Um estudo sobre a associação entre abordagens de localização de bugs e ações/padrões de reparo dos bugs

Júlia Manfrin Dias

Universidade Federal de Uberlândia
Faculdade de Computação
Programa de Pós-Graduação em Ciência da Computação

Uberlândia
2024

Júlia Manfrin Dias

Um estudo sobre a associação entre abordagens de localização de bugs e ações/padrões de reparo dos bugs

Dissertação de mestrado apresentada ao Programa de Pós-graduação da Faculdade de Computação da Universidade Federal de Uberlândia como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Ciência da Computação

Orientador: Marcelo de Almeida Maia

Uberlândia
2024

Ficha Catalográfica Online do Sistema de Bibliotecas da UFU
com dados informados pelo(a) próprio(a) autor(a).

D541 Dias, Júlia Manfrin, 1992-
2024 Um estudo sobre a associação entre abordagens de localização de bugs e ações/padrões de reparo dos bugs [recurso eletrônico] / Júlia Manfrin Dias. - 2024.

Orientador: Marcelo de Almeida Maia.
Dissertação (Mestrado) - Universidade Federal de Uberlândia, Pós-graduação em Ciência da Computação.
Modo de acesso: Internet.
Disponível em: http://doi.org/10.14393/ufu.di.2024.641
Inclui bibliografia.

1. Computação. I. Maia, Marcelo de Almeida,1969-, (Orient.). II. Universidade Federal de Uberlândia. Pós-graduação em Ciência da Computação. III. Título.

CDU: 681.3

Bibliotecários responsáveis pela estrutura de acordo com o AACR2:

Gizele Cristine Nunes do Couto - CRB6/2091
Nelson Marcos Ferreira - CRB6/3074

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
Coordenação do Programa de Pós-Graduação em Ciência da Computação
Av. João Naves de Ávila, 2121, Bloco 1A, Sala 243 - Bairro Santa Mônica, Uberlândia-MG, CEP 38400-902
Telefone: (34) 3239-4470 - www.ppgco.facom.ufu.br - cpgfacom@ufu.br

ATA DE DEFESA - PÓS-GRADUAÇÃO

Programa de Pós-Graduação em:	Ciência da Computação				
Defesa de:	Dissertação, 36/2024, PPGCO				
Data:	02 de setembro de 2024	Hora de início:	14:00	Hora de encerramento:	16:20
Matrícula do Discente:	11912CCP015				
Nome do Discente:	Júlia Manfrin Dias				
Título do Trabalho:	Um estudo sobre a associação entre abordagens de localização de bugs e ações/padrões de reparo dos bugs				
Área de concentração:	Ciência da Computação				
Linha de pesquisa:	Engenharia de Software				
Projeto de Pesquisa de vinculação:	Smart Developer - Assistentes Automatizados para Apoio ao Desenvolvedor de Software				

Reuniu-se por videoconferência, a Banca Examinadora, designada pelo Colegiado do Programa de Pós-graduação em Ciência da Computação, assim composta: Professores Doutores: Stéphane Julia - FACOM/UFU, Eduardo Magno Lages Figueiredo - Instituto de Ciências Exatas, Departamento de Ciência da Computação/UFMG e Marcelo de Almeida Maia - FACOM/UFU, orientador do candidato.

Os examinadores participaram desde as seguintes localidades: Eduardo Magno Lages Figueiredo - Belo Horizonte/MG. Os outros membros da banca e o aluno participaram da cidade de Uberlândia.

Iniciando os trabalhos o presidente da mesa, Prof. Dr. Marcelo de Almeida Maia, apresentou a Comissão Examinadora e o candidato, agradeceu a presença do público, e concedeu á Discente a palavra para a exposição do seu trabalho. A duração da apresentação da Discente e o tempo de arguição e resposta foram conforme as normas do Programa.

A seguir o senhor presidente concedeu a palavra, pela ordem sucessivamente, aos examinadores, que passaram a arguir á candidato. Ultimada a arguição, que se desenvolveu dentro dos termos regimentais, a Banca, em sessão secreta, atribuiu o resultado final, considerando o candidato:

Aprovado

Esta defesa faz parte dos requisitos necessários à obtenção do título de Mestre.

O competente diploma será expedido após cumprimento dos demais requisitos, conforme as normas do Programa, a legislação pertinente e a regulamentação

interna da UFU.

Nada mais havendo a tratar foram encerrados os trabalhos. Foi lavrada a presente ata que após lida e achada conforme foi assinada pela Banca Examinadora.

Documento assinado eletronicamente por **Stéphane Julia**, **Professor(a) do Magistério Superior**, em 04/09/2024, às 09:29, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Eduardo Magno Lages Figueiredo**, **Usuário Externo**, em 04/09/2024, às 14:53, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Marcelo de Almeida Maia**, **Professor(a) do Magistério Superior**, em 06/09/2024, às 11:23, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site https://www.sei.ufu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **5644972** e o código CRC **9B6AAAB1**.

Referência: Processo nº 23117.055650/2024-66 SEI nº 5644972

Dedico a meu filho Rui.

Agradecimentos

Agradeço ao meu orientador pela paciência e compreensão.

"Eu só confio numa ferramenta na qual alguém ganha dinheiro com ela "
(Borges, Wellyngton M.)

Resumo

Em um processo de desenvolvimento de software ocorrem problemas que podem atrapalhar a execução do mesmo por envolver questões de custos e de tempo. Um problema bastante frequentes é a ocorrência de erros, que podem requerer um esforço considerável de reparo. A engenharia de software propõe abordagens para minimizar este problema. O tema de estudo deste trabalho envolve este processo de correção de erros, focando especialmente em uma tarefa preliminar, chamada de localização de bugs, que consiste em localizar onde está o erro no código. Para auxiliar o trabalho do desenvolvedor na tarefa de localização de erros, foram propostas diversas abordagens automatizadas. A proposta deste estudo é analisar o desempenho, em relação a acurácia, de diversos tipos de localizadores, com base nas características dos bugs. Estas características se referem as ações e padrões de reparo que são conduzidas para a correção. São exemplos de ações de reparo adições, remoções e modificações de linhas no código-fonte. Já os padrões de reparo são abstrações de alto nível de recorrências de estruturas de ações nos códigos reparados. O objetivo do trabalho é entender se existe uma relação entre os diferentes tipos de ações/padrões de reparo com a acurácia dos diferentes tipos de localizadores. O trabalho comparou diferentes técnicas de localização de bugs, como DStar, Ochiai, Metallaxis, Muse, Predicate Switching, Fatiamento e Stack Trace. Observou-se que técnicas baseadas em cobertura e mutação são mais eficazes para bugs que envolvem remoção ou mudança de linhas, enquanto a adição de linhas apresentou maior dificuldade. Além disso, bugs em expressões foram mais facilmente localizados, enquanto aqueles relacionados a tipos e declarações de métodos foram mais difíceis de identificar. A análise de padrões de reparo mostrou que mudanças constantes e correções de API são mais facilmente detectadas, enquanto bugs de verificação de referências nulas e movimentação de código são os mais desafiadores.

Palavras-chave: Correção de Bugs. Localização de bugs. Bugs. Defects4j.

Abstract

In a software development process, issues can arise that may hinder its execution due to cost and time concerns. A common problem is the occurrence of errors, which may require considerable effort to repair. Software engineering proposes approaches to minimize this problem. The study topic of this work involves the bug fixing process, focusing especially on a preliminary task, named bug localization, which consists in locating where the error is in the code. To assist the developer's task in bug localization, various automated approaches have been proposed. This study aims to analyze the performance, in terms of accuracy, of different types of locators, based on the characteristics of the bugs. These characteristics refer to the actions and repair patterns that are conducted for fixing. Examples of repair actions include additions, removals, and modifications of lines in the source code. Meanwhile, repair patterns are high-level abstractions of recurrences of action structures in repaired code. The objective of the work is to understand if there is a relationship between the different types of actions/repair patterns and the accuracy of the different types of locators. The study compared different bug localization techniques, such as DStar, Ochiai, Metallaxis, Muse, Predicate Switching, Slicing, and Stack Trace. It was observed that coverage-based and mutation-based techniques are more effective for bugs involving line removal or modification, while line addition presented more difficulty. Additionally, bugs in expressions were more easily located, whereas those related to types and method declarations were harder to identify. The analysis of repair patterns showed that constant changes and API fixes are more easily detected, while missing null checks and code movement were the most challenging.

Keywords: Bug fixes. Bug location. Bugs. Defects4j.

Lista de ilustrações

Lista de tabelas

Sumário

Capítulo 1

Introdução

A Engenharia de Software se preocupa com o ciclo de vida do desenvolvimento de sistemas, em especial com os procedimentos a serem seguidos para melhor conduzir este desenvolvimento. Resumidamente, as principais etapas do ciclo de desenvolvimento de software se constituem em análise de requisitos, projeto do sistema, codificação, testes, localização de bugs[1], correção do código e implantação do sistema (SOMMERVILLE, 2011). Esta pesquisa se concentra na fase de localização de bugs, com um estudo relacionado a fatores que podem estar associados ao desempenho, em relação à acurácia, das abordagens de localização.

Antes que ocorra uma atividade de localização de bug, testes prévios indicaram alguma falha na operação do sistema. Os testes são importantes para o processo de desenvolvimento de software pois indicam se o projeto está sendo executado conforme as especificações descritas pelo cliente. Quando o sistema entra em fase de testes e não tem o comportamento esperado de acordo com os requisitos, conclui-se que o programa não passou no teste (ou que o teste falhou). Após esta fase de testes relatar os bugs, a próxima etapa é localizar os componentes do sistema que estão causando os erros, para posteriormente repará-los.

A localização de bugs é uma etapa no processo de correção de código que se preocupa em procurar qual a classe, método ou mesmo comandos e elementos de código que estão causando as falhas no sistema. Todo este processo pode despender muito tempo, levando consequentemente, a uma diminuição da produtividade e eficiência da equipe envolvida no desenvolvimento do projeto(WEN et al., 2017).

Com o aumento da demanda por software ao longo dos anos, o processo de detecção, localização e correção de bugs nos códigos tem se tornado custoso e exaustivo para os desenvolvedores e empresas. Em resposta a essa situação, muitos pesquisadores têm se dedicado ao estudo e à proposta de técnicas para automatizar as diversas fases deste processo, em particular, a fase de localização de bugs (WONG et al., 2016).

As abordagens automatizadas para localização de bugs, além de eventualmente faci-

[1] Neste trabalho usaremos o termo bug, dado o seu uso generalizado e sua inclusão no dicionário Houaiss.

litar o trabalho dos desenvolvedores para reparar bugs, também tem uma importância direta em outra classe de ferramentas, chamada de reparo automático de bugs. A área de reparo automático de software foi uma das mais pesquisadas nos últimos anos, tendo produzido um grande número de ferramentas, dentre as quais pode-se citar o Nopol (XUAN et al., 2017), ARJA (YUAN; BANZHAF, 2019), GenProg (GOUES et al., 2012), JMutRepair (MARTINEZ; MONPERRUS, 2019), JKali (NILIZADEH et al., 2021), Cardumen (MARTINEZ; MONPERRUS, 2018), DynaMoth (DURIEUX; MONPERRUS, 2016) e NPEFix (DURIEUX et al., 2017). Esta importância decorre do fato de que a localização de bugs é uma etapa das ferramentas de reparo automático de bugs, as quais basicamente possuem uma etapa de identificação da existência do bug, localização do bug, i.e., dos trechos de código que precisam ser reparado, e finalmente da síntese e aplicação do reparo (DURIEUX et al., 2019).

Trabalhos recentes na literatura de reparo automático de software tem mostrado que existe características dos bugs que tem influência no desempenho das ferramentas de reparo de software em relação a sua acurácia (DURIEUX et al., 2019). Neste sentido, diversos *benchmarks* para avaliação de ferramentas de reparo de bugs tem sido propostas com o objetivo de verificar o quão uma ferramenta desempenha efetivamente em bugs de diferentes natureza, tais como o Bears (MADEIRAL et al., 2019), Swarm (TOMASSI et al., 2019), Defects4j (JUST; JALALI; ERNST, 2014) , Bugs.jar (SAHA et al., 2018), IntroClassJava (GOUES et al., 2015) e QuixBugs (LIN et al., 2017). Além disso, a comunidade tem tido interesse específico na análise detalhada dos *benchmarks* em relação às diversas características dos bugs, em particular, em relação à características dos reparos[2] (SOBREIRA et al., 2018). Este interesse vem da necessidade de entender se as ferramentas estão sendo customizadas para tipos mais específicos de reparos ou se estão sendo desenvolvidas para alcançar a variedade existente de bugs.

Portanto, são objetos de estudo deste trabalho, as abordagens de localização de bugs (LB) e características dos bugs relacionadas ao tipo de reparo (detalhadas na próxima seção). Assim, pretende-se investigar se a acurácia das abordagens de localização automática de bugs pode estar associada às características dos bugs, de tal forma, a entender melhor onde possíveis pontos de melhoria para as respectivas abordagens podem ser trabalhados e assim indiretamente contribuir para melhorar essa etapa crítica para desenvolvedores. A descrição detalhada de como se encaminhará esta pesquisa é apresentada nas seções e subseções que se seguem. A seção dos resultados mostram que existe pouca associação em relação às abordagens de localização de bugs e as ações e padrões de reparo. Tendo a estrutura sintática do tipo Expression a ação na qual os localizadores obtiveram melhor acurácia em termos de quantidade de bugs encontrados.

[2] Em inglês, *patches*

1.1 Motivação

Após a conclusão dos testes do sistema, onde os bugs são relatados é iniciado o processo de localizar estes bugs para corrigi-los.

As abordagens de localização de bugs (LB) utilizam seus próprios algoritmos para encontrar o bug e retornam uma lista contendo o número das possíveis linhas de código que provavelmente contém os bugs (WONG et al., 2016). Diversas abordagens foram propostas para a automatização deste processo de encontrar os bugs, e as técnicas estudadas nesta dissertação são Ochiai (ABREU; ZOETEWEIJ; GEMUND, 2007), DStar (WONG et al., 2014b), Muse (MOON et al., 2014), Metallaxis (PAPADAKIS; TRAON, 2015a), Fatiamento (AGRAWAL et al., 1995), Stacktrace (WONG et al., 2014a) e Predicateswitching (ZHANG; GUPTA; GUPTA, 2006).

Assim como as abordagens de localização podem automatizar o processo de encontrar os erros para a correção de um programa, é interessante também direcionar a atenção para os bugs, pois alguns *benchmarks*, como o Defects4j, fornecem algumas informações dos bugs. Estas informações podem auxiliar este processo de automatização de encontrar os erros, pois fornecem informações como histórico de versões, versões corrigidas por programadores e as versões do sistema com os erros.

Sendo assim, entende-se que pode ser relevante estudar algumas características próprias que os reparos de bugs apresentam. Sobreira e outros mostraram que os reparos de bugs podem ser classificados em relação a ações de reparo e padrões de reparo (SOBREIRA et al., 2018) . Estes últimos são de mais alto nível em relação às ações e são definidos como estruturas de modificações no código fonte original que foram importantes para o reparo.

As ações e padrões de reparo foram escolhidas nesta pesquisa a fim de se descobrir se elas podem explicar parte do resultado do processo de localização de bugs.

1.2 Objetivos e Desafios

O objetivo deste trabalho é contribuir indiretamente para a melhoria dos sistemas automatizados de localização de bugs por meio do entendimento de se (e como) as ações e padrões encontrados nos reparos dos bugs podem influenciar o desempenho das diferentes abordagens, e portanto indicar possíveis caminhos para uma automatização mais eficiente.

Mais especificamente, este trabalho tem o objetivo de verificar se há alguma relação entre ao desempenho (**em relação ao ranqueamento dos erros localizados**) de diferentes abordagens de localização de bugs e o tipo de ações e padrões de reparo de bugs. Para organizar o estudo, foram formuladas três perguntas de pesquisa para verificar se e como ocorre esta possível associação. As perguntas de pesquisa estão listadas a seguir.

- Pergunta 1: Existe uma associação entre o desempenho dos localizadores de bugs e as ações de reparo de acordo com o tipo de ação?

  Esta questão também tem o objetivo de verificar se os localizadores conseguem encontrar mais erros, isto é, possui melhor desempenho em bugs com os tipos de ações de reparo, em relação, a adição, modificação ou remoção de código.

- Pergunta 2: Existe uma associação entre o desempenho dos localizadores e a estrutura sintática das ações de reparo?

  Esta questão também tem o objetivo de verificar se os localizadores possuem melhor desempenho em bugs com ações de reparo de estruturas sintáticas específicas.

- Pergunta 3: Existe uma associação entre a acurácia dos localizadores e os padrões de reparo?

  Essa questão se preocupa em verificar se o desempenho dos localizadores, em relação a quantidade de erros encontrados, é melhor quando focados exclusivamente nos padrões de reparo.

1.3 Contribuições

A contribuição que espera-se deste trabalho é a indicação de classes de bugs, onde os localizadores apresentam melhor desempenho ou aspectos que podem ser melhorados nestes programas.

Neste estudo busca-se aprofundar nas informações e características dos bugs e seus reparos para verificar se estes elementos poderiam auxiliar na eficiência do processo de localização e reparo de bugs.

1.4 Organização da Dissertação

Esta dissertação está organizada em capítulos, que seguem da seguinte forma:

- Capítulo 2 - Fundamentação teórica. Neste capítulo será detalhado o que há na literatura sobre o assunto, com citações de artigos que abordam o tema e que são base para esta pesquisa.

