UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ARTES, CIÊNCIAS E HUMANIDADES
PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

DENNIS LOPES DA SILVA

Impacto da Relação de Subsunção na Localização de Defeitos baseados em Espectros de Fluxo de Dados

São Paulo
2023

DENNIS LOPES DA SILVA

Impacto da Relação de Subsunção na Localização de Defeitos baseados em Espectros de Fluxo de Dados

Versão corrigida

Dissertação apresentada à Escola de Artes, Ciências e Humanidades da Universidade de São Paulo para obtenção do título de Mestre em Ciências pelo Programa de Pós-graduação em Sistemas de Informação.

Área de concentração: Metodologia e Técnicas da Computação

Versão corrigida contendo as alterações solicitadas pela comissão julgadora em 30 de março de 2023. A versão original encontra-se em acervo reservado na Biblioteca da EACH-USP e na Biblioteca Digital de Teses e Dissertações da USP (BDTD), de acordo com a Resolução CoPGr 6018, de 13 de outubro de 2011.

Orientador: Prof. Dr. Marcos Lordello Chaim

São Paulo

2023

Ficha catalográfica elaborada pela Biblioteca da Escola de Artes, Ciências e Humanidades,
com os dados inseridos pelo(a) autor(a)
Brenda Fontes Malheiros de Castro CRB 8-7012; Sandra Tokarevicz CRB 8-4936

Lopes da Silva, Dennis
Impacto da Relação de Subsunção na Localização de Defeitos baseados em Espectros de Fluxo de Dados / Dennis Lopes da Silva; orientador, Marcos Lordello Chaim. -- São Paulo, 2023.
84 p: il.

Dissertacao (Mestrado em Ciencias) - Programa de Pós-Graduação em Sistemas de Informação, Escola de Artes, Ciências e Humanidades, Universidade de São Paulo, 2023.
Versão corrigida

1. Testes de Software. 2. Localização de defeitos. 3. Relação de subsunção. 4. Fluxo de Dados. I. Chaim, Marcos Lordello, orient. II. Título.

Dissertação de autoria de Dennis Lopes da Silva, sob o título **"Impacto da Relação de Subsunção na Localização de Defeitos baseados em Espectros de Fluxo de Dados"**, apresentada à Escola de Artes, Ciências e Humanidades da Universidade de São Paulo, para obtenção do título de Mestre em Ciências pelo Programa de Pós-graduação em Sistemas de Informação, na área de concentração Metodologia e Técnicas da Computação, aprovada em 30 de março de 2023 pela comissão julgadora constituída pelos doutores:

Prof. Dr. Marcos Lordello Chaim

Escola de Artes, Ciências e Humanidades - Universidade de São Paulo

Presidente

Profa. Dra. Ellen Francine Barbosa

Instituto de Ciências Matemáticas e de Computação - Universidade de São Paulo

---

Prof. Dr. Delano Medeiros Beder

Universidade Federal de São Carlos

Para Michelle, Gabriel, Maria, José e Ana Cristina.

Agradecimentos

À Escola de Artes, Ciências e Humanidades da Universidade de São Paulo, onde me senti acolhido desde o primeiro dia, e de maneira muito especial ao meu orientador, Marcos Lordello Chaim, pela sua paciência, zelo e dedicação durante toda a minha jornada.

Resumo

SILVA, Dennis Lopes da. **Impacto da Relação de Subsunção na Localização de Defeitos baseados em Espectros de Fluxo de Dados**. 2023. 84 f. Dissertação (Mestrado em Ciências) – Escola de Artes, Ciências e Humanidades, Universidade de São Paulo, São Paulo, 2023.

Depuração tem por objetivo localizar e corrigir os defeitos do software. Para auxiliar o desenvolvedor foram desenvolvidas técnicas de localização de defeitos que utilizam métricas de associação e dados de cobertura de código (*espectros*) para identificar os trechos de código mais suspeitos. Elas auxiliam o desenvolvedor por meio de um *ranking* dos espectros mais suspeitos que pode ser usado para orientar a *caça* aos defeitos. Essas técnicas, quando baseadas em espectros de fluxo de dados, utilizam as associações definição uso (ADU) para cálculo das posições no *ranking*. No entanto, a cobertura de dadas ADUs, muitas vezes, garantem a cobertura de outras ADUs, numa relação entre elas denominada *subsunção*. Na prática, a relação de subsunção significa que se uma determinada ADU é coberta, outras também são, em determinadas condições, garantidamente cobertas. Com base na propriedade de subsunção, esse trabalho apresenta um experimento no qual é avaliada a eficácia da localização de defeitos com a utilização apenas dos espectros do conjunto de ADUs *não-limitadas*, ou seja, o conjunto minimal de ADUs que garante a cobertura de todas as outras ADUs do software em teste. Para tal experimento são utilizados um subconjunto dos programas do repositório *Defects4J*, espectros de fluxo de dados e a métrica de associação *Ochiai*. Os resultados do experimento indicam que a maioria dos defeitos localizados por espectros de fluxo de dados podem ser localizados inspecionando apenas as ADUs não-limitadas, sobretudo quando são consideradas apenas as ADUs posicionadas nos primeiros *rankings*. Além disso, o número de linhas de código a serem inspecionadas pelo programador é reduzido.

Palavras-chaves: Testes de Software, Localização de defeitos, Relação de subsunção, Fluxo de Dados.

Abstract

SILVA, Dennis Lopes da. **Data flow Subsumption and its Impact on Spectrum-based Fault Localization**. 2023. 84 p. Dissertation (Master of Science) – School of Arts, Sciences and Humanities, University of São Paulo, São Paulo, 2023.

Debugging aims at finding and correcting software defects. To help the developer, fault localization techniques were developed using association metrics and code coverage data spectra to identify the most suspicious code snippets. They assist the developer by means of a ranking of the most suspicious spectra that guides the developer in his or her "hunt" for defects. These techniques, when based on data flow spectra, use definition use associations (DUA) for ranking calculation. However, the coverage of given DUAs often guarantees the coverage of other DUAs, in a relationship between DUAs called subsumption. In practice, the subsumption relationship means that if a given DUA is covered, others are also guaranteed to be covered in certain conditions. Based on the subsumption property, this work presents an experiment in which fault localization effectiveness is assessed using only the spectra of the set of unconstrained DUAs, that is, the minimal set of DUAs that may guarantee coverage of all other DUAs of the software under test. For this experiment, we use a subset of programs of the Defects4J repository, data flow spectra, and the Ochiai association metric. Our results compare the rankings produced by the set of unconstrained DUAs against those produced by all DUAs for fault localization. They indicate that most of the faults reached by DUA spectra can be found by inspecting only the unconstrained DUAs, especially when only the ADUs positioned at the first rankings are taken into account. Furthermore, the number of lines of code to be inspected by the programmer is reduced.

Keywords: Software Testing, Software Fault Localization, Subsumption Relationship, Data flow.

Lista de figuras

Lista de tabelas

Sumário

[1] De acordo com a Associação Brasileira de Normas Técnicas. NBR 6023.

1 Introdução

A influência dos sistemas de software em nossas vidas é praticamente ubíqua. O software não só está presente como também é crítico em muitas atividades econômicas tais como a medicina, aeronáutica e energia nuclear (WONG; GAO; LI; ABREU; WOTAWA, 2016).

