tão críticos, mas que pode ser interessante de serem analisados após os de prioridade 1; e os avisos de prioridade 3, que indicam avisos relacionados a estilo de codificação e são considerados mais informativos.

Como exemplos de avisos relatados pela FindBugs podem ser citados, conforme apresentado na Tabela 2.2: o aviso CN_IMPLEMENTS_CLONE_BUT_NOT_CLONEABLE (linha 5) é um aviso do tipo CN (coluna *Bug Kind*), tem prioridade 1 (coluna P), é da categoria BAD_PRACTICE; o aviso BC_IMPOSSIBLE_CAST (linha 49) é um aviso do tipo BC, tem prioridade 1 e é da categoria CORRECTNESS; já o aviso DB_DUPLICATE_BRANCHES (linha 178) é um aviso do tipo DB, tem prioridade 2 e pertence à categoria STYLE.

2.3 Teste de Mutação

O Teste de Mutação permite gerar um número maior de versões de um determinado sistema. Essa geração é efetuada devido às pequenas alterações sintáticas que são produzidas no sistema original por meio dos operadores de mutação que simulam os enganos mais frequentes cometidos por desenvolvedores. Para cada alteração realizada

[1] Informação obtida de http://findbugs.sourceforge.net/bugDescriptions.html em 10/08/2015

Tabela 2.2: *Detalhes de Avisos Relatados pela FindBugs*

#	Bug Kind	Aviso *Bug Pattern*	P	Categoria	Descrição
1	BC	BC_EQUALS_METHOD_SHOULD_WORK_FOR_ALL_OBJECTS	2	BAD_PRACTICE	Equals method should not assume anything about the type of its argument
2	BIT	BIT_SIGNED_CHECK	3	BAD_PRACTICE	Check for sign of bitwise operation
3	CN	CN_IDIOM	2	BAD_PRACTICE	Class implements Cloneable but does not define or use clone method
4	CN	CN_IDIOM_NO_SUPER_CALL	2	BAD_PRACTICE	clone method does not call super.clone()
5	CN	CN_IMPLEMENTS_CLONE_BUT_NOT_CLONEABLE	1	BAD_PRACTICE	Class defines clone() but doesn't implement Cloneable
6	DE	DE_MIGHT_IGNORE	2	BAD_PRACTICE	Method might ignore exception
7	Dm	DM_EXIT	3	BAD_PRACTICE	Method invokes System.exit(...)
8	DMI	DMI_ENTRY_SETS_MAY_REUSE_ENTRY_OBJECTS	2	BAD_PRACTICE	Adding elements of an entry set may fail due to reuse of Entry objects
9	DMI	DMI_USING_REMOVEALL_TO_CLEAR_COLLECTION	2	BAD_PRACTICE	Don't use removeAll to clear a collection
10	Eq	EQ_CHECK_FOR_OPERAND_NOT_COMPATIBLE_WITH_THIS	1	BAD_PRACTICE	Equals checks for incompatible operand
...	...	...	...	...	...
49	BC	BC_IMPOSSIBLE_CAST	1	CORRECTNESS	Impossible cast
50	BC	BC_IMPOSSIBLE_INSTANCEOF	1	CORRECTNESS	instanceof will always return false
51	BIT	BIT_ADD_OF_SIGNED_BYTE	2	CORRECTNESS	Bitwise add of signed byte value
52	BIT	BIT_AND	1	CORRECTNESS	Incompatible bit masks
53	BIT	BIT_AND_ZZ	1	CORRECTNESS	Check to see if ((...) & 0) == 0
54	BIT	BIT_IOR	1	CORRECTNESS	Incompatible bit masks
55	DLS	DLS_DEAD_LOCAL_INCREMENT_IN_RETURN	2	CORRECTNESS	Useless increment in return statement
56	DLS	DLS_DEAD_STORE_OF_CLASS_LITERAL	2	CORRECTNESS	Dead store of class literal
57	DLS	DLS_OVERWRITTEN_INCREMENT	1	CORRECTNESS	Overwritten increment
58	DMI	DMI_INVOKING_HASHCODE_ON_ARRAY	2	CORRECTNESS	Invocation of hashCode on an array
...	...	...	...	...	...
112	Dm	DM_CONVERT_CASE	3	I18N	Consider using Locale parameterized version of invoked method
113	Dm	DM_DEFAULT_ENCODING	1	I18N	Reliance on default encoding
114	DP	DP_CREATE_CLASSLOADER_INSIDE_DO_PRIVILEGED	2	MALICIOUS_CODE	Classloaders should only be created inside doPrivileged block
115	DP	DP_DO_INSIDE_DO_PRIVILEGED	3	MALICIOUS_CODE	Method invoked that should be only be invoked inside a doPrivileged block
116	EI	EI_EXPOSE_REP	2	MALICIOUS_CODE	May expose internal representation by returning reference to mutable object
...	...	...	...	...	...
123	AT	AT_OPERATION_SEQUENCE_ON_CONCURRENT_ABSTRACTION	2	MT_CORRECTNESS	Sequence of calls to concurrent abstraction may not be atomic
124	DC	DC_DOUBLECHECK	2	MT_CORRECTNESS	Possible double check of field
125	ESync	ESync_EMPTY_SYNC	2	MT_CORRECTNESS	Empty synchronized block
126	IS	IS2_INCONSISTENT_SYNC	2	MT_CORRECTNESS	Inconsistent synchronization
127	JLM	JLM_JSR166_UTILCONCURRENT_MONITORENTER	2	MT_CORRECTNESS	Synchronization performed on util.concurrent instance
...	...	...	...	...	...
142	Bx	BX_BOXING_IMMEDIATELY_UNBOXED	2	PERFORMANCE	Primitive value is boxed and then immediately unboxed
143	Bx	BX_BOXING_IMMEDIATELY_UNBOXED_TO_PERFORM_COERCION	2	PERFORMANCE	Primitive value is boxed then unboxed to perform primitive coercion
144	Bx	BX_UNBOXING_IMMEDIATELY_REBOXED	2	PERFORMANCE	Boxed value is unboxed and then immediately reboxed
145	Dm	DM_BOOLEAN_CTOR	2	PERFORMANCE	Method invokes inefficient Boolean constructor; use Boolean.valueOf(...) instead
146	Bx	DM_BOXED_PRIMITIVE_FOR_PARSING	1	PERFORMANCE	Boxing/unboxing to parse a primitive
147	Bx	DM_BOXED_PRIMITIVE_TOSTRING	2	PERFORMANCE	Method allocates a boxed primitive just to call toString
...	...	...	...	...	...
170	HRS	HRS_REQUEST_PARAMETER_TO_HTTP_HEADER	2	SECURITY	HTTP Response splitting vulnerability
171	SQL	SQL_NONCONSTANT_STRING_PASSED_TO_EXECUTE	3	SECURITY	Nonconstant string passed to execute method on an SQL statement
172	SQL	SQL_PREPARED_STATEMENT_GENERATED_FROM_NONCONSTANT_STRING	3	SECURITY	A prepared statement is generated from a nonconstant String
173	BC	BC_BAD_CAST_TO_CONCRETE_COLLECTION	3	STYLE	Questionable cast to concrete collection
174	BC	BC_UNCONFIRMED_CAST	3	STYLE	Unchecked/unconfirmed cast
175	BC	BC_UNCONFIRMED_CAST_OF_RETURN_VALUE	3	STYLE	Unchecked/unconfirmed cast of return value from method
176	BC	BC_VACUOUS_INSTANCEOF	2	STYLE	instanceof will always return true
177	CI	CI_CONFUSED_INHERITANCE	3	STYLE	Class is final but declares protected field
178	DB	DB_DUPLICATE_BRANCHES	2	STYLE	Method uses the same code for two branches
...	...	...	...	...	...

pelo operador, é gerada uma nova versão do sistema. Essa versão alterada recebe o nome de mutante. Do ponto de vista teórico, cada mutante representa um possível defeito que pode estar presente no sistema original (MALDONADO; DELAMARO; JINO, 2007). A troca de constantes por outros valores, permutação de instruções, troca de operadores lógicos, troca de operações aritméticas são exemplos de alterações sintáticas que originam mutantes.

Na Tabela 2.3 são apresentados exemplos de mutantes de acordo com cada trecho de código original. Como pode ser visto na Tabela 2.3, diferentes mutações podem ser realizadas simulando diferentes tipos de defeitos. O ponto de mutação aparece sublinhado na Tabela 2.3. Por exemplo, na linha 4 o operador de mutação que troca o operador aritmético criou um mutante, substituindo a operação de adição pela operação de divisão: (no original) `int a = b + c;` por (no mutante) `int a = b / c;`

Tabela 2.3: *Exemplos de Mutantes*

id	**Trecho de Código Original**	**Trecho de Código Mutante**

1	`int add(int a, int b) {`
	`  return a + b;`
	`}`
		`int add(int a, int b) {`
		`  return a + b++;`
		`}`

2	`if (a == b){...}`	`if (a != b){... }`
3	`if (a == b && c == d){}`	`if (a == b || c == d){}`
4	`int a = b + c;`	`int a = b / c;`
5	`private static int sum = 0;`	`private _ int sum = 0;`

O Teste de Mutação permite aumentar a taxa de concentração de defeitos por linha com o objetivo de investigar a correspondência entre defeitos e avisos estáticos. Observe que, nesse caso, os defeitos são representados por mutantes gerados por operadores de mutação. A opção por utilizar o Teste de Mutação baseia-se no fato de que este já demonstrou ser um modelo de defeitos reais, conforme estudo de Andrews, Briand e Labiche (2005). Além disso, cada operador de mutação pode ser compreendido como uma representação de determinada classe de defeitos, de modo que é possível utilizá-lo para se verificar quais dessas classes as ferramentas de análise estática são mais/menos sensíveis em detectar.

O Teste de Mutação (TM) foi criado por pesquisadores do Georgia Institute of Technology e Yale University na década 1970. Um dos primeiros artigos sobre o Teste de Mutação foi de DeMillo, Lipton e Sayward (1978). Eles apresentaram esta técnica baseando-se em duas hipóteses: Hipótese do Programador Competente e no Efeito de Acoplamento. A **Hipótese do Programador Competente** baseia-se no fato de que um programador competente desenvolve programas que estão corretos ou "próximos" do correto. Já a **Hipótese do Efeito de Acoplamento** baseia-se no fato de que defeitos complexos são compostos por defeitos simples, que são inseridos no código por meio de enganos simples cometidos por desenvolvedores.

O Teste de Mutação fornece ao testador uma abordagem sistematizada para dois cenários: 1) avaliar a qualidade de um determinado conjunto de teste já existente; 2) permite melhorar um conjunto de teste (DEMILLO; LIPTON; SAYWARD, 1978). Considere S uma especificação e P um programa que é uma implementação de S cuja qualidade deseja-se avaliar. Considere D como sendo o domínio de entrada de P e $T \subset D$ um conjunto de teste para P.

Basicamente, a aplicação do Teste de Mutação consiste em quatro etapas: geração de mutantes, execução de casos de teste sobre o programa original, execução de casos de teste sobre os mutantes e, por fim, análise de mutantes vivos. Um resumo de cada uma dessas etapas é apresentado a seguir:

1^o Etapa: Geração de Mutantes

A fase de geração dos mutantes é uma das mais importantes, uma vez que os mutantes são responsáveis por avaliar a qualidade de um dado conjunto de teste já existente e/ou serve também como subsídio para gerar novos conjuntos de teste. Um conjunto de implementações alternativas contendo alterações sintáticas simples é criado. As alterações sintáticas são modeladas pelos operadores de mutação, os quais podem ser vistos como um modelo de defeitos que representam os enganos mais comuns cometidos pelos programadores durante o desenvolvimento de software. Geralmente, quanto mais operadores de mutação são empregados, mais mutantes podem ser gerados, o que aumenta o custo da utilização do Teste de Mutação. Os operadores de mutação são dependentes da linguagens de programação, pois cada uma dessas linguaguem possuem paradigmas e estruturas sintáticas diferentes.

2^o Etapa: Execução do Programa Original

Nesta fase, o programa P é executado com o conjunto de teste T já existente. Neste cenário duas situações podem ocorrer:

1. Caso exista $t \in T$ tal que $P(t) \neq S(t)$ então uma falha foi detectada, e P deve ser corrigido. Considere que $P(t)$ é a saída de P com entrada t, e que $S(t)$ é o valor correto para t de acordo com S.
2. Caso para qualquer caso de teste $t \in T$, $P(t) = S(t)$, considera-se que P corresponde a S para o conjunto de teste T. Nesse caso, é necessário questionar a qualidade de T confrontando a sua capacidade em "matar" mutantes.

3^o Etapa: Execução dos Mutantes

Seja M o conjunto de mutantes gerados na primeira etapa. Para cada $m \in M$ e cada $t \in T$, $m(t)$ é comparado com $P(t)$. Se houver um $t \in T$ tal que $m(t) \neq P(t)$, m é dito ser "morto" por t. Por outro lado, quando $m(t) = P(t)$ para todo $t \in T$, m é dito estar "vivo" e deve ser analisado no passo seguinte. Após este passo, o escore de mutação é calculado. O **escore de mutação** é a razão entre o número de mutantes mortos e o número de mutantes não-equivalentes e fornece ao testador uma métrica para avaliar a qualidade da atividade de teste. Deste modo, o escore de mutação pode ser usado para avaliar se os objetivos do teste foram atingidos. Se o escore de mutação atingir $1,0$ (100%), diz-se que T é adequado em relação a P, isto é, T é $TM-adequado$ para P. Uma vez que a identificação de mutantes equivalentes é importante para calcular o escore de mutação, um valor de $1,0$ é obtido somente após a análise de todos os mutantes vivos. O escore de mutação é obtido por meio da Função 2-1.

4^o Etapa: Análise de Mutante

Um mutante m pode permanecer vivo depois de ser executado com T ou porque m é equivalente a P ($m \equiv P$) e vai sempre se comportam de forma idêntica a P, para todo $t \in D$; ou porque T não é bom o suficiente para mostrar a diferença no comportamento de m e P, e existe um $t' \in (D \backslash T)$ tal que $m(t') \neq P(t')$. Se o mutante m for equivalente, então m pode ser ignorado. Caso contrário, o testador pode desenvolver novos casos de testes para evidenciar a diferença no comportamento do mutante vivo m e o programa P.

Fórmula de Escore de Mutação (*mutation score*):

$$ms(P,T) = \frac{DM(P,T)}{M(P) - EM(P)} \tag{2-1}$$

Sendo:

- $DM(P,T)$: número total de mutantes de P mortos pelo conjunto de teste T;
- $M(P)$: número total de mutantes gerado a partir do programa P;
- $EM(P)$: número de mutantes equivalente.

Observe que no cálculo do escore de mutação apenas $DM(P,T)$ depende do conjunto de teste utilizado. $EM(P)$ é obtido à medida que o testador decide que determinado mutante m vivo é equivalente a P.

Ferramenta de Mutação μJava

Existem várias ferramentas de mutação disponíveis para sistemas em Java (MA; OFFUTT; KWON, 2005; MAJOR, 2015; PITEST, 2015). Neste trabalho são utilizados os operadores de mutação implementados pela ferramenta μJava, que suporta Teste de Mutação para sistemas em Java (MA; OFFUTT; KWON, 2005).

A ferramenta μJava cria mutantes para sistemas em Java de acordo com seus 47 tipos de operadores de mutação que são divididos em dois tipos: operadores cujas mutações são produzidas no nível de método (MA; OFFUTT, 2005) e operadores cujas mutações são produzidas no nível de classe (OFFUTT; MA; KWON, 2006). O primeiro grupo tem 19 operadores que estão apresentados na Tabela 2.4. Já o segundo grupo tem 28 operadores que estão apresentados na Tabela 2.5.

Cada operador de mutação da μJava tem um acrônimo para sua identificação. A primeira letra da sigla determina a característica da linguagem relacionada ao operador. Por exemplo, o operador de nível de método AORB significa "*Arithmetic Operator Replacement (Binary)*" e é um dos operadores do grupo aritmético (A = *Arithmetic*). O número reduzido de operadores de método implementado pela μJava é devido à

abordagem seletiva adotada para criar tal conjunto de mutantes (OFFUTT et al., 1996). Como apresentado na Tabela 2.4, as categorias dos operadores de método são: *Arithmetic*, com 6 operadores; *Relational*, com 1 operador; *Conditional*, com 3 operadores; *Shift*, com 1 operador; *Logical*, com 3 operadores; já as categorias *Assignment, Variable, Constant, Operator* e *Statement operators* possuem 1 operador cada uma.

Tabela 2.4: *Operadores de Mutação de Método da μJava (adaptado de (MA; OFFUTT, 2005))*

Característica da Linguagem	Operador de Mutação	Descrição
Arithmetic (6)	AORB	Arithmetic Operator Replacement (Binary)
	AORS	Arithmetic Operator Replacement (Short-cut)
	AOIU	Arithmetic Operator Insertion (Uniry)
	AOIS	Arithmetic Operator Insertion (Short-cut)
	AODU	Arithmetic Operator Deletion (Uniry)
	AODS	Arithmetic Operator Deletion (Short-cut)
Relational (1)	ROR	Relational Operator Replacement
Conditional (3)	COR	Conditional Operator Replacement
	COD	Conditional Operator Deletion
	COI	Conditional Operator Insertion
Shift (1)	SOR	Shift Operator Replacement
Logical (3)	LOR	Logical Operator Replacement
	LOI	Logical Operator Insertion
	LOD	Logical Operator Deletion
Assignment (1)	ASRS	Short-cut Assignment Operator Replacement
Statement (1)	SDL	Statement Deletion
Variable (1)	VDL	Variable Deletion
Constant (1)	CDL	Constant Deletion
Operator (1)	ODL	Operator Deletion

Os operadores de classe implementados pela μJava são desenvolvidos com base em recursos de linguagem orientada a objetos e características específicas de Java. Como apresentado na Tabela 2.5, as características de orientação a objetos implementadas são: *Inheritance, Polymorphism e Overloading*, com 8, 7 e 3 operadores de mutação por característica, respectivamente. Há ainda mais 10 operadores: 6 operadores de mutação específicos de Java e 4 operadores relacionados aos enganos mais comuns de programação

Para mais informações sobre o conjunto de operadores de mutação da μJava e exemplos de mutações que eles realizam, o leitor interessado pode conferir em (MA; OFFUTT, 2005; MA; OFFUTT; KWON, 2005)

2.4 Considerações Finais do Capítulo

Nesse capítulo foram apresentados os principais conceitos relacionados à atividade de Verificação e Validação, à análise estática automatizada e ao Teste de Mutação. A ferramenta de análise estática FindBugs a ferramenta de mutação μJava foram apresentadas. No Capítulo 3 é apresentado o arcabouço de experimentação utilizado no estudo para investigar a correspondência entre mutações e avisos estáticos.

Tabela 2.5: *Operadores de Mutação de Classe da µJava (adaptado de (OFFUTT; MA; KWON, 2006))*

Característica da Linguagem	Operador de Mutação	Descrição
Inheritance (8)	IHD	Hiding variable deletion
	IHI	Hiding variable insertion
	IOD	Overriding method deletion
	IOP	Overriding method calling position change
	IOR	Overriding method rename
	ISI	`super` keyword insertion
	ISD	`super` keyword deletion
	IPC	Explicit call of a parent's constructor deletion
Polymorphism (7)	PNC	`new` method call with child class type
	PMD	Instance variable declaration with parent class type
	PPD	Parameter variable declaration with child class type
	PCI	Type cast operator insertion
	PCD	Type cast operator deletion
	PCC	Cast type change
	PRV	Reference assignment with other comparable type
Overloading (3)	OMR	Overloading method contents change
	OMD	Overloading method deletion
	OAN	Argument number change
Java-Specific (6)	JTI	`this` keyword insertion
	JTD	`this` keyword deletion
	JSI	`static` modifier insertion
	JSD	`static` modifier deletion
	JID	Member variable initialization deletion
	JDC	Java-supported default constructor creation
Common Programming Mistakes (4)	EOA	Reference assignment and content assignment replacement
	EOC	Reference comparison and content comparison replacement
	EAM	Accessor method change
	EMM	Modifier method change

CAPÍTULO 3

Arcabouço de Experimentação

Este capítulo descreve o arcabouço de experimentação utilizado para investigar a correspondência entre mutações e avisos estáticos. Inicialmente, é apresentada uma visão geral dos conceitos utilizados no arcabouço. Em seguida, são identificados os dados necessários para execução do experimento proposto, a apresentação do processo de coleta dos dados e a validação por meio de um primeiro experimento controlado.

3.1 Visão Geral

Considere $S = \{s_1, s_2, \ldots, s_t\}$ um conjunto de t sistemas e $s_x \in S$ um sistema sob análise para $1 \leq x \leq t$. O sistema s_x é composto por um conjunto de arquivos fontes $F = \{f_1, f_2, \ldots, f_{n_x}\}$ sendo n_x o número de arquivos fontes de s_x.

Uma ferramenta de análise estática como, por exemplo, a FindBugs, quando executada sobre cada arquivo original $f_j \in s_x$, $1 \leq j \leq n_x$, permite definir os seguintes conjuntos de avisos:

- Wf_j é o conjunto de avisos relatados no arquivo original f_j. Tais avisos podem ser analisados nos níveis de linha, método, classe, arquivo e sistema, como mostrado na Figura 3.1. A FindBugs relata seus avisos em determinada linha (ou intervalo de linhas) de um arquivo fonte. Desse modo, como se trata de um sistema orientado a objetos, os avisos referentes a determinadas linhas podem ser agrupados, sendo possível identificar quais avisos foram relatados em determinado método, classe ou arquivo, possibilitando a análise em diferentes níveis de granularidade.
- $Ws_x = \{Wf_1 \cup Wf_2 \cup \ldots \cup Wf_n\}$ é o conjunto de avisos relatados para o sistema s_x completo. Assim, Ws_x é a união dos avisos Wf_j relatados em cada um dos arquivos $f_j \in s_x$.

Para cada arquivo original $f_j \in s_x$ são gerados mutantes por meio dos operadores da ferramenta μJava. Considere $m_i o_k f_j$ o i-ésimo mutante gerado pelo operador o_k sobre o arquivo f_j. A ferramenta μJava possui 47 operadores, portanto $1 \leq k \leq 47$. Considerando

o operador de mutação o_k e o arquivo f_j objeto de mutação, podem ser definidos os seguintes conjuntos referentes aos mutantes:

- $Mo_k f_j = \{m_1 o_k f_j, m_2 o_k f_j, \ldots, m_{l_k} o_k f_j\}$ é o conjunto de mutantes gerados da aplicação do operador o_k sobre f_j, dando origem a l_k mutantes. Por exemplo: $Mo_1 f_1 = \{m_1 o_1 f_1, m_2 o_1 f_1, \ldots, m_{l_1} o_1 f_1\}$ é o conjunto de l_1 mutantes gerados da aplicação do operador o_1 sobre o arquivo $f_1 \in s_x$.
- $Mo_k = \{Mo_k f_1 \cup Mo_k f_2 \cup \ldots \cup Mo_k f_n\}$ é o conjunto de mutantes gerados da aplicação do operador o_k sobre todos os arquivos de s_x. Por exemplo: $Mo_1 = \{Mo_1 f_1 \cup Mo_1 f_2 \cup \ldots \cup Mo_1 f_n\}$ é o conjunto de mutantes gerados da aplicação do operador o_1 em todos os arquivos de s_x.
- $Mf_j = \{Mo_1 f_j \cup Mo_2 f_j \cup \ldots \cup Mo_k f_j\}$ é o conjunto de mutantes gerados a partir do arquivo f_j, considerando todos os operadores de mutação.
- $Ms_x = \{Mo_1 \cup Mo_2 \cup \ldots \cup Mo_k\}$ é o conjunto de mutantes gerados de todos os operadores $(1 \leq k \leq 47)$ sobre todos os arquivos f_j. De forma equivalente, tem-se $Ms_x = \{Mf_1 \cup Mf_2 \cup \ldots \cup Mf_n\}$: união dos conjuntos de mutantes gerados de cada arquivo f_j.

Com a execução da ferramenta FindBugs sobre cada mutante $m_i o_k f_j$, podem ser definidos os seguintes conjuntos de avisos relatados nos mutantes:

- $Wm_i o_k f_j$ é o conjunto de avisos relatados no mutante $m_i o_k f_j$.
- $WMo_k f_j = \{Wm_1 o_k f_j \cup Wm_2 o_k f_j \cup \ldots \cup Wm_{l_k} o_k f_j\}$ é o conjunto de avisos relatados nos l_k mutantes do operador o_k sobre o arquivo f_j.
- $WMo_k = \{WMo_k f_1 \cup WMo_k f_2 \cup \ldots \cup WMo_k f_n\}$ é o conjunto de avisos relatados em todos os mutantes do operador o_k em todos os arquivos $f_j \in s_x$.
- $WMf_j = \{WMo_1 f_j \cup WMo_2 f_j \cup \ldots \cup WMo_k f_j\}$ é o conjunto de avisos relatados em todos os mutantes do arquivo f_j, considerando todos os operadores de mutação.
- $WMs_x = \{WMf_1 \cup WMf_2 \cup \ldots \cup WMf_n\}$ é o conjunto dos avisos relatados nos mutantes de todos os arquivos $f_j \in s_x$, considerando todos os operadores de mutação.