- Capítulo 3 - Descrição do Estudo. Nesta seção será explicada a metodologia utilizada para se chegar aos propósitos da pesquisa, bem como a base de dados utilizada, e as respectivas análises conduzidas.

- Capítulo 4 – Resultados. Nesta seção será detalhado os resultados obtidos.

- Capítulo 5 – Trabalhos Relacionados. Os trabalhos relacionados e que deram embasamento a esta pesquisa serão apresentados neste capítulo.
- Capítulo 6 - Conclusão. Nesta seção estará o que se pode concluir de toda a pesquisa.

Capítulo 2

Fundamentação Teórica

Nesta seção serão apresentados os conceitos das abordagens de Localização de Bugs (LB's), a base de dados utilizada, que foi o Defects4j e as características dos bugs, que se referem as ações e padrões de reparo. Estes são os elementos centrais para conduzir a pesquisa, uma vez que esta procura estudar se as características dos bugs podem ser associadas ao desempenho dos diferentes localizadores.

2.1 Abordagens de Localização de Bugs

As abordagens de localização de bugs são algoritmos que têm o objetivo de encontrar os elementos que podem ser a causa das falhas nos códigos. Eles podem ser implementados junto ao código de uma ferramenta de reparo ou estarem contidos em módulos ou funções separados. As abordagens de Localização de Bugs se diferem entre si dependendo da técnicas usadas nos algoritmos.

A tabela 2.1 abaixo indica as principais características das abordagens de localização de bugs.

Técnicas de Localização de Bugs	**Técnica usada para localizar o bug**	**Saída do algoritmo**
Oichiai Dstar	Utiliza um cálculo matemático com base nos casos de testes que falharam	Lista de classes suspeitas
Metallaxis Muse	Mutação nas expressões e sentenças do programa	Sentença modificada
Slicing *Slicing_count* *Slicing_intersection*	Fatiamento do código fonte com base em um conjunto de variáveis	Sentenças de códigos suspeitos
Stacktrace	Utiliza uma pilha de frames durante a execução	Função ou método
Predicateswitching	Executa o caso de teste várias vezes, forçando um resultado diferente a cada execução	Predicado suspeito de conter um erro

Tabela 1 – Características das abordagens de localização de bugs.

As técnicas descritas acima serão melhor detalhadas a seguir.

2.1.1 Ochiai e Dstar

O Ochiai (ABREU; ZOETEWEIJ; GEMUND, 2007) e o Dstar (WONG et al., 2014b) são abordagens de localização baseada na análise dos casos de testes que falham e dos

casos que também obtiveram sucesso. Quanto mais um elemento ou comando do código é executado nos casos de teste que falham e menos nos casos de teste com sucesso, mais suspeito é aquele elemento de código de conter o bug. Estas técnicas usam uma fórmula matemática para calcular o quão suspeito aquele elemento é. O que diferencia a abordagem Dstar da abordagem Ochiai é a fórmula matemática utilizada para ranquear os elementos de código suspeitos.

A figura abaixo foi retirada do artigo de (MARINHO et al., 2023). Os pontos indicam se os métodos foram cobertos pelos casos de testes. O método 6 foi coberto apenas pelos testes que apresentam falhas.

Application: OSMTracker	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	Ochiai
class GPSLogger {...											
(1) public void onCreate() {...}	●	●	●	●	●	●	●	●	●	●	0.63
(2) public int onStartCommand(Intent intent, int flags, int startId) {...}	●	●	●	●	●	●	●		●	●	0.67
(3) public void onDestroy() {...}	●	●	●	●	●	●	●	●	●	●	0.63
(4) private void startTracking(long trackId) {...}	●	●	●	●	●	●			●	●	0.53
(5) private void stopTrackingAndSave() {...}	●	●	●	●	●	●			●	●	0.53
(6) public void onLocationChanged(Location location) {...} /* FAULT */	●		●	●			●				1.00
(7) private Notification getNotification() {...}	●	●	●	●	●	●	●		●	●	0.67
(8) private void createNotificationChannel() {...}					●	●					0.00
...}											
Test case outcomes (pass=✓, fail=X)	X	✓	X	X	✓	✓	X	✓	✓	✓	

Figura 1 – Exemplo do funciomanento da técnica Ochiai. Fonte: (MARINHO et al., 2023)

Fórmula do Ochiai:

$$S(s) = \frac{failed(s)}{\sqrt{totalfailed \cdot (failed(s) + passed(s))}} \tag{1}$$

Fórmula do DStar:

$$S(s) = \frac{failed(s)^*}{passed(s) + (totalfailed - failed(s))} \tag{2}$$

onde:

- s é uma instrução de código.
- *S(s)* é o Score.
- *failed(s)* são os casos de testes em que a instrução s falhou.
- *passed(s)* são os casos de testes em que a instrução s passou.
- *totalfailed* são todos os casos de testes em que o programa falhou.
- Na fórmula do DStar, o * é um expoente variável. O artigo de (PEARSON et al., 2017) considera este valor igual a 2.

Por exemplo, considerando valores arbitrários para as variáveis, pode-se obter exemplos de resolução destas fórmulas.

Considerando os valores:

totalfailed = 100 casos no total falharam; *failed(s)* = 68 casos que executaram s falharam; *passed(s)* = 32 casos que executaram s passaram

Tem-se que:

Exemplo de resolução da fórmula do Ochiai:

$$S(s) = \frac{68}{\sqrt{100 \cdot (68 + 32)}} = 0,68 \tag{3}$$

Exemplo de resolução de fórmula do DStar:

$$S(s) = \frac{68^2}{32 + (100 - 68)} = 72,25 \tag{4}$$

2.1.2 Metallaxis e Muse

As abordagens Metallaxis (PAPADAKIS; TRAON, 2015a) e Muse (MOON et al., 2014) utilizam o conceito de mutação para descobrir os erros nos códigos. A mutação significa uma troca de um elemento de código, que pode ser um operador ou um comando, por outro elemento. Considerando uma sentença de um código que seja 'if(x < y)..', um exemplo de mutação que poderia ocorrer seria a troca do operador, alterando a sentença para 'if(x <= y)...', outro exemplo de mutação também poderia ser a inserção de pequenos elementos, ficando a sentença assim: 'if(x < y -1)...' (PAPADAKIS; TRAON, 2015b). Após a realização desta mutação o código é submetido aos casos de testes, e o comportamento dos mutantes quando exercitados é observado. No caso de um mutante exibir um comportamento diferente do programa original, ele é chamado de "morto", enquanto no caso oposto é chamado de "vivo". O objetivo de se aplicar mutação em algum elemento do código é para descobrir se o elemento mutado consegue solucionar o bug. Depois, segue-se os mesmos passos das técnicas Ochiai e Dstar. Quanto mais o código passa nos casos de testes que falham e menos nos que dão sucesso, mais suspeito de conter o bug é considerado aquele código.

Ambos o Metallaxis e o Muse utilizam fórmulas para calcular os suspeitos e geram uma lista ranqueada de suspeitos. A diferença entre O Metallaxis e o Muse também está na fórmula. O Metallaxis utiliza a mesma fórmula que o Ochiai.

Fórmula do Metallaxis:

$$S(m) = \frac{failed(m)}{\sqrt{totalfailed \cdot (failed(m) + passed(m)}} \tag{5}$$

O MUSE segue um método diferente para identificar os comandos suspeitos. Em vez de usar a maneira tradicional de julgar se os mutantes são mortos ou não (com base nas

saídas do programa), ele considera os mutantes como mortos (ou não) apenas com base nos testes que passam e falham. Assim, o MUSE considera apenas os casos em que os mutantes transformam um caso de teste que passa em um que falha e vice-versa. Portanto, ele ignora os casos em que tanto os mutantes quanto o programa original (com defeito) falham, mas de maneiras diferentes, ou seja, suas saídas diferem.

Fórmula do Muse:

$$S(m) = failed(m) - \frac{f2p}{p2f} \cdot passed(m) \tag{6}$$

onde:

- m é o comando que sofreu a mutação.
- *S(m)* é o Score.
- *failed(m)* são os casos de testes em que o comando m falhou.
- *passed(m)* são os casos de testes em que o comando m passou.
- *f2p* (failed to passed) são os números de casos de testes em que o comando m falhava e depois da mutação passa no caso de teste.
- *p2f* (passed to failed) são os números de casos de testes em que o comando m passava e depois da mutação falha no teste.

2.1.3 Fatiamento com União, Fatiamento com Frequência e Fatiamento com Interseção

As abordagens do tipo *Fatiamento*[1] utilizam o conceito de criar fatias, as quais são conjuntos de linhas do programa (comandos) que podem interferir no valor de uma variável (ZOU et al., 2021).

Quando um bug aparece no programa geralmente alguma variável deste programa apresenta um valor incorreto e o programa é pausado. É a partir desta variável incorreta que o algoritmo faz o fatiamento do programa.

O fatiamento ocorre com a identificação de todas as sentenças no código, desde o início do programa, que direta ou indiretamente afeta aquela variável com valor indesejável. Após esse processo de identificação é criado um conjunto (a fatia) com todas aquelas sentenças que de fato contribuem para que um valor de uma variável saia como o esperado ou não.

A figura acima foi extraída do artigo de (ZHANG; GUPTA; GUPTA, 2007). Os valores de entrada são i =3, j = 6 e k = 6. As linhas destacadas são as instruções que podem

[1] Do inglês, slicing

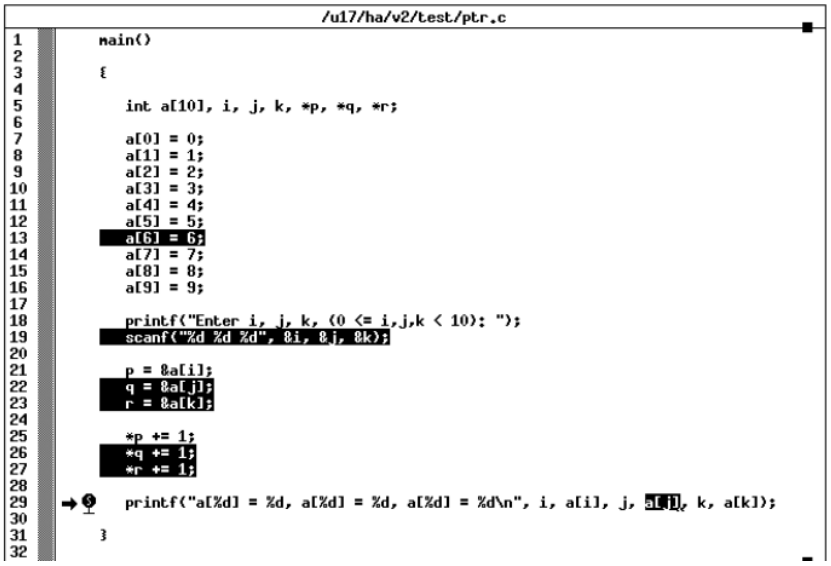

Figura 2 – Exemplo de como ocorre o fatiamento com base no valor incorreto de uma variável. Fonte: (ZHANG; GUPTA; GUPTA, 2007)

afetar o valor de saída da variável `a[j]`. Todas estas linhas fazem parte do fatiamento que o algoritmo cria.

Esta abordagem também utiliza os casos de testes que falharam para identificar as variáveis com valores errados e aplicar o algoritmo de fatiamento.

Como podem haver vários testes com falhas, são criadas mais de uma fatia. Devido a isso surgiram algumas diferentes estratégias para a abordagem de fatiamento, que utiliza diferentemente as fatias geradas, as quais são:

- Fatiamento com União (Slicing): Esta abordagem usa a união das fatias geradas pelo algoritmo para gerar a lista de elementos suspeitos.
- Fatiamento com Frequência (Slicing_count): Esta estratégia verifica quantas vezes uma sentença, comando ou o elemento é incluído na fatia de código. Quanto mais frequente for o elemento, mais suspeito ele é.
- Fatiamento com Interseção (Slicing_intersection): Usa a interseção de várias fatias de códigos para calcular os elementos suspeitos de erros. O algoritmo de *slicing intersection* considera somente as sentenças que são comuns nas fatias criadas.

2.1.4 Stack trace

Stack trace é uma abordagem que utiliza o rastreamento de pilha para auxiliar a depuração do sistema, pois fornece informação sobre as rotinas.

Durante a execução dos programas, é criada uma pilha de execução, que basicamente empilha as funções do programa a medida que vão sendo executadas. Esta pilha mostra tudo o que foi executado até então e quando acontece algum erro o programa e pilha são

pausadas. Sendo assim é possível observar em qual ponto o programa parou e apresentou o bug. Muitos IDEs de desenvolvimento mostram esta pilha indicando a classe e a linha onde o programa parou.

Como uma abordagem de localizador de bug, o *Stacktrace* utiliza essa pilha para encontrar os bugs e assim sugerir as instruções que contém os defeitos. O algoritmo segue empilhando cada chamada de método que é executada e assim que o respectivo método termina a respectiva chamada é desempilhada. Quando ocorre uma exceção, o programa e a pilha são pausados. Com isso, observa-se na pilha os métodos chamados e o último método executado antes da falha, o qual se encontra no primeiro *frame* (topo) da pilha, o qual é considerado a provável causa do problema.

A figura abaixo ilustra uma pilha de execução pausada no momento em que o programa falhou. No primeiro *frame* observa-se qual foi a classe, o método e a linha que foram executados por último, onde provavelmente está a localização do bug.

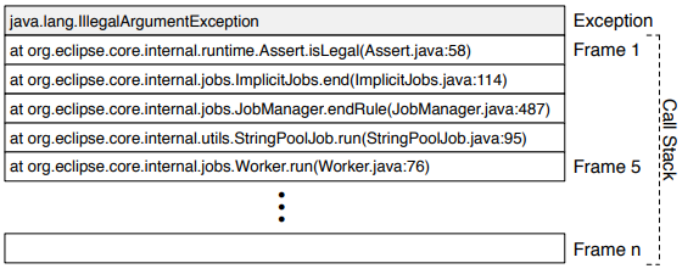

Figura 3 – Exemplo de uma pilha de execução usada pelo Stacktrace Fonte: (SCHROTER et al., 2010)

2.1.5 Predicate Switching

A abordagem de *Predicate Switching* é uma técnica de localização de bugs focada nos predicados e nos controles de fluxos do programa. Define-se um predicado como uma sentença que assume um valor lógico (verdadeiro ou falso). Os controles de fluxos são estruturas condicionais que podem tomar diferentes decisões, dependendo de seus valores de entrada. Sendo assim, um predicado pode controlar a execução de diferentes ramificações no sistema (ZOU et al., 2021).

Na abordagem *Predicate Switching*, o algoritmo percorre a pilha de execução, de um teste que falhou, e identifica todas as ramificações. Em seguida, o teste é executado várias vezes, porém em cada execução, o predicado que gerou a ramificação sofre uma mutação, de modo a gerar um resultado diferente.

Por exemplo, quando a execução de um programa encontra um predicado, que pode ser um comando `‘if else’`, `‘while’`, `‘for’`, a sequencia lógica do programa pode ir

para dois caminhos, dependendo da condição true ou false que o predicado assume. Esses caminhos são as ramificações que podem levar a diferentes tomadas de decisão.

Sendo assim, o algoritmo de *Predicate Switching* executa o caso de teste que falhou várias vezes, aplicando mutações no predicado até que uma delas conduza a uma ramificação que passe no caso de teste. Se esta modificação no predicado produziu a saída correta então este predicado é chamado de predicado crítico (ZOU et al., 2021).

O *Predicate Switching* se difere das abordagens que utilizam mutação (Metallaxis e Muse) no que se refere as mutações no fluxo de controle e não em qualquer parte do código, como ocorre com o Metallaxis e Muse.

2.2 *Benchmark* de Bugs Defects4j

Defects4J é um banco de dados que surgiu com 357 bugs de problemas reais de 5 projetos de código aberto. Esta base de dados pode ser extensível, podendo receber mais bugs com muita facilidade. Atualmente este banco já conta com 835 bugs e 17 projetos.

Esta base de dados vem sendo bastante utilizada em pesquisas de testes de softwares, em geral, por ser uma base com bugs reais e além disso, este benchmark fornece muitas informação sobre os bugs, como sua versão corrigida e os casos de testes para aquele bug. Tudo isso é possível porque este banco de dados também é um framework (JUST; JALALI; ERNST, 2014).

O framework do Defects4J provê muitas funcionalidades que facilitam o trabalho de pesquisas que envolvem bugs. Como exemplo, este framework fornece o acesso as versões com erros e as versões corrigidas de um bug e seus casos de testes.

A Tabela 2.2 abaixo exemplifica melhor quais são os projetos que compõe o Defects4j atualmente e quantos bugs cada um deles possui. Estes dados estão disponibilizados no repositório Github do Defects4j.

Cada bug do Defects4j possui as seguintes propriedades:

- ❏ O bug possui uma versão com o erro e uma versão com a correção do erro.
- ❏ O bug é reproduzível: todos os casos de testes passam na versão corrigida e pelo menos um deles falha na versão com erro, evidenciando o erro.
- ❏ O bug é minimizado: a versão com erro e a versão corrigida se diferem apenas pequenas alterações, não possuindo refatorações ou adição de recursos, por exemplo.
- ❏ O bug é corrigido em um único commit.
- ❏ O bug é corrigido pela modificação do código-fonte (em oposição aos arquivos de configuração, documentação ou arquivos de teste).