Embora muitos fatores afetem a engenharia de um software confiável, incluindo um projeto cuidadoso e uma sólida gestão de processos, o teste é a principal forma de avaliação utilizada pela indústria para avaliar o software durante o ciclo de desenvolvimento (AMMANN; OFFUTT, 2017). Um resultado do teste bem sucedido é a detecção da presença de defeitos nos sistemas de software. Porém, esses defeitos precisam ser localizados e corrigidos. Dessa forma, o teste detecta a existência de defeitos enquanto a depuração localiza e corrige os defeitos detectados (SOMMERVILLE, 2016). Estudos mostram que o teste e a depuração estão entre as atividades e tarefas mais caras do ciclo de desenvolvimento de software (NIST, 2002).

A atividade de depuração visa a atingir dois objetivos no desenvolvimento de software: localizar e corrigir defeitos (ZELLER, 2005). A depuração geralmente consiste na execução do programa por um determinado tempo, examinando conteúdos da memória que estejam incorretos e repetindo isso em incrementos menores, ao mesmo tempo em que se tenta limitar o escopo da execução ao ponto onde ocorre o primeiro defeito (BARR, 2018).

Existem alguns modelos para a atividade de depuração. Um deles define que a depuração é o processo recorrente de geração e validação de hipóteses. O objetivo é encontrar os defeitos que causam falhas (comportamento inadequado observável do software) a partir da confirmação ou refutação de hipóteses elaboradas pelo desenvolvedor (ARAKI; FURUKAWA; CHENG, 1991). Outro, por sua vez, avalia que a depuração de software é semelhante a uma caçada na qual o defeito é a caça e o caçador é o desenvolvedor de software (LAWRANCE *et al.*, 2013). A ideia é que o desenvolvedor não somente elabora e testa hipóteses para localizar e corrigir os defeitos, mas também utiliza o seu "faro"ou intuição. O "faro" do desenvolvedor advém da avaliação do comportamento dos testes e do seu conhecimento do código.

Esta dissertação visa estudar e aprimorar técnicas que auxiliam o desenvolvedor a identificar pontos mais associados a falhas — e, por consequência, aos defeitos —, potencializando tanto sua capacidade de geração de hipóteses como seu "faro" em relação ao defeito. Segundo Barr (2018), a correção geralmente é óbvia uma vez que o defeito seja encontrado e o código familiar.

1.1 Motivação

Uma vez que os defeitos foram detectados por meio do teste do software, o desenvolvedor tem a tarefa de corrigi-lo. Identificar a localização dos defeitos tem sido historicamente uma tarefa tediosa, demorada e proibitiavemente cara (VESSEY, 1985; WONG; GAO; LI; ABREU; WOTAWA, 2016; SOUZA; CHAIM; KON, 2016). Entre as técnicas tradicionais de localização de defeitos estão a inserção de comandos de escrita, asserções, *breakpoints* e *profiling* (WONG; GAO; LI; ABREU; WOTAWA, 2016):

- A inserção de comandos de escrita é a colocação, em pontos estratégicos do programa, de comandos para imprimir os valores de determinadas variáveis (CHAIM, 2001);
- Asserções são baseadas na inclusão de parte da especificação do programa no seu código fonte, de maneira que uma ação é ativada toda vez que a especificação parcial do programa é violada durante a execução (CHAIM, 2001);
- *Breakpoints* são usados para pausar o programa quando a execução atinge um ponto determinado, permitindo ao usuário examinar o estado corrente do software e, no momento da pausa, modificar valores de variáveis ou continuar a execução para observar a progressão do defeito (WONG; GAO; LI; ABREU; WOTAWA, 2016);
- *Profiling* é análise de métricas como velocidade de execução e uso de memória, que tipicamente são utilizadas na otimização de programas, para alavancar atividades de depuração como identificação de vazamentos de memória (*memory leaks*), frequências de execução de funções inesperadas, entre outros (WONG; GAO; LI; ABREU; WOTAWA, 2016).

As técnicas acima apoiam a localização manual de defeitos, isto é, aquela que depende fundamentalmente da ação humana. No entanto, localizar defeitos manualmente baseia-se na experiência, julgamento e intuição do desenvolvedor para identificar e priorizar o código com maior probabilidade de conter o defeito. Nos dias atuais, com o tamanho e a

escala dos softwares, essas técnicas não são efetivas no isolamento da origem do defeito (WONG; GAO; LI; ABREU; WOTAWA, 2016).

Com isso houve um esforço no desenvolvimento de novas técnicas de localização e correção de defeitos tais como fatiamento de programas (WEISER, 1981), localização baseada em espectros (COLLOFELLO; COUSINS, 1987), localização baseada em estatísticas (LIBLIT *et al.*, 2005; LIU *et al.*, 2006), entre outras (WONG; GAO; LI; ABREU; WOTAWA, 2016).

As técnicas de localização de defeitos baseadas em espectros são promissoras porque reduzem o espaço de busca dos defeitos a um custo em tempo de execução razoável (SOUZA; CHAIM; KON, 2016). Essas técnicas identificam os componentes (linhas, blocos, nós, ramos, funções, métodos, associações definição uso), também chamados *espectros*, com maior probabilidade de conter defeitos. Elas são baseadas em *métricas de associação* que atribuem valores mais altos aos espectros mais associados a casos de teste que falham e menos a casos de teste que passam. A ideia é que os espectros com maiores valores nas métricas de associação são mais suspeitos e, por isso, possuem maior chance de conter os defeitos que fazem o programa falhar.

Os espectros são, na maioria da vezes, derivados de requisitos de teste de critérios de teste estruturais baseados em fluxo de controle e de dados. Exemplos de critérios fluxo de controle são os critérios todos nós e todos ramos. *Todos-nós* requer que todo bloco de comandos sequenciais, chamado de nó, seja executado ao menos uma vez e *todos-ramos* requer que toda transferência de fluxo de controle, chamada de ramo, seja testada ao menos uma vez (RAPPS; WEYUKER, 1985). O critério de teste típico de fluxo de dados é o critério *todos-usos*. Basicamente, ele requer que os conjuntos de casos de teste exercitem pelo menos um caminho entre a definição (ponto do programa onde um valor é atribuído) e os subsequentes usos (referências ao valor atribuído) de uma variável. Os requisitos de teste do critério todos-usos são chamados de *associações definição uso* (ADU) (RAPPS; WEYUKER, 1985).

A maior parte dos estudos de localização de defeitos utilizam espectros baseadas em fluxo de controle como linhas, nós e ramos. Isso porque o código (fonte ou objeto) precisa ser instrumentado para rastrear os elementos executados pelo caso de teste. Essa instrumentação gera um custo que tende a ser menor para espectros baseados em fluxo de controle em comparação com os espectros de fluxo de dados (RIBEIRO; ROBERTO; CHAIM, 2019). Trabalhos anteriores, entretanto, indicam ser possível usar espectros

baseados em fluxo de dados a custos razoáveis de execução (ARAUJO; CHAIM, 2014; CHAIM; ARAUJO, 2013). O uso de associações definição uso — o espectro de fluxo de dados mais conhecido e utilizado — mostrou-se promissor na localização de defeitos (SANTELICES *et al.*, 2009; RIBEIRO; ROBERTO; CHAIM, 2019). Espectros baseados em fluxo de dados possuem mais informações contextuais (e.g., as variáveis definidas e usadas) que os baseados em fluxo de controle, o que, por hipótese, pode trazer mais eficácia à localização de defeitos.