As definições de conjuntos de mutantes, de avisos relatados nos arquivos originais e de avisos relatados nos mutantes estão ilustradas na Figura 3.1.

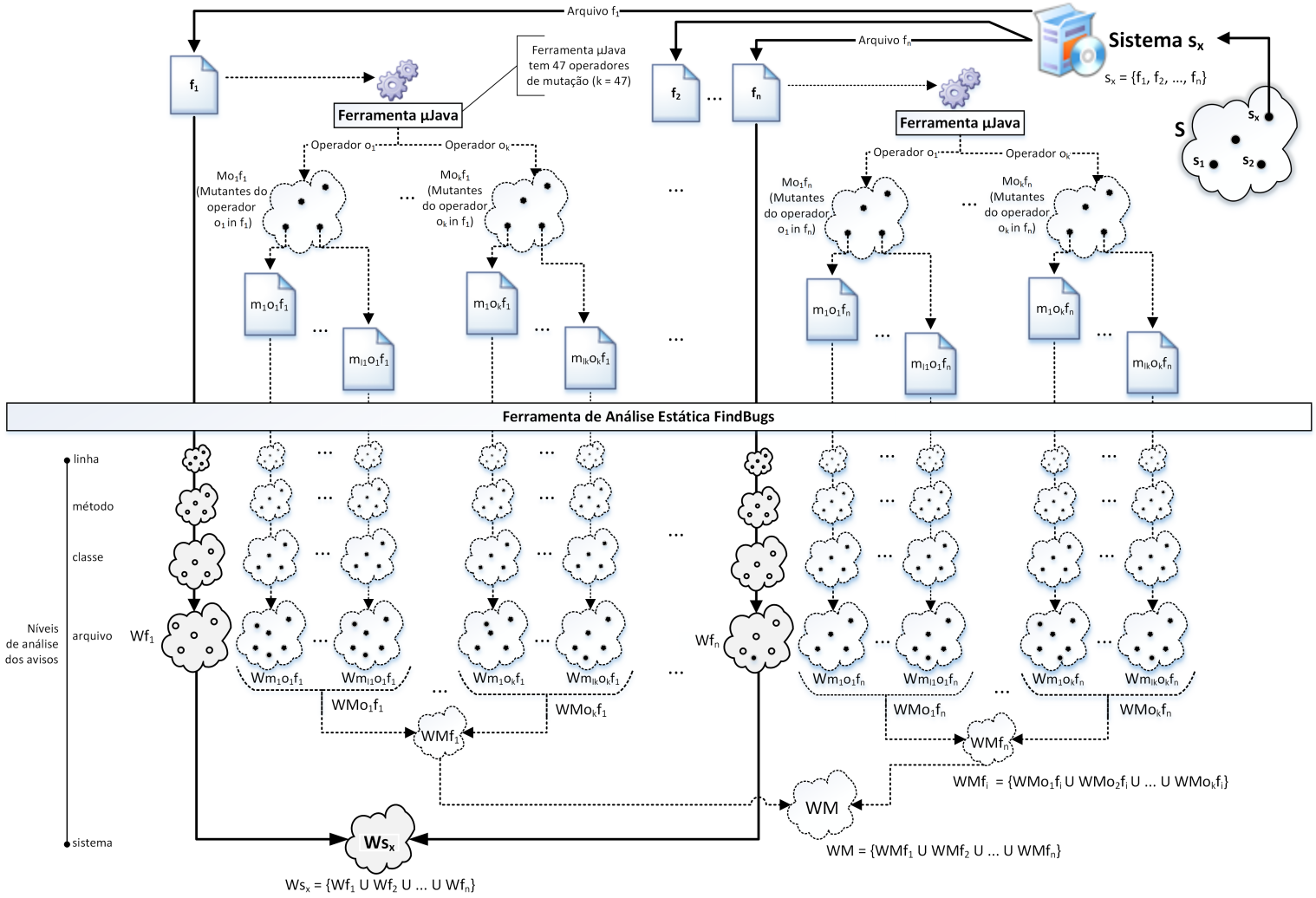

Figura 3.1: *Visão geral da estrutura comparativa entre avisos relatados em cada arquivo original f_j e em cada mutante $m_io_kf_j$, nos níveis de linha, método, classe, arquivo e sistema.*

Da execução da ferramenta FindBugs no arquivo original f_j e no mutante $m_i o_k f_j$ são relatados os conjuntos de avisos Wf_j e $Wm_i o_k f_j$, respectivamente. Tais conjuntos podem ser comparados nos níveis de linha, método, classe, arquivo e sistema, conforme ilustrado na Figura 3.1.

Nas Figuras 3.2, 3.3 e 3.4 são ilustradas situações que ocorrem quando um aviso w é relatado pela FindBugs no arquivo original f_j (dando origem ao conjunto de avisos Wf_j) e/ou no mutante $m_i o_k f_j$ (dando origem ao conjunto de avisos $Wm_i o_k f_j$);

Considere os avisos w_1, w_2 e $w_3 \in Wm_i o_k f_j \cup Wf_j$, de acordo com as Figuras 3.2, 3.3 e 3.4.

Caso 1: w_1 é um aviso que foi relatado no mutante $m_i o_k f_j$ ($w_1 \in \{Wm_i o_k f_j - Wf_j\}$) e que não foi relatado no arquivo original f_j, considerando a mesma linha na qual ocorreu a mutação. Desse modo, w_1 é um aviso verdadeiro positivo, pois ele indica a linha da mutação produzida no mutante. Esse cenário está ilustrado na Figura 3.2.

Caso 2: w_2 é um aviso que foi relatado no mutante $m_i o_k f_j$ ($w_2 \in \{Wm_i o_k f_j \cap Wf_j\}$), na mesma linha na qual ocorreu a mutação, porém w_2 também foi relatado no arquivo original f_j. Desse modo, o aviso w_2 independe da mutação produzida no mutante. Esse cenário está ilustrado na Figura 3.3.

Caso 3: w_3 é um aviso que foi relatado no arquivo original f_j ($w_3 \in \{Wf_j - Wm_i o_k f_j\}$), e que w_3 não foi relatado no mutante $m_i o_k f_j$, considerando a linha na qual ocorreu a mutação. Desse modo, a alteração produzida no mutante "corrigiu" o aviso relatado no arquivo original f_j. Esse cenário está ilustrado na Figura 3.4.

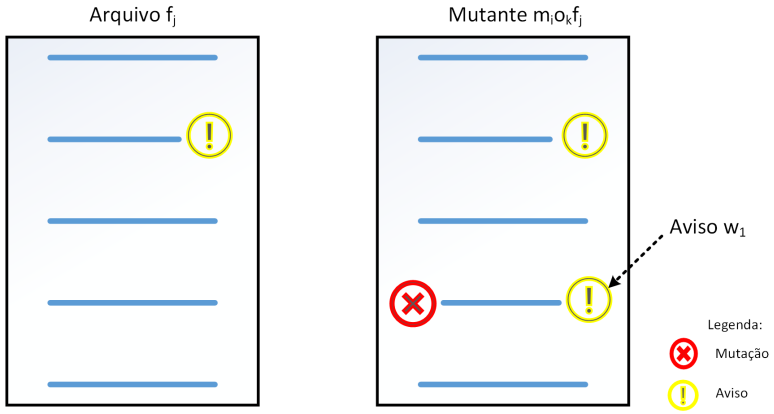

Figura 3.2: *Aviso w_1 relatado no mutante $m_i o_k f_j$ em virtude da mutação*

Como exemplo do Caso 1, no mutante apresentado na Figura 3.5(b) foi relatado o aviso w_1 na linha 4, na qual foi feita a mutação, e w_1 não foi relatado no código original correspondente (Figura 3.5(a)). A remoção do palavra chave *this* na linha 4 do código

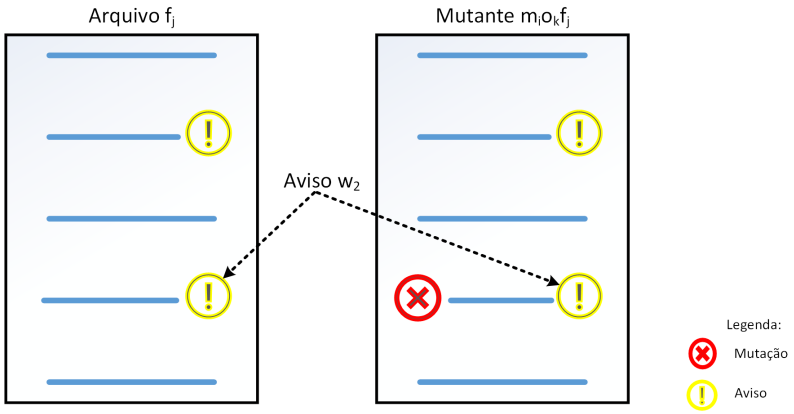

Figura 3.3: *Aviso w_2 relatado no mutante e no arquivo original, na mesma linha da mutação*

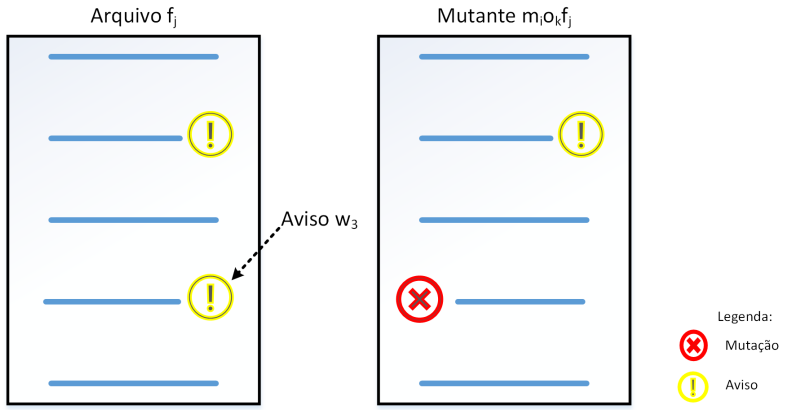

Figura 3.4: *Aviso w_3 relatado no arquivo original deixou de ser relatado no mutante*

original (`this.name = name;`) produziu no mutante a mutação `name = name;`, que foi percebida pela FindBugs por meio do aviso w_1.

Como exemplo do Caso 2, no mutante apresentado na Figura 3.6(b) foi relatado o mesmo aviso w_2 do arquivo original associado (Figura 3.6(a)). No código original e no código mutante ocorrem avisos em virtude da presença de código que nunca será executado (inalcançável). O fato de `value = false;` na linha 3 do código original faz com que a expressão booleana `if (value == true)` (linha 4) nunca seja avaliada como verdadeira, e isso torna o código da linha 5 inalcançável. De forma equivalente, a alteração produzida no mutante com a mutação – alterando a constante na linha 3 para `value = true;` – faz com que a expressão booleana `if (value == true)` (linha 4) seja sempre avaliada como verdadeira, e isso torna o código da linha 7 inalcançável no mutante.

Como exemplo do Caso 3, no mutante apresentado na Figura 3.7(b) não foi relatado o aviso w_3 que foi relatado no arquivo original correspondente (Figura 3.7(a)). Em Java um objeto da classe String é imutavél. Por isso, um aviso w_3 é relatado no código

original `name.toUpperCase();` na linha 4. Em virtude da mutação ter "removido" tal linha de código, o aviso w_3 em questão deixou de ser relatado.

```
1 public class Example1 {
2     private String name;
3     public void setName(String name){
4         this.name = name;
5     }
6 }
```

(a) *Código original 1*

```
1 public class Mutant1 {
2     private String name;
3     public void setName(String name){
4         name = name;//this.name = name;
5     }
6 }
```

(b) *Código mutante 1*

Figura 3.5: *Exemplo do Caso 1*

```
1  public class Example2 {
2      public void printValue() {
3          final boolean value = false;
4          if (value == true) {
5              System.out.println("value is true");
6          } else {
7              System.out.println("value is false");
8          }
9      }
10 }
```

(a) *Código original 2*

```
1  public class Mutant2 {
2      public void printValue() {
3          final boolean value = true;//false
4          if (value == true) {
5              System.out.println("value is true");
6          } else {
7              System.out.println("value is false");
8          }
9      }
10 }
```

(b) *Código mutante 2*

Figura 3.6: *Exemplo do Caso 2*

```
1 public class Example3 {
2     public void nameToUpperCase() {
3         String name = "string";
4         name.toUpperCase();
5     }
6 }
```

(a) *Código original 3*

```
1 public class Mutant3 {
2     public void nameToUpperCase() {
3         String name = "string";
4         //name.toUpperCase();
5     }
6 }
```

(b) *Código mutante 3*

Figura 3.7: *Exemplo do Caso 3*

Nesta dissertação, é investigado o cenário apresentado no Caso 1, onde $w_1 \in Wm_i o_k f_j$ e $w_1 \notin Wf_j$. Com isso, pode-se encontrar um conjunto de avisos capazes de detectarem tipos específicos de mutações. Esse resultado pode ser utilizado como subsídio para criar uma ordem de priorização de avisos a partir da capacidade da ferramenta de análise estática FindBugs em detectar as mutações. Tais avisos provavelmente sejam **verdadeiros positivos**. Casos 2 e 3 também são interessantes de serem investigados, mas estão fora do escopo do presente trabalho.

Filho et al. (2010) e Couto et al. (2013) utilizam o termo correlação estática quando o aviso e o defeito de campo[1] estão relativamente próximos: na mesma classe ou no mesmo método. Para esses autores, a correlação estática é feita no nível de classe e método. Isto é, se o aviso e o defeito pertencerem à mesma classe ou ao mesmo método que teve uma manutenção corretiva, então considera-se que há correlação estática entre eles.

[1]Defeito de campo foi definido como sendo um defeito reclamado pelo usuário.

Nesta dissertação avalia-se a correspondência no nível de linha de código. Essa correspondência é definida aqui como **Correspondência Direta por Linha** (*CDL*).

Neste sentido, considera-se que há correspondência direta por linha entre o aviso e a mutação, se ambos estiverem no mesmo ponto (linha) no código fonte. Assim, a análise é feita, neste trabalho, procurando verificar se o aviso e a mutação estão presentes no mesmo ponto do código fonte. Com isso, são identificados os tipos de mutações que os avisos são capazes de detectar com maior precisão, uma vez que cada mutante é gerado por um operador específico pode-se identificar os tipos de operadores cujas mutações os analisadores estáticos são mais/menos capazes de detectar.

3.2 Identificação dos Dados Necessários

O passo inicial foi identificar as informações necessárias para investigar o estudo proposto. Tais informações incluem dados relativos aos mutantes gerados pela μJava e aos avisos relatados pela FindBugs nos arquivos originais e nos mutantes.

Em resumo, os dados coletados devem permitir responder às seguintes questões:

- Dado um mutante *m*:
  - Qual é o operador de mutação que gerou *m*?
  - Qual é o ponto (linha) em que ocorre a mutação em cada mutante *m*?
  - Qual é a diferença de código entre cada arquivo original e cada um dos seus respectivos mutantes *m*?
- Dado um aviso *w*:
  - O aviso *w* é relatado em que ponto no código (linha inicial, linha final)?
  - O aviso *w* é relatado no mutante?
  - O aviso *w* é relatado exatamente no ponto de mutação?
  - O aviso *w* é relatado exclusivamente devido a mutação produzida?
  - O aviso *w* é relatado no programa original ou somente no mutante?

A partir de tais questões, foi desenvolvido o modelo de domínio mostrado na Figura 3.8. Procurou-se criar um modelo genérico o suficiente para que possa ser utilizado também por outras ferramentas de análise estática (além da FindBugs), outras ferramentas de mutação (além da μJava) e ser utilizado com defeitos reais (e não somente com mutantes).

O diagrama de classe da UML (*Unified Modeling Language*) apresentado na Figura 3.8 tem as seguintes informações:

- Um programa possui os atributos *id*, nome, versão, descrição e linguagem;

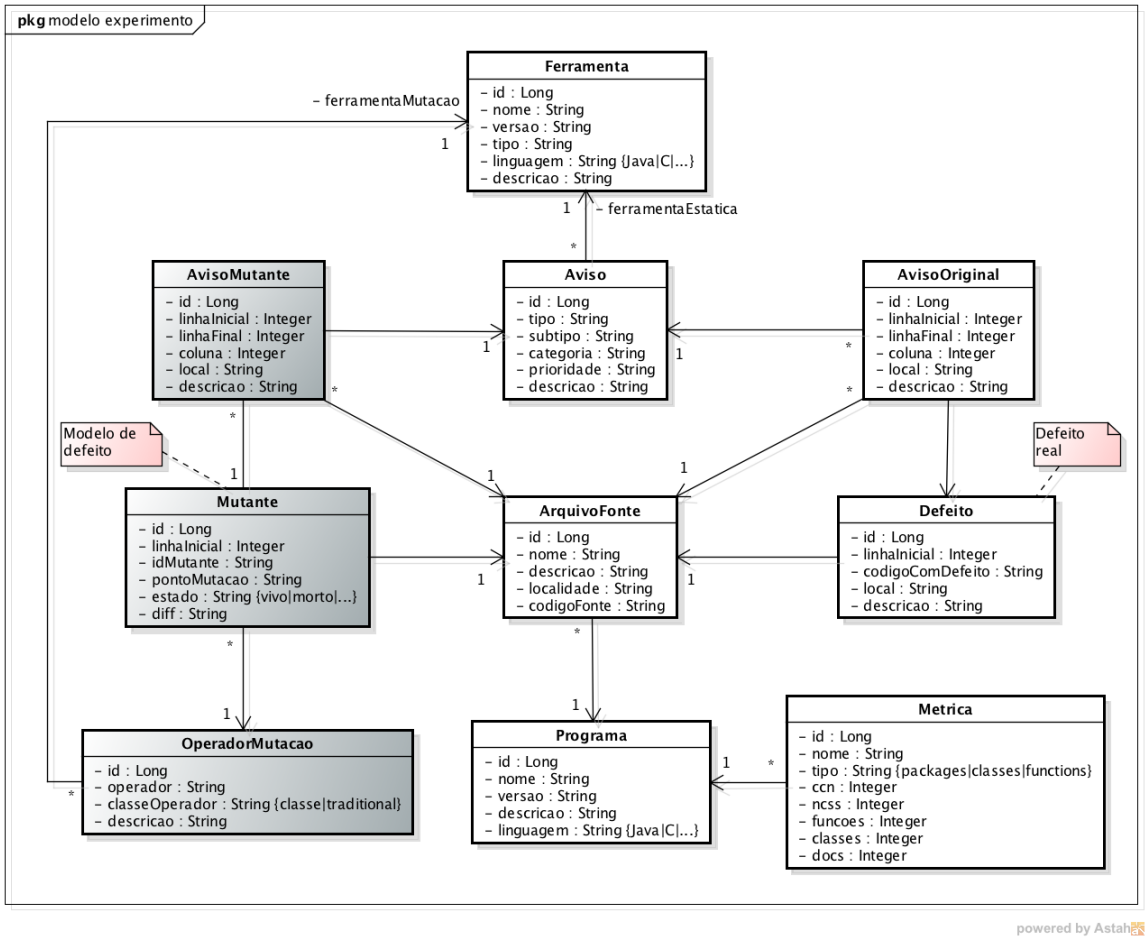

Figura 3.8: *Modelo de Domínio Utilizado no Experimento.*

- Um arquivo fonte (classe ArquivoFonte) está associado a um programa e tem os seguintes atributos: *id*, nome, descrição, código fonte e localidade;
- Um aviso representa informações dos avisos relatados pelas ferramentas de análise estática. Possui os seguintes atributos: *id*, tipo, subtipo, categoria, prioridade e descrição, e está associada a uma ferramenta de análise estática (classe Ferramenta);
- A classe Ferramenta representa as ferramentas utilizadas (de análise estática e de mutação) no experimento. Possui os seguintes atributos: *id*, nome, versão, tipo, linguagem e descrição;
- A classe AvisoOriginal representa cada aviso relatado em um arquivo original. Possui os seguintes atributos: *id*, linha inicial, linha final, coluna, local, descrição, e está associada obrigatoriamente a um tipo de aviso (classe Aviso), um arquivo fonte (classe ArquivoFonte) e a um defeito (classe Defeito);
- A classe Defeito representa um defeito real (defeito reclamado por usuário). Possui os seguintes atributos: *id*, linha inicial, trecho de código com defeito, local e descrição, e está associada a um arquivo fonte (classe *ArquivoFonte*);

- A classe OperadorMutacao representa os operadores de mutação de uma ferramenta de mutação (associação com a classe Ferramenta). Possui os seguintes atributos: *id*, operador, classe do operador e descrição;
- A classe Mutante representa informações de mutante. Possui os seguintes atributos: *id*, linha inicial, código do mutante, ponto de mutação, estado do mutante (vivo, morto, equivalente), diferença de código entre o mutante e o original, e está associado a um operador de mutação (classe OperadorMutacao);
- A classe AvisoMutante representa cada aviso relatado nos mutantes. Possui os seguintes atributos: *id*, linha inicial, linha final, coluna, local e descrição, e está associada a um tipo de aviso (classe Aviso), um arquivo fonte (classe ArquivoFonte) e a um mutante (classe Mutante);
- A classe Metrica representa as informações sobre a complexidade dos sistemas, como a complexidade ciclomática, a quantidade de linhas, entre outras.

Observe que as classes destacadas (sombreadas) na Figura 3.8 (AvisoMutante, Mutante e OperadorMutacao) são as classes criadas exclusivamente devido ao emprego do Teste de Mutação neste trabalho.

3.3 Processo Utilizado

O processo usado para coletar os dados necessários no experimento está ilustrado na Figura 3.9 e descrito a seguir:

1. Considere S um conjunto de sistema, ou seja, $S = \{s_1, s_2, \ldots, s_t\}$
2. Para cada sistema $s_x \in S, 1 \leq x \leq t$
   (a) Para cada arquivo fonte $f_j \in s_x, 1 \leq j \leq n = |s_x|$
      i. Execução da FindBugs no arquivo f_j e geração do arquivo Wf_j correspondente ao conjunto de avisos relatados no arquivo original f_j
      ii. Execução do *ParserXMLFindBugs* para analisar o arquivo Wf_j e salvar os dados relacionados aos avisos do arquivo original f_j no banco de dados
      iii. Execução da ferramenta μJava em cada f_j e geração dos mutantes $m_i o_k f_j$
      iv. Para cada mutante $m_i o_k f_j \in Mf_j$
         A. Execução da FindBugs no mutante $m_i o_k f_j$ e geração do arquivo $Wm_i o_k f_j$ correspondente ao conjunto de avisos relatados no respectivo mutante
         B. Execução do *ParserXMLFindBugs* para analisar o arquivo $Wm_i o_k f_j$ e salvar os dados dos avisos do mutante $m_i o_k f_j$ no banco de dados
         C. A mutação produzida, obtida da diferença textual entre f_j e $m_i o_k f_j$ pelo *ParserDiff*, é armazenada no banco de dados

3. Criação e execução de *scripts* SQL (*Structured Query Language*) para analisar a correspondência direta por linha de S

O processo utilizado para coletar os dados no experimento é ilustrado na Figura 3.9. A ferramenta de análise estática FindBugs é executada sobre cada um dos arquivos $f_j \in s_x$. O resultado de cada execução, ou seja, os avisos Wf_j relatados em f_j, é armazenado em arquivo no formato XML (*eXtensible Markup Language*). As informações do arquivo XML são analisadas pelo *ParserXMLFindBugs*, que armazena tais informações no banco de dados.

No passo seguinte são gerados mutantes do arquivo f_j por meio da aplicação dos operadores de mutação da ferramenta μJava. O operador de mutação o_k aplicado em cada arquivo f_j gera um conjunto de mutantes Mo_kf_j, os quais podem ser identificados como $m_io_kf_j$ (o i-ésimo mutante do operador o_k sobre f_j). A ferramenta FindBugs é, então, executada sobre o mutante $m_io_kf_j$. O resultado dessa execução, ou seja, os avisos $Wm_io_kf_j$ relatados em $m_io_kf_j$, é armazenado em arquivo XML, que também é analisado pelo *ParserXMLFindBugs*, que armazena os dados de $Wm_io_kf_j$ no banco de dados. A alteração produzida no mutante $m_io_kf_j$, em relação ao arquivo original f_j, é obtida pelo *ParserDiff*, que calcula a diferença textual entre o arquivo original f_j e o mutante $m_io_kf_j$, indicando inclusive a linha na qual ocorreu a mutação.