Nome do Projeto	Identificador	Número de bugs ativos
jfreechart	Chart	26
commons-cli	Cli	39
closure-compiler	Closure	174
commons-codec	Codec	18
commons-collections	Collections	4
commons-compress	Compress	47
commons-csv	Csv	16
gson	Gson	18
jackson-core	JacksonCore	26
jackson-databind	JacksonDatabind	112
jackson-dataformat-xml	JacksonXml	6
jsoup	Jsoup	93
commons-jxpath	JxPath	22
commons-lang	Lang	64
commons-math	Math	106
mockito	Mockito	38
joda-time	Time	26

Tabela 2 – Projetos Java que compõe o Defects4J

2.3 Ações e Padrões de Reparo

Após o bug ser localizado no código ele é reparado para que o programa em questão funcione corretamente. O reparo que ocorre no bug pode ser feito de várias maneiras. Podem ser adicionadas novas linhas no código ou podem ocorrer a remoção de alguns elementos ou mesmo linhas e podem acontecer também de apenas alguns elementos serem modificados, sem a necessidade de adição ou remoção de linhas.

Como os conjuntos de dados de bugs passaram a ser usados amplamente para avaliar o desempenho de diversas ferramentas relacionadas ao reparo de bugs, tais como as ferramentas de localização e de reparo automático de bugs, Sobreira e colegas (SOBREIRA et al., 2018) identificaram a necessidade de caracterizar conjuntos de dados de tal forma a mostrar as características dos bugs que podem estar influenciando o desempenho das ferramentas. Eles escolheram caracterizar o Defects4J e uma das maneiras utilizadas foi categorização das ações que acontecem na construção de um reparo. Sendo assim, padrões e ações de reparo foram definidas como categorias de blocos de código fonte modificados/inseridos/excluídos para fazer os reparos. Os padrões de reparo são abstrações de mais alto nível feitas a partir das ações as quais tem uma granularidade mais fina.

2.3.1 Ações de Reparo

São os blocos mais simples para fazer o reparo. Contemplam as ações de adições de linhas ou comandos, remoção de linhas ou comando e modificação de comando . Sobreira e colegas (SOBREIRA et al., 2018) identificaram e categorizaram 67 ações utilizando os bugs do Defects4j. Basicamente, as ações são operações de adição, remoção ou alteração de diferentes estruturas sintáticas da linguagem de programação, tais como, comandos de atribuição, comandos condicionais, comandos de repetição, chamadas de métodos, entre outras. A Tabela 3 apresenta as ações identificadas nos bugs do Defects4J.

Numeração	Ação de reparo	Descrição
1	assignAdd	envolvem as instruções de atribuição simples, operadores de incremento unário (x++), de decremento e de operadores aritméticos.
2	assignRem	
3	assignExpChange	
4	condBranIfAdd	São as ações de reparo que envolvem os blocos condicionais, como as instruções 'if... else' e 'case... switch'.
5	condBranIfElseAdd	
6	condBranElseAdd	
7	condBranCaseAdd	
8	condBranRem	
9	condExpExpand	
10	condExpRed	
11	condExpMod	
12	loopAdd	São ações relacionadas aos loops, como 'for', 'while' e 'do..while'. São ações que podem envolver adição ou remoção.
13	loopRem	
14	loopCondChange	
15	loopInitChange	
16	mcAdd	São ações relacionadas a uma adição de chamada de método, ou a remoção de uma chamada de método.
17	mcRem	
18	mcRepl	
19	mcMove	
20	mcParAdd	
21	mcParRem	
22	mcParSwap	
23	mcParValChange	
24	mdAdd	Nestas ações podem ocorrer adição ou remoção de uma declaração de método, ou modificações de elementos do método.
25	mdRem	
26	mdRen	
27	mdParAd	
28	mdParRem	
29	mdParTyChange	
30	mdRetTyChange	
31	mdModChange	
32	mdOverride	
33	objInstAdd	Essas ações de reparo se referem a instanciação de objetos observadas pela palavra-chave 'new'.
34	objInstRem	
35	objInstModAdd	
36	exTryCatchAdd	Essas ações se relacionam as exceções, como os blocos de código 'try-catch' e 'throw'
37	exTryCatchRem	
38	exThrowsAdd	
39	exThrowsRem	
40	retBranchAdd	Essas ações se referem a instrução 'return', que pode ser adicionada, modificada ou removida do código.
41	retRem	
42	retExpChange	
43	varAdd	São ações relacionadas a declaração de variáveis:novas declarações de variáveis, remoção ou mudança de tipo de variáveis.
44	varRem	
45	varTyChange	
46	varModChange	
47	varReplVar	
48	varReplMc	
49	tyAdd	Estas ações referem aos tipos de dados, como os tipos 'int', 'Double'.
50	tyImpInterf	
51	condBlockOthersAdd	São ações que envolvem os blocos condicionais, como 'if... else' e 'case... switch'. Consideram alterações do bloco inteiro.
52	condBlockRetAdd	
53	condBlockExcAdd	
54	condBlockRem	
55	expLogicExpand	Estas ações se referem a expressões lógicas ou aritméticas.
56	expLogicReduce	
57	expLogicMod	
58	expArithMod	
59	wrapsIf	Estas ações são muito semelhantes com as ações condicionais, porém incluem as ações de empacotamento de estruturas 'try-catch', e loops.
60	wrapsIfElse	
61	wrapsElse	
62	wrapsTryCatch	
63	wrapsMethod	
64	wrapsLoop	
65	unwrapIfElse	
66	unwrapMethod	
67	unwrapTryCatch	

Tabela 3 – Ações de reparo do Defects4J

A Figura 4 mostra o ranking das ações de reparo em elementos de código (eixo vertical) em relação ao número de reparos (eixo horizontal) onde ocorrem. As ações de reparo que pertencem ao mesmo grupo (por exemplo, adição de chamada de método e adição de parâmetro de chamada de método pertencem ao grupo *Adição de Chamada de Método*)

Acrônimo	Ação	Grupo
asgn	A/R/M	Atribuição
cnd	A/R/M	Condicional
lp	A/R/M	Loop
mc	A/R/M	Chamada de método
md	A/R/M	Declaração de método
obj	A/R/M	Instanciação de objetos
ex	A/R	Exceção
ret	A/R/M	Retorno
var	A/R/M	Variável
ty	A/M	Tipo

Tabela 4 – Acrônimos para ações de reparo por grupo sintático.

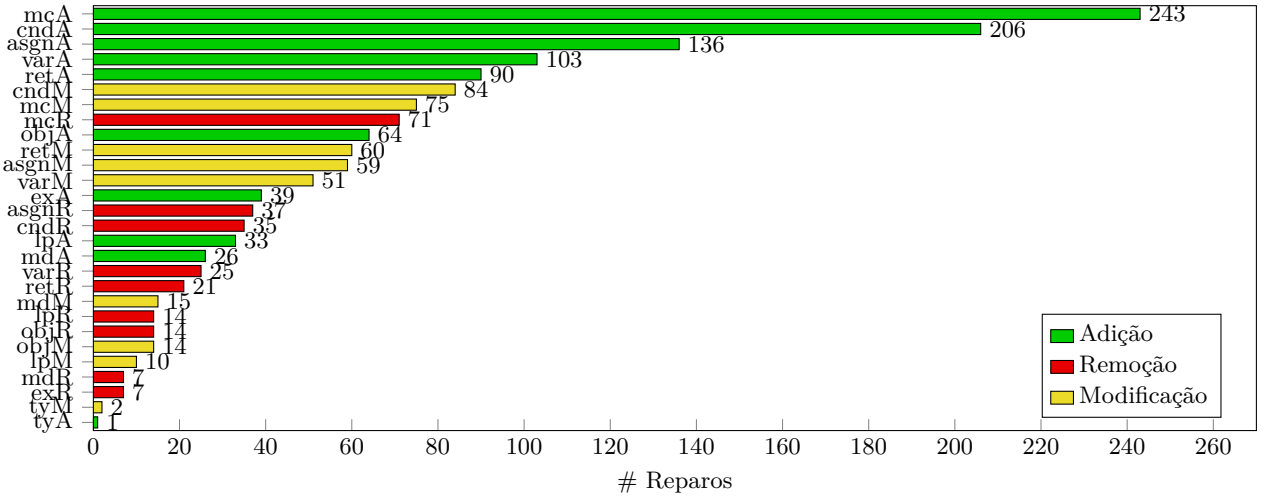

Figura 4 – Frequência das ações de reparos

foram agrupadas para evitar um gráfico muito fragmentado. Os nomes dos grupos de ações de reparo também foram contraídos. A Tabela 4 mostra, para cada grupo (por exemplo, *Chamada de Método*), suas siglas (por exemplo, mc) e letras de sufixo dos tipos de ação existentes para ele (A=Adição, R=Remoção e M=Modificação), que combinadas formam o nome contraído de um grupo de ação de reparo (por exemplo, "mcA" representa *Adição de Chamada de Método*). Para facilitar a compreensão do gráfico, barras verdes representam adição, barras vermelhas representam remoção e barras amarelas representam ações de modificação.

2.3.2 Padrões de Reparo

Os padrões de reparo são abstrações de alto nível para os reparos de bugs que podem envolver mudanças mais elaboradas, mas que seguem um padrão repetitivo. Enquanto as ações de reparo definem modificações mais simples feitas no código, como uma remoção

de linha, os padrões de reparo são definidos como aquelas ações que ocorrem com certa frequência no reparo dos bugs, e que evolvem em geral um contexto maior do código fonte.

Por exemplo, temos operações de *copiar e colar*, que podem incluir diferentes ações de reparo em diferentes estruturas sintáticas. Neste caso, é possível identificar o respectivo padrão *copyPaste* que não é capturável a partir de ações isoladas de reparo tal como foram definidas.

Sobreira e colegas (SOBREIRA et al., 2018) identificaram os seguintes padrões:

1. *Conditional Block*: este padrão está associado a adição ou remoção de blocos condicionais inteiros, incluíndo variantes que incluem um comando *return*, ou com lançamento de exceção;

2. *Expression Fix*: este padrão inclui reparos em expressões lógicas ou numéricas, incluíndo a redução, expansão ou outra modificação;

3. *Wraps-with/Unwraps-from*: este padrão inclui o encapsulamento ou desencapsulamento de um bloco inteiro com comandos *if*, *else*, *try-catch*, *loop* ou chamada de método;

4. *Wrong Reference*: este padrão inclui o reparo em referências a variáveis ou chamadas de métodos;

5. *Missing Null-Check*": Este padrão incluir a inclusão de um comando condicional para verificar se uma variável que vai ser acessada é nula ou não. Este padrão inclui tanto testes para ver se é nula ou se não é nula;

6. *Single Line*: Este padrão denota reparos onde ocorre um ajuste em uma única linha. Perceba que um determinado padrão não exclui outro padrão. Por exemplo, podemos ter um reparo do tipo *Expression Fix* e *Single Line*, porque o ajuste da expressão ocorreu em uma única linha;

7. *CopyPaste*: Este padrão ocorre quando é copiada uma parte do código para outro local;

8. *Constante Change*: Este padrão implica em uma mudança de constante no programa;

9. *Code Move*: Este padrão indica a movimentação de um trecho de código para outra área no programa.

A Figura 5 apresenta o ranking dos padrões de reparo (eixo vertical) em relação ao número de reparos (eixo horizontal) onde ocorrem. *Conditional Block* é o padrão de reparo mais prevalente encontrado nos reparos, seguido por *Expr. Fix* e *Wraps-with*. Por outro lado, *Const. Change* e *Code Moving* são os menos prevalentes entre os padrões de reparo.

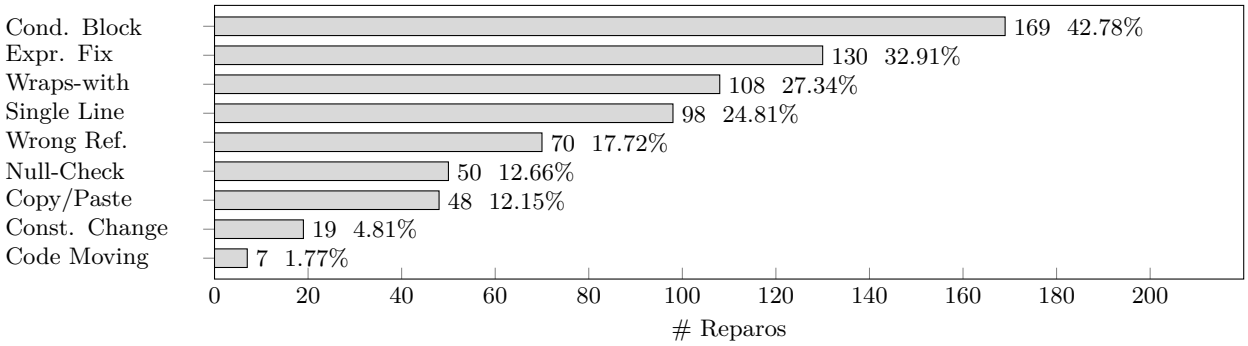

Figura 5 – Frequência dos padrões em reparos

De acordo com o foi apresentado nas seções acima, as 7 abordagens de localização de bugs, o framework Defects4j e as características dos bugs (ações e padrões de reparo) são os elementos que foram estudados para conduzir o estudo deste trabalho. No próximo capítulo será detalhado como estes dados foram utilizados na metodologia do estudo.

Capítulo 3

Descrição do Estudo

Neste capítulo é apresentado o método para desenvolver o estudo, incluíndo como os dados utilizados para o projeto foram organizados para se fazer a análise.

3.1 Proposta do Estudo

A proposta deste trabalho é apresentar um estudo empírico de uma análise qualitativa sobre as relações entre as abordagens de localização de bugs e as propriedades dos bug, mais precisamente as ações e padrões de reparo. Neste estudo é mostrado se existe alguma associação entre os localizadores de bugs e as ações e padrões de reparo que ocorrem nos respectivos bugs, de modo a avaliar a acurácia dos localizadores em termos de quantidade de bugs corrigidos.

Com isso, espera-se observar como os localizadores se comportam e qual sua eficácia em relação aos bugs corrigidos de acordo com as ações e padrões de reparo dos bugs. Desta análise espera-se que surjam pontos que possam servir para sugerir melhorias nas ferramentas de localização de bugs.

Três perguntas foram levantadas para direcionar este estudo na associação entre os localizadores e as ações e padrões de reparo. As questões, que já foram citadas anteriormente, estão listadas abaixo.

1. Pergunta 1: Existe associação entre a acurácia dos localizadores de bugs e os tipos ações de reparo, considerando adição, remoção e modificação?

2. Pergunta 2: Existe associação entre a acurácia dos localizadores e a estrutura sintática das ações de reparo?

3. Pergunta 3: Existe associação entre a acurácia dos localizadores e os padrões de reparo?

3.2 Método para a Avaliação

Os dados utilizados para validar a hipótese de pesquisa incluem os dados extraídos do estudo de Zou e colegas (ZOU et al., 2021), o qual descreve o resultado de localização de diferentes localizadores em relação ao benchmark Defects4j, o qual foi detalhado no capítulo anterior. A escolha dos localizadores de bugs e do benchmark Defects4j apresentados foi definida pelo fato de os localizadores estudados cobrirem uma parte relevante do universo de localizadores, com diferentes abordagens. Além disso, o Defects4J é um benchmark amplamente utilizado nos estudos de localização de bugs.

Os dados coletados para esta pesquisa contém informações sobre cada classe dos bugs que foram consideradas suspeitas e seus scores que foram dados pelos localizadores de bugs. Também foi considerado para este trabalho a disseção do Defects4J, a qual permitiu localizar quais ações e padrões de reparo estavam contidos nos bugs (SOBREIRA et al., 2018).

Os dados, que contém informações sobre as classes e características dos bugs precisaram ser organizados para facilitar a análise e sua manipulação. Para isso foi preciso gerar algumas tabelas para visualizar e processar melhor os dados. Com isso foi possível aplicar medidas de avaliação.

Foi utilizada a métrica *Einspect*, a mesma utilizada por Zhou e colegas para avaliar a eficácia dos localizadores (ZOU et al., 2021). A métrica *Einspect* consiste em analisar um agrupamento de n de posições de elementos suspeitos e verificar se o elemento defeituoso de fato está entre as n posições do ranking. Por exemplo, uma ferramenta de localização acerta a localização de um bug, utilizando a métrica Einspect com n = 3 (Einspect@3), se a localização certa estiver entre as 3 primeiras posições do ranking que o localizador retorna. Esta métrica foi escolhida neste trabalho por ser uma métrica que já foi utilizada em outros estudos e por ser uma medida mais significativa da qualidade de localização de falhas do que outras métricas, como o EXAM. Foram usados os valores n = 1, 3, 5 e 10, pois esta configuração também foi feita por Zhou e colegas (ZOU et al., 2021).

A Tabela 5 representa quais as abordagens de localizadores que conseguiram encontrar o erro no respectivo bug com n=1, onde o número 1 na tabela indica que o localizador daquela coluna encontrou a localização do bug e 0 quando o localizador não encontrou o bug.

project	bug	faulty	Oichiai	Dstar	Metallaxis	Muse	Slicing	Slicing_count	Slicing_intersection	Stacktrace	Predicateswitching
Chart	1	1	0	0	0	0	1	1	1	0	0
Chart	2	1	0	0	0	0	1	0	0	0	0
Chart	3	1	1	1	0	0	1	1	1	0	0
.											
.											
.											
Time	25	1	0	0	0	0	0	0	0	0	0
Time	27	1	0	0	0	0	1	1	1	0	0

Tabela 5 – Tabela Bugs vs Localizadores

A Tabela 6 mostra quais ações e padrões de reparo existem nos bugs. O número 1

indica que o bug possui aquela ação ou padrão de reparo. Um bug pode ter mais de uma ação e padrão. O número 0 indica que o bug não possui aquela ação.

project	bugId	assignAdd	assignRem	assignExpChange	. . .	codeMove
Chart	1	0	0	0	. . .	0
Chart	2	1	0	0	. . .	0
Chart	3	1	0	0	. . .	0
.						
.						
.						
Time	27	0	0	0	. . .	0

Tabela 6 – Tabela Bugs vs Ações e padrões de reparo

A Tabela 7 indica a somatória de bugs que possui aquela ação (indicada pelas colunas) que o localizador (indicado nas linhas) conseguiu encontrar nos 357 bugs. Por exemplo, dos 357 bugs, o localizador *Ochiai* conseguiu encontrar o erro em 30 bugs que precisa da ação *assignAdd* para ser reparado, e assim por diante.