Um problema, porém, das técnicas de localização baseadas em espectros é que elas ignoram as relações de dependência entre os espectros (ZHAO; WANG; YWIN, 2011). Muitas vezes, um espectro pode possuir um valor alto de suspeição porque ele é "dependente" de um outro espectro, este sim realmente associado aos casos de teste que falham. Essa relação de "dependência"entre os espectros, chamada de *subsunção*, pode levar à ocorrência de "empates" entre eles, fazendo com que o desenvolvedor dirija sua atenção a espectros que não estão diretamente associados a falhas.

O conceito de subsunção foi desenvolvido, inicialmente, para comparar diferentes critérios de teste (RAPPS; WEYUKER, 1985; CLARKE; PODGURSKI; RICHARDSON; ZEIL, 1985). De acordo com sua definição, um critério de cobertura C_1 *subsume* um outro critério C_2 se, e somente se, todo teste que satisfaz o critério C_1 também satisfaz o critério C_2. A ideia é que o critério *subsumissor* C_1 "herdaria" a capacidade de detecção de falhas do critério *subsumido* C_2.

O conceito de subsunção também foi aplicado para o teste baseado em mutantes por Ammann, Delamaro e Offutt (2014). Esses autores determinam um conjunto mínimo de mutantes de forma a garantir que todos os mutantes subsumidos sejam eliminados quando os elementos do conjunto mínimo são mortos. Dessa maneira, é possível manter a qualidade de um determinado conjunto de teste a um custo menor.

Marré e Bertolino (1996) estenderam a relação de *subsunção* entre critérios de teste para a relação de subsunção entre requisitos de teste. Sejam tr_1 e tr_2 dois requisitos de teste de um critério C, tr_1 *subsume* tr_2 se, e somente se, todo caminho de teste completo que exercita (cobre) tr_1 também exercita (cobre) tr_2. Esse resultado permite identificar um subconjunto minimal de requisitos de teste que maximizam a cobertura e, dessa maneira, minimizam o esforço do teste.

Denmat, Ducassé e Ridoux (2005) observaram que métricas de associação usadas para classificar trechos de código suspeitos em técnicas de localização de defeitos baseadas

em espectro sofrem com o problema da dependência (subsunção). Por exemplo, se determinada instrução é classificada como suspeita, também outras instruções dependentes (subsumidas) por essa primeira poderão ser erroneamente classificadas com suspeitas levando a falsos-positivos, ou seja, serão classificadas como suspeitas, quando na realidade não o são.

A relação entre os elementos de um espectro e a respectiva suspeição calculada pelas ferramentas de localização de defeitos baseada em espectros é direta. De acordo com Agrawal et al. (AGRAWAL; IMIELI'NSKI; ARUN, 1993), uma regra de associação Y→Z é dita falaciosa se existem outras duas regras: X→Y e Y→Z tal que a probabilidade P(Y|X,Z) = P(Y|X). Neste caso, a informação de que Y→Z está relacionada apenas ao fato de que X implica tanto Y como Z, não sendo uma informação relevante para explicar a associação Y→Z (DENMAT; DUCASSé; RIDOUX, 2005).

1.2 Justificativa

As métricas de associação utilizadas em técnicas de localização de defeitos baseada em espectro partem do princípio de que cada elemento dos espectros — seja ele uma linha, seja um ramo, seja uma associação definição uso — são independentes (DENMAT; DUCASSé; RIDOUX, 2005). Porém, os elementos de espectro não são independentes; eles possuem uma relaçao de subsunção cujo efeito pode influenciar a identificação dos trechos de programa mais suspeitos quando técnicas localização de defeitos baseada em espectros e métricas de associações são utilizadas.

1.3 Hipótese de Pesquisa

A hipótese de pesquisa desta dissertação é de que a relação de subsunção entre elementos de espectro de um programa pode ser explorada para produzir resultados mais efetivos na localização de defeitos por meio da redução do número de linhas a serem investigadas sem comprometer a capacidade de localização de defeitos. O uso da relação de subsunção para restringir o subconjunto de ADUs a serem utilizadas no cálculo das métricas, que seja do nosso conhecimento, ainda não investigado experimentalmente.

1.4 Objetivos

O objetivo geral desse projeto de pesquisa é explorar o efeito da relação de subsunção em técnicas de localização de defeitos baseadas em espectros. Mais especificamente, analisar como a relação de subsunção entre elementos do espectro de fluxo de dados (i.e., associações definição uso — ADUs) afeta o ranqueamento obtido por meio de métricas de associação.

Entre os objetivos específicos encontram-se:

- Investigar por meio de um experimento qual é o efeito — positivo ou negativo — das relações de subsunção entre ADUs no ranqueamento produzido pelas métricas de associação utilizadas para identificar trechos de código mais suspeitos de conter o defeito;
- Investigar por meio de um experimento se a relação de subsunção pode ser utilizada para diminuir o número de linhas a ser investigadas pelo desenvolvedor até a localização de defeitos.

1.5 Contribuições

Este trabalho apresenta uma avaliação experimental do impacto da relação de subsunção na localização de defeitos baseada em espectros de fluxo de dados, isto é, ADUs. O experimento consiste na comparação da eficácia na localização de defeitos das ADUs *não-limitadas* — o conjunto minimal de ADUs que garante a cobertura de demais ADUs — e todas as ADUs. No experimento, utilizaram-se um subconjunto dos programas do repositório *Defects4J* — um repositório de defeitos e arcabouço de execução de testes que permite a reprodução e o isolamento de defeitos reais encontrados em projetos de software —, espectros de fluxo de dados (ADUs) e a métrica de associação *Ochiai*. Os resultados do experimento indicam que o uso do subconjunto de ADUs não-limitadas é promissor na localização de defeitos, sobretudo quando são consideradas apenas as ADUs posicionadas nos primeiros *rankings*. Além disso, o número de linhas a serem inspecionadas pelo programador é reduzido.

1.6 Organização

Esse primeiro capítulo apresentou a introdução, objetivos, motivação, hipótese e justificativa desta pesquisa.

Os demais capítulos do trabalho estão organizados desta forma:

- **Capítulo 2**: Apresentação da fundamentação teórica sobre os conceitos básicos de depuração, técnicas de localização de defeitos baseadas em espectros do programa e subsunção de espectros de cobertura.
- **Capítulo 3**: Revisão bibliográfica apresentando trabalhos relacionados com a pesquisa.
- **Capítulo 4**: Apresentação das ferramentas Jaguar e SATool, bem como o repositório *Defects4j*, utilizados no experimento de avaliação das relações de subsunção entre elementos dos espectros de fluxo de dados.
- **Capítulo 5**: Apresentação do projeto experimental e dos resultados do experimento realizado para avaliar o impacto da relação de subsunção de fluxo de dados na localização de defeitos.
- **Capítulo 6**: Apresentação do resumo dos resultados do experimento, contribuições da pesquisa realizada e trabalhos futuros.

2 Fundamentação Teórica

Este capítulo descreve conceitos gerais de depuração de software bem como aqueles específicos da técnica de depuração que será estudada, a saber, localização de defeitos baseada em espectros de código. Inicialmente, os conceitos de defeito, infecção e falha são definidos (Seção 2.1). Em seguida, são apresentados os diferentes espectros de cobertura de código utilizados para fins de depuração (Seção 2.2). Por fim, são discutidas a técnica de depuração baseada em espectros de cobertura de código (Seção 2.3) e a subsunção de espectros (Seção 2.4).