A diferença entre o programa original e o mutante é coletada e armazenada no banco de dados para identificar a mutação produzida. No final do processo, *scripts* SQL são construídos para extrair a informação de correspondência direta por linha entre mutações e avisos estáticos.

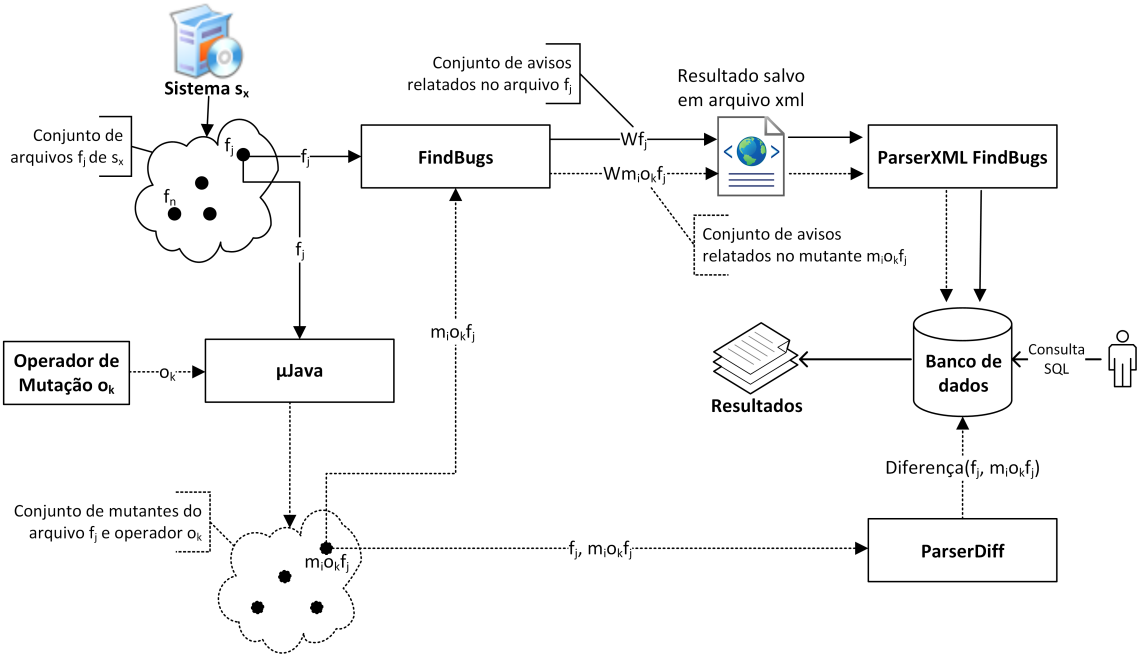

Figura 3.9: *Diagrama do Processo Utilizado no Experimento.*

3.4 Primeiro Experimento Controlado

Para validar o modelo de domínio construído (Figura 3.8) e o processo de coleta dos dados (Seção 3.3) foi desenvolvido um primeiro experimento controlado. Para este primeiro experimento, foram utilizados 7 programas, os quais somam 446 linhas de código e estão mostrados na Tabela 3.1. Esses programas foram selecionados por já terem sido utilizados em trabalhos anteriores (POLO; PIATTINI; GARCÍA-RODRÍGUEZ, 2009). Inicialmente, a FindBugs foi executada sobre os arquivos originais de cada programa. Em seguida, para cada um dos programas foram gerados mutantes por meio da ferramenta μJava, utilizando todos os 19 operadores tradicionais e os 28 operadores de classe disponíveis nessa ferramenta (OFFUTT, 2014). O conjunto de mutantes foi utilizado para verificar a correspondência direta por linha de dois pontos de vista: por operador de mutação e por tipo de aviso. Por operador de mutação, são identificadas quais categorias de defeitos (representadas por operadores) são mais/menos detectadas pela FindBugs. Já por tipo de aviso, são identificados quais avisos conseguem mais/menos perceber as mutações produzidas. Os resultados deste primeiro experimento são apresentados nas Tabelas 3.2 (por operador) e 3.3 (por tipo de aviso).

Tabela 3.1: *Programas Utilizados no Primeiro Experimento*

#	Programa	Quant. Arquivos	Quant. Métodos	LOC	Quant. Mutantes	Quant. Avisos
1	Bisect	1	3	26	192	3
2	BubCorrecto	1	7	30	94	1
3	Ciudad	1	23	183	294	1
4	Find	1	4	46	290	2
5	Fourballs	1	2	34	302	1
6	PluginTokenizer	1	16	67	132	0
7	Triangulo	1	5	60	566	0
Total		7	60	446	1.870	8

Na Tabela 3.1 são apresentadas as seguintes informações: as linhas apresentam dados sobre cada um dos programas; as colunas apresentam, por programa, o nome do programa, a quantidade de arquivos, a quantidade de métodos, a quantidade de linhas de código (coluna LOC), a quantidade de mutantes gerados, a quantidade de avisos relatados no programa original. Por exemplo, o programa Ciudad (linha 3) tem um arquivo, o qual possui 23 métodos, 183 linhas de código que permitiram gerar 294 mutantes, teve 1 aviso relatado no programa original.

Com relação a correspondência por tipo de operador, na Tabela 3.2 são exibidas informações relacionadas a quantidade de mutantes que foi gerada por operador e a quantidade de avisos que foi relatada na mesma linha onde ocorreu a mutação. Observe que dos 47 operadores de mutação da μJava (19 tradicionais e 28 de classe), somente

25 deles (15 tradicionais e 10 de classe) geraram mutantes nos 7 programas utilizados nesse primeiro experimento. Portanto, para 24 operadores, os 7 sistemas não possuem as estruturas sintáticas requeridas por tais operadores para gerar mutantes.

Tabela 3.2: *Correspondência por Tipo de Operador no Primeiro Experimento*

o_k	**Operador**	**Categoria do Operador**	**Quant. Mutantes**	**Quant. Mutantes com Aviso no Ponto de Mutação**	%
1	PRV	C	1	1	100,00%
2	JSD	C	1	1	100,00%
3	JTD	C	1	1	100,00%
4	CDL	T	5	3	60,00%
5	JTI	C	4	1	25,00%
6	AOIS	T	670	129	19,30%
7	SDL	T	115	9	7,80%
8	ROR	T	279	16	5,70%
9	ODL	T	83	3	3,60%
10	AORB	T	160	4	2,50%
11	COI	T	86	1	1,20%
12	AODU	T	1	0	0,00%
13	EMM	C	2	0	0,00%
14	JDC	C	2	0	0,00%
15	IOD	C	4	0	0,00%
16	JID	C	4	0	0,00%
17	COD	T	5	0	0,00%
18	AODS	T	7	0	0,00%
19	AORS	T	23	0	0,00%
20	JSI	C	24	0	0,00%
21	COR	T	24	0	0,00%
22	EAM	C	50	0	0,00%
23	VDL	T	58	0	0,00%
24	AOIU	T	101	0	0,00%
25	LOI	T	160	0	0,00%
Total			1.870	169	9,04%

Como exemplos de informações apresentadas na Tabela 3.2, para o operador CDL (linha 4) foram gerados 5 mutações e destas 5, 3 foram "percebidas" por algum aviso da FindBugs. Desse modo, para o operador CDL a correspondência direta por linha foi de 60% ($3/5$). Os três primeiros operadores (PRV, JSD e JTD) apresentados na Tabela 3.2 tiveram correspondência direta por linha de 100%. Por outro lado, os operadores AODU, EMM, JDC, IOD, JID, COD, AODS, AORS, JSI, COR, EAM, VDL, AOIU e LOI (linhas 12-25 da Tabela 3.2) tiveram correspondência de 0%, ou seja, as mutações produzidas por últimos operadores não foram "percebidas" pela FindBugs.

Já com relação a correspondência por tipo de aviso, na Tabela 3.3 são exibidas informações sobre a quantidade de avisos relatados pela FindBugs e a quantidade de avisos relatados na mesma linha onde ocorreu a mutação no mutante. O total de avisos relatados foi de 2.164 avisos de 28 diferentes tipos, sendo que 173 avisos, cerca de 8% do total, foram relatados no ponto de mutação. As 28 linhas da Tabela 3.3 representam os 28 diferentes tipos de avisos relatados nos mutantes dos 7 programas utilizados.

Tabela 3.3: *Correspondência por Tipo de Aviso no Primeiro Experimento*

#	Aviso	Quant. Avisos	Quant. Avisos no Ponto de Mutação	%
1	INT_BAD_COMPARISON_WITH_NONNEGATIVE_VALUE	20	20	100,00%
2	SA_FIELD_DOUBLE_ASSIGNMENT	12	12	100,00%
3	SA_FIELD_SELF_ASSIGNMENT	8	8	100,00%
4	UCF_USELESS_CONTROL_FLOW_NEXT_LINE	4	4	100,00%
5	FE_FLOATING_POINT_EQUALITY	4	4	100,00%
6	INT_BAD_REM_BY_1	4	4	100,00%
7	DLS_DEAD_LOCAL_STORE_IN_RETURN	2	2	100,00%
8	NP_LOAD_OF_KNOWN_NULL_VALUE	2	2	100,00%
9	SS_SHOULD_BE_STATIC	1	1	100,00%
10	SA_LOCAL_SELF_ASSIGNMENT	1	1	100,00%
11	SE_NO_SERIALVERSIONID	1	1	100,00%
12	SE_NONSTATIC_SERIALVERSIONID	1	1	100,00%
13	UCF_USELESS_CONTROL_FLOW	12	11	91,70%
14	RANGE_ARRAY_INDEX	20	5	25,00%
15	DLS_DEAD_LOCAL_STORE	573	95	16,60%
16	UC_USELESS_CONDITION	46	2	4,30%
17	UWF_FIELD_NOT_INITIALIZED_IN_CONSTRUCTOR	2	0	0,00%
18	MS_PKGPROTECT	4	0	0,00%
19	UC_USELESS_OBJECT	5	0	0,00%
20	UWF_UNWRITTEN_PUBLIC_OR_PROTECTED_FIELD	9	0	0,00%
21	UWF_UNWRITTEN_FIELD	9	0	0,00%
22	NP_ALWAYS_NULL	16	0	0,00%
23	NP_UNWRITTEN_FIELD	20	0	0,00%
24	ST_WRITE_TO_STATIC_FROM_INSTANCE_METHOD	26	0	0,00%
25	DM_NUMBER_CTOR	94	0	0,00%
26	UUF_UNUSED_FIELD	196	0	0,00%
27	EI_EXPOSE_REP	290	0	0,00%
28	URF_UNREAD_FIELD	782	0	0,00%
Total		2.164	173	7,99%

Como exemplo de informação apresentada na Tabela 3.3, o aviso DLS_DEAD_LOCAL_STORE (linha 15) foi relatado 573 vezes (terceira coluna), sendo que 95 avisos foram relatados no ponto de mutação. Desse modo, o aviso

DLS_DEAD_LOCAL_STORE teve correspondência direta por linha de $16,60\%$ $(95/573)$.

Observe que os 12 primeiros avisos apresentados na Tabela 3.3 tiveram correspondência de 100%. Por outro lado, os 12 últimos tipos de avisos tiveram correspondência de 0%, ou seja, estes 12 últimos não conseguiram "perceber" as mutações produzidas pelos operadores.

Resultados Obtidos no Primeiro Experimento

Em relação à correspondência obtida por operador, observe que os valores apresentados na Tabela 3.2 mostram uma variedade nos resultados entre os diferentes tipos de operadores. De forma geral, considerando todos os operadores, a correspondência entre a linha de código na qual ocorreu a mutação e a linha na qual o aviso foi relatado foi de $9,04\%$ $(169/1.870)$.

Há diversos operadores para os quais não foi relatado nenhum aviso na mesma linha onde ocorreu a mutação. Nenhuma mutação produzida pelos operadores AODU, EMM, JDC, IOD, JID, COD, AODS, AORS, JSI, COR, EAM, VDL, AOIU e LOI foram detectadas pela FindBugs. Portanto, a correspondência para estes últimos operadores é 0%. Isso significa que a ferramenta FindBugs não foi capaz de "perceber" o defeito modelado por tais operadores.

Já os operadores PRV, JSD, JTD e CDL apresentaram correspondência superior a 60%, sendo que os três primeiros tiveram correspondência de 100%.

Em relação à correspondência obtida por tipo de aviso, os valores apresentados na Tabela 3.3 mostram uma variedade entre os diferentes tipos de avisos. De forma geral, considerando todos os tipos de avisos, a correspondência entre a linha de código na qual ocorreu a mutação e a linha na qual o aviso foi relatado foi de $7,99\%$ $(173/2.164)$.

Há diversos avisos para os quais não foi relatado nenhum aviso na mesma linha onde ocorreu a mutação. Nenhum aviso foi relatado no ponto de mutação para os últimos 12 avisos (linhas 17-28) da Tabela 3.3. Contudo, os 12 primeiros avisos (linhas 1-12) apresentam correspondência de 100%.

Os resultados obtidos foram considerados bastante promissores e motivaram a condução do segundo experimento controlado (apresentado nos Capítulos 4 e 5).

Espera-se que, com o aumento do número de programas avaliados, seja possível identificar outras categorias de defeitos, representados pelos operadores de mutação, para os quais exista a correspondência, além de identificar os avisos que mais/menos conseguem perceber as mutações produzidas nos mutantes. A correspondência pode ser utilizada para contribuir para uma otimização na realização dos testes e no estabelecimento de estratégias complementares que combinem análise estática (por meio de utilização de

ferramentas de análise estáticas) e dinâmica (por meio da utilização do Teste de Mutação) de forma mais eficiente.

3.5 Considerações Finais do Capítulo

Nesse capítulo foram apresentados os conceitos e definições utilizados no estudo proposto. Foi apresentado o arcabouço de experimentação por meio do modelo de domínio utilizado e do processo de coleta de tais dados. Um primeiro experimento foi desenvolvido por meio de um conjunto de 7 programas. O resultado desse primeiro experimento mostrou que para alguns tipos de operadores e alguns tipos de avisos há uma certa correspondência entre mutações e avisos estáticos. Esse resultado serviu como motivação para a realização do segundo estudo experimental apresentado nos Capítulos 4 e 5. No Capítulo 4 é apresentado a correspondência direta por linha por tipo de aviso e no Capítulo 5, por tipo de operador.

CAPÍTULO 4

Estudo Experimental I – CDL por Aviso

Este capítulo apresenta o estudo realizado para analisar a correspondência direta por linha (CDL) entre mutações e avisos estáticos. São apresentados os sistemas selecionados, a correspondência direta por linha por aviso, e um cenário de aplicação da correspondência obtida para o tipo de aviso, por meio de uma abordagem incremental para análise dos avisos da FindBugs.

4.1 Sistemas Utilizados

Para o estudo experimental apresentado neste capítulo e no Capítulo 5 foram selecionados 19 sistemas. Estes sistemas foram escolhidos por dois motivos: 1) são sistemas cujos códigos fontes (arquivos .java) e correspondentes executáveis (arquivos .class e .jar) estão publicamente disponíveis no site e repositório da Fundação Apache; 2) correspondem a um subconjunto (de aproximadamente 63%, 19 de 30) dos sistemas utilizados nos trabalhos de Filho et al. (2010) e Couto et al. (2013), cujos resultados serviram como motivação para a realização do estudo relatado nesta dissertação. Inclusive, procurou-se por utilizar no estudo, sempre que possível, a mesma versão utilizada nos trabalhos de Filho et al. (2010) e Couto et al. (2013).

Os dados referentes aos sistemas selecionados são apresentados na Tabela 4.1. Nas colunas da Tabela 4.1 são apresentadas as seguintes informações: a coluna s_x é um identificador de cada sistema; a coluna Sistema representa o nome do sistema; a coluna Descrição é uma descrição resumida; a coluna Versão indica a versão utilizada; a coluna LOC é a quantidade de linhas de código fonte de cada sistema; a coluna "Avisos (W)" é a quantidade de avisos relatados pela FindBugs por sistema; a coluna "Mutantes (M)" é a quantidade de mutantes gerados de cada sistema; a coluna W/KLOC é a média de avisos relatados pela quantidade de milhares de linhas de código; a coluna M/KLOC é a média de mutantes gerados pela quantidade de milhares de linhas de código.

A ferramenta FindBugs foi executada individualmente sobre cada um dos arquivos originais dos sistemas selecionados. Já os mutantes foram gerados por meio da aplicação dos operadores de mutação disponibilizados na ferramenta μJava sobre cada um dos

arquivos dos referidos sistemas. Em seguida, a ferramenta FindBugs foi executada sobre cada mutante gerado.

Tabela 4.1: *Sistemas Utilizados no Estudo Experimental Estendido*

s_x	Sistema	Descrição	Versão	LOC	Avisos (W)	Mutantes (M)	W/KLOC[1]	M/KLOC
s_1	Beehive	Java application framework	1.0	55.129	340	27.282	6,16	494,87
s_2	Cayenne	Object/Relational framework	2.0.2	68.523	517	76.140	7,54	1.111,16
s_3	Cfx	Web service framework	2.1	5.344	51	3.065	9,543	573,54
s_4	Ddlutils	Database Definition (DDL) API	2.0.0	13.892	133	15.294	9,57	1.100,92
s_5	Hadoop-Hbase	Distributed database system	0.2.0	24.718	61	3.827	2,46	154,82
s_6	Ibatis	Object/Relational framework	2.3.0	10.781	135	13.102	12,52	1.215,28
s_7	Ivy	Dependency manager system	2.1	26.893	135	21.570	5,02	802,06
s_8	James-Server	Mail enterprise server	2.2.0	18.599	204	17.43	10,97	937,20
s_9	JDO	Persistence API	2.1	2.748	27	1.166	9,82	424,30
s_{10}	Lucene	Information retrieval library	2.9.4	38.152	215	28.316	5,63	742,18
s_{11}	OpenJPA	Persistence API	1.0.0	102.682	359	65.529	3,49	638,17
s_{12}	Roller	Blog server	4.0	40.146	65	6.790	1,61	169,13
s_{13}	Shidig	OpenSocial container	1.1b5	21.843	105	10.708	4,80	490,22
s_{14}	Solr	Text search platform	1.3.0	26.327	313	23.115	11,88	877,99
s_{15}	Tapestry	Web application framework	4.1.2	39.874	220	33.246	5,51	833,77
s_{16}	Tuscany-Sca	SOA-based framework	1	68.697	431	42.121	6,27	613,14
s_{17}	Wicket	Web application framework	1.4.0	45.356	219	79.968	4,82	1.763,11
s_{18}	Xalan_J	XSLT implementation	2.7.2	93.335	76	12.280	0,81	131,56
s_{19}	Xmlbeans	XML Parser	2.0.0	46.927	103	12.572	2,19	267,90
Total				749.966	3.709	493.522	–	–
Média				39.472	195	25.975	6,35	702,18
Mediana				38.152	135	17.431	5,63	638,17
Mínimo				2.748	27	1.166	0,814	131,57
Máximo				102.682	517	79.968	12,52	1.763,11

Como exemplos de informações contidas na Tabela 4.1, podem ser citados para o sistema Open JPA (s_{11}) cuja descrição é *Persistence API*: foi utilizada a versão 1.0.0; tal sistema possui 102.682 linhas de código[2]; teve 359 avisos relatados pela ferramenta FindBugs; permitiu gerar 65.529 mutantes por meio da ferramenta μJava; possui médias de, aproximadamente, $3,5$ avisos por KLOC e de 638 mutantes por KLOC. Importante observar que os avisos apresentados na Tabela 4.1 foram relatados exclusivamente no código original dos sistemas, e não nos mutantes gerados – informações de avisos relatados nos mutantes são apresentadas a seguir, nas Tabelas 4.2, 4.3 e 5.1.

Para coletar dados de avisos em cada um dos sistemas, a FindBugs foi executada com a configuração *default*, isto é, empregando todas as categorias de avisos disponíveis na FindBugs. Para coletar os dados dos avisos relatados em cada mutante, a FindBugs, também com a configuração *default*, foi executada sobre cada um deles. Para cada um dos sistemas selecionados foram gerados mutantes empregando todos os 47 operadores de mutação da ferramenta μJava. Todos os operadores de mutação geraram mutantes.

[2]informação obtida por meio da ferramenta JavaNCSS

Sobre os sistemas da Tabela 4.1, como exemplos de informações estatísticas, podem ser destacados:

- **Tamanho dos sistemas**: os sistemas somam 749.966 (quase 750 KLOC) linhas de código fonte; o tamanho médio dos sistemas é de 39.472 linhas de código; o tamanho mediano dos sistemas é 38.152 linhas de código; o menor sistema é sistema JDO (s_9), com 2.748 linhas; o maior sistema é o OpenJPA (s_{11}), com 102.682 linhas de código;
- **Quantidade de avisos**: ao todo foram relatados 3.709 avisos; em média, foram relatados 195 avisos por sistema; a mediana da quantidade de avisos é de 135 avisos; o sistema com a menor quantidade de avisos é o JDO (s_9), com 27 avisos; o sistema com a maior quantidade de avisos é o Cayenne (s_2), com 517 avisos;
- **Quantidade de mutantes**: o total de mutantes gerados foi de 493.522; em média, foram gerados 25.975 mutantes por sistema; a mediana da quantidade de mutantes entre os sistemas é de 17.431; o sistema para o qual foi gerada a menor quantidade de mutantes é o JDO (s_9), com 1.166 mutantes; o sistema para o qual foi gerada a maior quantidade de mutantes é o Wicket (s_{17}), com 79.968 mutantes.

Para verificar a correspondência direta por linha, todos os avisos relatados pela FindBugs nos arquivos originais e nos mutantes foram armazenados em um banco de dados relacional.

Após os mutantes terem sidos gerados, o passo seguinte foi executar, individualmente, a ferramenta FindBugs sobre cada um dos 493.522 mutantes. Os dados resultantes da execução foram armazenados na base de dados e estão apresentados na Tabela 4.2. Nessa tabela são mostradas informações sobre a quantidade de avisos, por tipo, relatados nos mutantes de acordo com os 47 tipos de operadores de mutação da μJava.

Nas colunas da Tabela 4.2 são apresentadas as seguintes informações: a coluna *id* é um identificador de linha; a coluna "Aviso *w*" é o nome do tipo de aviso relatado; as colunas ISD, JTI, JTD, JSD, ... (por razões de espaço, os outros operadores foram omitidos desta tabela), representam os operadores de mutação da μJava; a coluna $TW(w)$ é o total de avisos, por tipo, relatados nos mutantes.

Como exemplos de informações apresentadas na Tabela 4.2 têm-se: o aviso BC_UNCONFIRMED_CAST_OF_RETURN_VALUE (linha 7) foi relatado nenhuma vez nos mutantes do operador ISD, 27 vezes nos mutantes do operador JTI, 27 vezes nos mutantes do operador JTD, 12 vezes nos mutantes do operador JSD, e assim por diante. Considerando os mutantes dos 47 operadores, o aviso BC_UNCONFIRMED_CAST_OF_RETURN_VALUE foi relatado 4.894 vezes (linha 7, coluna $TW(w)$).