Localizadores	assignAdd	assignRem	assignExpChange	condBranIfAdd	. . .	notClassified
Oichiai	30	10	18	31	. . .	4
Dstar	31	11	20	30	. . .	4
Metallaxis	35	11	28	32	. . .	6
Muse	10	5	5	9	. . .	1
Slicing	67	16	34	63	. . .	5
Slicing_count	55	14	32	52	. . .	5
Slicing_intersection	52	12	29	46	. . .	5
Stacktrace	5	0	3	7	. . .	0
Predicateswitching	7	6	0	7	. . .	0

Tabela 7 – Tabela Localizadores de Bugs vs Ações e padrões de reparo

3.2.1 Associação entre Localizadores e Tipo de Reparo

Seja a pergunta de pesquisa 1: Existe associação entre a acurácia dos localizadores de bugs e os tipos de ações de reparo, considerando adição, remoção e modificação?

A questão a ser discutida com esta pergunta é se os localizadores conseguem encontrar mais erros pelo tipo de ação de reparo. Por exemplo, se um bug possui uma determinada ação 'A' e um localizador 'L´ consegue encontrar melhor esse tipo de bug proporcionalmente aos demais, então deve existir uma associação positiva entre a ação 'A' e o localizador 'L´. Para esta pergunta os tipos de ações de reparo foram definidos como 3 tipos, ações do tipo Adição de linhas de código, Remoção de linhas e Modificação de linhas.

A metodologia usada para se chegar a uma conclusão desta questão foi a partir da Tabela 7, onde consta os dados dos localizadores de bugs e quantas e ações especificamente eles conseguiram detectar nos bugs que encontraram. As colunas das ações agrupadas em 3, cada uma referente aos tipos de Adição, Remoção e Modificação. Foram avaliados *Einspect* para n = 1, 3, 5 e 10. As células das colunas foram somadas e depois foram transformadas em valores percentuais. A última etapa da metodologia foi gerar um gráfico de linhas para dispor estes valores e assim verificar mais facilmente quais os localizadores que obtiveram as melhores taxas de acerto em relação ao 3 grupos de ações.

3.2.2 Associação entre Localizadores e Estruturas Sintáticas em Reparo

Seja a pergunta de pesquisa 2: Existe associação entre a acurácia dos localizadores e a estrutura sintática das ações de reparo?

A lógica da pergunta 2 é semelhante a pergunta 1, diferindo-se no tipo de ações. Enquanto na questão 1 as ações foram dividas em 3 grupos, para a segunda questão as ações foram dividas de acordo com suas estruturas sintáticas.

A estrutura sintática de um código se refere ao arranjo dos comandos numa linha que eles desempenhem uma função. Comandos 'if', 'for', 'while', ou uma simples atribuição de valor a uma variável são considerados estruturas sintáticas. Para discutir a questão de pesquisa 2, as ações foram dividas em 10 grupos de acordo com suas estruturas sintáticas, como foi citado no artigo de (SOBREIRA et al., 2018).

A metodologia utilizada também referenciou a Tabela 7. As colunas referentes às ações foram organizadas em 10 grupos (*Assignment, Conditional, Expression, Loop, Meth. Call, Meth. Declaration, Obj. Instantiantion, Return, Type,* e *Variable*). As células das colunas também foram somadas e transformadas em valores percentuais. Foram avaliados *Einspect* para n = 1, 3, 5 e 10.

Tal como ocorreu na questão 1, também houve a geração de um gráfico de linhas dos localizadores em relação aos 10 grupos de ações de reparo.

3.2.3 Associação entre Localizadores e Padrões de Reparo

Seja a pergunta 3: Existe associação entre a acurácia dos localizadores e os padrões de reparo?

Esta questão tem o objetivo de discutir se existe uma associação ou não dos localizadores de bugs com os padrões de reparo que os bugs apresentam. Os padrões são abstrações que são encontradas nas ações com comportamentos que se repetem.

A Tabela 7 também foi usada para responder a esta pergunta. O método foi semelhante ao das outras perguntas, porém neste experimento foram consideradas apenas as colunas referentes aos padrões e as demais que se referiam as ações foram desconsideradas. Os padrões foram desdobrados em 16 categorias, sejam elas: *Block Remove, Code Move, Cond. Block, Const. Change, Copy-Paste, Arith.Expr., Logic Expr. API fix, Init Fix, Miss. Comput., Miss Null Check, Single Line, Wrap, Unwrap, Wrong Reference*, além de terem havido bugs *Not-Classified*. Com esta configuração foi gerado um gráfico de linhas, onde se pôde observar qual a porcentagem de acerto de cada localizador em relação aos 16 tipos de padrões acima.

Foram apresentados nesta seção a metodologia, bem como a métrica utilizada para classificar a acurácia das abordagens de localização. Também foram apresentados os métodos utilizados para responder as 3 perguntas de pesquisa. No capitulo a seguir será apresentado as repostas destas perguntas.

Capítulo 4

Resultados

Neste capítulo serão apresentados os resultados obtidos para responder as perguntas de pesquisa, as quais foram organizadas de acordo com a caracterização dos reparos dos bugs definida por Sobreira e colegas (SOBREIRA et al., 2018). Antes será apresentada uma breve caracterização dos dados relacionados a características e desempenho dos diferentes localizadores de uma maneira geral, sem levar em consideração as especificidades dos bugs em relação a ações e padrões de reparo. Depois os resultados para responder as três perguntas serão mostrados e finalmente os mesmos serão discutidos.

4.1 Caracterização geral do desempenho dos localizadores

A seguir, será apresentada a caracterização da distribuição dos *scores* para cada linha de código que é indicada como suspeita. Quanto mais próximo de 1 é o *score*, maior é a suspeição da linha de acordo com a abordagem, onde *um* é o score (suspeição) máximo e *zero* o score mínimo. Vamos inicialmente analisar quantidades de suspeitos por bug com score *um*, e em seguida quantidades de bugs com score máximo igual *zero*, significando que não houve suspeito para aquele bug.

Na Figura 6 é apresentada a distribuição de bugs com número de linhas com scores igual a *um* por bug. Como existem *outliers*, no gráfico a esquerda limitamos a bugs com número máximo de 500 linhas com score *um* e no gráfico a direita são representados todos os bugs. Quando um bug em uma abordagem tem um elevado número de localizações com score *um*, significa que a respectiva abordagem está suspeitando de forma máxima de um elevado número de localizações, a grande parte das quais são falso positivos. Nestes casos, a ordenação entre as localizações com score *um* é aleatória, pois fica sendo considerado um empate entre as respectivas localizações.

Podemos observar que as abordagens de fatiamento são as que mais contém bugs com elevados número de suspeitos com suspeição máxima. As abordagens Ochiai, Dstar e Me-

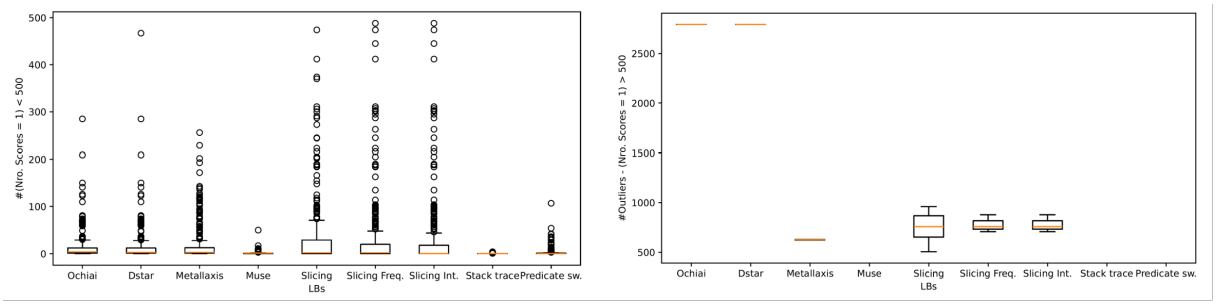

Figura 6 – Distribuição das localizações com scores 1. Esquerda: Limitada a 500 localizações. Direita: Acima de 500 localizações

tallaxis tem distribuição similar e intermediária de suspeitos iguais a *um*, e as abordagens Muse, *Stack trace* e *Predicate Switching* tem uma distribuição menor de bugs com score *um*. Podemos entender que as abordagens Muse, *Stack trace* e *Predicate switching* são construídas de forma a estabelecerem um alvo mais claro para as classes suspeitas, enquanto a abordagens de fatiamento geram um número grande de fatias suspeitas, mesmo aquela baseada em interseção das fatias. De qualquer forma, podemos observar na Tabela 8, que as abordagens para a grande parte dos bugs (3º quartil) não indicam mais que 20 suspeitos máximos, exceto *Fatiamento* que indica 29 suspeitos no 3º quartil. Isso é importante porque avaliaremos a qualidade dos localizadores com a métrica Einspect que avalia os localizadores analisando as n primeiras localizações do ranking, onde n=1, 3, 5, 10. Quando há empates nas primeiras posições do ranking, as localizações são inseridas aleatoriamente, e podemos ver que os empates além do n da métrica Einspect vão representar uma porção menor dos dados.

LB	Média	Mediana	Mínimo	Máximo	1º Quartil	3º Quartil
Ochiai	14,75	4,0	0	285	1,0	12,25
Dstar	15,84	3,0	0	467	1,0	12,0
Metallaxis	17,72	3,0	0	256	1,0	12,5
Muse	1,44	1,0	0	50	0,0	1,0
Slicing	32,39	2,0	0	474	0,0	29,0
Slicing Freq.	29,60	2,0	0	488	0,0	19,75
Slicing Int.	28,19	0,0	0	488	0,0	17,75
Stack trace	0,29	0,0	0	4	0,0	0,0
Predicate sw .	2,75	0,0	0	107	0,0	1,0

Tabela 8 – Estatísticas das localizações por bug com scores 1. Esquerda: Limitada a 500 localizações.

Na Figura 7 é apresentada a frequência de bugs em cada localizador para os quais o score máximo é igual a *zero*. Quando o score máximo das localizações de um bug é zero, significa que o respectivo localizador não conseguiu identificar nenhum suspeito, o que

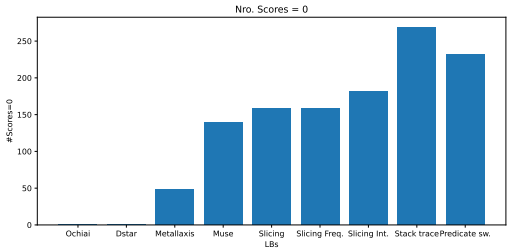

Figura 7 – Distribuição dos frequência de bugs com score máximo zero.

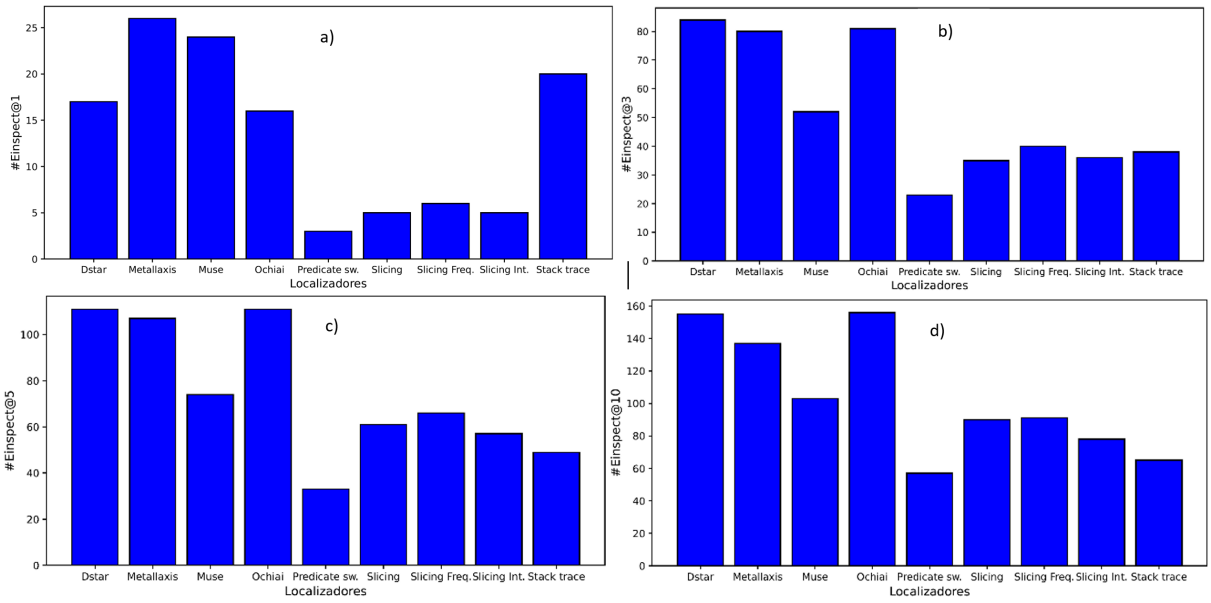

Figura 8 – Métrica Einspect@n para os localizadores. a) n=1 b) n=3 c) n=5 d) n=10

definitivamente já impõe um limite à sua capacidade de localização.

Observamos que Ochiai e Dstar sempre indicam alguma localização com score diferente de *zero*, indicando que elas conseguem indicar pelo menos um suspeito. Por outro lado, as abordagens *Stack trace* e *Predicate Switching*, que eram abordagens que apresentavam poucos bugs com elevado número de localizações com suspeição máxima, também são os localizadores que tem um elevado número de bugs para os quais as respectivas abordagens não conseguem determinar nenhum suspeito. As abordagens Muse e de fatiamento também tem um elevado número de bugs para os quais não conseguem indicar suspeito.

Na Figura 8 são apresentados os valores da métrica Einspect para n = 1, 3, 5 e 10, para os diversos localizadores. Note que a métrica Einspect denota o número de bugs que são corretamente localizados considerando os n primeiros elementos do *ranking* de localizações retornado pela respectiva abordagem.

De acordo com os gráficos, pode-se observar que:

1. Quando n = 1, os localizadores Dstar, Metallaxis, Muse, Ochiai e Stacktrace se

destacam como aqueles que conseguem localizar maior quantidade de bugs, em relação ao Predicate Switching e aos de fatiamento, os quais tem pior desempenho.

2. Quando n = 1, dos 357 bugs, os localizadores Metallaxis, Muse e Stacktrace conseguem localizar mais de 20 deles, o que corresponde a menos que 10% do total de bugs.

3. Para n = 3, 5 e 10, os localizadores Ochiai e Dstar se destacam por ter o melhor ganho de desempenho se tornando aqueles que melhor localizam os bugs considerando o respectivo *n*.

4. *Predicate Switching* invariavelmente tem o pior desempenho nos 4 cenários.

Nas seções seguintes apresentaremos os resultados para cada uma das três perguntas de pesquisa elaboradas neste trabalho.

4.2 Associação entre Localizadores e Tipos de Ações de Reparo

A análise da primeira pergunta de pesquisa está relacionada a entender se existe alguma associação entre algum dos tipos de reparo, sejam eles, *Add* cujo reparo envolve adição de linha ou comando, *Rem* cujo reparo envolve remoção de linha ou comando e *Change* envolvendo modificação de linha ou comando. Como em um reparo podem coexistir diferentes tipos de ações, vamos dividir a análise em 3 partes. A primeira análise envolve três classes, sejam elas *Add*, *Rem*, *Change*, onde para um bug pertencer a uma classes, basta que tenha pelo menos um tipo de ação da respectiva classe. A segunda análise envolve três classes contendo pares de tipos de ações, sejam elas: *Add-Rem*, *Add-Change* e *Rem-Change*, onde em cada classe contém bugs que contém pelo menos os dois tipos de ação. Finalmente, terceira análise envolve bugs que contém todos os tipos de ação, sejam elas *Add-Rem-Change*.

Na primeira parte da análise, inicialmente as ações de reparo foram separadas em 3 grupos. Os agrupamentos foram os seguintes: o grupo *Rem*, que incluem ao menos uma remoção de comandos ou linhas; o grupo *Change*, que incluem ao menos uma ação que faça modificação em comandos ou uma modificação de um elemento numa linha de comando; e o grupo *Add*, que incluem ao menos uma ação que adicione linhas no código. Cada bug pode conter um ou mais tipo de ação.

A Figura 9 mostra os 3 grupos de ações e a métrica Einspect dos localizadores com bug com pelo menos uma daquelas ações, enquanto a Figura 10 mostra os 3 grupos de ações e a métrica Einspect percentualmente em relação ao total de bugs classificados no respectivo grupo.