2.1 Defeito, infecção e falha

De acordo com Zeller (2005), os termos defeito, infecção e falha são distintos. Esta distinção é caracterizada abaixo.

2.1.1 Defeito

Um defeito é um trecho incorreto do código que pode causar uma infecção. O defeito pode ser causado pela falta de conhecimento do programador sobre os requisitos ou a tecnologia empregada, um estado do programa não previsto pelos requisitos originais, incompatibilidade de interface entre dois módulos ou uma inesperada integração de diversos componentes.

Um defeito pode ser atingido durante a execução de um caso de teste, mas nem sempre ele causa uma infecção e uma subsequente falha. Alguns defeitos apenas ocasionam uma infecção se algumas condições são satisfeitas.

Para ilustrar os conceitos, foi elaborado o programa Max (CHAIM; ARAUJO, 2013) apresentado na Tabela 1. Esse programa possui um único método, denominado *max*, que recebe como parâmetros um vetor de inteiros e o seu respectivo tamanho. O retorno desse método é o maior inteiro contido no vetor de inteiros. Um conjunto de entradas válidas para esse programa poderia ser:

- Vetor de Inteiros: $\{0, 9, 7, 4, 2\}$
- Tamanho do vetor: 5

Nesse exemplo, o retorno do método seria o inteiro 9.

Tabela 1 – Programa Max

Linha	Código
1	package br.usp.each.sfl;
2	
3	public class Max {
4	int max(int array [], int length)
5	{
6	int i = 0;
7	int max = array[++i]; // Correto: array[i++]
8	while (i < length)
9	{
10	if (array[i] > max)
11	max = array[i];
12	i = i + 1;
13	}
14	System.out.println(max);
15	return max;
16	}
17	}

Fonte: Dennis Lopes, 2022

Nesse código foi inserido um defeito na linha 7 onde, ao invés de se realizar o pós-incremento na variável i, é realizado o pré-incremento. Isso faz com que o valor da primeira posição do vetor de inteiros não seja avaliada e, com isso, o defeito irá se manifestar sempre que a primeira posição do vetor contiver o maior número do vetor. Nesse exemplo ilustrativo estamos simulando defeitos que podem ocorrer por falta de atenção do programador ao lidar com as condições lógicas dos seus algoritmos.

2.1.2 Infecção ou erro

Uma infecção ou erro ocorre quando um estado de programa não previsto é atingido quando o programa é executado sob certas condições. Uma infecção pode causar mais infecções em partes de código sem defeitos sendo que essa infecção pode ou não revelar o defeito.

No exemplo, ilustrado por meio do programa Max, a infecção ocorre sempre que o programa Max inicia a sua checagem em busca do maior inteiro do vetor a partir da sua segunda posição (ao invés de percorrer o vetor desde a sua primeira posição). Neste caso, a infecção sempre irá ocorrer na execução do programa ainda que se possa perceber, pela análise do algoritmo, que uma falha irá se manifestar se, e somente se, o inteiro contido na primeira posição do vetor for o maior inteiro deste vetor. Interessante observar que a infecção pode ocorrer mesmo que não produza uma falha observável no programa.

2.1.3 Falha

A falha é uma infecção ou erro observável externamente. Uma infecção propaga-se e gera um comportamento anormal do sistema. Uma falha é visível aos usuários finais, como uma mensagem de erro ou saídas incorretas.

No programa Max, a falha ocorre quando o inteiro que ocupa a primeira posição do vetor é de fato o maior inteiro do vetor. Neste caso, Max apresentará como saída o segundo maior elemento do vetor.

2.2 Cobertura de código

A cobertura de código pode ser utilizada como uma métrica de qualidade de um conjunto de testes e quanto maior essa cobertura, menores são as chances de um defeito não ser detectado, embora essa métrica isolada não possa garantir ausência de defeitos no código. Se defeitos não causam infecções (erros) que provocam falhas, todos os casos de teste irão passar. De acordo com Dijkstra, os testes podem apenas mostrar a presença de defeitos, mas nunca sua ausência (ZELLER, 2005).

O termo cobertura de código, chamada a partir de agora de apenas cobertura, refere-se a informações extraídas de execuções do programa que indicam quais elementos do código foram exercitados. Esses componentes podem ser comandos (YOU; HUANG; PENG; HSU, 2013), nós ou blocos de comandos, fatias (MAO; LEI; DAY; QI; WANG, 2014) e associações de definição uso (CHAIM; MALDONADO; JINO, 2003), por exemplo.

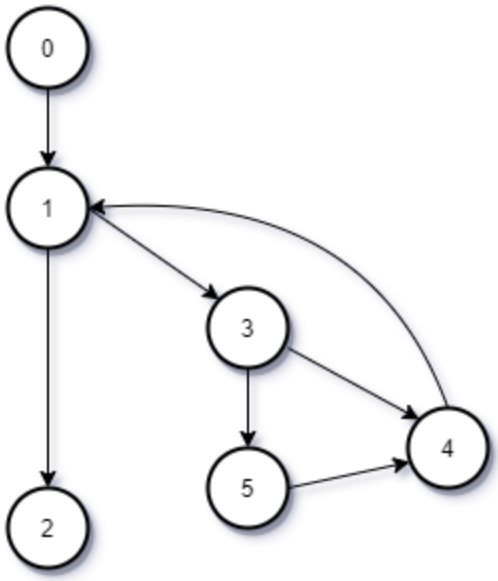
O programa precisa ser instrumentado para que a cobertura seja coletada durante a execução de casos de teste (ARAUJO; CHAIM, 2014). Essa etapa consiste em adicionar

código extra para rastrear cada componente, guardando a informação de quando ele é executado ou não. Portanto, a informação de cobertura é coletada durante a execução do conjunto de teste.

2.2.1 Cobertura de fluxo de controle

A informação de fluxo de controle de um programa é representada por meio de um grafo de fluxo de controle (GFC), como o apresentado na Figura 1, que é uma representação de possíveis sequências de execução de um programa. Os nós ou vértices desse grafo correspondem a comandos que são executados sempre em sequência, e os ramos representam os possíveis fluxos de execução que o programa pode seguir (DELAMARO; CHAIM; VINCENZI, 2010). O mapeamento das linhas e nós do programa Max pode ser observado na Tabela 2. É importante ressaltar que no grafo de fluxo de controle cada nó representa uma ou mais instruções e, portanto, uma ou mais linhas de código.