Considerando todos os mutantes gerados, foram relatados neles 218 tipos diferentes de avisos. Estes 218 tipos diferentes de avisos estão representados pelas linhas 1, 2,

Tabela 4.2: *Avisos Relatados nos Mutantes por Operador*

id	**Aviso** w	**Operadores de mutação**					$TW(w)$
		ISD	**JTI**	**JTD**	**JSD**	⋯	
1	AT_OPERATION_SEQUENCE_ON_CONCURRENT_ABSTRACTION *	0	0	0	1	⋯	5
2	BC_BAD_CAST_TO_CONCRETE_COLLECTION *	0	0	0	0	⋯	1
3	BC_EQUALS_METHOD_SHOULD_WORK_FOR_ALL_OBJECTS	0	2	0	43	⋯	530
4	BC_IMPOSSIBLE_CAST *	0	0	0	0	⋯	15
5	BC_IMPOSSIBLE_INSTANCEOF	0	0	0	0	⋯	14
6	BC_UNCONFIRMED_CAST *	0	0	0	53	⋯	13.511
7	BC_UNCONFIRMED_CAST_OF_RETURN_VALUE *	0	27	27	12	⋯	4.894
8	BC_VACUOUS_INSTANCEOF *	0	16	8	9	⋯	3.584
9	BIT_ADD_OF_SIGNED_BYTE	0	0	0	0	⋯	4
10	BIT_AND	0	0	0	0	⋯	15
11	BIT_AND_ZZ	0	0	0	0	⋯	8
12	BIT_IOR	0	0	0	0	⋯	97
13	BIT_SIGNED_CHECK	0	0	0	0	⋯	358
14	BX_BOXING_IMMEDIATELY_UNBOXED	0	0	1	0	⋯	1
15	BX_BOXING_IMMEDIATELY_UNBOXED_TO_PERFORM_COERCION *	0	0	0	0	⋯	368
16	BX_UNBOXING_IMMEDIATELY_REBOXED	0	2	0	0	⋯	4
17	CI_CONFUSED_INHERITANCE *	0	2	0	1	⋯	188
18	CN_IDIOM *	0	23	11	0	⋯	441
19	CN_IDIOM_NO_SUPER_CALL *	0	0	0	3	⋯	3.003
20	CN_IMPLEMENTS_CLONE_BUT_NOT_CLONEABLE *	0	163	155	10	⋯	6.906
21	DB_DUPLICATE_BRANCHES	0	2	2	1	⋯	686
22	DB_DUPLICATE_SWITCH_CLAUSES *	0	0	0	0	⋯	589
23	DC_DOUBLECHECK *	0	17	15	1	⋯	1.634
24	DE_MIGHT_IGNORE	0	119	65	44	⋯	15.692
25	DLS_DEAD_LOCAL_INCREMENT_IN_RETURN	0	0	0	0	⋯	372
26	DLS_DEAD_LOCAL_STORE	2	75	2.647	177	⋯	44.987
27	DLS_DEAD_LOCAL_STORE_IN_RETURN	0	0	0	0	⋯	88
28	DLS_DEAD_LOCAL_STORE_OF_NULL	0	1	9	1	⋯	822
29	DLS_DEAD_STORE_OF_CLASS_LITERAL	0	0	0	0	⋯	13
30	DLS_OVERWRITTEN_INCREMENT	0	0	0	0	⋯	4
31	DMI_BLOCKING_METHODS_ON_URL *	0	15	8	0	⋯	50
32	DMI_COLLECTION_OF_URLS *	2	45	13	15	⋯	1.719
33	DMI_HARDCODED_ABSOLUTE_FILENAME	0	0	0	0	⋯	3
34	DMI_INVOKING_HASHCODE_ON_ARRAY *	0	45	15	9	⋯	1.276
35	DMI_INVOKING_TOSTRING_ON_ARRAY *	0	12	9	10	⋯	7.883
36	DMI_USELESS_SUBSTRING	0	0	0	0	⋯	154
37	DMI_USING_REMOVEALL_TO_CLEAR_COLLECTION *	0	0	0	0	⋯	17
38	DM_BOOLEAN_CTOR *	0	3	1	11	⋯	4.933
39	DM_BOXED_PRIMITIVE_FOR_PARSING *	0	5	2	59	⋯	2.583
40	DM_BOXED_PRIMITIVE_TOSTRING *	0	0	0	0	⋯	1.468
…	…	…	…	…	…	⋱	…
210	UUF_UNUSED_PUBLIC_OR_PROTECTED_FIELD *	0	53	36	24	⋯	3.093
211	UWF_FIELD_NOT_INITIALIZED_IN_CONSTRUCTOR	0	441	288	14	⋯	24.667
212	UWF_NULL_FIELD	1	143	137	7	⋯	2.275
213	UWF_UNWRITTEN_FIELD	0	1.402	1.141	1	⋯	5.682
214	UWF_UNWRITTEN_PUBLIC_OR_PROTECTED_FIELD	0	378	347	43	⋯	3.546
215	UW_UNCOND_WAIT *	0	2	0	0	⋯	450
216	VO_VOLATILE_REFERENCE_TO_ARRAY *	0	10	4	0	⋯	1.011
217	WA_NOT_IN_LOOP *	0	5	3	9	⋯	1.474
218	WMI_WRONG_MAP_ITERATOR *	0	0	0	5	⋯	3.072
219	$TW(o_k)$	88	14.303	11.789	7.882	⋯	980.533

..., 218 da Tabela 4.2. Ainda nesta tabela, na linha 219 ($TW(o_k)$) há o total de avisos relatados nos mutantes, de acordo com o operador de mutação. Por exemplo: nos mutantes do operador ISD foram relatados 88 avisos de diferentes tipos, nos mutantes do operador JTI foram relatados 14.303 avisos, nos mutantes do operador JTD foram relatados 11.789 avisos, nos mutantes do operador JSD foram relatados 7.882 avisos, e assim por diante. Nos 493.522 mutantes, independente do operador, foram relatados 980.533 avisos (linha 219, coluna $TW(w)$). Observe que os avisos nos mutantes apresentados na Tabela 4.2, não foram, necessariamente, relatados no mesmo ponto em que ocorreu a mutação.

A partir dos dados coletados e apresentados na Tabela 4.2 foram aplicadas as funções definidas nas Seções 4.2 e 5.1 para obter a *CDL* por aviso (Seção 4.3) e a *CDL* por operador (Seção 5.2), respectivamente.

4.2 CDL por Aviso

A *CDL* (correspondência direta por linha) por tipo de aviso ocorre quando um aviso é relatado pela FindBugs no mesmo ponto em que ocorre uma mutação produzida por algum operador de mutação, ou seja, um aviso é relatado em virtude da mutação criada, conforme apresentado por meio da Figura 3.2. O objetivo da *CDL* por aviso é identificar os tipos de avisos que mais/menos conseguem "perceber" os pontos de mutações.

Para calcular a *CDL* de cada tipo de aviso da FindBugs, foram definidas as seguintes funções:

- $TW(w)$ = 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. 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.

O objetivo das funções $TW(w)$, $CDL_A(w)$ e $CDL_R(w)$ é classificar cada tipo de aviso w de acordo com sua precisão em relatar os defeitos representados nos mutantes. Os tipos de avisos com taxas $CDL_R(w)$ maiores (mais próximos de 1) deveriam ser priorizados em relação aos tipos de aviso com taxas $CDL_R(w)$ menores (mais próximos de 0),

pois os avisos com $CDL_R(w)$ mais alta tem maior probabilidade de serem, teoricamente, avisos verdadeiros positivos, e, nesse caso, deveriam ser analisados com maior prioridade.

A partir dos dados coletados e apresentados na Tabela 4.2 foram aplicadas as funções $TW(w)$, $CDL_A(w)$ e $CDL_R(w)$, cujos resultados permitiram gerar a Tabela 4.3 com o cálculo da *CDL* por aviso.

4.3 Cálculo da CDL por Aviso

Para obter a correspondência direta por linha de cada tipo de aviso w foram aplicadas as funções $TW(w)$, $CDL_A(w)$ e $CDL_R(w)$ sobre os dados apresentados na Tabela 4.2. Os resultados da aplicação destas funções são apresentados por meio da Tabela 4.3.

Nas colunas da Tabela 4.3 são apresentadas as seguintes informações: a primeira coluna (*id*) é um identificador de linha; a coluna "Aviso w" representa os avisos que tiveram pelo menos um aviso relatado no ponto de mutação, isto é, $CDL_A(w) \geq 1$; a coluna P é o nível de prioridade de cada aviso definido pela FindBugs (1 = alta, 2 = média, 3 = baixa); a coluna seguinte é a categoria do aviso; as colunas ISD, JTI, JTD, JSD, ..., representam os 47 operadores da μJava; as colunas $CDL_A(w)$, $TW(w)$ e $CDL_R(w)$ são os resultados das respectivas funções que foram definidas na Seção 4.2.

A Tabela 4.3 está ordenada de forma decrescente, respectivamente, pelas colunas $CDL_R(w)$ e $CDL_A(w)$. Portanto, nas primeiras linhas desta tabela há os avisos da FindBugs que mais detectaram proporcionalmente ($CDL_R(w)$ mais próximo de 1) as mutações produzidas nos mutantes. De forma semelhante, nas últimas linhas há os avisos que menos detectaram ($CDL_R(w)$ mais próximo de 0) as mutações produzidas nos mutantes.

Nas primeiras linhas da Tabela 4.3 estão os avisos que mais foram capazes de detectar defeitos modelados pelos operadores de mutação da μJava. Já nas últimas linhas, há os avisos com as menores correspondências com mutações.

Observe que na Tabela 4.3 são apresentados apenas os 98 tipos de avisos que tiveram pelo menos um aviso relatado na mesma linha de mutação (coluna $CDL_A(w) \geq 1$). Na Tabela 4.2 são apresentados todos os 218 tipos de avisos reportados no experimento. Portanto, existem 120 tipos de avisos com $CDL_A(w) = 0$. Esse resultado significa que ou não há aviso no ponto de mutação, ou, se existir, esse tipo de aviso já existia no arquivo original, de forma que o aviso em si não está associado a mutação produzida (neste caso, o aviso não é computado na função $CDL_A(w)$). Os avisos que tiveram $CDL_A(w) = 0$ estão identificados com o símbolo '$*$' na Tabela 4.2. Por exemplo, apesar do aviso CN_IMPLEMENTS_CLONE_BUT_NOT_CLONEABLE, na linha 20 da Tabela 4.2, ter sido relatado 6.906 vezes nos mutantes, nenhum

Tabela 4.3: *Correspondência Direta por Linha por Aviso*

id	**Aviso** *w*	**P**	**Categoria**	**Operadores de Mutação**					$CDL_A(w)$	$TW(w)$	$CDL_R(w)$
				ISD	**JTI**	**JTD**	**JSD**	⋯			
1	DLS_DEAD_LOCAL_STORE_IN_RETURN	3	STYLE	0	0	0	0	⋯	88	88	100,00%
2	QF_QUESTIONABLE_FOR_LOOP	2	STYLE	0	0	0	0	⋯	81	81	100,00%
3	MS_FINAL_PKGPROTECT	2	MALICIOUS_CODE	0	0	0	0	⋯	22	22	100,00%
4	INT_BAD_COMPARISON_WITH_SIGNED_BYTE	3	CORRECTNESS	0	0	0	0	⋯	18	18	100,00%
5	BIT_AND	1	CORRECTNESS	0	0	0	0	⋯	15	15	100,00%
6	INT_VACUOUS_BIT_OPERATION	2	STYLE	0	0	0	0	⋯	13	13	100,00%
7	RE_BAD_SYNTAX_FOR_REGULAR_EXPRESSION	1	CORRECTNESS	0	0	0	0	⋯	13	13	100,00%
8	ICAST_INT_CAST_TO_DOUBLE_PASSED_TO_CEIL	1	CORRECTNESS	0	0	0	0	⋯	6	6	100,00%
9	BIT_ADD_OF_SIGNED_BYTE	2	CORRECTNESS	0	0	0	0	⋯	4	4	100,00%
10	BX_UNBOXING_IMMEDIATELY_REBOXED	2	PERFORMANCE	0	2	0	0	⋯	4	4	100,00%
11	DLS_OVERWRITTEN_INCREMENT	1	CORRECTNESS	0	0	0	0	⋯	4	4	100,00%
12	NS_DANGEROUS_NON_SHORT_CIRCUIT	1	STYLE	0	0	0	0	⋯	4	4	100,00%
13	RV_01_TO_INT	1	CORRECTNESS	0	0	0	0	⋯	4	4	100,00%
14	UM_UNNECESSARY_MATH	3	PERFORMANCE	0	0	0	0	⋯	4	4	100,00%
15	DMI_HARDCODED_ABSOLUTE_FILENAME	2	STYLE	0	0	0	0	⋯	3	3	100,00%
16	QBA_QUESTIONABLE_BOOLEAN_ASSIGNMENT	1	CORRECTNESS	0	0	0	0	⋯	2	2	100,00%
17	SA_LOCAL_SELF_COMPUTATION	1	CORRECTNESS	0	0	0	0	⋯	2	2	100,00%
18	BX_BOXING_IMMEDIATELY_UNBOXED	2	PERFORMANCE	0	0	1	0	⋯	1	1	100,00%
19	EC_UNRELATED_CLASS_AND_INTERFACE	1	CORRECTNESS	0	0	0	0	⋯	1	1	100,00%
20	SA_FIELD_SELF_ASSIGNMENT	1	CORRECTNESS	0	3.296	0	0	⋯	3.661	3.663	99,95%
21	SA_LOCAL_SELF_ASSIGNMENT	2	STYLE	2	0	2.763	0	⋯	2.915	2.920	99,83%
22	UCF_USELESS_CONTROL_FLOW_NEXT_LINE	1	STYLE	0	0	0	0	⋯	489	490	99,80%
23	DLS_DEAD_LOCAL_INCREMENT_IN_RETURN	2	CORRECTNESS	0	0	0	0	⋯	370	372	99,46%
24	INT_BAD_REM_BY_1	1	STYLE	0	0	0	0	⋯	848	864	98,15%
25	IL_INFINITE_RECURSIVE_LOOP	1	CORRECTNESS	35	0	0	183	⋯	750	775	96,77%
26	INT_VACUOUS_COMPARISON	2	STYLE	0	0	0	0	⋯	66	69	95,65%
27	INT_BAD_COMPARISON_WITH_NONNEGATIVE_VALUE	1	CORRECTNESS	0	0	0	0	⋯	952	999	95,30%
28	BIT_IOR	1	CORRECTNESS	0	0	0	0	⋯	90	97	92,78%
29	BIT_SIGNED_CHECK	3	BAD_PRACTICE	0	0	0	0	⋯	330	358	92,18%
30	UR_UNINIT_READ	2	CORRECTNESS	0	1.855	0	0	⋯	2.292	2.593	88,39%
31	DMI_USELESS_SUBSTRING	2	STYLE	0	0	0	0	⋯	123	154	79,87%
32	NP_NULL_ON_SOME_PATH_MIGHT_BE_INFEASIBLE	2	STYLE	0	0	0	0	⋯	9	12	75,00%
33	UCF_USELESS_CONTROL_FLOW	2	STYLE	0	0	0	0	⋯	7.130	9.556	74,61%
34	SA_FIELD_DOUBLE_ASSIGNMENT	2	STYLE	0	0	0	0	⋯	88	124	70,97%
35	BIT_AND_ZZ	1	CORRECTNESS	0	0	0	0	⋯	5	8	62,50%
36	DLS_DEAD_STORE_OF_CLASS_LITERAL	2	CORRECTNESS	0	0	0	0	⋯	8	13	61,54%
37	SA_FIELD_SELF_COMPUTATION	2	CORRECTNESS	0	0	0	0	⋯	8	15	53,33%
38	NP_GUARANTEED_DEREF	2	CORRECTNESS	0	0	0	0	⋯	28	57	49,12%
39	SQL_BAD_PREPARED_STATEMENT_ACCESS	1	CORRECTNESS	0	0	0	0	⋯	22	57	38,60%
40	DLS_DEAD_LOCAL_STORE	2	STYLE	2	0	2.621	0	⋯	17.294	44.987	38,44%
…	…	…	…	…	…	…	…	⋱	…	…	…
86	DE_MIGHT_IGNORE	2	BAD_PRACTICE	0	0	0	0	⋯	14	15.692	0,09%
87	OS_OPEN_STREAM	2	BAD_PRACTICE	0	0	0	0	⋯	5	5.703	0,09%
88	IS2_INCONSISTENT_SYNC	2	MT_CORRECTNESS	0	0	0	0	⋯	18	21.114	0,09%
89	SBSC_USE_STRINGBUFFER_CONCATENATION	2	PERFORMANCE	0	0	0	0	⋯	2	2.882	0,07%
90	NP_BOOLEAN_RETURN_NULL	2	BAD_PRACTICE	0	0	0	0	⋯	1	1.813	0,06%
91	DM_STRING_CTOR	2	PERFORMANCE	0	0	0	0	⋯	1	2.841	0,04%
92	REC_CATCH_EXCEPTION	3	STYLE	0	0	0	0	⋯	15	54.741	0,03%
93	DM_CONVERT_CASE	3	I18N	0	0	0	0	⋯	6	37.082	0,02%
94	ES_COMPARING_STRINGS_WITH_EQ	2	BAD_PRACTICE	0	1	0	0	⋯	1	9.122	0,01%
95	DM_DEFAULT_ENCODING	1	I18N	0	0	0	0	⋯	2	44.280	0,00%
96	IIO_INEFFICIENT_INDEX_OF	3	PERFORMANCE	0	0	0	0	⋯	1	26.388	0,00%
97	EI_EXPOSE_REP2	2	MALICIOUS_CODE	0	0	0	0	⋯	1	27.574	0,00%
98	DM_NUMBER_CTOR	2	PERFORMANCE	0	0	0	0	⋯	2	77.952	0,00%
Total				39	6.362	5.431	2.249	⋯	50.768	980.533	5,18%

aviso foi relatado por causa da mutação, ou seja, a $CDL_A(w) = 0$ para o aviso CN_IMPLEMENTS_CLONE_BUT_NOT_CLONEABLE.

Na linha Total da Tabela 4.3 é mostrada a quantidade de avisos relatados nos mutantes de acordo com o operador. O total de avisos relatados em todos os mutantes foi

de 980.533, sendo que 50.768 avisos foram relatados exatamente no ponto de mutação, o que equivale a uma taxa $CDL_R(w)$ geral de $5,18\%$ $(50.768/980.533)$.

Apesar da $CDL_R(w)$ geral ter sido apenas de $5,18\%$, há 37 tipos de avisos que apresentaram $CDL_R(w) > 50\%$. Estes 37 tipos de avisos são apresentados nas 37 primeiras linhas da Tabela 4.3. Desses 37 tipos de avisos, os 29 primeiros têm $CDL_R(w)$ superior a 90%. Melhor ainda: os 19 primeiros tipos de avisos têm $CDL_R(w) = 100\%$, ou seja, todos os avisos relatados por estes 19 tipos foram relatados no ponto de mutação.

Por outro lado, alguns tipos de avisos apresentaram $CDL_R(w) \leq 0,1\%$. Esses avisos estão apresentados nas últimas linhas da Tabela 4.3. Como exemplos de tipos de avisos com baixa taxa de *CDL* podem ser citados os 14 avisos das linhas 85-98 da Tabela 4.3. Por exemplo o aviso ES_COMPARING_STRINGS_WITH_EQ (linha 94), um aviso da categoria BAD_PRACTICE, teve taxa de correspondência de $0,01\%(1/9.122)$, ou seja, apenas um aviso foi relatado no ponto de mutação em um universo de 9.122 avisos relatados nos mutantes.

Os tipos de avisos com $CDL_R(w)$ próximos de 0% representam os tipos de avisos que menos foram capazes de detectar de forma efetiva a mutação produzida nos mutantes.

Observe que isso não significava que esses avisos são bons ou maus preditores de defeitos ou geram avisos verdadeiros positivos / falso positivos, indicam apenas que eles são bons / maus preditores na detecção de defeitos modelados por operadores de mutação. Por outro lado, uma vez que o Teste de Mutação é considerado um bom critério para avaliação da qualidade de conjuntos de casos de testes (ANDREWS; BRIAND; LABICHE, 2005), o conjunto de defeitos modelados por operadores mutação é utilizado para criar uma abordagem de priorização dos avisos da FindBugs.

A variação na correspondência entre os avisos e as mutações ilustra que alguns tipos de avisos são mais sensíveis na identificação de certos tipos de defeitos, representados por operadores de mutação específicos. Os resultados obtidos para a *CDL* por aviso podem ser utilizados como subsídio para estabelecer uma abordagem de priorização para analisar os avisos relatados pela FindBugs. Na Seção 4.4 é apresentado um exemplo de aplicação da *CDL* por meio de uma abordagem incremental de análise dos avisos da FindBugs. Com base nos dados apresentados na Tabela 4.3, foi criada uma abordagem incremental para análise dos avisos. Os detalhes desta abordagem são descritos na Seção 4.4 a seguir.

4.4 Exemplo de Aplicação da CDL por Aviso

Para ilustrar como a *CDL* por aviso pode ser utilizada na prática, é apresentada nessa seção uma abordagem de priorização incremental para analisar os tipos de avisos relatados pela FindBugs. Para isso, foram selecionados três sistemas adicionais: Cassandra, Hibernate e Apache POI que estão apresentados na Tabela 4.4. Nas colunas da Tabela 4.4

são apresentadas informações relacionadas a estes sistemas: a coluna *id* é um identificador do sistema, seguido pelo nome, versão utilizada, quantidade de linhas de código (coluna LOC) e quantidade de avisos relatados pela FindBugs em cada um dos sistemas (coluna "Avisos w").

Tabela 4.4: *Sistemas para Exemplificar a Aplicação da $CDL_R(w)$*

id	**Sistema**	**Versão**	**LOC**	**Avisos w**
C	Cassandra	2.1.0	43.792	946
H	Hibernate	4.3.6	129.337	2.131
P	Apache POI	3.9	61.460	776
Total			234.589	3.853

Por exemplo, para o sistema Hibernate (id H) foi utilizada a versão 4.3.6, a qual possui 129.337 linhas de código, e nas quais foram relatados 2.131 avisos pela FindBugs. Os três sistemas (Cassandra, Hibernate e Apache POI), em conjunto, somam cerca de 235 mil linhas de código e tiveram 3.853 avisos relatados pela FindBugs.

Em seguida, para cada um dos 98 tipos de avisos apresentados na Tabela 4.3, foi obtida a quantidade de vezes que a FindBugs relatou este mesmo aviso nos sistemas Cassandra, Hibernate e Apache POI. O resultado deste procedimento permitiu gerar os dados apresentados na Tabela 4.5.

Na Tabela 4.5 é apresentada a quantidade de avisos, por tipo de aviso, relatados nos sistemas Cassandra (coluna W_C), Hibernate (coluna W_H) e Apache POI (coluna W_P). A coluna W_{total} é igual a soma $W_C + W_H + W_P$. Como exemplo, na linha 40 da Tabela 4.5 há o aviso DLS_DEAD_LOCAL_STORE que foi relatado 5 vezes no sistema Cassandra (coluna W_C), 383 vezes no sistema Hibernate (coluna W_H) e 25 vezes no sistema Apache POI (coluna W_P). Considerando os três sistemas, o aviso DLS_DEAD_LOCAL_STORE foi relatado 413 $(5+383+25)$ vezes (coluna W_{total}).

Observe que algumas linhas da Tabela 4.5 apresentam $W_{total} = 0$, o que significa que o aviso da linha em questão não foi relatado em nenhum dos três sistemas. Isso ocorre, por exemplo, com os avisos das linhas 1, 2, 3, 7, 9, entre outras. Esses avisos poderiam ser omitidos da Tabela 4.5 sem prejuízo do resultado obtido da aplicação da abordagem incremental (apresentada a seguir), mas foram mantidos por questões didáticas.

Priorização Baseada na CDL por Aviso

A abordagem de priorização incremental sugerida é baseada na ordem decrescente da taxa de $CDL_R(w)$ obtida de cada aviso w. O objetivo é analisar primeiro os avisos com maiores $CDL_R(w)$, ou seja, aqueles que mais perceberam as mutações produzidas.

Desse modo, baseado nos resultados da $CDL_R(w)$ de cada aviso apresentados na Tabela 4.3, é proposto que sejam analisados os avisos da FindBugs na seguinte ordem:

- 1º DLS_DEAD_LOCAL_STORE_IN_RETURN, com a maior $CDL_R(w)$
- 2º QF_QUESTIONABLE_FOR_LOOP, com a 2ª maior $CDL_R(w)$
- 3º MS_FINAL_PKGPROTECT, com a 3ª maior $CDL_R(w)$
- 4º INT_BAD_COMPARISON_WITH_SIGNED_BYTE com a 4ª maior $CDL_R(w)$
- ...
- 96º IIO_INEFFICIENT_INDEX_OF, com a 96ª maior $CDL_R(w)$
- 97º EI_EXPOSE_REP2, com a 97ª maior $CDL_R(w)$
- 98º DM_NUMBER_CTOR, com a 98ª maior $CDL_R(w)$

Observe que os avisos apresentados da linha 1 até a linha 98 na Tabela 4.5 estão na ordem de priorização sugerida.

Para aplicar a abordagem incremental proposta, foram definidas as seguintes funções:

Função de Custo Acumulado de Aviso (*Cumulate Cost* $CC(w_i)$):

$$CC(w_i) = W_1 + W_2 + \cdots + W_{i-1} + W_i \tag{4-1}$$

Função de Redução de Custo de Aviso (*Cost Reduction* $CR(w_i)$):

$$CR(w_i) = 1 - (CC(w_i)/Total\ de\ avisos) \tag{4-2}$$

A Função 4-1 representa a quantidade acumulada de avisos para analisar os avisos da linha 1 até o aviso da linha i da Tabela 4.5, seguindo o ordem de priorização sugerida.

A Função 4-2 representa a redução de custo referente aos avisos que deixariam de ser verificados, caso sejam analisados apenas os avisos da linha 1 até a linha i da Tabela 4.5. Assim, os avisos da linha $i+1$ em diante na Tabela 4.5 seriam, por algum motivo (por exemplo: falta de recursos), ignorados.