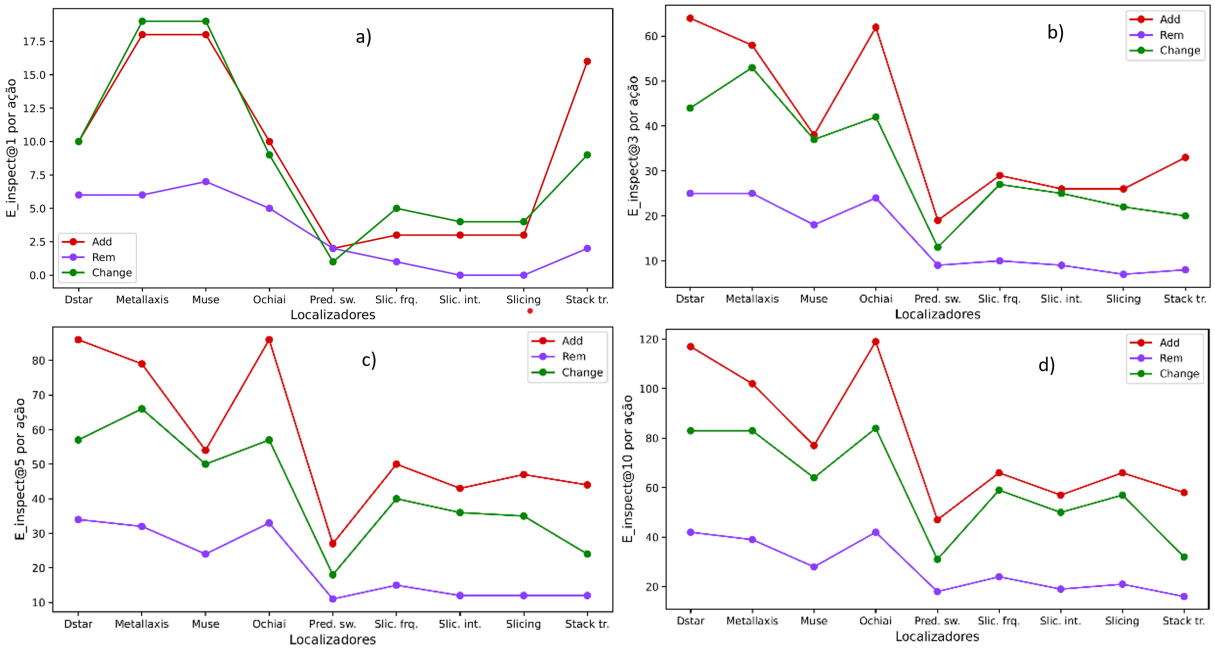

Figura 9 – Métrica Einspect@n por ação para os localizadores. a) n=1 b) n=3 c) n=5 d) n=10

Uma diferença que pode ser observada entre as Figuras 9 e 10 é que em termos absolutos todos os localizadores localizam menos bugs com remoção de linhas, e localizam mais bugs com adição de linhas. Quando analisa-se o Einspect proporcionalmente ao número de tipos de bugs, observamos pouca diferença entre os diferentes tipos de mudança.

Algumas observações podem ser feitas em relação ao Einspect percentual:

1. Para n=1, os localizadores Metallaxis e Muse apresentam melhor acurácia, em relação aos demais para as ações *Add*.

2. Para n=3, observa-se que Dstar, Metallaxis, Muse e Predicate switching conseguem capturar melhor bugs com remoção de linhas. As técnicas de fatiamento capturam melhor bugs com ações de mudança e Stack trace capturam melhor bugs com adição de linhas. Apesar das diferenças serem pequenas intra-localizadores, percebe-se uma certa complementariedade entre os mesmos.

A Figura 11 mostra 3 grupos de ações em pares, e a métrica Einspect dos localizadores com bug com pelos menos uma ação de cada elemento do par. Em outras palavras, um bug do grupo [‘Add’, ‘Rem’] tem pelo menos uma adição e pelo menos uma ação de remoção, podendo ou não ter ação de mudança. O mesmo se aplica para os outros dois pares [‘Add’, ‘Change’] e [‘Rem’, ‘Change’].

A Figura 12 mostra os 3 grupos de ações em pares, e a métrica Einspect dos localizadores com bug com pelos menos uma ação de cada elemento do par, diferenciando-se da

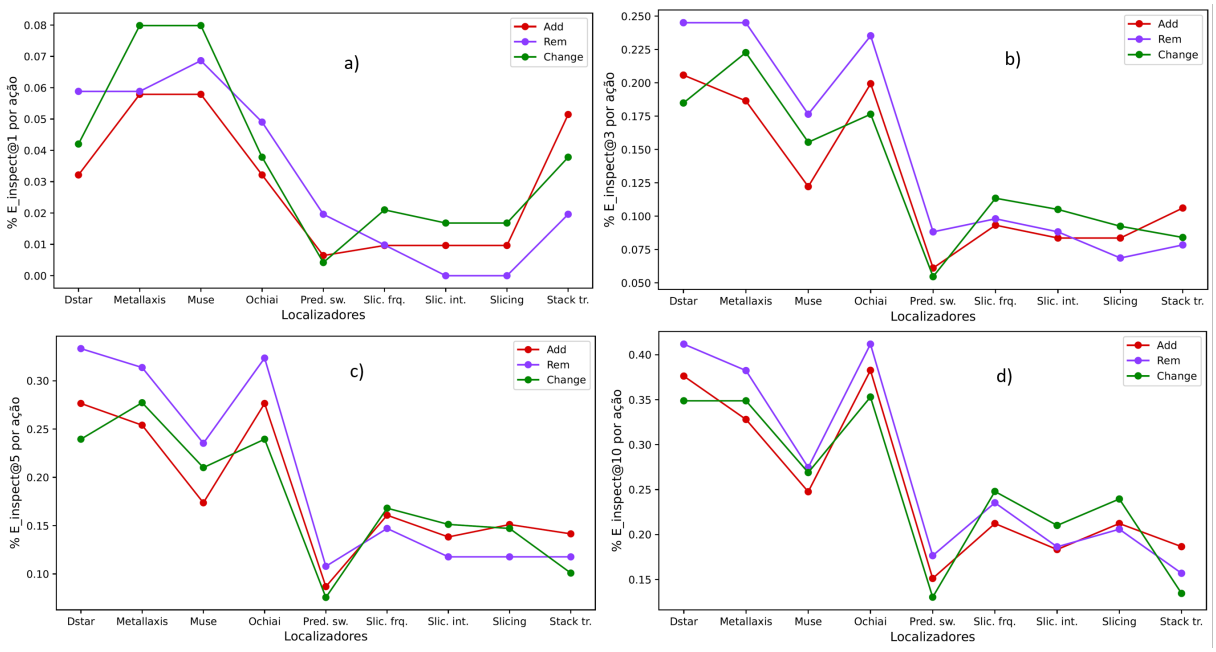

Figura 10 – Métrica Einspect@n percentual por ação para os localizadores. a) n=1 b) n=3 c) n=5 d) n=10

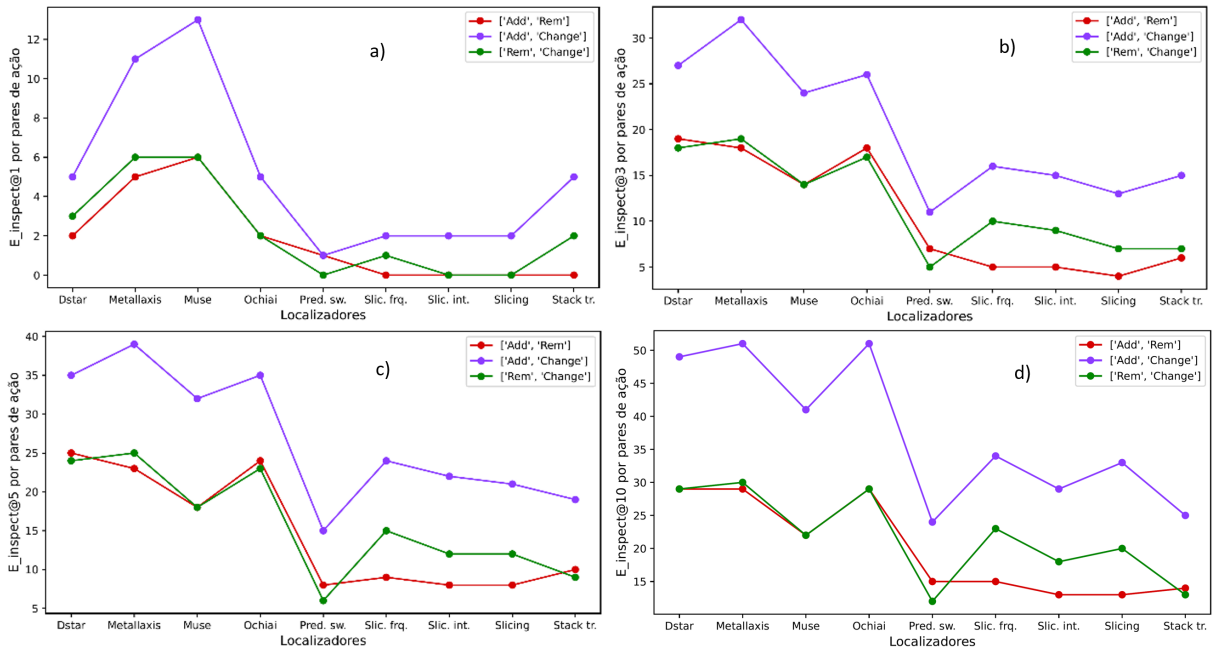

Figura 11 – Métrica Einspect@n por pares de ações para os localizadores. a) n=1 b) n=3 c) n=5 d) n=10

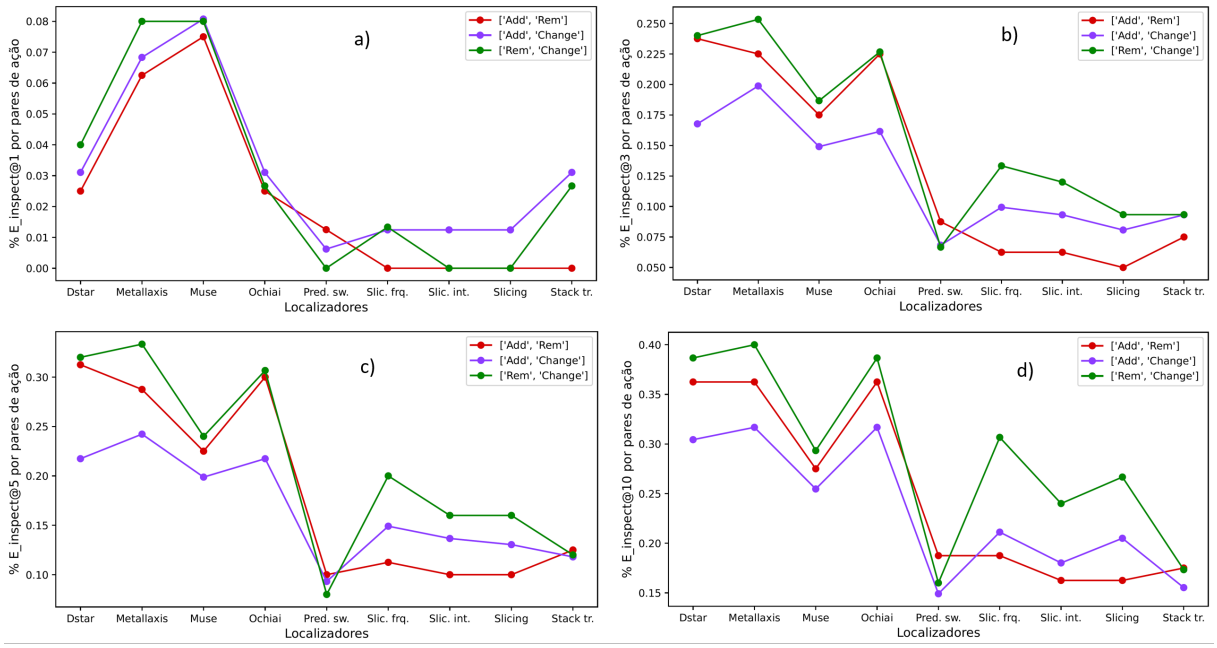

Figura 12 – Métrica Einspect@n percentual por pares de ações para os localizadores. a) n=1 b) n=3 c) n=5 d) n=10

figura anterior por apresentar o valor percentual do Einspect em relação ao total de bugs no respectivo grupo.

Em termos absolutos, observa-se uma localização melhor de bugs com *Add* e *Change*. Entretanto, este padrão não se repete em termos percentuais, sugerindo um número absoluto maior de bugs com adição e mudança. Além disso, em termos absolutos o Einspect cai generalizadamente para cerca da metade, comparado com bugs que apresentam pelo menos um tipo de ação como mostrado anteriormente, ocasionado por uma filtragem mais restritiva dos tipos de reparo. Sobre o ponto de vista relativo, as diferenças entre os diferentes tipos de ação são menos perceptíveis, entretanto podemos citar algumas observações:

1. As técnicas em geral se comportam melhor com bugs *Rem-Change*.

2. As técnicas de fatiamento especialmente para n $>$ 1, se comportam melhor com bugs do tipo *Rem-Change*, e pior com bugs do tipo *Add-Rem*

3. As demais técnicas se comportam melhor também com bugs do tipo *Rem-Change*, mas o pior desempenho fica com *Add-Change*.

Finalmente, na Figura 13 apresenta-se a métrica Einspect no grupo de bugs os quais contém pelo menos um tipo de cada ação *Add*, *Rem*, e *Change*, ou seja, o bug teve todos estes tipos de ação de reparo.

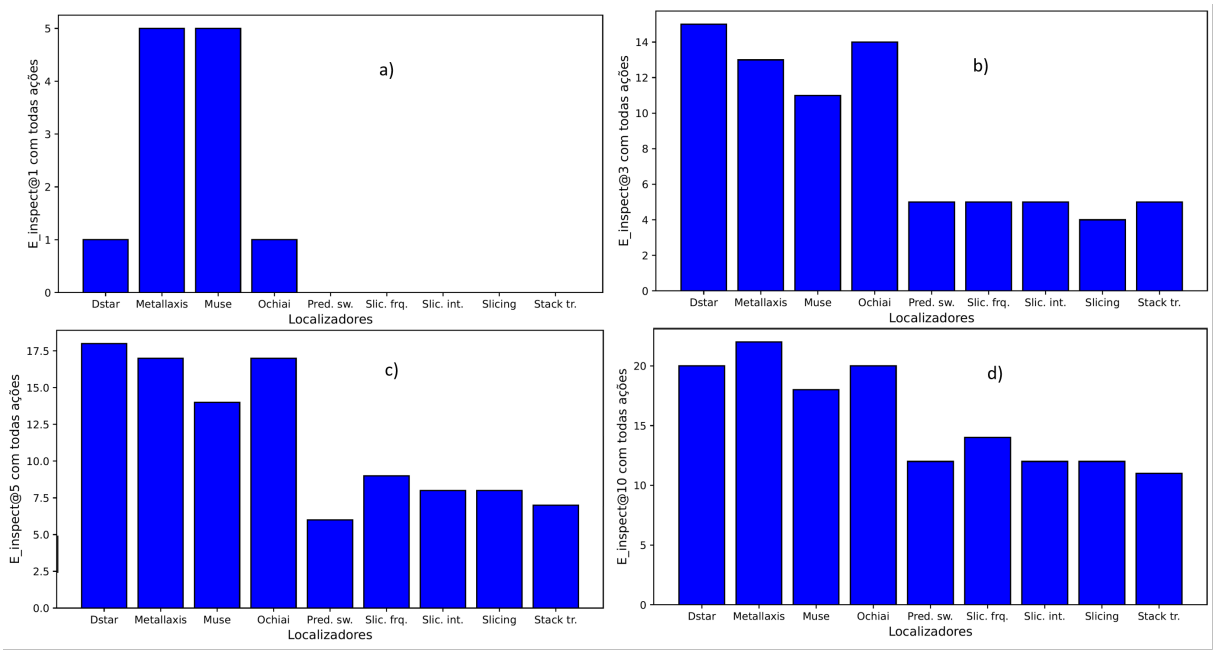

Figura 13 – Métrica Einspect@n para bugs com todas ações para os localizadores. a) n=1 b) n=3 c) n=5 d) n=10

A Figura 14 mostra a métrica Einspect para os bugs que contém todas as ações, diferenciando do gráfico anterior por apresentar o valor percentual do Einspect em relação ao total de bugs que contém todas os tipos *Add*, *Rem* e *Change* de ação de reparo.

Pode-se observar que:

1. Para n=1, somente as técnicas Dstar, Metallaxis, Muse e Ochiai conseguiram encontrar os bugs corretamente.

2. O restante dos localizadores conseguem encontrar o bug quando n>=3.

3. Quando n>=3, as abordagens Dstar, Ochiai, Metallaxis são as abordagens que mais listaram os elementos suspeitos nas primeiras posições.

4. Não houve uma diferença significativa entre os padrões de performance comparado com as outras classes de erro envolvendo outras possibilidades relacionadas a inserção, remoção e mudança.