Figura 1 – Grafo de fluxo de controle do programa Max



Fonte: Dennis Lopes, 2021

Formalmente, o GFC de um programa P é definido por uma quádrupla $G(N, E, e, s)$ onde N é o conjunto de nós, E é o conjunto de ramos, e é o nó de entrada e s é o nó de saída. Um caminho é uma sequência de nós $(n_i, ..., n_k, n_{k+1}, ...n_j)$, onde $i \leq k < j$, e $(n_k, n_{k+1}) \in E$ (CHAIM; ARAUJO, 2013). Um caminho $(n_i, ..., n_k, n_{k+1}, ...n_j)$ é *completo* se $n_i = e$, isto é, n_i é o nó de entrada, e $n_j = s$, ou seja, n_j é o nó de saída. A Tabela 2

Tabela 2 – Linhas e nós do programa Max

Linha	Nó	Código
1	-	package br.usp.each.sfl;
2	-	
3	-	public class Max {
4	0	int max(int array [], int length)
5	0	{
6	0	int i = 0;
7	0	int max = array[++i];
8	1	while (i < length)
9	1	{
10	3	if (array[i] > max)
11	5	max = array[i];
12	4	i = i + 1;
13	4	}
14	2	System.out.println(max);
15	2	return max;
16	-	}
17	-	}

Fonte: Dennis Lopes, 2022

lista o código e apresenta a associação entre as linhas e os nós no grafo de fluxo de controle para o programa Max.

Um nó (ramo) é considerado coberto se há um caso de teste que percorre um caminho que inclui o nó (ramo). Dois critérios de teste, *todos-nós* e t*odos-ramos* (DELAMARO; CHAIM; VINCENZI, 2010), requerem que todos os nós e todos os ramos do programa, respectivamente, sejam cobertos por pelo menos um caso de teste.

A Tabela 3 lista os requisitos de teste para os critérios todos-nós e todos-ramos. O espectro de cobertura de nós e ramos obtido da execução dos casos de teste pode ser usado para localizar defeitos.

Tabela 3 – Cobertura de fluxo de controle do Programa Max

Todos-nós	Todos-ramos
0	(0,1)
1	(1,2) (1,3)
2	-
3	(3,4) (3,5)
4	(4,1)
5	(5,4)

Fonte: Dennis Lopes, 2021

2.2.2 Cobertura de fluxo de dados

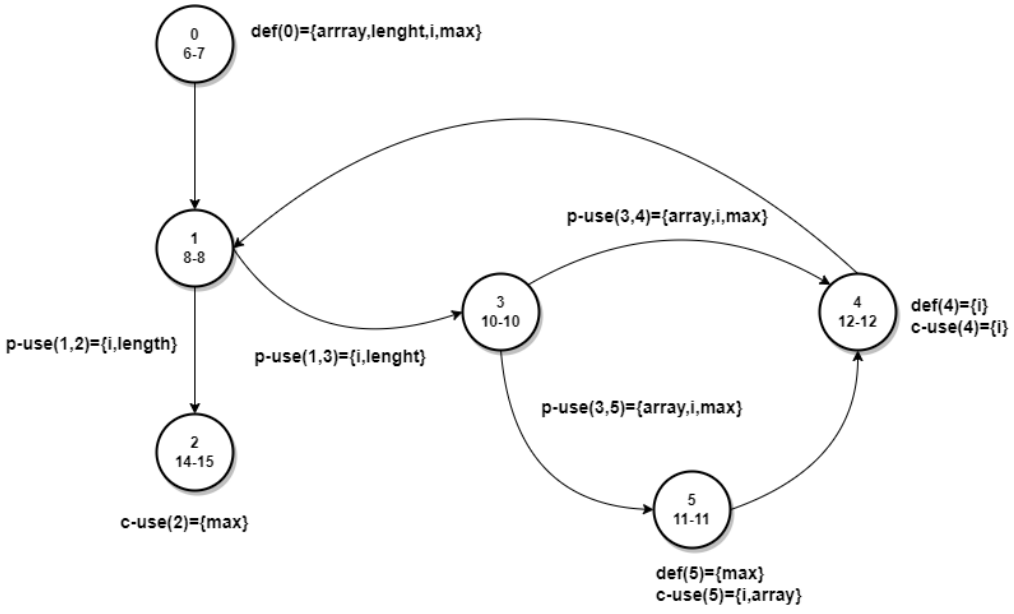
A cobertura de fluxo de dados baseia-se nas definições e usos das variáveis de um programa. Uma variável é definida quando recebe um valor que é subsequentemente referenciado, isto é, usado, em outros pontos do programa (DELAMARO; CHAIM; VINCENZI, 2010). Um uso de variável ocorre quando ela é referenciada e isso pode acontecer de duas maneiras:

- Uso predicativo (ou p-uso), quando esse uso ocorre na condição de um comando de controle de fluxo do programa (e.g., while, for, if);
- Uso computacional (ou c-uso), quando o uso ocorre em uma computação.

O uso predicativo é associado aos ramos do grafo e o uso computacional é associado aos nós ou vértices de um grafo de fluxo de dados conforme apresentado na Figura 3. Um caminho livre de definição com respeito à variável x é um caminho $(n_i, ..., n_k, n_{k+1}, ..., n_j)$, onde $i \leq k < j$, de modo que x não é redefinido, exceto possivelmente no último nó.

A Figura 2 apresenta o grafo de fluxo de controle (GFC) do programa Max. Para facilitar o entendimento, cada nó possui duas informações: o número de identificação atribuído ao nó e o intervalo de linhas do programa associadas a esse nó. Associado a cada nó temos ainda a informação de quais variáveis são definidas (def) e quais delas são usadas (use).

Figura 2 – Grafo de fluxo de dados anotado do programa Max



Fonte: Dennis Lopes, 2021

Uma associação definição uso (ADU) (i, j, x) ocorre quando a definição da variável x ocorre no nó i e um c-uso ocorre no nó j e há um caminho livre de definição com relação a x de i a j. De forma semelhante, a tripla (i, (j, k), x) representa uma associação de fluxo de dados na qual a definição de x ocorre no nó i e um p-uso na aresta (j,k). Além disso, há um caminho (i, ..., j, k) que é um livre de definição com relação a x.

O critério de teste estrutural todos-usos (RAPPS; WEYUKER, 1985) estabelece que um conjunto de teste adequado a ele inclui casos de testes que exercitam ao menos uma vez as associações definição uso presentes no programa. Tabela 4 lista os requisitos de teste para o critério todos-usos. Observe-se que na Tabela 4 as associações definição uso do programa Max estão associadas a números de linhas e não aos nós do grafo de fluxo de controle. As associações definição uso cobertas por casos de teste — isto é, o espectro de fluxo de dados — podem ser usadas para localizar defeitos e identificar as variáveis do programa associadas a eles.

Tabela 4 – Requisitos de teste de fluxo de dados do programa Max

(6,(10,11),array)	(7,(10,11),i)	(11,(10,11),max)	(12,12,i)
(6,(10,12),array)	(7,(10,12),i)	(11,(10,12),max)	(12,11,i)
(6,11,array)	(7,12,i)	(11,14,max)	
(6,(8,14),length)	(7,11,i)	(12,(8,14),i)	
(6,(8,10),length)	(7,(10,11),max)	(12,(8,10),i)	
(7,(8,14),i)	(7,(10,12),max)	(12,(10,11),i)	
(7,(8,10),i)	(7,14,max)	(12,(10,12),i)	

Fonte: Dennis Lopes, 2021

2.3 Localização de defeitos baseada em espectros de cobertura de código

A técnica de localização de defeitos baseada em espectros (em inglês, *Spectrum-based Fault Localization* — SFL) é uma técnica que está sendo muito estudada (SOUZA; CHAIM; KON, 2016). SFL possui custo relativamente baixo de execução porque utiliza espectros, isto é, dados sobre a cobertura (ou não) de componentes do código por casos de teste automatizados; esses dados são usados para verificar quais componentes (linhas, blocos ou associações de definição uso) possui potencial de conter um defeito. Esse potencial é calculado a partir de métricas de associação, como Ochiai (ABREU; ZOETEWEIJ; GEMUND, 2007) e Tarantula (JONES; HARROLD; STASKO, 2002a).