As colunas CC_C, CC_H e CC_P armazenam, respectivamente, o resultado da Função 4-1 sobre os avisos relatados nos sistemas Cassandra, Hibernate e Apache POI. A coluna CC_{total} é o resultado da Função 4-1 aplicado aos três sistemas. Por exemplo, os custos acumulados para analisar os 40 primeiros tipos de avisos são 9 avisos para o sistema Cassandra (coluna CC_C), 401 para o Hibernate (coluna CC_H), 46 para o Apache POI (coluna CC_P) e 456 para os três sistemas (coluna CC_{total}).

As colunas CR_C, CR_H e CR_P armazenam, respectivamente, o resultado da Função 4-2 sobre os avisos relatados nos sistemas Cassandra, Hibernate e Apache POI. A coluna CR_{total} é o resultado da Função 4-2 aplicado aos três sistemas em conjunto. Por exemplo, as reduções de custos proporcionadas para analisar apenas os 40 primeiros tipos de avisos são $99,05\%$ para o sistema Cassandra (coluna CR_C), $81,18\%$ para o Hibernate

(coluna CR_H), 94,07% para o Apache POI (coluna CR_P) e 88,17% considerando os três sistemas (coluna CR_{total}).

Tabela 4.5: *Priorização de Avisos de acordo com a $CDL_R(w)$*

W_i	Aviso W	Cassadra			Hibernate			Apache POI			Total		
		W_C	CC_C	CR_C	W_H	CC_H	CR_H	W_P	CC_P	CR_P	W_{total}	CC_{total}	CR_{total}
1	DLS_DEAD_LOCAL_STORE_IN_RETURN	0	0	100,00%	0	0	100,00%	0	0	100,00%	0	0	100,00%
2	QF_QUESTIONABLE_FOR_LOOP	0	0	100,00%	0	0	100,00%	0	0	100,00%	0	0	100,00%
3	MS_FINAL_PKGPROTECT	0	0	100,00%	0	0	100,00%	0	0	100,00%	0	0	100,00%
4	INT_BAD_COMPARISON_WITH_SIGNED_BYTE	2	2	99,79%	0	0	100,00%	2	2	99,74%	4	4	99,90%
5	BIT_AND	0	2	99,79%	0	0	100,00%	1	3	99,61%	1	5	99,87%
6	INT_VACUOUS_BIT_OPERATION	0	2	99,79%	0	0	100,00%	1	4	99,48%	1	6	99,84%
7	RE_BAD_SYNTAX_FOR_REGULAR_EXPRESSION	0	2	99,79%	0	0	100,00%	0	4	99,48%	0	6	99,84%
8	ICAST_INT_CAST_TO_DOUBLE_PASSED_TO_CEIL	0	2	99,79%	0	0	100,00%	2	6	99,23%	2	8	99,79%
9	BIT_ADD_OF_SIGNED_BYTE	0	2	99,79%	0	0	100,00%	0	6	99,23%	0	8	99,79%
10	BX_UNBOXING_IMMEDIATELY_REBOXED	2	4	99,58%	1	1	99,95%	0	6	99,23%	3	11	99,71%
11	DLS_OVERWRITTEN_INCREMENT	0	4	99,58%	0	1	99,95%	0	6	99,23%	0	11	99,71%
12	NS_DANGEROUS_NON_SHORT_CIRCUIT	0	4	99,58%	2	3	99,86%	0	6	99,23%	2	13	99,66%
13	RV_01_TO_INT	0	4	99,58%	0	3	99,86%	0	6	99,23%	0	13	99,66%
14	UM_UNNECESSARY_MATH	0	4	99,58%	0	3	99,86%	0	6	99,23%	0	13	99,66%
15	DMI_HARDCODED_ABSOLUTE_FILENAME	0	4	99,58%	0	3	99,86%	0	6	99,23%	0	13	99,66%
16	QBA_QUESTIONABLE_BOOLEAN_ASSIGNMENT	0	4	99,58%	0	3	99,86%	0	6	99,23%	0	13	99,66%
17	SA_LOCAL_SELF_COMPUTATION	0	4	99,58%	0	3	99,86%	0	6	99,23%	0	13	99,66%
18	BX_BOXING_IMMEDIATELY_UNBOXED	0	4	99,58%	0	3	99,86%	0	6	99,23%	0	13	99,66%
19	EC_UNRELATED_CLASS_AND_INTERFACE	0	4	99,58%	0	3	99,86%	0	6	99,23%	0	13	99,66%
20	SA_FIELD_SELF_ASSIGNMENT	0	4	99,58%	0	3	99,86%	0	6	99,23%	0	13	99,66%
21	SA_LOCAL_SELF_ASSIGNMENT	0	4	99,58%	0	3	99,86%	0	6	99,23%	0	13	99,66%
22	UCF_USELESS_CONTROL_FLOW_NEXT_LINE	0	4	99,58%	0	3	99,86%	0	6	99,23%	0	13	99,66%
23	DLS_DEAD_LOCAL_INCREMENT_IN_RETURN	0	4	99,58%	0	3	99,86%	0	6	99,23%	0	13	99,66%
24	INT_BAD_REM_BY_1	0	4	99,58%	0	3	99,86%	0	6	99,23%	0	13	99,66%
25	IL_INFINITE_RECURSIVE_LOOP	0	4	99,58%	1	4	99,81%	1	7	99,10%	2	15	99,61%
26	INT_VACUOUS_COMPARISON	0	4	99,58%	0	4	99,81%	0	7	99,10%	0	15	99,61%
27	INT_BAD_COMPARISON_WITH_NONNEGATIVE_VALUE	0	4	99,58%	0	4	99,81%	0	7	99,10%	0	15	99,61%
28	BIT_IOR	0	4	99,58%	0	4	99,81%	0	7	99,10%	0	15	99,61%
29	BIT_SIGNED_CHECK	0	4	99,58%	0	4	99,81%	3	10	98,71%	3	18	99,53%
30	UR_UNINIT_READ	0	4	99,58%	1	5	99,77%	3	13	98,32%	4	22	99,43%
31	DMI_USELESS_SUBSTRING	0	4	99,58%	0	5	99,77%	0	13	98,32%	0	22	99,43%
32	NP_NULL_ON_SOME_PATH_MIGHT_BE_INFEASIBLE	0	4	99,58%	0	5	99,77%	3	16	97,94%	3	25	99,35%
33	UCF_USELESS_CONTROL_FLOW	0	4	99,58%	13	18	99,16%	5	21	97,29%	18	43	98,88%
34	SA_FIELD_DOUBLE_ASSIGNMENT	0	4	99,58%	0	18	99,16%	0	21	97,29%	0	43	98,88%
35	BIT_AND_ZZ	0	4	99,58%	0	18	99,16%	0	21	97,29%	0	43	98,88%
36	DLS_DEAD_STORE_OF_CLASS_LITERAL	0	4	99,58%	0	18	99,16%	0	21	97,29%	0	43	98,88%
37	SA_FIELD_SELF_COMPUTATION	0	4	99,58%	0	18	99,16%	0	21	97,29%	0	43	98,88%
38	NP_GUARANTEED_DEREF	0	4	99,58%	0	18	99,16%	0	21	97,29%	0	43	98,88%
39	SQL_BAD_PREPARED_STATEMENT_ACCESS	0	4	99,58%	0	18	99,16%	0	21	97,29%	0	43	98,88%
40	DLS_DEAD_LOCAL_STORE	5	9	99,05%	383	401	81,18%	25	46	94,07%	413	456	88,17%
41	BC_IMPOSSIBLE_INSTANCEOF	0	9	99,05%	0	401	81,18%	0	46	94,07%	0	456	88,17%
42	NP_ALWAYS_NULL	0	9	99,05%	0	401	81,18%	0	46	94,07%	0	456	88,17%
43	SS_SHOULD_BE_STATIC	0	9	99,05%	1	402	81,14%	0	46	94,07%	1	457	88,14%
44	RCN_REDUNDANT_COMPARISON_TWO_NULL_VALUES	0	9	99,05%	0	402	81,14%	0	46	94,07%	0	457	88,14%
45	EQ_ALWAYS_FALSE	0	9	99,05%	0	402	81,14%	0	46	94,07%	0	457	88,14%
46	NP_NULL_INSTANCEOF	0	9	99,05%	0	402	81,14%	0	46	94,07%	0	457	88,14%
47	NP_NONNULL_PARAM_VIOLATION	0	9	99,05%	0	402	81,14%	0	46	94,07%	0	457	88,14%
48	RCN_REDUNDANT_COMPARISON_OF_NULL_AND...	0	9	99,05%	0	402	81,14%	0	46	94,07%	0	457	88,14%
49	NP_NULL_ON_SOME_PATH	5	14	98,52%	7	409	80,81%	1	47	93,94%	13	470	87,80%
50	DB_DUPLICATE_BRANCHES	0	14	98,52%	2	411	80,71%	0	47	93,94%	2	472	87,75%
...	...	...	...	...	...	...	...	...	...	...	...	...	...
Total		946	–	–	2.131	–	–	776	–	–	3.853	–	–

Para melhor compreensão, na Figura 4.1 são comparados o custo acumulado ($CC(w_i)$) e a redução de custo ($CR(w_i)$) obtidos na abordagem de priorização sugerida de acordo com o $CDL_R(w)$. No eixo X da Figura 4.1, *wi* representa o aviso da linha *i* na Tabela 4.5. Já no eixo Y, há o valor da redução de custo (*CR*) e o valor do custo acumulado (*CC*) à medida que os tipos de avisos *wi* são analisados.

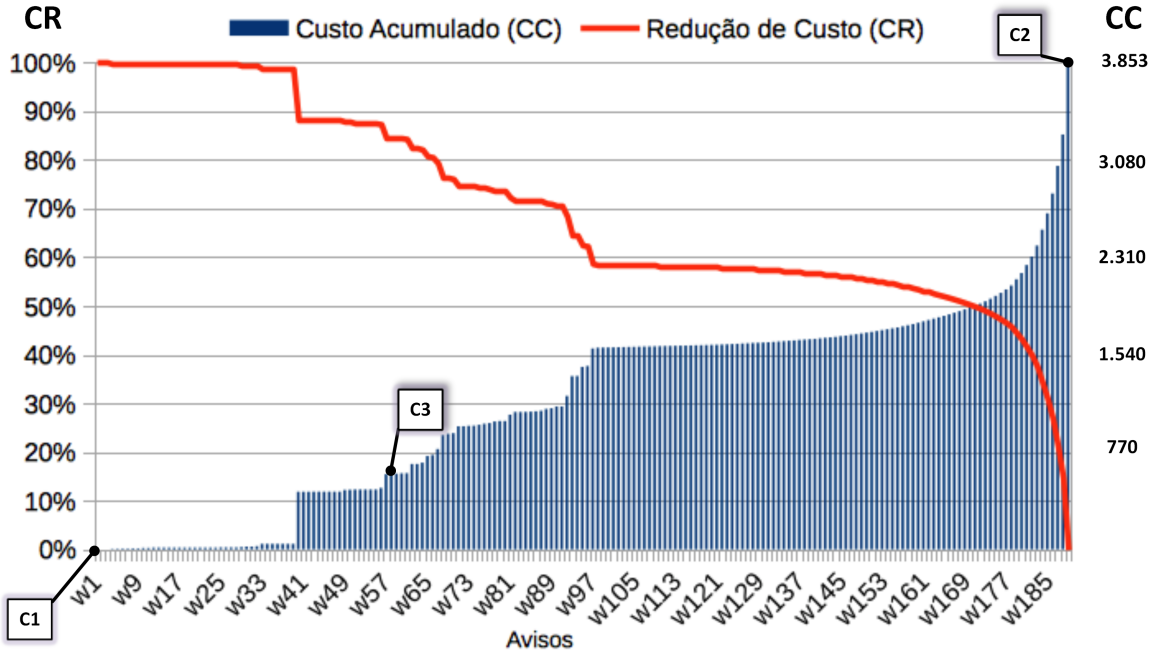

Figura 4.1: *Custo acumulado ($CC(w_i)$) versus redução de custo ($CR(w_i)$) na abordagem de priorização para análise dos avisos, de acordo com a $CDL_R(w)$.*

Didaticamente, a Figura 4.1 pode ser compreendida por meio de três cenários:

Cenário 1: nenhum aviso é analisado. Este caso ilustra o cenário no qual todos os avisos relatados por ferramentas de análise estática são ignorados. Por falta de recursos ou por acreditarem que tais ferramentas não agregam qualidade ao produto de software, nenhum aviso é analisado pelo desenvolvedor. Na Figura 4.1, este cenário ocorre no início do gráfico (ponto $C1$) com $CC(w_i) = 0$ (nenhum aviso é analisado) e $CR(w_i) = 100\%$ (a redução é de 100% porque todos os avisos deixaram de ser analisados);

Cenário 2: todos os avisos são analisados. Este caso ilustra o cenário no qual todos os avisos relatados por ferramentas de análise estática são levados em consideração: qualquer aviso relatado é cuidadosamente analisado e resolvido pelo desenvolvedor. Na Figura 4.1, este cenário ocorre no final do gráfico (ponto $C2$) com $CC(w_i) = 3.853$ (todos os avisos são analisados) e $CR(w_i) = 0\%$ (a redução de custo é de 0% porque nenhum aviso deixou ser analisado);

Cenário 3: apenas alguns avisos são analisados. Este caso ilustra um cenário intermediário, no qual alguns avisos são analisados, mas não todos. Na Figura 4.1, este cenário ocorre no gráfico com $0 < CC(w_i) < 3.853$ e $100\% > CR(w_i) > 0\%$, como, por exemplo, no ponto $C3$ apresentado no gráfico. Com isso, os avisos analisados na ordem de priorização sugerida baseada na $CDL_R(w)$ permite otimizar os recursos, pois os avisos considerados com maior probabilidade de ser um aviso verdadeiro

positivo são analisados primeiro. Isso permite a otimização da análise dos avisos e a maximização da redução de custos de análise, ao deixar que os avisos com menor $CDL_R(w)$, considerados "menos importantes", para serem analisados em um segundo momento, se mais recursos estiverem disponíveis.

Um segundo exemplo de aplicação da *CDL* por aviso é apresentado a seguir. A ferramenta FindBugs foi executada sobre cerca de 13.000 mutantes que foram gerados para os sistemas Cassandra, Apache POI e Hibernate. A Tabela 4.6 apresenta os 59 diferentes tipos de avisos que foram relatados nesses mutantes. Nesta tabela tem-se: a coluna aviso é o nome do aviso relatado; $TW(w)$ é o total de aviso de cada tipo; $CDL_A(w)$ é o número de avisos, por tipo, relatado no ponto de mutação; já as colunas seguintes indicam a prioridade do aviso, a categoria e a ordem do aviso de acordo com correspondência sugerida e apresentada na Tabela 4.3.

Tabela 4.6: *Avisos Relatados nos Mutantes dos Sistemas Adicionais*

#	Aviso	$TW(w)$	$CDL_A(w)$	P	Categoria	Ordem CDL
1	SA_FIELD_SELF_ASSIGNMENT	102	102	1	Correctness	20
2	UCF_USELESS_CONTROL_FLOW	53	53	2	Dodgy code	33
3	SS_SHOULD_BE_STATIC	49	49	2	Performance	43
4	BIT_SIGNED_CHECK	48	48	3	Bad practice	29
5	SA_LOCAL_SELF_ASSIGNMENT	36	36	2	Dodgy code	21
6	IL_INFINITE_RECURSIVE_LOOP	23	23	1	Correctness	25
7	DLS_DEAD_LOCAL_INCREMENT_IN_RETURN	12	12	2	Correctness	23
8	BIT_IOR	12	12	1	Correctness	28
9	UCF_USELESS_CONTROL_FLOW_NEXT_LINE	12	12	1	Dodgy code	22
10	INT_BAD_REM_BY_1	10	10	1	Dodgy code	24
11	NP_NULL_PARAM_DEREF_NONVIRTUAL	8	8	1	Correctness	59
12	INT_BAD_COMPARISON_WITH_SIGNED_BYTE	7	7	3	Correctness	4
13	INT_BAD_COMPARISON_WITH_NONNEGATIVE_VALUE	5	5	1	Correctness	27
14	SA_FIELD_DOUBLE_ASSIGNMENT	4	4	2	Dodgy code	34
15	BIT_IOR_OF_SIGNED_BYTE	3	3	N/A	Correctness	N/A
16	VA_FORMAT_STRING_EXTRA_ARGUMENTS_PASSED	3	3	N/A	Correctness	N/A
17	FE_FLOATING_POINT_EQUALITY	2	2	3	Dodgy code	78
18	DLS_DEAD_LOCAL_STORE_IN_RETURN	2	2	3	Dodgy code	1
19	DMI_USELESS_SUBSTRING	1	1	2	Dodgy code	31
20	DM_STRING_TOSTRING	1	1	3	Performance	85
...	...	...	...	...	...	...
59	EI_EXPOSE_REP	11.863	0	2	Malicious code vulnerability	N/A
Total		42.636	1.104	–	–	–

Os avisos foram ordenados para serem analisados segundo dois critérios: 1) ordem baseada na *CDL* apresentada na Tabela 4.3; 2) ordem baseada na categoria e prioridade do aviso. A categoria *Correctness* é aquela que possui a menor taxa de aviso falso positivo, segundo as pesquisas de Shen, Fang e Zhao (2011). Já as prioridades 1 e

2 indicam os tipos de avisos que estão mais relacionados a possíveis defeitos, segundo a FindBugs.

A Tabela 4.7 mostra os 16 avisos de prioridade 1 ou 2 da categoria *Correctness*. Seguindo esta abordagem, a linha Total mostra que o total de avisos é 353 avisos analisados da categoria *Correctness* com prioridade 1 ou 2. Destes 353 avisos analisados, 302 avisos foram relatados no mesmo ponto de mutação, o que equivale a um custo de análise de $1,17$ aviso por mutação encontrada.

Tabela 4.7: *Avisos Analisados pela Categoria e Prioridade*

#	**Aviso**	$TW(w)$	$CDL_A(w)$	**P**	**Categoria**	**Ordem CDL**
1	SA_FIELD_SELF_ASSIGNMENT	102	102	1	Correctness	20
2	IL_INFINITE_RECURSIVE_LOOP	23	23	1	Correctness	25
3	BIT_IOR	12	12	1	Correctness	28
4	NP_NULL_PARAM_DEREF_NONVIRTUAL	8	8	1	Correctness	59
5	INT_BAD_COMPARISON_WITH_NONNEGATIVE_VALUE	5	5	1	Correctness	27
6	IL_INFINITE_LOOP	3	1	1	Correctness	63
7	EQ_ALWAYS_TRUE	3	0	1	Correctness	52
8	DLS_DEAD_LOCAL_INCREMENT_IN_RETURN	12	12	2	Correctness	23
9	UR_UNINIT_READ	73	72	2	Correctness	30
10	NP_NULL_ON_SOME_PATH	4	3	2	Correctness	49
11	NP_ALWAYS_NULL	61	45	2	Correctness	42
12	UWF_UNWRITTEN_FIELD	28	19	2	Correctness	51
13	SA_FIELD_SELF_COMPARISON	1	0	2	Correctness	54
14	SA_FIELD_SELF_COMPUTATION	1	0	2	Correctness	37
15	UWF_NULL_FIELD	2	0	2	Correctness	74
16	NP_UNWRITTEN_FIELD	15	0	2	Correctness	N/A
Total		353	302	–	–	–

Já a Tabela 4.8 mostra os 16 primeiros avisos ordenados de acordo com a *CDL* obtida para cada tipo de aviso. Seguindo esta abordagem, a linha Total mostra que o total é de 401 avisos analisados. Destes 401 avisos analisados, 399 avisos foram relatados no mesmo ponto de mutação, o que equivale a um custo de análise de cerca de 1 aviso por mutação produzida.

A abordagem de priorização baseada na *CDL* apresentou ser melhor do que a abordagem baseada da categoria *Correctness*. A análise dos avisos por meio da *CDL* teve custo de 85% em relação ao custo da análise dos avisos por meio da categoria *Correctness*. Além disso, por meio da *CDL* há uma cobertura maior de avisos de diferentes categorias e prioridades. Contudo, outros experimentos precisam ser realizados para avaliar a capacidade de tais avisos em detectar defeitos reais.

Tabela 4.8: *Avisos Analisados pela CDL*

#	Aviso	$TW(w)$	$CDL_A(w)$	P	Categoria	Ordem CDL
1	DLS_DEAD_LOCAL_STORE_IN_RETURN	2	2	3	Dodgy code	1
2	INT_BAD_COMPARISON_WITH_SIGNED_BYTE	7	7	3	Correctness	4
3	SA_FIELD_SELF_ASSIGNMENT	102	102	1	Correctness	20
4	SA_LOCAL_SELF_ASSIGNMENT	36	36	2	Dodgy code	21
5	UCF_USELESS_CONTROL_FLOW_NEXT_LINE	12	12	1	Dodgy code	22
6	DLS_DEAD_LOCAL_INCREMENT_IN_RETURN	12	12	2	Correctness	23
7	INT_BAD_REM_BY_1	10	10	1	Dodgy code	24
8	IL_INFINITE_RECURSIVE_LOOP	23	23	1	Correctness	25
9	INT_BAD_COMPARISON_WITH_NONNEGATIVE_VALUE	5	5	1	Correctness	27
10	BIT_IOR	12	12	1	Correctness	28
11	BIT_SIGNED_CHECK	48	48	3	Bad practice	29
12	UR_UNINIT_READ	73	72	2	Correctness	30
13	DMI_USELESS_SUBSTRING	1	1	2	Dodgy code	31
14	UCF_USELESS_CONTROL_FLOW	53	53	2	Dodgy code	33
15	SA_FIELD_DOUBLE_ASSIGNMENT	4	4	2	Dodgy code	34
16	SA_FIELD_SELF_COMPUTATION	1	0	2	Correctness	37
Total		401	399	–	–	–

4.5 Considerações Finais do Capítulo

Nesse capítulo foi apresentado o estudo experimental estendido para avaliar a correspondênca entre mutações e avisos estáticos para um conjunto de 19 sistemas. Foram definidas as funções utilizadas para calcular a correspondência entre mutações e avisos estáticos, de acordo com o aviso. Um exemplo de aplicação da abordagem de análise incremental a partir da correspondência obtida para cada aviso foi sugerida.

CAPÍTULO 5

Estudo Experimental II – CDL por Operador

Este capítulo apresenta o estudo realizado para analisar a correspondência direta por linha (CDL) entre mutações e avisos estáticos, de acordo com o operador de mutação da μJava. Com o objetivo de reduzir o custo do Teste de Mutação, é apresentado um cenário de aplicação da *CDL* por operador.

5.1 CDL por Operador

A *CDL* (correspondência direta por linha) por operador ocorre quando uma mutação é percebida pela ferramenta de análise estática FindBugs, ou seja, um aviso é relatado em virtude da mutação produzida, conforme ilustrado por meio da Figura 3.2. O objetivo da *CDL* por operador é identificar os operadores cujos mutações mais/menos são percebidas pela FindBugs.

Para calcular a *CDL* por operador, foram definidas as seguintes funções:

- $TM(o_k)$ = 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 por ferramentas de análise estáticas. Assim, na teoria adotada neste trabalho, é o grau de precisão dos analisadores estáticos em detectar o defeito modelado pelo operador o_k. Desse modo, quanto mais próximo de 1, maior a chance dos defeitos modelos pelo operador o_k de serem detectados pela ferramenta FindBugs. De forma simétrica, quanto mais próximo de zero, menor a chance dos defeitos modelados pelo operador o_k de serem detectados por avisos da FindBugs.

O objetivo das funções $TM(o_k)$, $CDL_A(o_k)$ e $CDL_R(o_k)$ é classificar os operadores de mutação de acordo com a precisão com que tais defeitos – representadas pelas

mutações – são detectados por ferramentas de análise estática. Os operadores de mutação com maiores taxas de correspondência direta por linha $CDL_R(o_k)$ representam, teoricamente, os tipos de defeitos que mais são detectados pela FindBugs. Por outro lado, os operadores de mutação com menores $CDL_R(o_k)$ representam os tipos de defeitos que são poucos detectados pela FindBugs e, neste caso, por exemplo, novos detectores de defeitos poderiam ser sugeridos para serem incorporados na ferramenta FindBugs.

A partir dos dados coletados e apresentados na Tabela 4.2 foram aplicadas as funções $TM(o_k)$, $CDL_A(o_k)$ e $CDL_R(o_k)$, cujos resultados permitiram gerar a Tabela 5.1 (*CDL* por tipo de operador).

5.2 Cálculo da CDL por Operador

Para obter a correspondência direta por linha por operador ($CDL_R(o_k)$), foram aplicadas as funções $TM(o_k)$, $CDL_A(o_k)$ e $CDL_R(o_k)$ que foram definidas na Seção 5.1, sobre os dados apresentados na Tabela 4.2. Os resultados da aplicação dessas funções sobre a Tabela 4.2 são apresentados na Tabela 5.1.