5. Nem sempre a ferramenta de localização vai atribuir um score = 1 para a linha que contém o erro de fato.

6. Podemos observar que Muse e Predicate Switching se beneficiam mais desta classe de bugs, se compararmos as Figuras 8 e 13

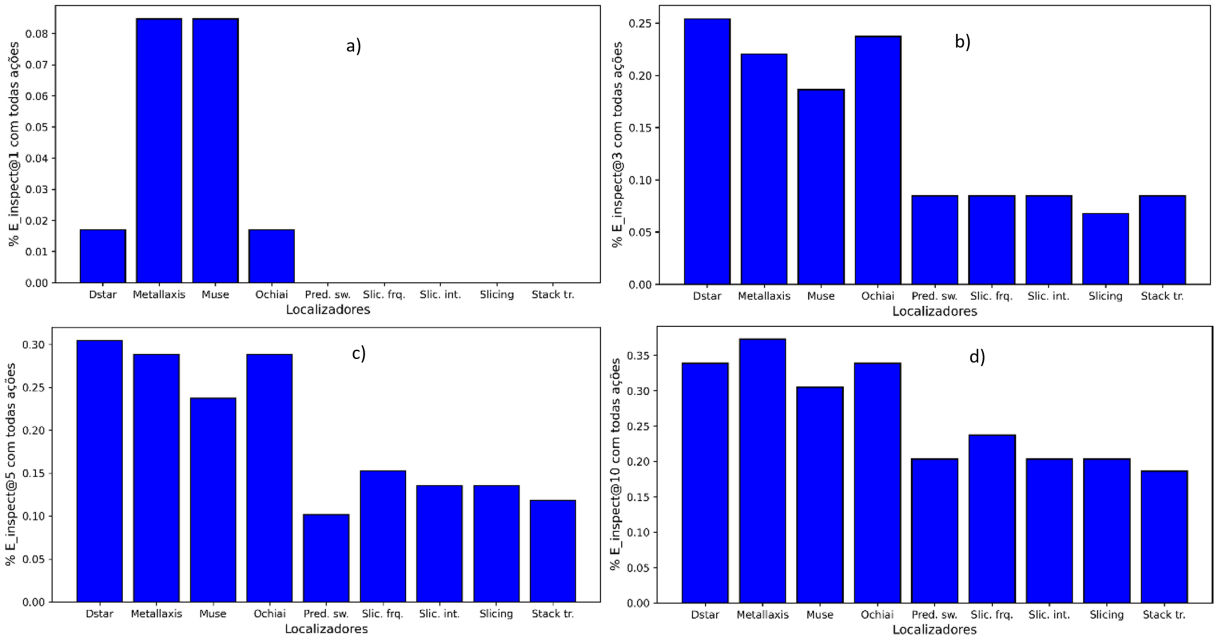

Figura 14 – Métrica Einspect@n percentual para bugs com todas ações para os localizadores. a) n=1 b) n=3 c) n=5 d) n=10

Resposta para a Pergunta 1: *Existe uma associação entre o desempenho dos localizadores de bugs e as ações de reparo de acordo com o tipo de ação?*
Pode-se observar percentualmente em relação aos tipos de modificação que não existe uma diferença considerável entre o tipo de modificação e a influência da mesma em alguns localizadores.
Entretanto, pode-se observar que quando os bugs são corrigidos com Remoção e Mudança os localizadores do tipo Fatiamento passam a se comportam melhor do que com outros tipos de correções.
Além disso, Muse comparativamente aos demais localizadores tem uma melhoria de performance com bugs que contém todos os tipos *Add*, *Rem*, e *Change*.

4.3 Associação entre Localizadores e Estrutura Sintática das Ações de Reparo

O segundo estudo dividiu as ações de reparo em 10 grupos, seguindo o artigo de (SOBREIRA et al., 2018). Neste agrupamento não houve distinção entre ações que adicionam, removem ou modificam linhas de código. Os grupos formados foram os seguintes:

1. Assigment: incluem as ações que envolvem atribuições.
2. Conditional: incluem as ações relacionadas aos blocos condicionais, como os 'if-else', 'case-switch'.

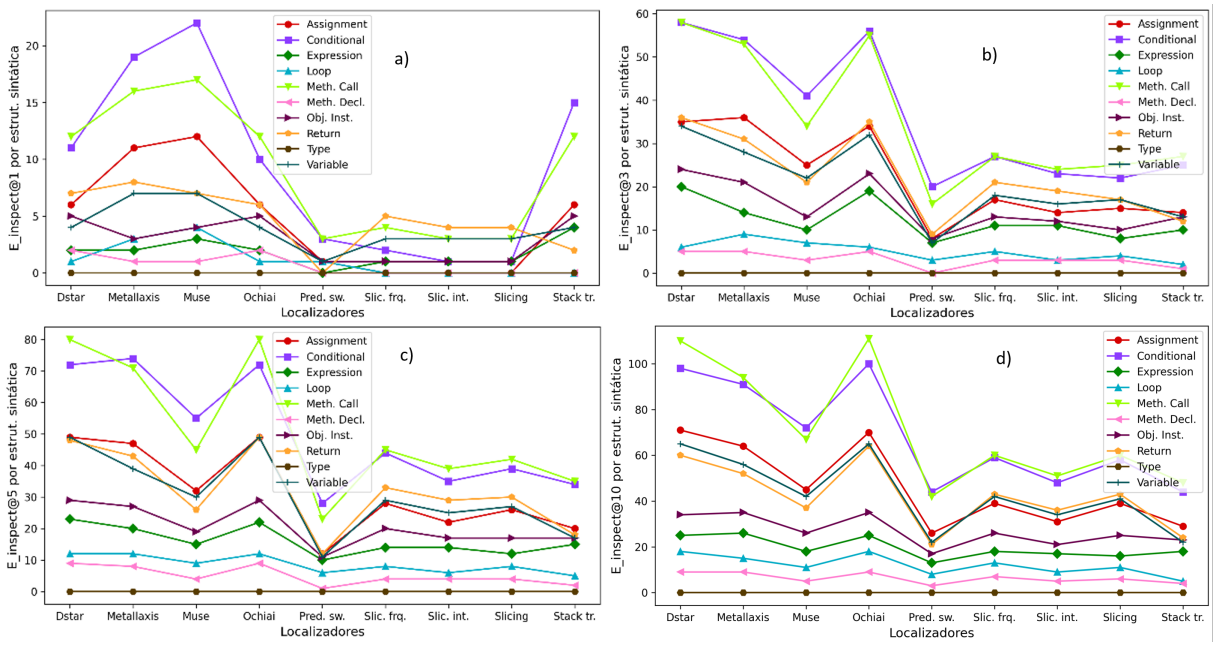

Figura 15 – Métrica Einspect@n dos localizadores por ações divididas em 10 grupos sintáticos. a) n=1 b) n=3 c) n=5 d) n=10

3. Expression: incluem ações em expressões.
4. Loop: incluem as ações dos blocos de código do tipo ‘for’, ‘while’, ‘do-while’.
5. Method Call: são as ações que incluem chamadas de métodos.
6. Method Declaration: incluem as ações que declaram um método.
7. Object Instantiation: incluem as ações que envolve a instanciação de um objeto.
8. Return: são as ações de reparo que envolvem as linhas de código com o comando ‘return’.
9. Variable: são as ações de reparo que envolvem as declarações de variáveis.
10. Type: são as ações que envolvem os Tipos de dados, que são o int, float, double, bolean, String.

A Figura 15 apresenta a métrica Einspect do localizadores em relação a quantidade de bugs encontrados pelas ações divididas nestes 10 grupos de ações que precisam ser feitas para o bug ser corrigido.

A Figura 16 apresenta a quantidade percentual de bugs da métrica Einspect do localizadores em relação a quantidade de total de bugs envolvendo cada um dos 10 tipos sintáticos que são reparados.

A partir destes gráficos pode-se fazer algumas observações:

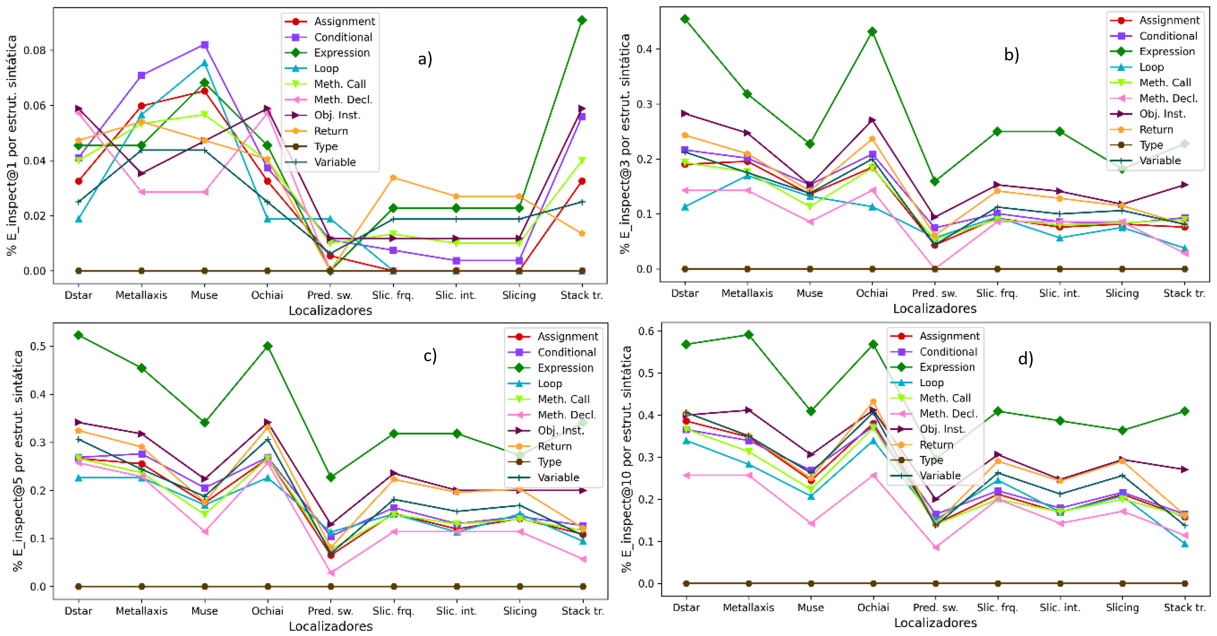

Figura 16 – Métrica Einspect@n Percentual dos localizadores por ações divididas em 10 grupos sintáticos. a) n=1 b) n=3 c) n=5 d) n=10

1. Um aspecto interessante, é que não observamos localizadores muito mais especializados em um tipo de estrutura em comparação a outros localizadores, o que pode ser percebido por não haver muitos cruzamentos das linhas referentes a cada estrutura à medida que mudam os localizadores. Ou seja, não houve interação de dados entre estruturas e localizadores, ou seja, uma estrutura que era mais bem localizada em um localizador, por exemplo, expressões, era a mais bem localizada nos outros localizadores. Um das poucas observações neste sentido é que Stack Tracing se comporta um pouco pior em bugs em estruturas *Return*, e também Condicionais para Einspect@1.

2. Sob o ponto de vista absoluto, ao observamos Einspect@10, vemos que existe uma ordenação entre as estruturas sintáticas sendo *Declaração de Métodos* e *Tipos* as estruturas com menos localizações corretas, e *Chamada de Método* e *Condicionais* são onde os localizadores tem maior número de bugs localizados.

3. Sob o ponto de vista percentual, o grupo *Expression*, que são as ações que envolvem o correções em expressões, foi o mais assertivo pelo localizadores, que foram mais eficientes em encontrar o erro. Isto sugere, que apesar de haver muitas localizações corretas em *Chamadas de Método* e *Condicionais*, também há um número maior de bugs com correções nestas estrutura. Sob o ponto de vista percentual, bugs em *Tipos* e *Declaração de métodos* tem maior grau de dificuldade de localização.

4. Percentualmente não existe uma diferença clara entre as estrutura que não seja

Tipos, *Declaração de Métodos* e *Expressões*.

Resposta para a Pergunta 2: *Existe uma associação entre o desempenho dos localizadores de bugs e a estrutura sintática das ações de reparo?*
Ao contrário do tipo de mudança (Add, Rem, Change), pode-se observar alguns padrões importantes em relação às estruturas sintáticas. Bugs em Expressões são mais fáceis de localizar, e bugs em Tipos e Declaração de Métodos são mais difíceis de localizar. Nenhum localizador foi capaz de localizar bugs em Tipos.

4.4 Associação entre Localizadores e Padrões de Reparo

No terceiro estudo foi analisada a performance dos localizadores de bugs, em termos de quantidade de bugs encontrados, em relação aos padrões definidos por (SOBREIRA et al., 2018).

Os padrões de reparo são os seguintes:

1. Block Remove: é um padrão onde o reparo é feito por meio da remoção de um bloco de código.
2. Code Move: é um padrão que inclue ações que fazem a movimentação de linhas de código, ou seja, pegam um trecho de código e colocam em outro lugar no código.
3. Conditional Block: é um padrão que envolve adição ou remoção de blocos condicionais.
4. Constant Change: são incluídas neste padrão as ações que fazem mudanças de constantes no código.
5. Copy-Paste: são incluídos neste padrão as ações que copiam e colam trechos em diferentes pontos no código- fonte.
6. Arith. Expression: padrão que incluem ações para correção de expressões aritméticas.
7. Logic Expression: padrão que incluem ações para correção de expressões lógicas.
8. API fix: padrão que inclui ações para correção em uso de APIs.
9. Init fix: padrão que inclui ações para correção em inicialização de variáveis.
10. Missing Computation: padrão para inclusão de computação faltante em algoritmos.

11. MissNullCheck: padrão que incluem reparos para verificação de referências nulas (null).

12. NotClassified: incluem os reparos cujas ações não se encaixam em nenhum padrão.

13. SingleLine: são padrões que incluem os reparos com adição de uma linha, ou remoção ou modificação de uma linha.

14. WrongRef: incluem os padrões onde as variáveis ou métodos foram referenciadas de forma errada.

15. Wrap: padrão que incluem reparos que onde um trecho de código é envolvido em alguma estrutura de controle (if, loop, ...).

16. Unwrap: padrão que incluem reparos que onde um trecho de código é retirado de alguma estrutura de controle que o envolve.

A Figura 17 apresenta a métrica Einspect do localizadores em relação a quantidade de bugs encontrados pelas ações divididas nos 16 grupos de padrões de reparo que precisam ser feitas para o bug ser corrigido.

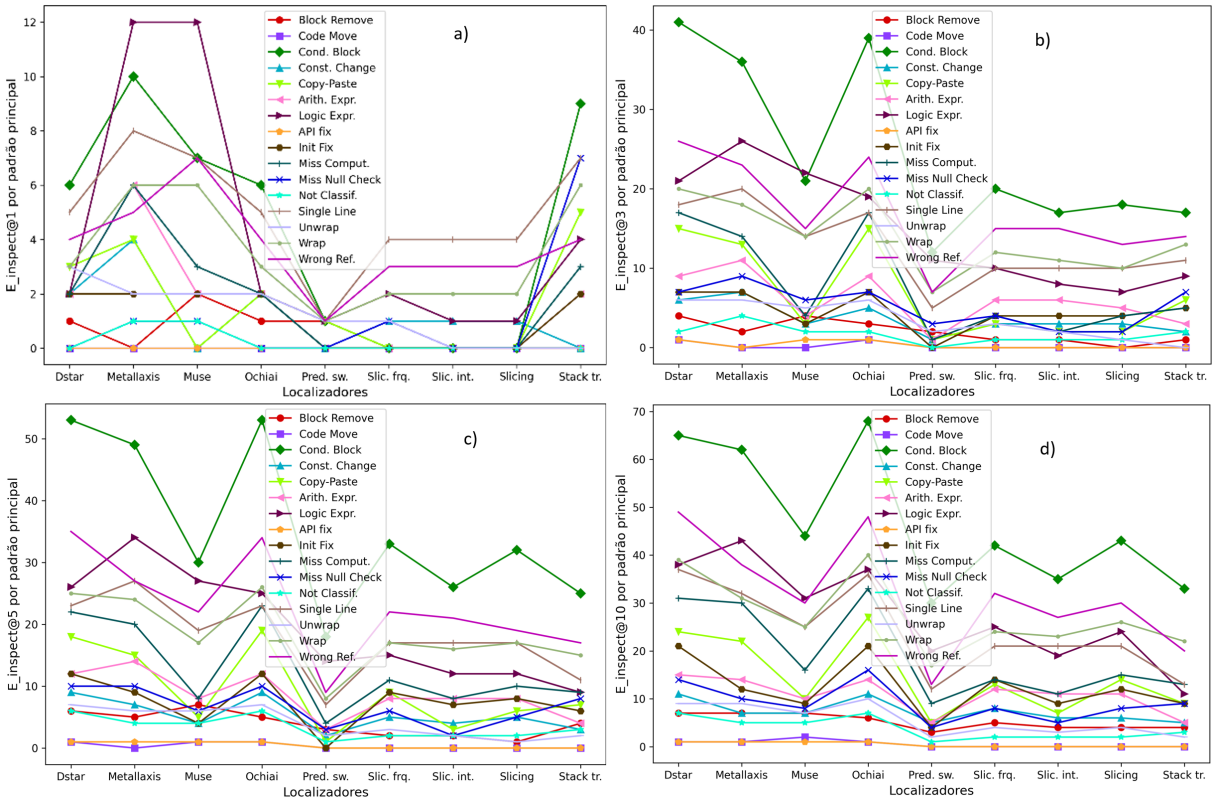

Figura 17 – Métrica Einspect@n dos localizadores por ações divididas em 16 grupos de padrões de reparo. a) n=1 b) n=3 c) n=5 d) n=10

A Figura 18 apresenta a quantidade percentual de bugs da métrica Einspect do localizadores em relação a quantidade de total de bugs envolvendo cada um dos 16 padrões de reparo apresentados.