Essas métricas são baseadas em coeficientes que levam em consideração os componentes que foram, ou não foram, executados por casos de testes que passam ou que falham. Quanto mais um componente for executado por casos de teste que falham, maior a probabilidade desse componente conter um defeito. As métricas também consideram que quando um componente não é executado por casos de teste que falham, ele tem menor probabilidade de conter um defeito. De forma contrária, um componente que não é executado por casos de teste que passam tem maior chance de conter um defeito, desde que sejam executados por casos de teste que falham.

Observe-se que essa verificação é baseada nos testes de cada método individualmente, o que em programas orientados a objetos é chamado de teste intra-método (HARROLD; ROTHERMEL, 1994)

A Tabela 5 apresenta os coeficientes de cada componente j. Eles indicam o número de vezes que o componente j foi executado por um teste que passou (c_{ep}), que foi executado por um teste que falhou (c_{ef}), que não foi executado por um teste que passou (c_{np}) e que não foi executado por um teste que falhou (c_{nf}).

Tabela 5 – Coeficientes do componente j

	Teste que falha	Teste que passa
Executado j	$c_{ef}(j)$	$c_{ep}(j)$
Não executado j	$c_{nf}(j)$	$c_{np}(j)$

Fonte: Ribeiro (2016)

Para ilustrar o uso da técnica SFL, será utilizado o programa Max, apresentado na Tabela 1. A Tabela 6 descreve os resultados esperados, os resultados obtidos e o veredito, sucesso ou falha, dos casos de teste. O código da classe de teste, bem como os parâmetros para cada um dos testes é exibido na Tabela 7

Os casos de teste listados na Tabela 6 foram executados e a cobertura das linhas executadas para cada caso de teste coletada. Os coeficientes descritos na Tabela 3 associados a cada linha foram obtidos com os dados de cobertura.

As métricas *Ochiai* e *Tarantula* foram então calculadas utilizando as fórmulas seguir:

Tabela 6 – Resultados execução dos casos de testes para o programa Max

Tn	Caso de Teste	Resultado Esperado	Resultado Obtido	Veredito
t1	max([1,2,3],3)	3	3	✓
t2	max([5,5,6],3)	6	6	✓
t3	max([2,1,10],3)	10	10	✓
t4	max([4,3,2],3)	4	3	✗
t5	max([4],1)	4	Exceção	✗

Fonte: Dennis Lopes, 2022

$$Ochiai\ H_{\mathrm{O}} = \frac{c_{\mathrm{ef}}}{\sqrt{(c_{\mathrm{ef}}+c_{\mathrm{nf}})\cdot(c_{\mathrm{ef}}+c_{\mathrm{ep}})}}$$

$$Tarantula\ H_{\mathrm{T}} = \frac{\frac{c_{\mathrm{ef}}}{c_{\mathrm{ef}}+c_{\mathrm{nf}}}}{\frac{c_{\mathrm{ef}}}{c_{\mathrm{ef}}+c_{\mathrm{nf}}}+\frac{c_{\mathrm{ep}}}{c_{\mathrm{ep}}+c_{\mathrm{np}}}}$$

onde H_O é o valor de suspeição atribuído pela métrica *Ochiai* e H_T o valor de suspeição atríbuido pela métrica *Tarantula*. Ambos os valores foram calculados a partir dos coeficientes c_{np}, c_{ep}, c_{nf} e c_{ef} da Tabela 5 para cada linha do programa.

O resultado da aplicação das métricas para a cobertura de código baseada em fluxo de controle, no programa Max, pode ser visto na Tabela 8. A primeira e segunda coluna representam, respectivamente, a linha e o nó do programa e as cinco colunas posteriores representam a execução dos casos de teste. Nelas, os componentes que foram cobertos pelos respectivos casos de teste são marcados com uma esfera preenchida (●) e os não cobertos com a esfera não preenchida (❍). As quatro colunas seguintes (c_{np}, c_{ep}, c_{nf} e c_{ef}) apresentam os coeficientes apresentados na tabela 5 de acordo com a execução do conjunto de testes. Por fim, as duas últimas colunas apresentam o valor de suspeição do componente de acordo com as métricas Ochiai (H_o) e Tarantula (T_t). A última linha da tabela indica o resultado dos casos de teste e verifica-se que os testes t_4 e t_5 falharam.

De maneira similar, as métricas *Ochiai* e *Tarantula* foram aplicadas para a cobertura de código baseada em fluxo de dados e os resultados são apresentados na Tabela 9.

Critérios de cobertura baseados em fluxo de dados são geralmente mais difíceis de satisfazer do que os critérios baseados em fluxo de controle embora ambos tenham o

Tabela 7 – Programa MaxTest

Linha	Código
1	package br.usp.each.sfl;
2	import static org.junit.Assert.*;
3	import org.junit.Test;
4	
5	public class MaxTest {
6	
7	Max max = new Max();
8	int result;
9	
10	@Test
11	public void t1() {
12	int[] array = new int[]1, 2, 3;
13	assertEquals(3, max.max(array, 3));
14	}
15	@Test
16	public void t2() {
17	int[] array = new int[]5, 5, 6;
18	assertEquals(6, max.max(array, 3));
19	}
20	@Test
21	public void t3() {
22	int[] array = new int[]2, 1, 10;
23	assertEquals(10, max.max(array, 3));
24	}
25	@Test
26	public void t4() {
27	int[] array = new int[]4, 3, 2;
28	assertEquals(4, max.max(array, 3));
29	}
30	@Test
31	public void t5() {
32	int[] array = new int[]4;
33	assertEquals(4, max.max(array, 3));
34	}
35	}

Fonte: Dennis Lopes, 2022

Tabela 8 – Cobertura do Programa Max com as métricas *Ochiai* e *Tarantula*

Linha	Nó	t_1	t_2	t_3	t_4	t_5	C_{np}	C_{ep}	C_{nf}	C_{ef}	H_0	H_T
6	0	●	●	●	●	●	0	3	0	2	0.63	0.5
7	0	●	●	●	●	●	0	3	0	2	0.63	0.5
8	1	●	●	●	●	❍	0	3	1	1	0.35	0.33
9	-	-	-	-	-	-	-	-	-	-	-	-
10	3	●	●	●	●	❍	0	3	1	1	0.35	0.33
11	5	●	●	●	❍	❍	0	3	2	0	0.0	0.0
12	4	●	●	●	●	❍	0	3	1	1	0.35	0.33
13	-	-	-	-	-	-	-	-	-	-	-	-
14	2	●	●	●	●	❍	0	3	1	1	0.35	0.33
15	2	●	●	●	●	❍	0	3	1	1	0.35	0.33
Resultado		✓	✓	✓	✗	✗						

Fonte: Dennis Lopes, 2022

mesmo objetivo, que é o de ajudar na seleção de alguns dos muitos caminhos possíveis de um programa. Traduzindo em termos de detecção de defeitos, os critérios de cobertura baseados em fluxo de dados possuem maior probabilidade de revelar defeitos do que os critérios baseados exclusivamente no fluxo de controle (MATHUR, 2013).