Na Tabela 5.1, que está ordenada pelas colunas $CDL_R(o_k)$ (decrescente), $CDL_A(o_k)$ (decrescente) e $TM(o_k)$ (crescente), são apresentadas informações obtidas para os 47 operadores da ferramenta μJava.

Nas colunas da Tabela 5.1 são apresentadas as seguintes informações: a primeira coluna (k) é um identificador de linha; a coluna Operador o_k representa os operadores de mutação; a coluna Tipo é a categoria de cada operador ((C) classe e (T) tradicional); a coluna $TM(o_K)$ é o resultado da função de mesmo nome (definida na Seção 5.1); a coluna "$TM(o_k)$ com Aviso" contém a quantidade absoluta de mutantes do operador o_k que tiveram avisos relatados (coluna Total_1) e a quantidade relativa desses mutantes sobre o total (coluna $\text{Total}_1/TM(o_k)$); a coluna "$TM(o_k)$ sem Aviso" contém, para cada operador, a quantidade de mutantes que não tiveram avisos (coluna Total_2) e a quantidade relativa desses mutantes sobre o total (coluna $\text{Total}_2/TM(o_k)$); por fim, as colunas $CDL_A(o_k)$ e $CDL_R(o_k)$ são os resultados das respectivas funções, definidas na Seção 5.1.

As 47 linhas da Tabela 5.1 representam os 47 operadores de mutação da μJava. Por exemplo, o operador de mutação COI (linha 10), um operador da categoria tradicional, possibilitou gerar um total de 27.399 mutantes. Dos 27.399 mutantes do operador COI, 18.963 mutantes, cerca de 69,20% do total, tiveram pelo menos um aviso relatado em seu código. Nos demais mutantes desse operador, ou seja, em 8.436 mutantes, cerca de 30,80% do total, nenhum aviso foi relatado. Para o operador COI, a $CDL_A(o_k)$ foi de 2.533, o que corresponde a uma $CDL_R(o_k)$ de cerca de $9,20\%$ ($2.533/27.399$). Assim, 9 de cada 100 mutações geradas pelo operador COI foram percebidas por algum aviso da FindBugs.

Tabela 5.1: *Correspondência Direta por Linha por Operador*

k	Operador o_k	Tipo	$TM(o_k)$	$TM(o_k)$ com Avisos		$TM(o_k)$ sem Aviso		$CDL_A(o_k)$	$CDL_R(o_k)$
				Total_1	$\text{Total}_1/TM(o_k)$	Total_2	$\text{Total}_2/TM(o_k)$		
1	JTD	C	3.243	3.094	95,40%	149	4,60%	2.850	87,90%
2	JTI	C	4.969	4.372	88,00%	597	12,00%	3.605	72,50%
3	JSD	C	3.229	2.806	86,90%	423	13,10%	2.139	66,20%
4	ISD	C	75	51	68,00%	24	32,00%	37	49,30%
5	OAN	C	2.632	2.073	78,80%	559	21,20%	561	21,30%
6	AOIS	T	55.182	40.189	72,80%	14.993	27,20%	10.718	19,40%
7	LOR	T	484	341	70,50%	143	29,50%	93	19,20%
8	COR	T	8.659	5.629	65,00%	3.030	35,00%	1.243	14,40%
9	SDL	T	102.500	66.644	65,00%	35.856	35,00%	13.030	12,70%
10	COI	T	27.399	18.963	69,20%	8.436	30,80%	2.533	9,20%
11	EOC	C	67	45	67,20%	22	32,80%	6	9,00%
12	AORB	T	10.287	6.844	66,50%	3.443	33,50%	871	8,50%
13	PRV	C	8.010	5.344	66,70%	2.666	33,30%	661	8,30%
14	ROR	T	47.917	32.617	68,10%	15.300	31,90%	3.529	7,40%
15	AODU	T	451	308	68,30%	143	31,70%	22	4,90%
16	PCD	C	168	127	75,60%	41	24,40%	6	3,60%
17	JSI	C	8.755	8.240	94,10%	515	5,90%	310	3,50%
18	SOR	T	88	70	79,50%	18	20,50%	2	2,30%
19	COD	T	2.446	1.404	57,40%	1.042	42,60%	29	1,20%
20	CDL	T	7.825	4.603	58,80%	3.222	41,20%	88	1,10%
21	ODL	T	34.024	20.318	59,70%	13.706	40,30%	389	1,10%
22	PNC	C	678	373	55,00%	305	45,00%	7	1,00%
23	AODS	T	765	225	29,40%	540	70,60%	7	0,90%
24	ASRS	T	1.274	899	70,60%	375	29,40%	11	0,90%
25	VDL	T	7.897	5.185	65,70%	2.712	34,30%	61	0,80%
26	OMR	C	6.361	5.549	87,20%	812	12,80%	26	0,40%
27	EAM	C	37.450	21.135	56,40%	16.315	43,60%	157	0,40%
28	LOI	T	24.814	15.174	61,20%	9.640	38,80%	63	0,30%
29	AORS	T	2.419	1.613	66,70%	806	33,30%	4	0,20%
30	EMM	C	9.612	4.686	48,80%	4.926	51,20%	21	0,20%
31	AOIU	T	13.620	8.462	62,10%	5.158	37,90%	16	0,10%
32	IHD	C	19	4	21,10%	15	78,90%	0	0,00%
33	LOD	T	21	6	28,60%	15	71,40%	0	0,00%
34	OMD	C	73	27	37,00%	46	63,00%	0	0,00%
35	JDC	C	144	72	50,00%	72	50,00%	0	0,00%
36	EOA	C	166	91	54,80%	75	45,20%	0	0,00%
37	ISI	C	170	88	51,80%	82	48,20%	0	0,00%
38	PPD	C	209	174	83,30%	35	16,70%	0	0,00%
39	PCC	C	211	46	21,80%	165	78,20%	0	0,00%
40	PMD	C	221	87	39,40%	134	60,60%	0	0,00%
41	IOR	C	426	190	44,60%	236	55,40%	0	0,00%
42	IOP	C	489	177	36,20%	312	63,80%	0	0,00%
43	IPC	C	866	159	18,40%	707	81,60%	0	0,00%
44	IHI	C	2.304	2.045	88,80%	259	11,20%	0	0,00%
45	JID	C	2.828	1.882	66,50%	946	33,50%	0	0,00%
46	IOD	C	3.631	1.430	39,40%	2.201	60,60%	0	0,00%
47	PCI	C	48.444	11.769	24,30%	36.675	75,70%	0	0,00%
Total			493.522	305.630	61,93%	187.892	38,07%	43.095	8,73%
Média			10.500	6.503	60,23%	3.998	39,77%	917	9,11%
Mediana			2.446	1.613	65,00%	540	35,00%	16	0,90%
Mínimo			19	4	18,40%	15	4,60%	0	0,00%
Máximo			102.500	66.644	95,40%	36.675	81,60%	13.030	87,90%

A linha Total da Tabela 5.1 mostra que foram gerados, considerando todos os operadores, 493.522 mutantes (coluna $TM(o_k)$). Deste total, em 187.892 mutantes, cerca de $38,07\%$, não foi relatado nenhum aviso (coluna "$TM(o_k)$ sem Aviso"). Nos demais mutantes, ou seja, em 305.630 ($493.522 - 187.892$) mutantes, cerca de $61,93\%$ do total, foi relatado pelo menos um aviso pela FindBugs (coluna "$TM(o_k)$ com Avisos"). Ainda sobre os dados da Tabela 5.1, como informações estatísticas, podem ser destacados:

- As médias de mutantes gerados por operador, mutantes que tiveram avisos e mutantes que não tiveram avisos são, respectivamente, de 10.500, 6.503 e 3.998;
- As medianas de mutantes gerados por operador, mutantes que tiveram avisos e mutantes que não tiveram avisos são, respectivamente, de 2.446, 1.613 e 540;
- O operador que menos mutantes gerou foi o IHD (linha 32) com 19 mutantes, dos quais 4 tiveram avisos e 15 não tiveram avisos;
- O operador que mais gerou mutantes foi o SDL (linha 9) com 102.500 mutantes, dos quais 66.644 tiveram avisos relatados e 35.856 não tiveram nenhum aviso.

Os resultados das funções $TM(o_k)$, $CDL_A(o_k)$ e $CDL_R(o_k)$ são apresentados, respectivamente, nas colunas de mesmo nome da Tabela 5.1. O resultado da $CDL_R(o_k)$ de cada operador é apresentado na última coluna da Tabela 5.1. Como a Tabela 5.1 está ordenada de forma decrescente pela coluna $CDL_R(o_k)$, nas primeiras linhas desta tabela há os tipos de operadores cujas mutações mais foram percebidas, isto é, detectadas pela FindBugs. Nas últimas linhas, há os operadores cujas mutações menos foram detectadas pela FindBugs.

Os resultados obtidos mostram uma variedade na taxa $CDL_R(o_k)$ entre diferentes tipos de operadores. Considerando todos os operadores, a $CDL_R(o_k)$ geral foi de $8,73\%$ ($43.095/493.522$) (última coluna da linha Total). Desse modo, do total de 493.522 mutantes, em 43.095 mutantes a FindBugs foi capaz de detectar a mutação produzida por algum dos 47 operadores de mutação.

Apesar da $CDL_R(o_k)$ geral ter sido menor que 10%, alguns operadores apresentam taxas bem superiores. Como destaques de melhores taxas $CDL_R(o_k)$, os operadores apresentados nas quatro primeiras linhas da Tabela 5.1 têm $CDL_R(o_k) > 49\%$. Estes operadores são o JTD (com $87,90\%$), o JTI (com $72,50\%$), o JSD (com $66,20\%$) e o ISD (com $49,30\%$). Observe que esses quatro operadores são todos operadores de classe (coluna Tipo).

Os defeitos modelados pelos operadores com $CDL_R(o_k) > 49\%$ têm, teoricamente, maior chance de serem relatados pela FindBugs, isto é, de um aviso verdadeiro positivo ter sido relatado.

Por outro lado, há 16 tipos de operadores com $CDL_R(o_k) = 0\%$, aproximadamente 33% ($16/47$) do total de operadores. Este resultado significa que as mutações produzidas por esses 16 operadores de mutação não foram detectadas pela FindBugs. Estes

operadores, que estão apresentados nas linhas 32 a 47 da Tabela 5.1, são IHD, LOD, OMD, JDC, EOA, ISI, PPD, PCC, PMD, IOR, IOP, IPC, IHI, JID, IOD e PCI. Desse modo, para os 19 sistemas utilizados nesse estudo (Seção 4.1), a FindBugs não foi capaz de perceber a alteração produzida nos mutantes gerados tais operadores. Observe que destes 16 operadores, há apenas um operador do tipo tradicional: LOD, na linha 33. Os demais são todos operadores de classes.

Os defeitos modelados pelos operadores que tiveram $CDL_R(o_k) = 0\%$ podem ser utilizados para melhorar, por meio de criação de novas regras de detecção, a capacidade da FindBugs em detectar tais tipos de defeitos.

A partir dos resultados apresentados na Tabela 5.1, foi criada uma abordagem incremental para análise dos mutantes cujas mutações foram menos percebidas pela ferramenta FindBugs. Essa abordagem usa informações sobre a correspondência por operador para diminuir (ou otimizar) o custo do Teste de Mutação. Neste caso, a ideia é a priorização dos operadores de mutação que não tiveram correspondência com avisos, isto é, operadores de mutação que representam os defeitos que foram mais difíceis de serem detectados pela FindBugs. Os detalhes desta abordagem são descritos na Seção 5.3 a seguir.

5.3 Exemplo de Aplicação da CDL por Operador

Para ilustrar como a $CDL_R(o_k)$ por tipo de operador pode ser utilizada, com o objetivo de servir como subsídio para reduzir o custo do Teste de Mutação, é apresentada nesta seção uma abordagem incremental para análise de mutantes dos operadores de mutação da μJava.

A ideia foi utilizar os dados históricos coletados anteriormente sobre a taxa de correspondência entre avisos e mutações, na definição de uma estratégia incremental que prioriza, inicialmente, o uso dos operadores que geram mutantes cujas mutações não foram detectadas pela ferramenta de análise estática FindBugs.

Para exemplificar foram selecionados os sistemas Cassandra, Apache POI e Hibernate, para os quais foram gerados mutantes utilizando todos os operadores de mutação da μJava. Os dados sobre tais sistemas e dos mutantes gerados são apresentados na Tabela 5.2, cujas colunas contêm as seguintes informações: a coluna *id* é um identificador de cada sistema (C = Cassandra, P = Apache POI, H = Hibernate), seguido pelo nome, versão utilizada, quantidade de linhas de código (coluna LOC) e a quantidade de mutantes gerados.

Por exemplo, para o sistema Hibernate (id *H*) foi utilizada a versão 4.3.6, a qual possui 129.337 linhas de código, e para as quais foram gerados 110.677 mutantes. Os três

sistemas (Cassandra, Apache POI e Hibernate) somam, em conjunto, cerca de 235 mil linhas de código e geraram 133.683 mutantes.

Tabela 5.2: *Sistemas para Exemplificar a Aplicação da $CDL_R(o_k)$*

ID	Sistema	Versão	LOC	N^o Mutantes
C	Cassandra	2.1.0	43.792	3.289
P	Apache POI	3.9	61.460	19.717
H	Hibernate	4.3.6	129.337	110.677
Total			234.589	133.683

Na Tabela 5.3 é apresentada a quantidade de mutantes gerados, por operador, nos sistemas Cassandra (coluna M_C), Apache POI (coluna M_P) e Hibernate (coluna M_H). A coluna M_{total} é igual a $M_C + M_P + M_H$. Por exemplo, na linha 5 da Tabela 5.3 há a quantidade de mutantes gerados referentes ao operador IPC, o qual permitiu gerar 18 mutantes no sistema Cassandra (coluna M_C), 30 mutantes no sistema Apache POI (coluna M_P) e 32 mutantes no sistema Hibernate (coluna M_H). Considerando os três sistemas, o operador IPC gerou 80 ($18+30+32$) mutantes (coluna M_{total}).

Com o objetivo de reduzir o custo de análise de mutantes, a Tabela 5.3 apresenta os operadores de mutação da μJava aplicados de forma incremental. O objetivo é priorizar a análise dos mutantes relacionados aos operadores de mutação com menor $CDL_R(o_k)$, porque estes operadores representam os defeitos que foram mais difíceis de serem percebidos pela ferramenta FindBugs. Assim, considerando os dados coletados, são priorizados os operadores com menores $CDL_R(o_k)$ pois eles indicam mutações que a FindBugs foi pouco eficaz ou até mesmo incapaz de relatar avisos para a sua detecção.

Priorização Baseada na CDL por Operador

A abordagem de priorização incremental sugerida é baseada na ordem crescente da taxa de $CDL_R(o_k)$ obtida de cada operador. O objetivo é analisar primeiro os mutantes dos operadores com menores $CDL_R(o_k)$, ou seja, os operadores cujas mutações foram menos percebidas pela FindBugs.

Desse modo, baseando nos resultados da $CDL_R(o_k)$ de cada operador, apresentados na Tabela 5.1, é proposto que sejam analisados os mutandes dos operadores na seguinte ordem:

- $1^{\underline{o}}$ PCI, com a menor $CDL_R(o_k)$
- $2^{\underline{o}}$ IOD, com a $2^{\underline{a}}$ menor $CDL_R(o_k)$
- $3^{\underline{o}}$ JID, com a $3^{\underline{a}}$ menor $CDL_R(o_k)$
- $4^{\underline{o}}$ IHI, com a $4^{\underline{a}}$ menor $CDL_R(o_k)$
- ...
- $45^{\underline{o}}$ JSD, com a $3^{\underline{a}}$ maior $CDL_R(o_k)$
- $46^{\underline{o}}$ JTI, com a $2^{\underline{a}}$ maior $CDL_R(o_k)$

- 47^{o} JTD, com a maior $CDL_R(o_k)$

Observe que na Tabela 5.3 os tipos de operadores estão ordenados de forma crescente de acordo com a $CDL_R(o_k)$ obtida por cada operador de mutação.

Para aplicar a abordagem incremental para análise de mutantes, foram definidas as seguintes funções:

Função de Custo Acumulado de Operador (*Cumulate Cost* $CC(o_k)$):

$$CC(o_k) = M_1 + M_2 + \cdots + M_{k-1} + M_k \tag{5-1}$$

Função de Redução de Custo de Operador (*Cost Reduction* $CR(o_k)$):

$$CR(o_k) = 1 - (CC(o_k)/Total\ de\ mutantes) \tag{5-2}$$

A Função 5-1 representa a quantidade acumulada de mutantes para serem analisados à medida que a abordagem de priorização incremental apresentada na Tabela 5.3 é seguida. A Função 5-1 é o total acumulado de mutantes analisados do operador da primeira linha (isto é, PCI) até os mutantes do operador da k-ésima linha da Tabela 5.3.

As colunas CC_C, CC_P e CC_H armazenam, respectivamente, o resultado da Função 5-1 sobre os mutantes gerados nos sistemas Cassandra, Apache POI e Hibernate. A coluna CC_{total} é o resultado da Função 5-1 aplicado aos três sistemas. Os custos acumulados para analisar os mutantes referentes aos 5 primeiros tipos de operadores (PCI, IOD, JID, IHI e IPC) são de 165 mutantes no sistema Cassandra, 1.001 mutantes no sistema Apache POI, 9.752 mutantes no sistema Hibernate e 10.918 mutantes considerando os três sistemas em conjunto.

Considerando os 16 operadores com $CDL_R(o_k) = 0\%$ (linhas 1-16 da Tabela 5.3), os custos acumulados para analisar os mutantes referentes a esses 16 operadores (PCI, IOD, JID, IHI, IPC, IOP, IOR, PMD, PCC, PPD, ISI, EOA, JDC, OMD, LOD e IHD) são de 174 mutantes no sistema Cassandra, 1.043 mutantes no sistema Apache POI, 10.033 mutantes no sistema Hibernate e 11.250 mutantes considerando os três sistemas em conjunto. O custo acumulado para análise dos mutantes destes 16 operadores corresponde a cerca de $8,42\%$ $(11.250/133.683)$ do total de mutantes.

Já a Função 5-2 representa a redução de custo (em %) referente ao total de mutantes que deixariam de ser analisados, caso sejam analisados apenas os mutantes do operador da primeira linha até o operador da k-ésima linha da Tabela 5.3. Assim, os mutantes do operador da linha $k+1$ da Tabela 5.3 não seriam analisados, por algum motivo (por exemplo: falta de recursos), ou seriam analisados posteriormente.

As colunas CR_C, CR_P, CR_H e CR_{total} armazenam, respectivamente, o resultado da Função 5-2 sobre os mutantes analisados nos sistemas Cassandra, Apache POI,

Tabela 5.3: *Priorização de Operadores de Acordo com a* $CDL_R(o_k)$

k	*Operador*	**Cassandra**			**Apache POI**			**Hibernate**			**Total**		
		M_C	CC_C	CR_C	M_P	CC_P	CR_P	M_H	CC_H	CR_H	M_{total}	CC_{total}	CR_{total}
1	PCI	90	90	97,26%	880	880	95,54%	7.975	7.975	92,79%	8.945	8.945	93,31%
2	IOD	47	137	95,83%	70	950	95,18%	1.389	9.364	91,54%	1.506	10.451	92,18%
3	JID	6	143	95,65%	16	966	95,10%	254	9.618	91,31%	276	10.727	91,98%
4	IHI	4	147	95,53%	5	971	95,08%	102	9.720	91,22%	111	10.838	91,89%
5	IPC	18	165	94,98%	30	1.001	94,92%	32	9.752	91,19%	80	10.918	91,83%
6	IOP	2	167	94,92%	1	1.002	94,92%	43	9.795	91,15%	46	10.964	91,80%
7	IOR	5	172	94,77%	16	1.018	94,84%	83	9.878	91,07%	104	11.068	91,72%
8	PMD	0	172	94,77%	0	1.018	94,84%	8	9.886	91,07%	8	11.076	91,71%
9	PCC	1	173	94,74%	20	1.038	94,74%	16	9.902	91,05%	37	11.113	91,69%
10	PPD	0	173	94,74%	0	1.038	94,74%	27	9.929	91,03%	27	11.140	91,67%
11	ISI	1	174	94,71%	1	1.039	94,73%	35	9.964	91,00%	37	11.177	91,64%
12	EOA	0	174	94,71%	0	1.039	94,73%	0	9.964	91,00%	0	11.177	91,64%
13	JDC	0	174	94,71%	0	1.039	94,73%	64	10.028	90,94%	64	11.241	91,59%
14	OMD	0	174	94,71%	3	1.042	94,72%	2	10.030	90,94%	5	11.246	91,59%
15	LOD	0	174	94,71%	0	1.042	94,72%	0	10.030	90,94%	0	11.246	91,59%
16	IHD	0	174	94,71%	1	1.043	94,71%	3	10.033	90,93%	4	11.250	91,58%
17	AOIU	172	346	89,48%	1.647	2.690	86,36%	2.907	12.940	88,31%	4.726	15.976	88,05%
18	EMM	0	346	89,48%	9	2.699	86,31%	1.437	14.377	87,01%	1.446	17.422	86,97%
19	AORS	9	355	89,21%	57	2.756	86,02%	408	14.785	86,64%	474	17.896	86,61%
20	LOI	238	593	81,97%	2.274	5.030	74,49%	4.830	19.615	82,28%	7.342	25.238	81,12%
21	EAM	38	631	80,81%	780	5.810	70,53%	11.851	31.466	71,57%	12.669	37.907	71,64%
22	OMR	15	646	80,36%	3	5.813	70,52%	2.215	33.681	69,57%	2.233	40.140	69,97%
23	VDL	66	712	78,35%	612	6.425	67,41%	1.719	35.400	68,02%	2.397	42.537	68,18%
24	ASRS	52	764	76,77%	674	7.099	64,00%	256	35.656	67,78%	982	43.519	67,45%
25	AODS	2	766	76,71%	3	7.102	63,98%	87	35.743	67,71%	92	43.611	67,38%
26	PNC	6	772	76,53%	0	7.102	63,98%	1.751	37.494	66,12%	1.757	45.368	66,06%
27	ODL	241	1.013	69,20%	2.087	9.189	53,40%	8.489	45.983	58,45%	10.817	56.185	57,97%
28	CDL	46	1.059	67,80%	523	9.712	50,74%	2.092	48.075	56,56%	2.661	58.846	55,98%
29	COD	12	1.071	67,44%	22	9.734	50,63%	727	48.802	55,91%	761	59.607	55,41%
30	SOR	2	1.073	67,38%	2	9.736	50,62%	0	48.802	55,91%	4	59.611	55,41%
31	JSI	40	1.113	66,16%	141	9.877	49,91%	1.509	50.311	54,54%	1.690	61.301	54,14%
32	PCD	0	1.113	66,16%	1	9.878	49,90%	5	50.316	54,54%	6	61.307	54,14%
33	AODU	4	1.117	66,04%	7	9.885	49,87%	53	50.369	54,49%	64	61.371	54,09%
34	ROR	528	1.645	49,98%	1.246	11.131	43,55%	9.999	60.368	45,46%	11.773	73.144	45,29%
35	PRV	21	1.666	49,35%	19	11.150	43,45%	1.183	61.551	44,39%	1.223	74.367	44,37%
36	AORB	192	1.858	43,51%	1.484	12.634	35,92%	2.284	63.835	42,32%	3.960	78.327	41,41%
37	EOC	0	1.858	43,51%	0	12.634	35,92%	26	63.861	42,30%	26	78.353	41,39%
38	COI	152	2.010	38,89%	290	12.924	34,45%	6.542	70.403	36,39%	6.984	85.337	36,16%
39	SDL	467	2.477	24,69%	2.502	15.426	21,76%	24.284	94.687	14,45%	27.253	112.590	15,78%
40	COR	56	2.533	22,99%	46	15.472	21,53%	2.442	97.129	12,24%	2.544	115.134	13,88%
41	LOR	12	2.545	22,62%	88	15.560	21,08%	8	97.137	12,23%	108	115.242	13,79%
42	AOIS	602	3.147	4,32%	3.858	19.418	1,52%	7.792	104.929	5,19%	12.252	127.494	4,63%
43	OAN	15	3.162	3,86%	16	19.434	1,44%	2.102	107.031	3,29%	2.133	129.627	3,03%
44	ISD	0	3.162	3,86%	0	19.434	1,44%	20	107.051	3,28%	20	129.647	3,02%
45	JSD	27	3.189	3,04%	191	19.625	0,47%	736	107.787	2,61%	954	130.601	2,31%
46	JTI	89	3.278	0,33%	49	19.674	0,22%	2.133	109.920	0,68%	2.271	132.872	0,61%
47	JTD	11	3.289	0,00%	43	19.717	0,00%	757	110.677	0,00%	811	133.683	0,00%
Total		3.289	–	–	19.717	–	–	110.677	–	–	133.683	–	–

Hibernate e nos três sistemas. As reduções de custos proporcionadas para analisar apenas os mutantes dos 5 primeiros operadores (PCI, IOD, JID, IHI e IPC) são de 94,98% para o sistema Cassandra, 94,92% para o Apache POI, 91,19% para o Hibernate e 91,83% considerando os três sistemas.