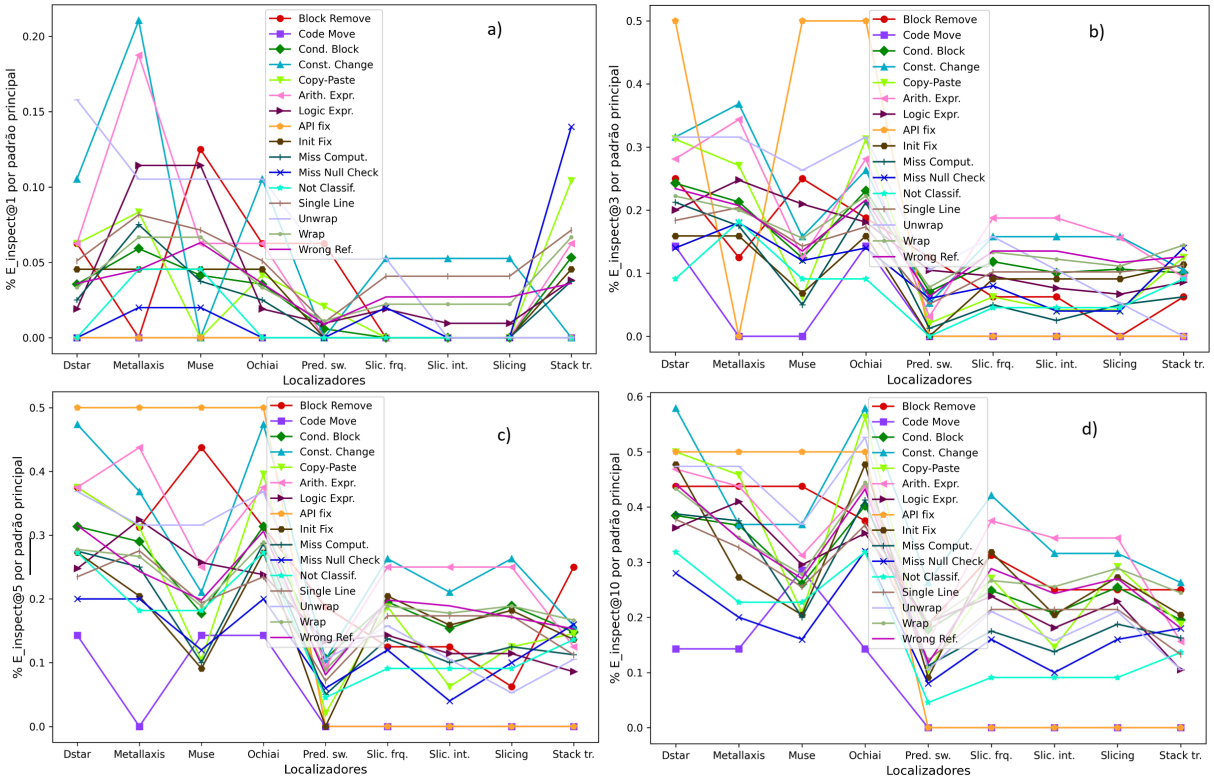

Figura 18 – Métrica Einspect@n Percentual dos localizadores por ações divididas em 16 grupos de padrões de reparo. a) n=1 b) n=3 c) n=5 d) n=10

Avaliando o resultado do gráfico acima, tem-se que:

1. De maneiras similar às estruturas sintáticas, em termos absolutos podemos observar uma certa ordenação nos padrões de reparo de maneira homogênea em relação aos localizadores.

2. Para n >=3, podemos ver que Conditional Block é o padrão com maior valor absoluto de localizações eficazes, o que é consistente com o achado anterior sobre estrutura sintática. Wrong Reference também se destaca, e é consistente com um número alto de localizações em chamadas de métodos e variáveis, pois Wrong Reference é correção de uma referência a método ou variáveis.

3. Podemos observar em termos absolutos que API fix, Code Move, Code Remove, e Unwrap tem um número baixo de localizações bem sucedidas.

4. Em termos relativos, API Fix tem um comportamento interessante. Para n=5 e 10, os localizadores Dstar, Metallaxis, Muse e Ochiai tem entre suas melhores

performances este tipo de padrão. Por outro lado, Predicate Switching, Fatiamento, e Stack Trace não conseguem localizar este tipo de bug.

5. Em termos relativos para n = 10, Constant Change é o bug melhor localizado, tendo sido assim, para Dstar e Ochiai, apesar de não ter tido performance similar com Metallaxis e Muse.

6. MissNullCheck também é um tipo de bug cuja performance de localização não é boa comparada com os demais tipos.

7. CodeMove é o padrão com mais grau de dificuldade de localização, tendo o Muse conseguido uma melhor performance de localização se considerado n=5 e 10.

Resposta para a Pergunta 3: *Existe uma associação entre o desempenho dos localizadores de bugs e os diferentes padrões de reparo?*
Pode-se observar uma associação entre alguns tipos de padrões e uma pior ou melhor performance em determinados localizadores. Os melhores casos são 1) Constant Change que tem um performance alta nos localizadores DStar e Ochiai e 2) API fixing que tem performance alta em relação em DStar, Metallaxis, Muse e Ochiai, mas por outro lado não são localizados com as demais abordagens (Pred. Sw., Fatiamento, Stack trace). Do lado negativo, Miss Null Check e Code Move são bugs com piores performances.

4.5 Discussão dos Resultados

Em relação ao tipos de reparo em relação a adição/mudança/remoção de linhas, pode-se observar percentualmente em relação aos tipos de modificação que não existe uma diferença considerável entre o tipo de modificação e a influência da mesma em alguns localizadores.

Entretanto, pode-se verificar que quando os bugs são corrigidos com Remoção e Mudança os localizadores do tipo Fatiamento se comportam melhor. Possíveis explicações incluem o fato que as técnicas de fatiamento se baseiam no rastreamento de dependências e fluxo de controle. Assim, quando o reparo implica em alteração ou remoção de algo existente, permite que o localizador explore tais dependências existentes, isto é, quando uma linha é mudada ou removida, as dependências de dados e fluxo de controle são diretamente afetadas. As técnicas de fatiamento são projetadas para analisar precisamente essas dependências, tornando mais fácil rastrear o impacto da alteração ou remoção em outras partes do código. Por outro lado, a inclusão de novas linhas pode introduzir novas dependências que não existiam anteriormente. Como essas dependências são novas, pode ser mais difícil para as técnicas de fatiamento rastrearem seu impacto completo sem um histórico de como elas se relacionam com o código existente.

Além disso, Muse comparativamente aos demais localizadores se comporta melhor com bugs que contém todos os tipos *Add*, *Rem*, e *Change*. Um possível explicação é que Muse, como um algoritmo de localização baseado em mutação, tem uma abordagem distinta para identificar falhas, considerando especificamente os casos em que os mutantes transformam um caso de teste que passa em um que falha e vice-versa. Esta característica pode explicar por que o Muse é particularmente eficaz na localização de bugs que envolvem múltiplos tipos de alterações (Add, Rem, Change) pois ele se concentra nas mudanças que têm um impacto direto e significativo no comportamento do programa, ignorando mutantes que causam falhas superficiais ou que não afetam os resultados dos testes de maneira clara. Ao focar em mutantes que causam transformações significativas, Muse consegue identificar mudanças cruciais no código que estão diretamente ligadas ao comportamento errôneo, o que é vital em cenários onde múltiplas alterações (Add, Rem, Change) estão envolvidas. Além disso, bugs que requerem adição, remoção e mudança de linhas muitas vezes implicam em interações complexas entre essas alterações. O Muse, ao analisar como diferentes mutantes afetam os casos de teste, teria maior habilidade de detectar essas interações complexas e identificar os pontos onde estas combinações de alterações estariam causando o problema. Ou seja, o Muse teria uma melhor capacidade de cobrir uma ampla gama de cenários, incluindo aqueles onde adições, remoções e mudanças de linhas interagem de maneiras inesperadas para causar falhas. Além do mais, ao ignorar os casos em que tanto os mutantes quanto o programa original falham de maneiras diferentes, o Muse consegue filtrar mutantes que não contribuem diretamente para a localização do bug. Isso permite que o algoritmo se concentre nos pontos que realmente impactam os casos de teste, tornando a análise mais eficiente.

Em relação às estruturas sintáticas, observou-se a ausência de interação de dados entre estruturas e os diferentes localizadores, ou seja, uma estrutura mais bem localizada em um localizador, por exemplo, expressões, era a mais bem localizada nos outros localizadores. Uma explicação para este fato decorre da natureza genérica dos algoritmos, ou seja, amaioria dos algoritmos de localização de falhas, como Ochiai, Dstar, Metallaxis, Muse, Predicate Switching e técnicas de fatiamento, utiliza abordagens baseadas em cobertura e diferenças de comportamento de execução. Essas abordagens são amplamente aplicáveis a várias estruturas de código, resultando em padrões de desempenho semelhantes para diferentes tipos de estruturas sintáticas.

Além disso, observou-se que bugs em Expressões são mais fáceis de localizar. Uma explicação para isto é que expressões geralmente são pequenas e têm um impacto direto e imediato no comportamento do programa. Um erro em uma expressão pode ser mais facilmente detectável porque geralmente resulta em uma mudança clara e localizada no comportamento do código. Ou seja, como em geral os algoritmos de localização baseiam-se em dados de cobertura e comportamento de execução, as alterações em expressões frequentemente resultam em diferenças claras nos resultados dos testes, facilitando

a identificação da localização da falha.

Por outro lado, bugs em Tipos e Declaração de Métodos são mais difíceis de localizar, tanto que nenhum localizador foi capaz de localizar bugs em Tipos. Talvez a principal explicação esteja relacionada ao fato de que os algoritmos de localização dependem de diferenças claras de comportamento entre execuções bem-sucedidas e falhas, e portanto ficam em dificuldade de capturar adequadamente os bugs em tipos e declarações de métodos, pois bugs nessas estruturas podem não resultar em falhas diretas ou imediatas, mas em comportamentos sutis e difíceis de rastrear. Outra possível explicação, pode estar relacionada à complexidade e escopo, ou seja, bugs em tipos e declarações de métodos geralmente afetam uma parte maior do código, causando mudanças mais sutis e espalhadas no comportamento do programa. Isso torna mais difícil para os localizadores isolarem a causa específica da falha. Além, testes de unidade e até mesmo de integração podem não cobrir todas as interações complexas que envolvem tipos e declarações de métodos. Isso resulta em menos dados disponíveis para os algoritmos de localização de falhas, tornando a identificação da origem da falha mais desafiadora.

Em relação aos padrões de reparo, pode-se observar uma associação entre alguns tipos de padrões e uma pior ou melhor performance em determinados localizadores.

Observou-se que *Constant Change* tem um performance alta em relação aos demais especialmente em DStar e Ochiai. Uma explicação é que nos localizadores é possível isolar se as constantes fazem parte ou não de execuções bem ou mal sucedidas, facilitando a localização.

Observou-se também que *API fixing* tem performance alta em DStar, Metallaxis, Muse e Ochiai, mas por outro lado não são localizados com as demais abordagens (Pred. Sw., Fatiamento, Stack trace). Uma justificativa para este desempenho é que assim como com Constante, o uso de APIs pode ser isolado com facilidade nas execuções bem/mal sucedidas. A mutações com APIs (Metallaxis, Muse) também parecem surtir bom efeito na expressão e localização do bug.

Por outro lado, *Predicate Switching*, *Fatiamento* e *Stack Trace* não cobrem chamadas de API em seus algoritmos.

Do lado negativo, Miss Null Check e Code Move são bugs com piores performances. Uma possível explicação para Miss Null Check é que uma determinada variável pode aparecer em várias regiões do programa. A necessidade de descobrir quais são as possíveis ocorrências de variáveis de instância que levaram ao comportamento inadequado se torna desafiador considerando o número de variáveis e suas respectivas ocorrências. Em relação ao Code Move, é de se esperar a complexidade envolvida em uma movimentação de código torne difícil de o algoritmo de localização entender o problema no fluxo de controle e as dependências entre diferentes partes do programa. Mudanças desta natureza são difíceis para os localizadores interpretarem Além disso, os algoritmos podem não capturar completamente os impactos indiretos das movimentações de código, especialmente se as

mudanças afetarem áreas distantes ou complexas do programa.

4.6 Ameaças a Validade

Ameaças à Validade Externa. Está relacionada com a generalização dos resultados. Os resultados obtidos podem não ser generalizáveis para além do dataset específico (Defects4J com bugs Java). A aplicabilidade dos resultados pode variar dependendo das características específicas dos sistemas estudados, e também da linguagem de programação.

Ameaças à Validade Interna. Em relação a variáveis externas não controladas que influenciam os resultados, como configurações específicas de ambiente, versões de software, ou características não documentadas nos bugs podem implicar em diferenças em relação à base de dados utilizada. Este trabalho mitigou em parte isto usando uma base de dados já curada e publicada em outro trabalho correlato.

A definição e categorização das estruturas sintáticas e padrões de reparo foram feitas manualmente no trabalho de Sobreira et al (SOBREIRA et al., 2018). Alguma inconsistência naquele trabalho pode afetar a precisão e validade dos resultados aqui apresentados. Um aspecto que mitiga este ponto é que esta classificação está em um repositório público e que não houve nenhuma solicitação de errata até o momento.

Além disso, podem ter havido inconsistências ou erros no script Python que utilizamos para o cálculo da métrica Einspect. Este ponto foi mitigado por meio de uma conferência por um terceiro das métricas coletadas.

Ameaças à Validade de Construção. Em relação à validade da métrica Einspect, existem outras alternativa, e ela pode não capturar completamente todas as nuances nos resultados dos localizadores. Entretanto é uma métrica amplamente adotada e aceita. Ainda a escolha do Defects4J como dataset pode introduzir viéses de seleção, pois nem todos os tipos de bugs ou características podem estar adequadamente representados neste dataset específico. Entretanto, talvez seja um dos datasets mais utilizados na literatura de bugs.

Ameaças à Validade de Conclusão. Em relação à interpretação dos resultados a mesma foi realizada por meio da observação dos gráficos, a qual é dependente da individualidade do pesquisador. Entretanto, o uso de métodos mais analíticos não traria a riqueza de análise que pudemos mostrar.

Capítulo 5

Trabalhos Relacionados

Nesta seção será apresentado os artigos que tiveram como objeto de estudo a implicação de fatores ou elementos que poderiam contribuir de alguma maneira com o resultado do localizador de bugs.

Os autores (GARNIER; FERREIRA; GARCIA, 2017) avaliam técnicas de LB que se baseiam em recuperação estruturada de informação para encontrar os erros, utilizando uma base de dados de programas escritos na linguagem C#. No entanto, os estudos sobre estas técnicas são simplistas e duvidosos. Sendo assim, os autores sugerem avaliar como construções de programas, incluindo construções C# inexistentes em Java, podem impactar no desempenho destas técnicas. Os autores concluem em sua análise que métodos e classes são as construções que mais contribuem para a eficácia da localização de bugs. Este artigo possui relação com esta pesquisa no ponto em que o estudo vai além dos localizadores e busca elementos no códigos que possam interferir, tanto positivamente ou negativamente no resultado de busca dos localizadores.

Zou e colegas (ZOU et al., 2021) observaram que diferentes localizações de bugs têm melhor desempenho em classes específicas de bugs e combiná-los melhora o desempenho geral dos localizadores. Este trabalho combinou abordagens de diferentes famílias, (Ochiai com Fatiamento, por exemplo) e mediu o custo de tempo destas técnicas.

Kui e colegas (LIU et al., 2019) mostram que "não se pode consertar o que você não consegue encontrar". Eles propõe destacar as diferentes configurações de localização de falhas utilizadas na literatura e seu impacto nos sistemas APR (Reparo automático de programas) quando aplicados ao benchmark Defects4J. Estes autores descobriram que apenas uma parte dos bugs do Defects4J pode ser localizada corretamente pelas técnicas de localização de bugs mais utilizadas. Nosso trabalho investigou quais classes de bugs, em relação às ações/padrões de reparo, os diferentes localizadores podem apresentam melhor acurácia.

Kim e colegas (KIM et al., 2008) analisaram 7 projetos de código fonte aberto com mais de 200 mil revisões. O objetivo deste artigo foi estudar o cache gerado por estes sistemas para verificar a possibilidade de obter os arquivos que apresentam as classes

defeituosas. Com base em uma falha conhecida, os pesquisadores armazenam em cache todos os demais arquivos que estão provavelmente relacionados com aquela falha. Eles concluíram que o cache seleciona 10% dos arquivos que são responsáveis por 73% a 95% das falhas no código.

Pearson e colegas (PEARSON et al., 2017) fizeram um estudo sobre a localização de bugs utilizando falhas de projetos reais, uma vez que estes projetos possuem algumas diferenças com bugs artificiais. Eles replicaram estudos existentes na literatura, utilizando 10 pares de combinações de técnicas de localização de bugs (abordagens baseadas em espectro e baseadas em mutação), afim de compará-las. Os pesquisadores usaram 2.995 falhas artificiais em 6 programas do mundo real e 310 falhas reais dos mesmos programas. A conclusão obtida foi que os resultados anteriores foram estatisticamente insignificantes. O experimento destes autores mostrou que as falhas artificiais não são úteis para prever quais técnicas de localização de falhas apresentam melhor acurácia em falhas reais. Este artigo e nosso trabalho se assemelham no momento em que buscam elementos no próprio código para realizar a busca por bugs.

Rahman e colegas (RAHMAN; ROY, 2018) descobriram que as técnicas de localização de bugs baseadas em recuperação de informações (IR) não funcionam bem se o relatório de bugs não tiver informações estruturadas ricas, como os nomes de entidades. Com isso, eles propõem uma ferramenta de localização de bugs (Blizzard) que localiza automaticamente entidades com bugs a partir da fonte do projeto usando queries de consultas reformuladas para a recuperação eficaz de informações. Para este experimento, os autores utilizaram 5.139 relatórios de bugs e eles concluíram que a nova técnica é capaz de aumentar a acurácia da localização de bugs em 19% em relação as demais técnicas existentes.