Isso pode ser observado no caso concreto do programa Max: as métricas obtidas a partir da cobertura baseada em fluxo de controle foram úteis para a localização do defeitos já que a linha que apresenta o defeito teve o maior *score* para as duas métricas calculadas (0.63 para *Ochiai* e 0.5 para a *Tarantula*). No caso do uso das associações definição uso, houve um empate no *score* obtido por duas ADUs – (7,14,max) e (12,(10,12),i) –, observe-se que a primeira ADU localiza a linha com o defeito e a segunda ADUs revela a variável afetada (variável i).

Em muitos casos, entretanto, pode-se observar que os efeitos do defeito são sentidos *a posteriori* no caso de teste, ou seja, tendo sido o defeito injetado em uma linha qualquer, essa infecção só é detectada pelas métricas a partir do momento em que a infecção afeta o resultado esperado do teste. Nesse caso, para localizar a origem do defeito, o desenvolvedor deve retroceder um pouco a partir da linha onde o defeito foi revelado, possivelmente, analisando a pilha de execução.

Esses blocos de código que compreendem o caminho desde a instrução origem do código defeituoso até onde o defeito é revelado contribuem tanto para a frequência dos casos de teste que passam quanto para a frequência dos casos de testes que falham e, portanto, constituem-se ruídos inerentes à forma como os coeficientes são calculados.

Tabela 9 – Cobertura do Programa Max com as métricas *Ochiai* e *Tarantula*

ADU	t_1	t_2	t_3	t_4	t_5	C_{np}	C_{ep}	C_{nf}	C_{ef}	H_0	H_T
(6,(10,11),array)	●	●	●	❍	❍	0	3	2	0	0	0
(6,(10,12),array)	●	●	●	●	❍	0	3	1	1	0.35	0.33
(6,11,array)	●	●	●	❍	❍	0	3	2	0	0	0
(6,(8,14),length)	●	●	●	●	❍	0	3	1	1	0.35	0.33
(6,(8,10),length)	●	●	●	●	❍	0	3	1	1	0.35	0.33
(7,(8,14),i)	❍	❍	❍	❍	❍	3	0	2	0	0	0
(7,(8,10),i)	●	●	●	●	❍	0	3	1	1	0.35	0.33
(7,(10,11),i)	❍	❍	❍	❍	❍	3	0	2	0	0	0
(7,(10,12),i)	●	●	●	●	❍	0	3	1	1	0.35	0.33
(7,12,i)	●	●	●	●	❍	0	3	1	1	0.35	0.33
(7,11,i)	❍	❍	❍	❍	❍	3	0	2	0	0	0
(7,(10,11),max)	●	●	●	❍	❍	0	3	2	0	0	0
(7,(10,12),max)	●	●	●	●	❍	0	3	1	1	0.35	0.33
(7,14,max)	❍	❍	❍	●	❍	3	0	1	1	0.71	1
(11,(10,11),max)	❍	❍	❍	❍	❍	3	0	2	0	0	0
(11,(10,12),max)	❍	❍	❍	❍	❍	3	0	2	0	0	0
(11,14,max)	●	●	●	❍	❍	0	3	2	0	0	0
(12,(8,14),i)	●	●	●	●	❍	0	3	1	1	0.35	0.33
(12,(8,10),i)	●	●	●	●	❍	0	3	1	1	0.35	0.33
(12,(10,11),i)	●	●	●	❍	❍	0	3	2	0	0	0
(12,(10,12),i)	❍	❍	❍	●	❍	3	0	1	1	0.71	1
(12,12,i)	●	●	●	●	❍	0	3	1	1	0.35	0.33
(12,11,i)	●	●	●	❍	❍	0	3	2	0	0	0
Resultado	✓	✓	✓	✗	✗						

Fonte: Dennis Lopes, 2022

No caso da cobertura por fluxo de dados, há ainda um número maior de caminhos, já que os caminhos são associados aos pares definição-uso, isto é, ADUs, de cada uma das variáveis do programa. No entanto, os caminhos determinados por cada uma das ADUs podem se sobrepor. Essa sobreposição de caminhos entre ADUs determina a relação de subsunção entre ADUs que pode levar a ruídos nos *rankings* gerados pelas métricas de associação (e.g., Ochiai e Tarantula).

2.4 Subsunção

Os critérios de cobertura são utilizados para a criação de requisitos de teste a partir de uma modelo abstrato do software (AMMANN; OFFUTT, 2017). Esses requisitos de teste especificam caminhos do programa que devem ser cobertos por meio de um teste.

Ao executarmos um teste, um caminho no grafo do programa é exercitado e, como consequência, determinados requisitos de teste são cobertos. No entanto, em determinadas condições, alguns requisitos são sempre cobertos quando um outro requisito de teste o é. A intuição é que existe um ordenamento entre os requisitos de teste, sendo alguns mais fáceis de serem cobertos do que outros. Essa relação entre os requisitos de teste é denominada *subsunção* e será formalmente definida nas próximas seções.

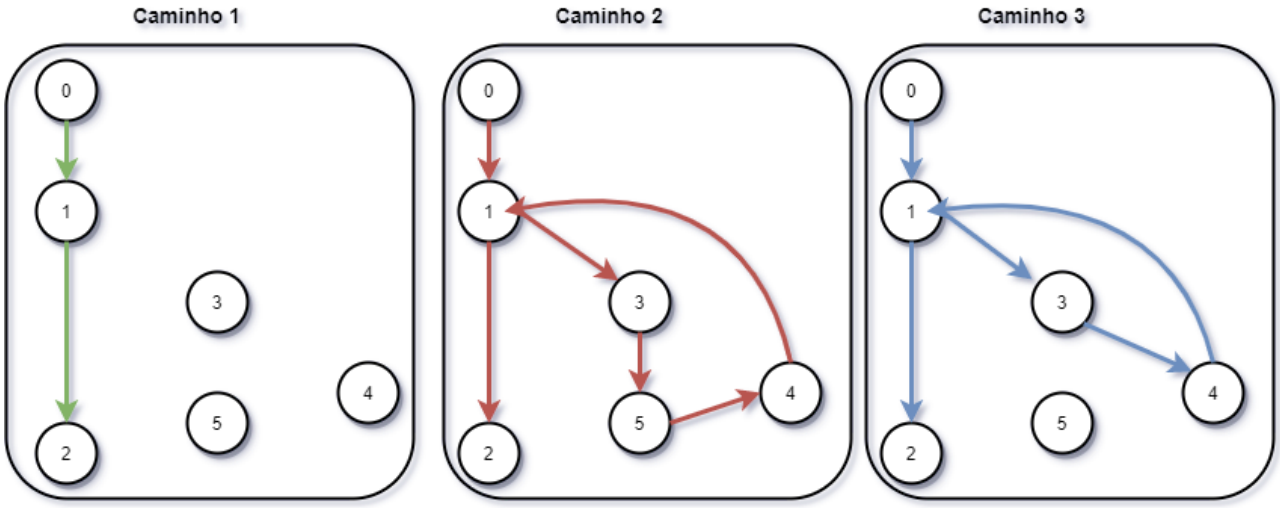
2.4.1 Subsunção de nós

A partir do grafo apresentado na Figura 1, podemos determinar caminhos completos para o programa Max. Como o grafo possui um ciclo, sem que haja limitação para a compreensão do conceito de subsunção, limitamos a um o número de vezes em que o ciclo é exercitado em um caminho completo. A partir dessa premissa, obtivemos os seguintes caminhos completos:

Caminho 1: $0 \rightarrow 1 \rightarrow 2$
Caminho 2: $0 \rightarrow 1 \rightarrow 3 \rightarrow 5 \rightarrow 4 \rightarrow 1 \rightarrow 2$
Caminho 3: $0 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 1 \rightarrow 2$

Esses caminhos estão representados na Figura 3. O caminho 1 é representado em verde no primeiro grafo, o caminho 2 em vermelho no segundo e o caminho 3 no terceiro grafo em azul.