Observe que a redução de custo ainda se mantém acima de 90% aplicando os 16 operadores com $CDL_R(o_k) = 0\%$ (linhas 1-16 da Tabela 5.3). As reduções de custos proporcionadas para analisar os mutantes referentes a estes 16 operadores (PCI, IOD, JID, IHI, IPC, IOP, IOR, PMD, PCC, PPD, ISI, EOA, JDC, OMD, LOD e IHD) são de 94,71% para o sistema Cassandra, 94,71% para o Apache POI, 90,93% para o Hibernate e 91,58% considerando os três sistemas.

Na Figura 5.1 são ilustrados o custo acumulado ($CC(o_k)$) e a redução de custo ($CR(o_k)$) obtidos na abordagem de priorização de análise de mutantes sugerida de acordo com o $CDL_R(o_k)$. No eixo X da Figura 5.1, há os operadores (e valor do respectivo custo acumulado) na mesma ordem apresentada na Tabela 5.3. Já no eixo Y, há o valor da redução de custo (*RC*) e o valor do custo acumulado (*CC*) a medida que os mutantes dos operadores são analisados na ordem sugerida.

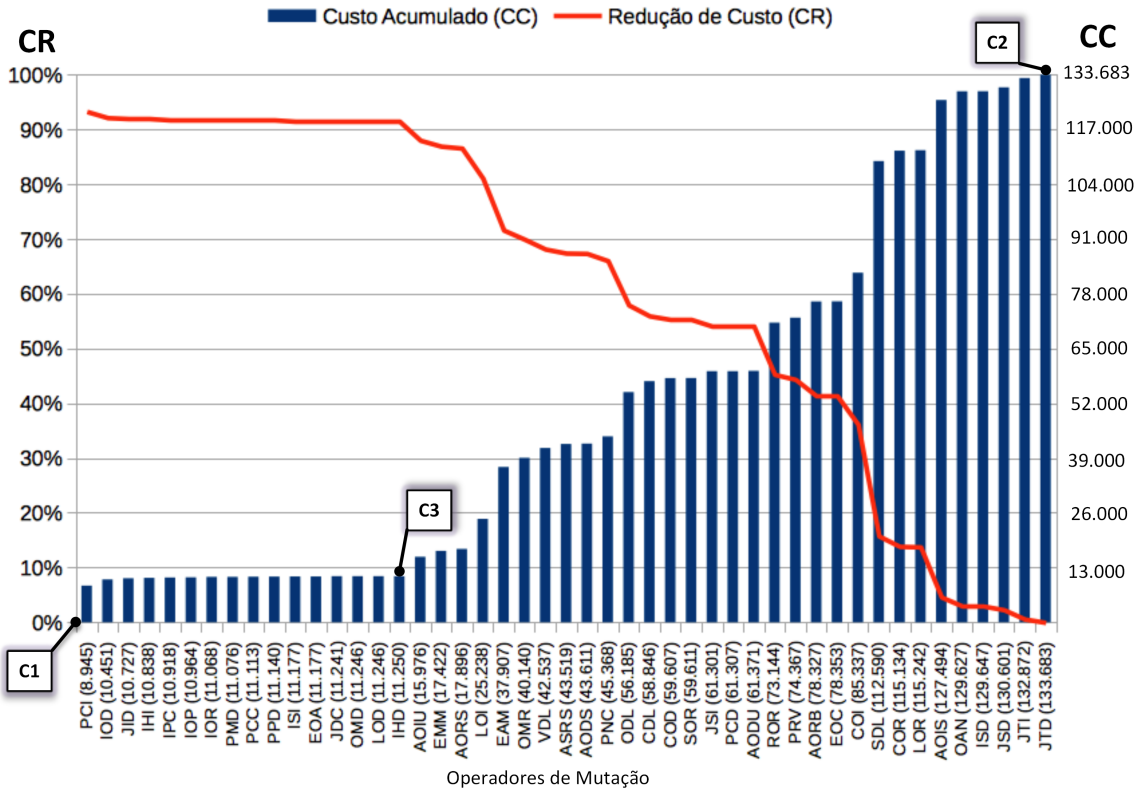

Figura 5.1: *Custo acumulado ($CC(o_k)$) versus redução de custo ($CR(o_k)$) na abordagem de priorização para análise de mutantes, de acordo com a $CDL_R(o_k)$*

Didaticamente, a Figura 5.1 que apresenta o custo acumulado e a redução de custo referentes à análise dos mutantes por operador pode ser compreendida por meio de três cenários:

Cenário 1: mutantes de nenhum tipo de operador é analisado. Este caso ilustra o cenário no qual nenhum dos mutantes gerados (de qualquer operador) é analisado, por exemplo, por falta de recursos. Na Figura 5.1, este cenário ocorre no início do gráfico (representado por $C1$) com $CC(o_k) = 0$ (nenhum mutante é analisado) e $CR(o_k) = 100\%$ (a redução é de 100%, pois todos os mutantes deixaram de ser analisados);

Cenário 2: mutantes de todos os operadores são analisados. Este caso ilustra o cenário no qual todos os mutantes, de todos os operadores, são analisados, o que pode ser impraticável devido ao custo da aplicação do Teste de Mutação. Na Figura 5.1, este cenário ocorre no final do gráfico (representado por $C2$), com $CC(o_k) = 133.683$ (todos os mutantes gerados são analisados) e $CR(o_k) = 0\%$ (a redução de custo é de 0%, porque nenhum mutante gerado ficou sem ser analisado);

Cenário 3: apenas os mutantes de alguns tipos de operadores são analisados. Este caso ilustra um cenário intermediário, no qual apenas mutantes de alguns tipos de operadores são analisados. Na Figura 5.1, este cenário ocorre no gráfico com $0 < CC(o_k) < 133.683$ e $100\% > CR(o_k) > 0\%$, como por exemplo o ponto representado por $C3$. Assim, os mutantes são analisados na ordem de priorização sugerida de acordo com a $CDL_R(o_k)$. Isso pode permitir a otimização dos recursos para a aplicação do Teste de Mutação, porque mutantes dos operadores de mutação com as menores $CDL_R(o_k)$ seriam analisados primeiro, pela razão da FindBugs não ter percebido as mutações produzidas por tais operadores.

5.4 Considerações Finais do Capítulo

Nesse capítulo foi apresentado o estudo experimental para avaliar a correspondênca entre mutações e avisos estáticos, do ponto de vista dos operadores de mutação. Os 19 sistemas utilizados nesse estudo são os mesmos apresentados no Capítulo 4. Foram definidas as funções utilizadas para calcular a correspondência direta entre mutações e avisos estáticos, de acordo com o operador de mutação Foi apresentado um exemplo de aplicação da abordagem de análise incremental, sugerida a partir da $CDL_R(o_k)$ por operador.

CAPÍTULO 6

Análise dos Resultados

Este capítulo apresenta uma análise mais detalhada dos resultados obtidos nos Capítulos 4 (*CDL* por aviso) e 5 (*CDL* por operador), documenta as lições aprendidas e identifica as ameaças de validação do presente estudo.

6.1 Análise Geral

Como pode ser observado na Tabela 5.1, há uma variedade na taxa de correspondência entre diferentes tipos de operadores. A Tabela 5.1 (linhas 32 - 47) mostra que 16 operadores (cerca de 33% do total) tiveram taxa de correspondência igual a zero ($CDL_R(o_k) = 0\%$). Assim, para esses operadores a ferramenta FindBugs não foi capaz de relatar qualquer aviso no mesmo local onde ocorreu a mutação. Isso evidência que a FindBugs não é capaz de detectar o defeito modelado por tais operadores. 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.

As melhores taxas de correspondência por tipo de operador são do operador JTD com $87,90\%$, JTI com $72,50\%$, JSD com $66,20\%$ e ISD com $49,30\%$. Esses resultados podem ser considerados promissores e é esperado que com o aumento do número de sistemas, seja possível identificar outras categorias de defeitos representados por outros operadores de mutação, os quais podem contribuir para uma otimização de testes e para o estabelecimento de estratégias incrementais que combinem análise estática e análise dinâmica de modo mais eficiente e efetivo.

A partir dos resultados mostrados nas Tabelas 4.3 (*CDL* por aviso) e 5.1 (*CDL* por operador), são apresentadas abordagens de priorização incremental para análise dos tipos de avisos da FindBugs (Seção 4.3) e para análise dos mutantes dos operadores de mutação da μJava (Seção 5.2).

No caso dos avisos, a ordem sugerida é que os avisos com maiores taxas de correspondência sejam analisados antes dos avisos com menores taxas de correspondência. No caso dos operadores de mutação, a ordem sugerida é que os mutantes dos operadores

com menores taxas sejam analisados antes dos mutantes dos operadores de maiores taxas de correspondência.

Nas Seções 6.2 e 6.3 são apresentados mais detalhes relacionados aos resultados obtidos para o tipo de operador e para o tipo de aviso, respectivamente.

6.2 Análise da CDL por Operador

A Tabela 6.1 mostra os operadores de mutação que obtiveram as maiores taxas de correspondência direta por linha com avisos da FindBugs.

Tabela 6.1: *Operadores com Alta Correspondência*

#	Operador	Categoria	Descrição do Operador
1	JTD	Java Specific	`this` keyword deletion
2	JTI	Java Specific	`this` keyword insertion
3	JSD	Java Specific	`static` modifier deletion
4	ISD	Inheritance	`super` keyword deletion

O operador JTD, que pertence à categoria `Java Specific feature`, tem taxa de correspondência de 87,90% e é responsável por simular defeitos por meio da remoção da palavra reservada `this`. Os operadores JTI e JSD, também da categoria `Java Specific feature`, são responsáveis por modelar defeitos relacionados, respectivamente, à inserção do termo `this` e à remoção do termo `static`. Tais operadores têm taxas de correspondência de 72,50% e 66,20%, respectivamente. O quarto operador de mutação é o ISD, que pertence à categoria *Inheritance*, tem taxa de correspondência de 49,30% e modela defeitos relacionados à remoção do termo `super`.

Observe que isso não significa que a FindBugs não seja capaz de relatar avisos para outros tipos de defeitos mas, considerando o conjunto de defeitos modelados por operadores de mutação implementados na μJava, os operadores de mutação JTD, JTI, JSD e ISD tiveram as maiores taxas de correspondência direta por linha.

Outra avaliação é a identificação das categorias de defeito que tiveram baixa correspondência, isto é, o conjunto de operadores com correspondência igual a zero. Eles correspondem aos operadores apresentados nas últimas linhas da Tabela 5.1 (linhas 32 – 47). O único operador da categoria de método entre os 16 operadores com menores taxas de correspondência é o LOD – responsável pela remoção de operadores lógicos (Tabela 6.2).

Tabela 6.2: *Operadores de Método com Baixa Correspondência*

#	Operador	Categoria	Descrição do Operador
33	LOD	Logical	Logical Operator Deletion

Há 15 operadores de classe (Tabela 6.3) que geraram mutações que não foram detectadas pela FindBugs. Eles modelam defeitos de diferentes categorias:

- 1 operador relacionado a enganos comuns sobre o uso de atribuição de referência em vez de clonar o conteúdo do objeto (EOA);
- 7 operadores relacionados a herança sobre exclusão e remoção de um atributo em uma subclasse com o mesmo nome de um atributo na superclasse (IHD e IHI), ou excluindo ou renomeando métodos em uma subclasse com os mesmos nomes de métodos da superclasse (IOD e IOR), ou remoção da chamada para `super()` no construtor da subclasse (IPC), ou inserção da palavra chave `super` (ISI), ou mover a posição da chamada de métodos de substituição (IOP);
- 2 operadores de características específicas de Java relacionados à remoção do construtor padrão se ele existir (JDC) à remoção da inicialização de variáveis de instância em declarações (JID);
- 1 operador está relacionado com a remoção da sobrecarga em uma subclasse (OMD);
- 4 operadores relacionados ao recurso de polimorfismo em uma declaração de variável para um tipo de superclasse (PMD), alterando o tipo de uma classe superclasse para sua subclasse (PPD), e do tipo de inserção antes de lançar variáveis de referência (PCI).

Tais informações podem utilizadas como subsídio para melhorar a FindBugs, definindo-se novas regras que possam tentar capturar esses possíveis tipos de defeitos. Como, por exemplo, por meio da criação de nova regra para verificar a existência do construtor padrão (JDC) ou verificar se há a necessidade de sobrecarregar os métodos da superclasse (OMD). Além disso, assumindo estratégias incrementais quando se combina a análise estática e dinâmica, este conjunto de operadores de mutação cujas mutações não são detectadas pela FindBugs pode ser utilizado primeiro, baseado no conceito de mutação seletiva, visando a reduzir o custo total do Teste de Mutação.

6.3 Análise da CDL por Aviso

Cada tipo de aviso da FindBugs tem uma prioridade e pertence a uma categoria específica indicando a que tipo de falha tal aviso está relacionado (HOVEMEYER; PUGH, 2004).

Em geral, os resultados obtidos da taxa de correspondência, considerando todos os avisos, foi de $5,18\%$. Contudo, alguns avisos apresentaram taxa de correspondência superior a 50%. Tais avisos são apresentados na Tabela 6.4. Foram 37 tipos de avisos com

Tabela 6.3: *Operadores de Classe com Baixa Correspondência*

#	Operador	Categoria	Descrição do Operador
32	IHD	Inheritance	Hiding variable deletion
34	OMD	Overloading	Overloading method deletion
35	JDC	Java Specific	Java-supported default constructor creation
36	EOA	Common Mistake	Reference assignment and content assignment replacement
37	ISI	Inheritance	`super` keyword insertion
38	PPD	Polymorphism	Parameter variable declaration with child class type
39	PCC	Polymorphism	Cast type change
40	PMD	Polymorphism	Instance variable declaration with parent class type
41	IOR	Inheritance	Overriding method rename
42	IOP	Inheritance	Overriding method calling position change
43	IPC	Inheritance	Explicit call of a parent's constructor deletion
44	IHI	Inheritance	Hiding variable insertion
45	JID	Java Specific	Member variable initialization deletion
46	IOD	Inheritance	Overriding method deletion
47	PCI	Polymorphism	Type cast operator insertion

$CDL_R(w) > 50\%$, os quais são de diversas prioridades e categorias. A Tabela 6.4 apresenta informações adicionais relacionadas aos avisos que obtiveram taxa de $CDL_R(w) > 50\%$.

Estes 37 tipos de avisos estão divididos nas seguintes prioridades: $43,2\%$ são de prioridade 1 $(16/37)$, 46% são de prioridade 2 $(17/37)$ e $10,8\%$ são de prioridade 3 $(4/37)$.

Estes 37 tipos de avisos estão divididos nas seguintes categorias (Tabela 6.4): $51,3\%$ dos avisos são da categoria CORRECTNESS $(19/37)$, $35,2\%$ são da categoria STYLE $(13/37)$, $8,1\%$ de PERFORMANCE $(3/37)$, $2,7\%$ BAD_PRACTICE $(1/37)$ e $2,7\%$ MALICIOUS_CODE $(1/37)$.

Nenhum aviso das categorias INTERNATIONALIZATION, MULTITHREADED CORRECTNES e SECURITY está entre os 37 com $CDL_R(w) > 50\%$.

A Tabela 6.5 apresenta informações adicionais relacionadas aos tipos de avisos que obtiveram taxa de correspondência $CDL_R(w) \approx 0.00\%$. Esses avisos são das seguites categorias: $33,3\%$ são da categoria PERFORMANCE $(6/18)$, $22,2\%$ da categoria BAD_PRACTICE $(4/18)$, $11,1\%$ da categoria INTERNATIONALIZATION $(2/18)$, $11,1\%$ são da categoria STYLE $(2/18)$, $16,2\%$ da categoria CORRECTNESS $(3/18)$ e $5,5\%$ da categoria MALICIOUS_CODE $(1/18)$. Com relação a prioridade, os avisos com $CDL_R(w) \approx 0.00\%$ estão divididos nas seguintes prioridades: $5,5\%$ são de prioridade 1 $(1/18)$, $61,2\%$ são de prioridade 2 $(11/18)$ e $33,3\%$ são de prioridade 3 $(6/18)$.

6.4 Lições Aprendidas

Os resultados aqui apresentados mostram evidências de que uma correspondência entre mutações e avisos existe. Esse resultado pode ser utilizado para o estabeleci-

Tabela 6.4: *Detalhes dos Avisos com Alta Correspondência*

#	Bug Kind	Bug Pattern (Tipo de Aviso)	P	Categoria	Descrição
1	DLS	DLS_DEAD_LOCAL_STORE_IN_RETURN	3	Dodgy code	Useless assignment in return statement
2	QF	QF_QUESTIONABLE_FOR_LOOP	2	Dodgy code	Complicated, subtle or wrong increment in for-loop
3	MS	MS_FINAL_PKGPROTECT	2	Malicious code ...	Field should be both final and package protected
4	INT	INT_BAD_COMPARISON_WITH_SIGNED_BYTE	3	Correctness	Bad comparison of signed byte
5	BIT	BIT_AND	1	Correctness	Incompatible bit masks
6	INT	INT_VACUOUS_BIT_OPERATION	2	Dodgy code	Vacuous bit mask operation on integer value
7	RE	RE_BAD_SYNTAX_FOR_REGULAR_EXPRESSION	1	Correctness	Invalid syntax for regular expression
8	ICAST	ICAST_INT_CAST_TO_DOUBLE_PASSED_TO_CEIL	1	Correctness	Integral value cast to double and then passed to Math.ceil
9	BIT	BIT_ADD_OF_SIGNED_BYTE	2	Correctness	Bitwise add of signed byte value
10	Bx	BX_UNBOXING_IMMEDIATELY_REBOXED	2	Performance	Boxed value is unboxed and then immediately reboxed
11	DLS	DLS_OVERWRITTEN_INCREMENT	1	Correctness	Overwritten increment
12	NS	NS_DANGEROUS_NON_SHORT_CIRCUIT	1	Dodgy code	Potentially dangerous use of non-short-circuit logic
13	RV	RV_01_TO_INT	1	Correctness	Random value from 0 to 1 is coerced to the integer 0
14	UM	UM_UNNECESSARY_MATH	3	Performance	Method calls static Math class method on a constant value
15	DMI	DMI_HARDCODED_ABSOLUTE_FILENAME	2	Dodgy code	Code contains a hard coded reference to an absolute pathname
16	QBA	QBA_QUESTIONABLE_BOOLEAN_ASSIGNMENT	1	Correctness	Method assigns boolean literal in boolean expression
17	SA	SA_LOCAL_SELF_COMPUTATION	1	Correctness	Nonsensical self computation involving a variable (e.g., x & x)
18	Bx	BX_BOXING_IMMEDIATELY_UNBOXED	2	Performance	Primitive value is boxed and then immediately unboxed
19	EC	EC_UNRELATED_CLASS_AND_INTERFACE	1	Correctness	Call to equals() comparing unrelated class and interface
20	SA	SA_FIELD_SELF_ASSIGNMENT	1	Correctness	Self assignment of field
21	SA	SA_LOCAL_SELF_ASSIGNMENT	2	Dodgy code	Self assignment of local variable
22	UCF	UCF_USELESS_CONTROL_FLOW_NEXT_LINE	1	Dodgy code	Useless control flow to next line
23	DLS	DLS_DEAD_LOCAL_INCREMENT_IN_RETURN	2	Correctness	Useless increment in return statement
24	INT	INT_BAD_REM_BY_1	1	Dodgy code	Integer remainder modulo 1
25	IL	IL_INFINITE_RECURSIVE_LOOP	1	Correctness	An apparent infinite recursive loop
26	INT	INT_VACUOUS_COMPARISON	2	Dodgy code	Vacuous comparison of integer value
27	INT	INT_BAD_COMPARISON_WITH_NONNEGATIVE_VALUE	1	Correctness	Bad comparison of nonnegative value with negative constant or zero
28	BIT	BIT_IOR	1	Correctness	Incompatible bit masks
29	BIT	BIT_SIGNED_CHECK	3	Bad practice	Check for sign of bitwise operation
30	UR	UR_UNINIT_READ	2	Correctness	Uninitialized read of field in constructor
31	DMI	DMI_USELESS_SUBSTRING	2	Dodgy code	Invocation of substring(0), which returns the original value
32	NP	NP_NULL_ON_SOME_PATH_MIGHT_BE_INFEASIBLE	2	Dodgy code	Possible null pointer dereference on branch that might be infeasible
33	UCF	UCF_USELESS_CONTROL_FLOW	2	Dodgy code	Useless control flow
34	SA	SA_FIELD_DOUBLE_ASSIGNMENT	2	Dodgy code	Double assignment of field
35	BIT	BIT_AND_ZZ	1	Correctness	Check to see if ((...) & 0) == 0
36	DLS	DLS_DEAD_STORE_OF_CLASS_LITERAL	2	Correctness	Dead store of class literal
37	SA	SA_FIELD_SELF_COMPUTATION	2	Correctness	Nonsensical self computation involving a field (e.g., x & x)
...	...	...	...	...	...

mento de estratégias de teste complementares combinando técnicas de análise estática e dinâmica.

Custo benefício da abordagem sugerida para o tipo de aviso:

Benefício: o valor obtido da $CDL_R(w)$ de cada aviso w é usado para estabelecer uma ordem de priorização para analisar os tipos de avisos. Com a utilização da ordem de prioridade sugerida, é de esperar que os avisos verdadeiros positivos sejam analisados tão rapidamente quanto possível, deixando os avisos considerados "menos importante" (com menor *CDL*) para serem analisados mais adiante, se ainda existirem recursos disponíveis.

Custo: No experimento realizado, há o custo para executar a FindBugs sobre o programa original e em seus mutantes para gerar os dados armazenados no banco de dados. No entanto, como os mutantes não necessitam de ser executados, e o tempo de geração é menor do que o tempo exigido pelo Teste de Mutação, uma vez determinada

Tabela 6.5: *Detalhes dos Avisos com Baixa Correspondência*

#	Bug Kind	Bug Pattern (Tipo de Aviso)	P	Categoria	Descrição
81	UwF	UWF_FIELD_NOT_INITIALIZED_IN_CONSTRUCTOR	3	Dodgy code	Field not initialized in constructor but dereferenced without null check
82	UrF	URF_UNREAD_FIELD	2	Performance	Unread field
83	MSF	MSF_MUTABLE_SERVLET_FIELD	3	Multithreaded ...	Mutable servlet field
84	IP	IP_PARAMETER_IS_DEAD_BUT_OVERWRITTEN	2	Correctness	A parameter is dead upon entry to a method but overwritten
85	Dm	DM_STRING_TOSTRING	3	Performance	Method invokes toString() method on a String
86	DE	DE_MIGHT_IGNORE	2	Bad practice	Method might ignore exception
87	OS	OS_OPEN_STREAM	2	Bad practice	Method may fail to close stream
88	IS	IS2_INCONSISTENT_SYNC	2	Multithreaded ...	Inconsistent synchronization
89	SBSC	SBSC_USE_STRINGBUFFER_CONCATENATION	2	Performance	Method concatenates strings using + in a loop
90	NP	NP_BOOLEAN_RETURN_NULL	2	Bad practice	Method with Boolean return type returns explicit null
91	Dm	DM_STRING_CTOR	2	Performance	Method invokes inefficient new String(String) constructor
92	REC	REC_CATCH_EXCEPTION	3	Dodgy code	Exception is caught when Exception is not thrown
93	Dm	DM_CONVERT_CASE	3	Internationalization	Consider using Locale parameterized version of invoked method
94	ES	ES_COMPARING_STRINGS_WITH_EQ	2	Bad practice	Comparison of String objects using == or !=
95	Dm	DM_DEFAULT_ENCODING	1	Internationalization	Reliance on default encoding
96	IIO	IIO_INEFFICIENT_INDEX_OF	3	Performance	
97	EI2	EI_EXPOSE_REP2	2	Malicious code ...	May expose internal representation by incorporating reference to mutable object
98	Bx	DM_NUMBER_CTOR	2	Performance	Method invokes inefficient Number constructor; use static valueOf instead

a ordem de priorização, conforme apresentado na Seção 4.3, o custo de utilizá-lo, como mostrado na Seção 4.4, é só o de aplicar a FindBugs no programa desejado e analisar os avisos relatados, seguindo a ordem sugerida.