Saha e colegas (SAHA et al., 2013) desenvolveu uma ferramenta chamada BLUiR que é baseada na recuperação estruturada de informações com base em construções de código, como nomes de classes e métodos, o que permite uma localização de bugs mais precisa. Este estudo fornece uma base completa da pesquisa de localização de bugs baseada em recuperação da informação (IR). A ferramenta BLUiR foi analisada em 4 projetos de código aberto com aproximadamente 3.400 bugs e se mostrou superior às ferramentas de últimas geração.

Capítulo 6

Conclusão

O objetivo deste estudo experimental foi analisar a possibilidade de existência de associação entre o desempenho de diferentes localizadores de bugs, avaliados com a métrica Einspect, e diversas características dos bugs definidas por características dos reparo de bugs, sejam elas os tipos de alterações (Add, Rem, Change), as estruturas sintáticas alteradas e os padrões de reparo.

Para conduzir este estudo, utilizamos o dataset Defects4J, que contém 395 bugs Java provenientes de diversos projetos de código aberto. O Defects4J é uma coleção bem estabelecida e amplamente utilizada na pesquisa de engenharia de software, proporcionando um ambiente controlado e padronizado para avaliação de técnicas de depuração.

Os dados de desempenho dos localizadores de bugs foram extraídos de um estudo de Zou et al (ZOU et al., 2021) que executou os localizadores sobre os bugs presentes no Defects4J e obteve listas de elementos suspeitos. A métrica Einspect foi utilizada para avaliar a eficácia dos localizadores, considerando a precisão na identificação de linhas de código defeituosas em diferentes posições na lista de suspeitos.

Os localizadores de bugs estudados foram:

- **DStar**: Uma técnica de localização baseada em cobertura que utiliza a diferença na execução de casos de teste passando e falhando para atribuir pontuações de suspeita a linhas de código.

- **Ochiai**: Uma técnica baseada em cobertura que calcula a suspeita de linhas de código usando o coeficiente de Ochiai, que considera a frequência de execução de linhas em testes passando e falhando.

- **Metallaxis**: Um localizador baseado em mutação que aplica mutações ao código e observa se os casos de teste falham como resultado, ajudando a identificar linhas de código defeituosas.

- **Muse**: Outro localizador baseado em mutação que se concentra em mutações que transformam casos de teste que passam em casos que falham e vice-versa, ignorando

mutações onde ambos, mutantes e o programa original, falham.

- **Predicate Switching**: Uma técnica que altera condições lógicas no código para observar o impacto nas falhas dos testes, ajudando a identificar a origem dos bugs.
- **Técnicas de Fatiamento**: Técnicas que extraem subconjuntos de programas relacionados a variáveis ou instruções específicas, facilitando a identificação de dependências e causas de falhas.
- **Stack Trace**: Utiliza informações de rastreamento de pilha geradas durante exceções para identificar a localização das falhas.

A análise inicial focou-se nos tipos de alterações necessárias para reparar os bugs, dividindo-os em três categorias: adição de linhas (Add), remoção de linhas (Rem) e modificação de linhas (Change). Observamos que:

- Para EInspect@1, os localizadores Metallaxis e Muse apresentaram melhor acurácia em bugs que requerem adição de linhas.
- Para EInspect@3, Dstar, Metallaxis, Muse e Predicate Switching foram mais eficazes em capturar bugs que requerem remoção de linhas. As técnicas de fatiamento se destacaram na captura de bugs que envolvem mudanças, enquanto Stack Trace foi mais eficaz em bugs que envolvem adição de linhas.
- Geralmente, todas as técnicas se comportaram melhor com bugs que envolvem remoção e mudança de linhas (Rem-Change). Em contraste, a eficácia foi inferior para bugs que envolvem adição e mudança de linhas (Add-Change).
- Para EInspect@3, apenas Dstar, Metallaxis, Muse e Ochiai conseguiram identificar corretamente os bugs. As demais técnicas conseguiram localizar os bugs somente quando Einspect@n com n >= 3. Para n>=3, Dstar, Ochiai, Metallaxis foram as abordagens que mais frequentemente listaram os elementos suspeitos nas primeiras posições.

Esses resultados sugerem que a natureza das alterações necessárias para corrigir um bug tem alguma associação com a eficácia dos localizadores de falhas. Técnicas baseadas em cobertura, como Dstar e Ochiai, bem como técnicas baseadas em mutação, como Muse, tendem a se sair melhor em cenários de remoção e mudança, onde a alteração do fluxo de controle ou da lógica do programa pode ser mais perceptível.

Quando avaliamos o desempenho dos localizadores em diferentes estruturas sintáticas, observamos que:

- **Bugs em expressões** são mais fáceis de localizar, pois influenciam diretamente a saída do programa de maneira perceptível.

- **Bugs em tipos e declarações de métodos** são os mais difíceis de localizar. Nenhum localizador conseguiu identificar bugs relacionados a tipos.

- De maneira consistente, uma estrutura melhor localizada por um localizador, como expressões, também foi melhor localizada pelos outros localizadores, indicando uma homogeneidade no desempenho relativo entre os diferentes localizadores para cada tipo de estrutura sintática.

Esses achados podem ser atribuídos ao fato de que alterações em expressões tendem a ter um impacto direto e imediato no comportamento do programa, sendo assim bugs que implicam em alterações nesta estrutura facilitam técnicas de localização que analisam a execução e as saídas dos testes. Em contraste, problemas em tipos e declarações de métodos podem introduzir falhas sutis e indiretas, tornando-as mais difíceis de localizar.

Finalmente, ao considerar 16 diferentes padrões de reparo, verificamos que:

- **Constant Change** indicou uma eficácia alta em relação aos demais padrões nos diversos localizadores, exceto Metallaxis e Muse. Valores de constante podem influenciar a saída do programa de maneira direta e perceptível, facilitando a localização de um possível bug.

- **API Fixing** teve um desempenho alto nos localizadores DStar, Metallaxis, Muse e Ochiai, mas não foi bem localizado pelas demais abordagens (Predicate Switching, Fatiamento, Stack Trace). A correção de uso de APIs pode alterar significativamente a interação do programa com bibliotecas externas, sendo bugs que tem este tipo de reparo localizados mais facilmente por técnicas que analisam o comportamento geral do programa.

- **Miss Null Check** e **Code Move** foram os padrões de reparo com pior desempenho. A detecção de bugs relacionados a verificações de null é difícil porque eles podem se manifestar em situações específicas e complexas, dificultando a cobertura completa dos casos de teste. Bugs cujo reparo envolve movimentação de código (Code Move) que altera significativamente o fluxo de controle e as dependências, tende a tornar a localização mais desafiadora.

Esses padrões de reparo mostram como diferentes características dos bugs influenciam a eficácia das técnicas de localização. Técnicas baseadas em cobertura e análise dinâmica tendem a ter melhor desempenho em bugs cujos padrões de reparo causam alterações diretas e perceptíveis no comportamento do programa, enquanto bugs cujos padrões que introduzem mudanças complexas no fluxo de controle ou dependências são mais desafiadores.

6.1 Trabalhos Futuros

Com base nos achados deste estudo, sugerimos as seguintes direções para trabalhos futuros:

- **Exploração de Diversos Datasets**: Avaliar a replicabilidade dos resultados em múltiplos datasets para assegurar a generalização dos achados em diferentes contextos e tipos de sistemas.
- **Melhoria das Técnicas de Localização**: Desenvolver novas técnicas de localização que abordem especificamente as limitações observadas em padrões de reparo complexos, como Code Move e Miss Null Check, melhorando a detecção de falhas indiretas e sutis.
- **Integração de Abordagens Complementares**: Explorar novas formas de integração de múltiplas técnicas de localização de falhas para aproveitar as complementariedades observadas. Por exemplo, combinar técnicas baseadas em cobertura com técnicas baseadas em mutação e análise dinâmica para obter melhores resultados em diferentes tipos de bugs.

Em resumo, este estudo forneceu poucos achados sobre a associação entre o desempenho dos localizadores de bugs e diferentes características dos bugs, destacando as áreas onde técnicas de localização podem ser melhoradas e adaptadas para lidar com desafios específicos. As direções para trabalhos futuros sugeridas aqui oferecem um caminho promissor para avanços na área de localização de falhas, com o objetivo final de melhorar a qualidade e a manutenção de software.

6.2 Publicação

Este trabalho foi publicado nos Anais do XI Workshop de Visualização, Evolução e Manutenção de Software (VEM 2023).

DIAS, JÚLIA MANFRIN ; MAIA, MARCELO DE ALMEIDA . *On the relationship of repair actions and patterns on bug localization approaches: a comparative study.* In: **Anais do XI Workshop de Visualização, Evolução e Manutenção de Software (VEM 2023)**. p. 21-25.

Referências

ABREU, R.; ZOETEWEIJ, P.; GEMUND, A. J. van. On the accuracy of spectrum-based fault localization. In: **Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)**. [S.l.: s.n.], 2007. p. 89–98.

AGRAWAL, H. et al. Fault localization using execution slices and dataflow tests. In: **Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95**. [S.l.: s.n.], 1995. p. 143–151.

DURIEUX, T. et al. Dynamic patch generation for null pointer exceptions using metaprogramming. In: **2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)**. [S.l.: s.n.], 2017. p. 349–358.

_____. Empirical Review of Java Program Repair Tools: A Large-Scale Experiment on 2,141 Bugs and 23,551 Repair Attempts. In: **Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19)**. [s.n.], 2019. Disponível em: <https://arxiv.org/abs/1905.11973>.

DURIEUX, T.; MONPERRUS, M. Dynamoth: dynamic code synthesis for automatic program repair. In: **Proceedings of the 11th International Workshop on Automation of Software Test**. New York, NY, USA: Association for Computing Machinery, 2016. (AST '16), p. 85–91. ISBN 9781450341516. Disponível em: <https://doi.org/10.1145/2896921.2896931>.

GARNIER, M.; FERREIRA, I.; GARCIA, A. On the influence of program constructs on bug localization effectiveness. **Journal of Software Engineering Research and Development**, v. 5, p. 6:1 – 6:29, Aug. 2017. Disponível em: <https://sol.sbc.org.br/journals/index.php/jserd/article/view/434>.

GOUES, C. L. et al. The manybugs and introclass benchmarks for automated repair of c programs. **IEEE Transactions on Software Engineering**, v. 41, n. 12, p. 1236–1256, 2015.

_____. Genprog: A generic method for automatic software repair. **IEEE Trans. Softw. Eng.**, IEEE Press, v. 38, n. 1, p. 54–72, jan 2012. ISSN 0098-5589. Disponível em: <https://doi.org/10.1109/TSE.2011.104>.

JUST, R.; JALALI, D.; ERNST, M. D. Defects4J: A Database of existing faults to enable controlled testing studies for Java programs. In: **ISSTA 2014, Proceedings of the 2014 International Symposium on Software Testing and Analysis**. San Jose, CA, USA: [s.n.], 2014. p. 437–440. Tool demo.

KIM, S. et al. Predicting faults from cached history. In: **Proceedings of the 1st India Software Engineering Conference**. New York, NY, USA: Association for Computing Machinery, 2008. (ISEC '08), p. 15–16. ISBN 9781595939173. Disponível em: <https://doi.org/10.1145/1342211.1342216>.

LIN, D. et al. Quixbugs: a multi-lingual program repair benchmark set based on the quixey challenge. In: **Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity**. New York, NY, USA: Association for Computing Machinery, 2017. (SPLASH Companion 2017), p. 55–56. ISBN 9781450355148. Disponível em: <https://doi.org/10.1145/3135932.3135941>.

LIU, K. et al. **You Cannot Fix What You Cannot Find! An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems**. 2019.

MADEIRAL, F. et al. Bears: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In: **Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '19)**. [s.n.], 2019. Disponível em: <https://arxiv.org/abs/1901.06024>.

MARINHO, E. H. et al. Applying spectrum-based fault localization to android applications. In: **Proceedings of the XXXVII Brazilian Symposium on Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2023. (SBES '23), p. 257–266. ISBN 9798400707872. Disponível em: <https://doi.org/10.1145/3613372.3613397>.

MARTINEZ, M.; MONPERRUS, M. Ultra-large repair search space with automatically mined templates: The cardumen mode of astor. In: COLANZI, T. E.; MCMINN, P. (Ed.). **Search-Based Software Engineering**. Cham: Springer International Publishing, 2018. p. 65–86. ISBN 978-3-319-99241-9.

_____. Astor: Exploring the design space of generate-and-validate program repair beyond genprog. **Journal of Systems and Software**, v. 151, p. 65–80, 2019. ISSN 0164-1212. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121219300159>.

MOON, S. et al. Ask the mutants: Mutating faulty programs for fault localization. In: **2014 IEEE Seventh International Conference on Software Testing, Verification and Validation**. [S.l.: s.n.], 2014. p. 153–162.

NILIZADEH, A. et al. Exploring true test overfitting in dynamic automated program repair using formal methods. In: **2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)**. [S.l.: s.n.], 2021. p. 229–240.

PAPADAKIS, M.; TRAON, Y. L. Metallaxis-fl: mutation-based fault localization. **Softw. Test. Verif. Reliab.**, John Wiley and Sons Ltd., GBR, v. 25, n. 5–7, p. 605–628, aug 2015. ISSN 0960-0833. Disponível em: <https://doi.org/10.1002/stvr.1509>.

_____. Metallaxis-fl: Mutation-based fault localization. **Softw. Test. Verif. Reliab.**, John Wiley and Sons Ltd., GBR, v. 25, n. 5–7, p. 605–628, aug 2015. ISSN 0960-0833. Disponível em: <https://doi.org/10.1002/stvr.1509>.

PEARSON, S. et al. Evaluating and improving fault localization. In: **2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)**. [S.l.: s.n.], 2017. p. 609–620.

RAHMAN, M. M.; ROY, C. K. Improving ir-based bug localization with context-aware query reformulation. In: **Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2018. (ESEC/FSE 2018), p. 621–632. ISBN 9781450355735. Disponível em: <https://doi.org/10.1145/3236024.3236065>.

SAHA, R. et al. Bugs.jar: A large-scale, diverse dataset of real-world Java bugs. In: **2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)**. [S.l.: s.n.], 2018. p. 10–13.

SAHA, R. K. et al. Improving bug localization using structured information retrieval. In: **Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering**. IEEE Press, 2013. (ASE '13), p. 345–355. ISBN 9781479902156. Disponível em: <https://doi.org/10.1109/ASE.2013.6693093>.

SCHROTER, A. et al. Do stack traces help developers fix bugs? In: **2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)**. [S.l.: s.n.], 2010. p. 118–121.

SOBREIRA, V. et al. Dissection of a bug dataset: Anatomy of 395 patches from Defects4J. In: **2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)**. [S.l.: s.n.], 2018. p. 130–140.

SOMMERVILLE, I. **Engenharia de software**. Pearson Prentice Hall, 2011. ISBN 9788579361081. Disponível em: <https://books.google.com.br/books?id=H4u5ygAACAAJ>.

TOMASSI, D. A. et al. Bugswarm: mining and continuously growing a dataset of reproducible failures and fixes. In: **Proceedings of the 41st International Conference on Software Engineering**. IEEE Press, 2019. (ICSE '19), p. 339–349. Disponível em: <https://doi.org/10.1109/ICSE.2019.00048>.

WEN, M. et al. **An Empirical Analysis of the Influence of Fault Space on Search-Based Automated Program Repair**. 2017.

WONG, C.-P. et al. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In: **2014 IEEE International Conference on Software Maintenance and Evolution**. [S.l.: s.n.], 2014. p. 181–190.

WONG, W. E. et al. The DStar method for effective software fault localization. **IEEE Transactions on Reliability**, v. 63, n. 1, p. 290–308, 2014.

_____. A survey on software fault localization. **IEEE Transactions on Software Engineering**, v. 42, n. 8, p. 707–740, 2016.

XUAN, J. et al. Nopol: Automatic repair of conditional statement bugs in Java programs. **IEEE Trans. Softw. Eng.**, IEEE Press, v. 43, n. 1, p. 34–55, jan 2017. ISSN 0098-5589. Disponível em: <https://doi.org/10.1109/TSE.2016.2560811>.

YUAN, Y.; BANZHAF, W. A hybrid evolutionary system for automatic software repair. In: **Proceedings of the Genetic and Evolutionary Computation Conference**. New York, NY, USA: Association for Computing Machinery, 2019. (GECCO '19), p. 1417–1425. ISBN 9781450361118. Disponível em: <https://doi.org/10.1145/3321707.3321830>.

ZHANG, X.; GUPTA, N.; GUPTA, R. Locating faults through automated predicate switching. In: **Proceedings of the 28th International Conference on Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2006. (ICSE '06), p. 272–281. ISBN 1595933751. Disponível em: <https://doi.org/10.1145/1134285.1134324>.

_____. A study of effectiveness of dynamic slicing in locating real faults. **Empirical Softw. Engg.**, Kluwer Academic Publishers, USA, v. 12, n. 2, p. 143–160, apr 2007. ISSN 1382-3256. Disponível em: <https://doi.org/10.1007/s10664-006-9007-3>.

ZOU, D. et al. An empirical study of fault localization families and their combinations. **IEEE Trans. Softw. Eng.**, IEEE Press, v. 47, n. 2, p. 332–347, feb 2021. ISSN 0098-5589. Disponível em: <https://doi.org/10.1109/TSE.2019.2892102>.