Custo benefício da abordagem sugerida para o tipo de operador:

Benefício: o valor obtido da $CDL_R(o_k)$ de cada operador o_k é utilizado para estabelecer uma ordem de priorização para analisar os mutantes dos operadores. Com a utilização da ordem de prioridade sugerida, espera-se que o Teste de Mutação concentre inicialmente nos defeitos que são mais difíceis de serem detectados por analisadores estáticos.

Custo: uma vez determinada a ordem de priorização, conforme apresentado nas Seções 5.2, o custo de utilizá-lo, como mostrado na Seção 4.4, é só o de aplicar a Teste de Mutação no programa desejado e analisar os operadores seguindo a ordem sugerida.

6.5 Ameaças à Validade do Estudo

Respectivamente nas Seções 6.5.1, 6.5.2 e 6.5.3 são identificadas as ameaças à validade externa, à validação interna e as limitações observadas no presente estudo.

6.5.1 Validade Externa

Algumas limitações observadas no estudo estão relacionados com a linguagem de programação e as ferramentas de mutacão e de análise estática utilizadas. No estudo foi empregada apenas a linguagem Java e, portanto, os resultados não podem ser estendidos automaticamente para outras linguagens, especialmente para aquelas linguagens

de tipagem dinâmica. Em relação à ferramenta de análise estática, foi utilizada apenas a FindBugs e, por isso, os resultados não podem ser automaticamente estendido para outras ferramentas de análise estática, tais como a PMD, Checkstyle. Em relação à ferramenta de mutação, foi utilizada apenas a μJava e, por isso, os resultados também não podem ser estendido para outras ferramentas de mutação, tais como a Pitest (PITEST, 2015) e Major (MAJOR, 2015). Entretanto, a opção por Java é que essa linguagem aparece a alguns anos consectivos como uma das linguagens de programação mais utilizadas no desenvolvimento de aplicações (CASS, 2015). Além disso, apesar de existirem outras ferramentas de análise estática e de mutação, a FindBugs e a μJava são duas muito utilizadas em estudos presentes na literatura e, por esse motivos foram escolhidas.

6.5.2 Validade Interna

Inicialmente foram gerados mutantes dos sistemas mostrados na Tabela 4.1 por meio da ferramenta μJava (versão 4). Com o objetivo de gerar a maior quantidade de mutantes possível, foram selecionados todos os operadores disponíveis na ferramenta μJava.

A ferramenta FindBugs (versão 3.0.0) foi empregada com a opção `-low` para que os avisos de qualquer prioridade sejam relatados. Para calcular o local onde mutação ocorreu, foi necessário armazenar na base de dados a diferença textual entre cada mutante e o respectivo arquivo original (diff). Dos 427 diferentes tipos de avisos presentes na FindBugs, apenas 218 tipos diferentes de avisos foram relatados nos sistemas da Tabela 4.1, ou seja, há cerca de 210 tipos de avisos que não foram relatados em tais sistemas. Mesmo assim, o conjunto de programas empregado no estudo também foi utilizado em estudos anteriores e apresenta uma quantidade significativa de linhas de código.

6.5.3 Risco de Construção

Para controlar o processo de experimentação e reduzir o risco de uma análise incorreta dos dados, do processo empregado e da utilização do Teste de Mutação, foram selecionados inicialmente sistemas mais simples, com menor quantidade de linhas de código (Tabela 3.1) utilizados em experimentos anteriores por (POLO; PIATTINI; GARCÍA-RODRÍGUEZ, 2009).

Sobre mutantes equivalentes: um dos objetivos deste estudo é verificar se a mutação produzida no mutante é detectada pela FindBugs. Observe que para isso não é necessário executar o mutante com um caso de teste. Desse modo, nesta etapa do estudo, os mutantes não foram executados e por isso não há dados sobre mutantes vivos, mortos ou equivalentes. Contudo, pode ser construído um banco histórico para

operadores Java, similar ao que já existe para os operadores da linguagem C, para que informações relacionados aos mutantes equivalentes sejam obtidas por meio da Técnica Bayesian (VINCENZI et al., 2002).

6.6 Considerações Finais do Capítulo

Neste capítulo foi apresentada uma análise mais detalhada dos resultados apresentados nos Capítulos 4 e 5. Foram feitas duas análises: uma análise dos resultados obtidos para operadores de mutação; uma análise para os tipos avisos. As lições aprendidas e as ameaças de validação do presente estudo foram relatadas.

CAPÍTULO 7

Trabalhos Relacionados

Neste capítulo são apresentados os principais trabalhos relacionados ao estudo desenvolvido nesta dissertação.

Como referências em análise estática foram utilizados os trabalhos de Louridas (2006) e Ayewah et al. (2008). Em Hovemeyer e Pugh (2004) é apresentada uma técnica para localizar bugs em software. Em Ayewah et al. (2007b), a FindBugs foi empregada na produção de software.

Estudos sobre correlação entre defeitos de campo e avisos relatados pela ferramenta FindBugs foram relatados nos trabalhos Filho et al. (2010) e Couto et al. (2013). Em tais trabalhos são analisados dois tipos de correlação: a direta e a indireta. Na análise da correlação direta, o objetivo foi verificar se o defeito de campo (defeito reclamado por usuário) e o aviso relacionado ocorrem no mesmo local, isto é, na mesma classe e, melhor ainda, no mesmo método. No caso de correlação indireta, procurou-se demonstrar estatisticamente a relação entre a quantidade de defeitos de campo e a quantidade de avisos relatados. Como resultados, os estudos mostraram a existência da correlação indireta e a não existência de correlação direta. Em outras palavras, na correlação indireta, mostraram que quanto mais avisos tem um determinado sistema, maior é a quantidade de defeitos que serão encontrados nele; e na correlação direta, mostraram que os avisos relatados pela FindBugs, em geral, ocorrem em locais (classes e métodos) diferentes dos locais onde ocorrem os defeitos.

Para chegar aos resultados apresentados, aqueles estudos analisaram informações sobre defeitos reais em sistemas disponibilizados nos repositórios iBugs e Fundação Apache. Entretanto, essa abordagem de investigação de defeitos por meio de repositório (iBugs ou Fundação Apache) poderia está limitado a quantidade de defeitos investigados ou não explorar características existentes entre diferentes defeitos.

Para investigar a correspondência estática, nos trabalhos apresentados por Filho et al. (2010) e Couto et al. (2013) foram utilizados três sistemas que juntos somam cerca de 118 mil linhas de código e para os quais foram relatados 277 defeitos corretivos por meio das ferramentas Bugzilla e Jira. Para investigar a correspondência indireta, foi utilizado um conjunto de 30 sistemas da Fundação Apache, os quais somam cerca de

um milhão e meio de linhas de código e para os quais foram analisados 1.956 defeitos registrados por meio da ferramenta Jira.

Estudos relacionados à qualidade de software relatam taxas de defeitos superiores, em média, às taxas de defeitos investigadas nos trabalhos de Filho et al. (2010) e Couto et al. (2013). Em média, é esperado que seja encontrada uma taxa de defeitos de 10 a 25 defeitos por mil linhas de código em produtos de software (MCCONNELL; JOHANNIS, 2004). Casos nos quais as taxas de defeitos são 10 vezes menores, ou seja, taxas de defeitos entre 0,1 e 2,5 defeitos por mil linhas de código, são tratados como exceções, tais como: taxas de 3 defeitos por 1.000 linhas de código durante teste internos, e 0,1 defeitos por linhas de código em produção foram relatados em (COBB; MILLS, 1990); aplicativos da Microsoft têm entre 10 e 20 defeitos por 1.000 linhas em testes internos e 0,5 defeitos por 1.000 linhas de código em produção (MCCONNELL; JOHANNIS, 2004).

Desse modo, comparando a taxa média de defeitos normalmente encontradas em produtos de software com as taxas empregadas no estudo de Filho et al. (2010) e Couto et al. (2013), observa-se que uma possível limitação seria a baixa quantidade de defeitos reais armazenados nas versões disponíveis nos repositórios do iBugs e da Fundação Apache. Para investigar a correspondência direta foram analisados cerca de 2,35 defeitos por mil linhas de código (277 defeitos cadastrados por 118 mil linhas de código fonte). Ou seja, as taxas de defeitos por mil linhas de código utilizadas nos trabalhos de Filho et al. (2010) e Couto et al. (2013) são menores do que as taxas médias relatadas em (MCCONNELL; JOHANNIS, 2004). Além disso, no trabalho de Filho et al. (2010) e Couto et al. (2013) não foi feita uma identificação de quais classes de defeitos que a ferramenta de análise estáticaFindBugs é mais/menos hábil em detectar.

Ayewah et al. (2007a) examinaram diferentes tipos de avisos relatados pela FindBugs classificando-os em falsos positivos, defeitos simples e defeitos complexos.

Nagappan e Ball (2005) desenvolveram uma metodologia empírica para a projeção inicial de densidade de defeitos pré-lançamento com base nos resultados de duas ferramentas de análise estática. Elas foram capazes de estabelecer uma correspondência forte entre o número de avisos relatados pelas ferramentas de análise estática e a densidade de defeitos reais de pré-lançamento para o sistema Windows Server 2003.

Daimi, Banitaan e Liszka (2013) também procederam uma avaliação da performance de cinco analisadores estáticos Java. Eles avaliaram cinco ferramentas, utilizando o Eclipse, de acordo com três critérios diferentes: o número total de violações (avisos) encontrados, tempo de execução e consumo de memória. Com base nestes critérios, cada ferramenta foi avaliada de acordo com seis categorias de defeitos: defeitos de dados, defeitos de controle, defeitos de interface, defeitos de medição, código duplicado e violações

de convenções de código. Com base nessas categorias, códigos defeituosos eram injetados em três programas diferentes e cada ferramenta foi avaliada sobre essa versão defeituosa.

O trabalho apresentado nesta dissertação pode ser visto como complementar aos estudos relacionados, no sentido de que foi proposta a utilização de mutantes para avaliar a capacidade dos analisadores estáticos em detectar tais mutações. Os resultados obtidos até o momento indicam que informações sobre a correspondência podem ser utilizadas para priorizar as categorias de avisos em uma estratégia incremental, permitindo que o usuário possa selecionar categorias específicas de aviso que sejam mais capazes de detectar mutações, aumentando a chance de analisar um aviso verdadeiro positivo.

Também foi possível identificar categorias de avisos que não foram capazes de detectar qualquer tipo de mutação, o que sugere que estas categorias de avisos devem ser analisadas após aqueles avisos com melhor taxa de correspondência, se houver tempo e recursos disponíveis.

Os operadores de mutação não detectados pelo analisador estático também podem ser utilizados como fonte de informação para melhorar a capacidade do analisador estático na detecção de novos tipos de defeitos.

Mendonça et al. (2014) desenvolveu um sistema que permite o uso combinado de diferentes analisadores estáticos com o objetivo de diminuir a quantidade de aviso falso positivo.

Nos trabalhos de Filho et al. (2010) e Couto et al. (2013) a correspondência direta foi avaliada no nível de método. No trabalho apresentado nesta dissertação foi avaliada a correspondência direta no nível de linha de código, buscando identificar com mais precisão a existência da correspondência entre avisos e determinados tipos de defeitos modelados pelos operadores de mutação.

CAPÍTULO 8

Conclusão, Trabalhos Futuros e Publicações

8.1 Conclusão

No estudo relatado nesta dissertação foram identificadas evidências da correspondência direta entre alguns tipos de defeitos representados por operadores mutação da μJava e alguns tipos de avisos relatados pela ferramenta de análise estática FindBugs.

Tais evidências podem colaborar para a criação de estratégias complementares de testes que permitem que atividades de análise estática e dinâmica sejam empregadas em conjunto para encontrar defeitos a um custo menor.

Observe que foram identificados 16 operadores de mutação (últimas linhas da Tabela 5.1) cujas mutações a FindBugs não foi capaz de "perceber". Neste caso, pode-se sugerir a criação de novas regras de detecção para serem adicionadas à FindBugs para que ela detecte novas categorias de defeitos.

A partir dos resultados obtidos para os operadores de mutação, é possível reduzir o custo do Teste de Mutação, procurando analisar inicialmente os mutantes dos operadores cujos defeitos sejam mais difíceis de serem detectados por analisadores estáticos. Essas informações foram utilizadas para definir uma estratégia incremental para a análise dos mutantes dos operadores de mutação. Observe que os 16 operadores de mutação com menor $CDL_R(o_K)$ foram responsáveis por menos de 10% do número total de mutantes gerados nos sistemas Cassandra, Apache POI e Hibernate, de forma que estes operadores podem ser considerados um bom ponto de partida para a redução do custo da aplicação do Teste de Mutação.

Por outro lado, observou-se que 19 tipos de avisos tiveram correspondência de 100% (primeiras linhas da Tabela 4.3). Esses avisos foram capazes de detectar as mutações produzidas nos mutantes, gerados por determinados operadores de mutação.

A partir dos resultados obtidos para os tipos de avisos, é possível otimizar o custo da revisão, procurando analisar, inicialmente, os tipos de avisos que mais perceberam mutações. Com base na correspondência obtida por aviso, foi apresentada uma estratégia de priorização incremental para análise dos avisos de acordo com o seu grau de correspondência com mutações.

Considerando os resultados obtidos nos Capítulos 4 e 5, as respostas para as questões definidas no Capítulo 1 são:

- **Questão** Q_1: Existe alguma correspondência entre o local dos avisos estáticos e o local de mutação? Sim, existe para alguns tipos de avisos e mutações.
- **Questão** Q_2: Quais tipos de avisos conseguem perceber um maior/menor número de mutações – ou seja, as alterações no código – produzidas nos mutantes? Os tipos de avisos que mais perceberam mutações são os 37 primeiros avisos apresentados na Tabela 4.3 (linhas 1 - 37). Os tipos de avisos que menos perceberam mutações são os últimos da Tabela 4.3.
- **Questão** Q_3: Quais operadores geram mutações que são/não são detectadas por analisadores estáticos? Os operadores JTD, JTI, JSD e ISD foram aqueles cujas mutações mais foram detectadas pela FindBugs. Já os operadores IHD, LOD, OMD, JDC, EOA, ISI, PPD, PCC, PMD, IOR, IOP, IPC, IHI, JID, IOD e PCI foram os que tiveram suas mutações não detectadas pela FindBugs.

O uso combinado de análise estática automatizada (que tem baixo custo, mas não é muito eficiente em encontrar defeitos) e do Teste de Mutação (que é eficiente para ajudar a encontrar defeitos, mas é muito caro de ser aplicado) permite o estabelecimento de estratégias de teste que maximizem as vantagens da análise estática automatizada e da análise dinâminca como, por exemplo, da aplicação do Teste de Mutação.

8.2 Principais Contribuições

Assim sendo, após a finalização desta dissertação, considera-se que a mesma produziu algumas contribuições relacionadas abaixo:

1. Uso alternativo do teste de mutação com a finalidade de avaliar analisadores estáticos. 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. Não se encontrou relato do uso do teste de mutação com a finalidade de comparar ou avaliar a eficácia de analisadores estáticos.

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. Tal resultado trouxe duas outras contribuições:

2.1 Uma sugestão de ordem de utilização das categorias de avisos visando, inicialmente, utilizar aqueles mais sensíveis em "perceber" mutações e, com isso, espera-se minimizar a ocorrência de falso positivo entre os avisos reportados. Observa-se que isso ainda precisa ser investigado experimentalmente mas, partindo-se do pressuposto que se tem pouco conhecimento sobre cada categoria de avisos e a taxa de falto positivo gerada em cada uma delas, a correspondência com mutações passa a ser uma informação adicional que permite sugerir essa priorização.

2.2 Uma sugestão de ordem de utilização de operadores de mutação uma vez que para alguns operadores, seus mutantes passaram despercebidos pela ferramenta de análise estática indicando, possivelmente, que eles modelam defeitos mais difíceis de serem "percebidos" estaticamente e que mereceriam ser utilizados no teste, para permitir uma melhor análise dinâmica do produto em teste.

3. Desenvolvimento de um arcabouço que possibilita o uso de mutação para a comparação de analisadores estáticos. A estrutura empregada na comparação da FindBugs com a μJava pode ser utilizada para comparar outras ferramentas de análise estática com mutantes gerados por outras ferramentas de mutação ou até mesmo defeitos reais obtidos a partir de sistemas gerenciadores de defeitos.

4. Publicação/submissão de trabalhos científicos (listados na Seção 8.4)

5. Formação de recursos humanos tanto em nível de mestrado quanto de iniciação científica.

Finalmente, tais contribuições abrem outras possibilidades de pesquisa, algumas delas listadas na Seção 8.3, que discute os trabalhos futuros.

8.3 Trabalhos Futuros

O estudo relatado nesta dissertação pode ser melhorado por meio dos seguintes trabalhos futuros:

- **Incorporar outras ferramentas de análise estática para Java:** neste trabalho foi utilizada apenas a ferramenta FindBugs. Outras ferramentas de análise estática (PMD, Checkstyle, etc) podem ser incorporadas no estudo para que sejam feitas novas análises a partir dos resultados obtidos por meio de diferentes analisadores estáticos;

- **Incorporar outras ferramentas de mutação:** neste trabalho foi utilizado a ferramenta μJava. Há várias ferramentas de mutação para diferentes linguagens: para Java há a Pitest, a Major, etc; para a linguagem C há a Proteum. Desse modo, outras ferramentas de mutação podem ser empregadas para que sejam feitos novos estudos a partir das diferentes mutações produzidas por tais ferramentas;

- **Estender o estudo para outras linguagens de programação:** este estudo pode ser estendido para outras linguagens de programação com seus respectivos analisadores estáticos e ferramentas de mutação;

- **Criar um serviço para ser utilizado como repositório de conhecimento sobre análise estática e defeitos:** pode-se criar um serviço para ser utilizado por diferentes linguagens, analisadores estáticos e ferramentas de mutação com o objetivo de estabelecer estratégias de testes que combinem atividades estáticas e dinâmicas de forma eficiente;

- **Estabelecer o Teste de Mutação como modelo para comparar diferentes analisadores estáticos:** o Teste de Mutação pode ser utilizado para comparar diferentes analisadores estáticos, ou seja, com o uso do Teste de Mutação avisos relatados por diferentes analisadores por ser comparados. Com isso, pode-se identificar relacionamentos entre avisos emitidos por diferentes analisadores estáticos.

8.4 Publicações

A pesquisa desenvolvida neste trabalho permitiu gerar, além desta dissertação, os seguintes artigos científicos:

1. Artigo: **Uma Investigação Inicial Sobre a Correlação entre Defeitos de Software Simulados por Mutantes e Avisos Relatados por uma Ferramenta de Análise Estática**
   Evento: SAST 2014 – 8th Brazilian Workshop on Systematic and Automated Software Testing
   Situação: Aceito (Artigo completo) e Apresentado

2. Artigo: ***An Investigation on the Correspondence between Mutations and Static Warnings***
   Evento: SEKE 2015 – 27th Conference on Software Engineering and Knowledge Engineering
   Situação: Aceito como *short paper*, mas não foi apresentado

3. Artigo: ***Investigating the Correspondence between Mutations and Static Warnings***
   Evento: SBES 2015 – 29ª edição Simpósio Brasileiro de Engenharia de Software
   Situação: Aceito (Artigo completo) e Apresentado.
   Selecionado entre os 5 melhores artigos do evento. Convidado para submissão de uma versão estendida do artigo para o JSERD.

4. Artigo: ***Combining Mutation Testing and Automatic Static Analyzer: Towards Improving Mutation-based Incremental Testing Strategies***
   Journal of Software Engineering Research and Development (JSERD)
   Situação: Enviado

Referências Bibliográficas

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.

AYEWAH, N. et al. Using static analysis to find bugs. Published by the IEEE Computter Society, 2008.

AYEWAH, N. et al. 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. et al. 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. Disponível em: <http://doi.acm.org/10.1145/1297846.1297897>.

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.

BURN, O. *Checkstyle–coding standard verifier*. jul. 2014. Tool's Homepage. Available at: <https://github.com/checkstyle/checkstyle>. Access on: 07/02/2014.

CASS, S. *The 2015 Top Ten Programming Languages*. 2015. <http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages>. Últimoacess23/11/2015.

COBB, R.; MILLS, H. Engineering software under statistical quality control. *Software, IEEE*, v. 7, n. 6, p. 45–54, Nov 1990. ISSN 0740-7459.

COPELAND, T. *PMD Applied: An Easy-to-use Guide for Developers*. Centennial Books, 2005. (An easy-to-use guide for developers). ISBN 9780976221418. Disponível em: <http://pmdapplied.com/>.

COUTO, C. et al. 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. Disponível em: <http://dx.doi.org/10.1007/s11219-011-9172-5>.

DAIMI, K.; BANITAAN, S.; LISZKA, K. Examining the performance of java static analyzers. In: *XI International Conference on Software Engineering Research and Practice – SERP'13*. Las Vegas, Nevada, EUA: CSREA Press, 2013. p. 225–230. ISBN 1-60132-260-7.

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. 34–41, abr. 1978.

EVANS, D.; LAROCHELLE, D. Improving security using extensible lightweight static analysis. *IEEE Softw.*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 19, n. 1, p. 42–51, jan. 2002. ISSN 0740-7459. Disponível em: <http://dx.doi.org/10.1109/52.976940>.

FILHO, J. E. de A. et al. A study on the correlation between field defects and warnings reported by a static analysis tool. 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. Disponível em: <http://doi.acm.org/10.1145/1052883.1052895>.

ISO, I. *IEEE, Systems and Software Engineering–Vocabulary*. [S.l.], 2010.

JOHNSON, B. et al. 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. Disponível em: <http://dl.acm.org/citation.cfm?id=2486788.2486877>.

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

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.; KWON, Y.-R.; OFFUTT, J. Inter-class mutation operators for Java. In: *13th International Symposium on Software Reliability Engineering - ISSRE'2002*. Annapolis, MD: [s.n.], 2002.

MA, Y.-S.; OFFUTT, J. *Description of Method-level Mutation Operators for Java*. nov. 2005. On-line document. Available at: <http://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf>. Accessed on: 08/10/2015.

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

MAJOR. *Major*. 2015. <http://mutation-testing.org/>. Último acesso 13/07/2015.

MALDONADO, J. C.; DELAMARO, M. E.; JINO, M. *Introdução ao Teste de Software*. [S.l.]: Editora Elsevier, 2007. 1st edition.

MCCONNELL, S.; JOHANNIS, D. *Code complete*. [S.l.]: Microsoft press Redmond, 2004.

MENDONÇA, V. R. L. d. et al. Estudo, definição e implementação de um sistema de recomendação para priorizar os avisos gerados por ferramentas de análise estática. Universidade Federal de Goiás, 2014.

MICROSOFT. *{ }StyleCop*. mar. 2014. Tool Homepage. Available at: <https://stylecop.codeplex.com/>. Access on: 07/02/2014.

NAGAPPAN, N.; BALL, T. Static analysis tools as early indicators of pre-release defect density. In: *Proceedings of the 27th International Conference on Software Engineering*. New York, NY, USA: ACM, 2005. (ICSE '05), p. 580–586. ISBN 1-58113-963-2. Disponível em: <http://doi.acm.org/10.1145/1062455.1062558>.

OFFUTT, A. J. et al. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering Methodology*, v. 5, n. 2, p. 99–118, abr. 1996.

OFFUTT, J. *MuJava*. 2014. Disponível em: <http://cs.gmu.edu/~offutt/mujava/>. Último acesso em 23/06/2014.

OFFUTT, J.; MA, Y.-S.; KWON, Y.-R. The class-level mutants of mujava. In: *Proceedings of the 2006 International Workshop on Automation of Software Test*. New York, NY, USA: ACM, 2006. (AST '06), p. 78–84. ISBN 1-59593-408-1. Disponível em: <http://doi.acm.org/10.1145/1138929.1138945>.

PITEST. *Pitest*. 2015. <http://pitest.org>. Último acesso 13/07/2015.

POHL, J. *Lint - C program verifier*. maio 2001. Tool's Man Page. Available at: <http://www.unix.com/man-page/FreeBSD/1/lint>. Access on: 07/02/2014.

POLO, M.; PIATTINI, M.; GARCÍA-RODRÍGUEZ, I. Decreasing the cost of mutation testing with second-order mutants. *Softw. Test. Verif. Reliab.*, John Wiley and Sons Ltd., Chichester, UK, v. 19, n. 2, p. 111–131, jun. 2009. ISSN 0960-0833. Disponível em: <http://dx.doi.org/10.1002/stvr.v19:2>.

SHEN, H.; FANG, J.; ZHAO, J. Efindbugs: Effective error ranking for findbugs. In: *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. Washington, DC, USA: IEEE Computer Society, 2011. (ICST '11), p. 299–308. ISBN 978-0-7695-4342-0.

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. et al. Bayesian-learning based guidelines to determine equivalent mutants. *International Journal of Software Engineering and Knowledge Engineering*, World Scientific, v. 12, n. 06, p. 675–689, 2002.