Figura 3 – Caminhos completos do programa Max



Fonte: Dennis Lopes, 2021

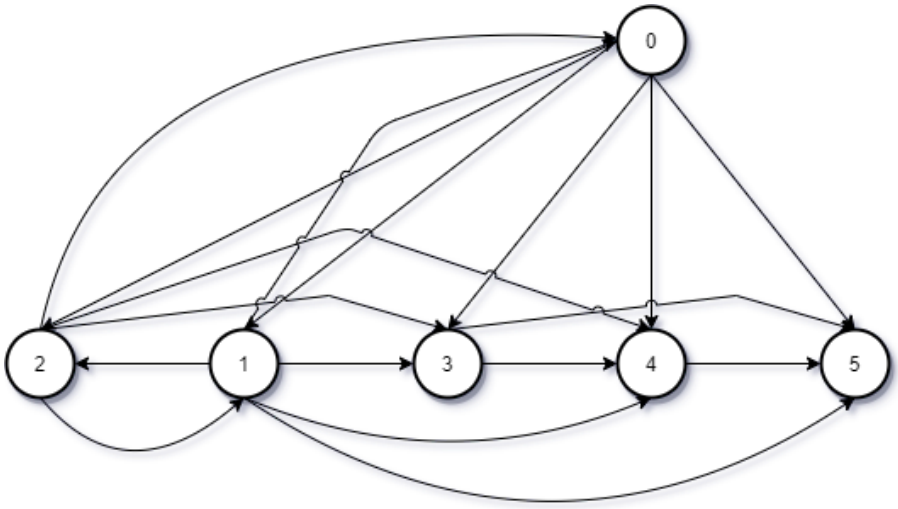
Analisando individualmente os nós do grafo podemos dizer, por exemplo, que a execução do nó 1, por meio de um caminho completo, garante também a execução do nó 0 e do nó 2. Observe-se, porém, que a execução do nó 1 não garante de maneira definitiva a execução dos nós 3, 4 ou 5.

Outro exemplo é a execução do nó 5 por meio de um caminho completo: observe-se que a sua execução por meio de um caminho completo garante a execução de todos os outros nós (0, 1, 2, 3, 4). Essa relação entre os nós é denominada *subsunção* de nós e possui a seguinte definição formal:

Definição 1 *Um nó n_1 é subsumido por um nó n_2 se, e somente se, todo caminho completo que incluir n_1 também incluir n_2. Essa relação de subsunção entre os nós n_1 e n_2 é representada da seguinte maneira: $n_1 \rightarrow n_2$ (n_1 é subsumido por n_2 ou n_2 subsume n_1).*

A Figura 4 apresenta as relações de subsunção entre os nós do programa Max.

Figura 4 – Subsunção de nós do programa Max



Fonte: Dennis Lopes, 2022

Utilizando o conceito de conexão forte entre nós de um grafo (que diz que um nó s é fortemente ligado a um nó t se existe um caminho de s a t e também um caminho de t a s) podemos simplificar esse grafo de subsunção agrupando em um único nó os vértices fortemente ligados. Esse grafo simplificado de subsunção pode ser visto na Figura 5.

Figura 5 – Subsunção de nós simplificado do programa Max

Fonte: Dennis Lopes, 2022

Considerando se individualmente cada requisito de teste do grafo de subsunção simplificado, ou entidades conforme originalmente estabelecido por Marré e Bertolino (2003), definimos como *conjunto de abrangência* (*spanning sets*) um subconjunto mínimo de requisitos de teste que, ao serem cobertos por um conjunto de teste, garantem a cobertura de todos os requisitos de teste do programa. No grafo de subsunção esses elementos estão

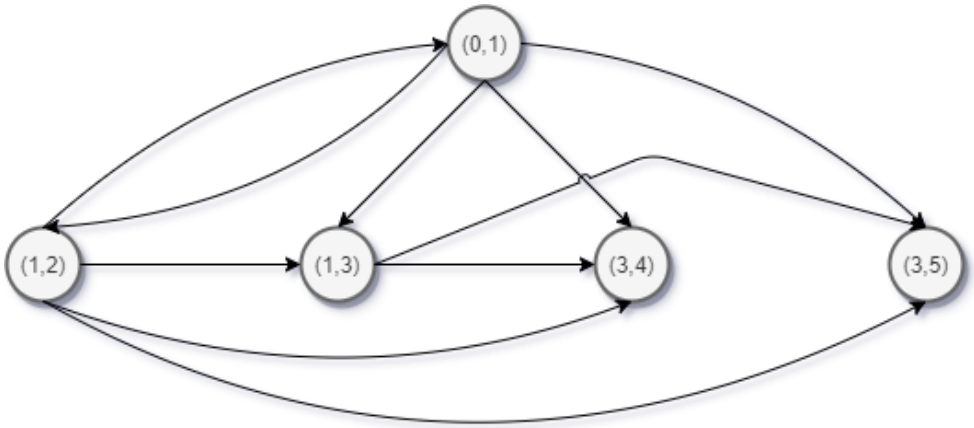
contidos nos nós finais (ou folhas) representados na Figura 5 como retângulos. Os requisitos contidos nos conjuntos de abrangência recebem também a denominação de elementos *não-limitados* (*uncontrained*) (MARRé; BERTOLINO, 2003). Note-se que para o programa Max o nó 5 é não-limitado pois, cobrindo-se este nó, todos os demais são cobertos, desde que a execução do programa não seja interrompida, por uma exceção ou pelo fim da execução (e.g., *exit*), depois da execução do nó 5.

2.4.2 Subsunção de ramos

O conceito de subsunção, aplicado anteriormente para nós, também pode ser aplicado para ramos de um grafo que representa um programa.

A partir do grafo do programa Max, podemos criar um grafo que representea a relação de subsunção para os ramos que estão associados a comando condicionais. Nesse grafo, apresentado na Figura 6, cada nó representa uma aresta do grafo do programa.

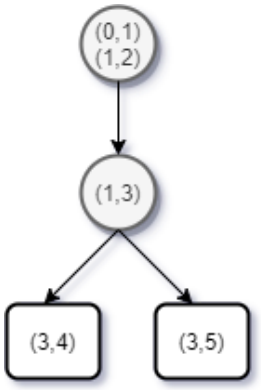
Figura 6 – Subsunção de ramos do programa Max



Fonte: Dennis Lopes, 2021

Utilizando-se do conceito de conexão forte entre nós podemos novamente simplificar o grafo de subsunção como apresentado na Figura 7. Nesse caso, o conjunto de folhas ((3,4), (3,5)) são ramos não-limitados e compõem o *conjunto de abrangência*:

Figura 7 – Subsunção de ramos simplificado do programa Max

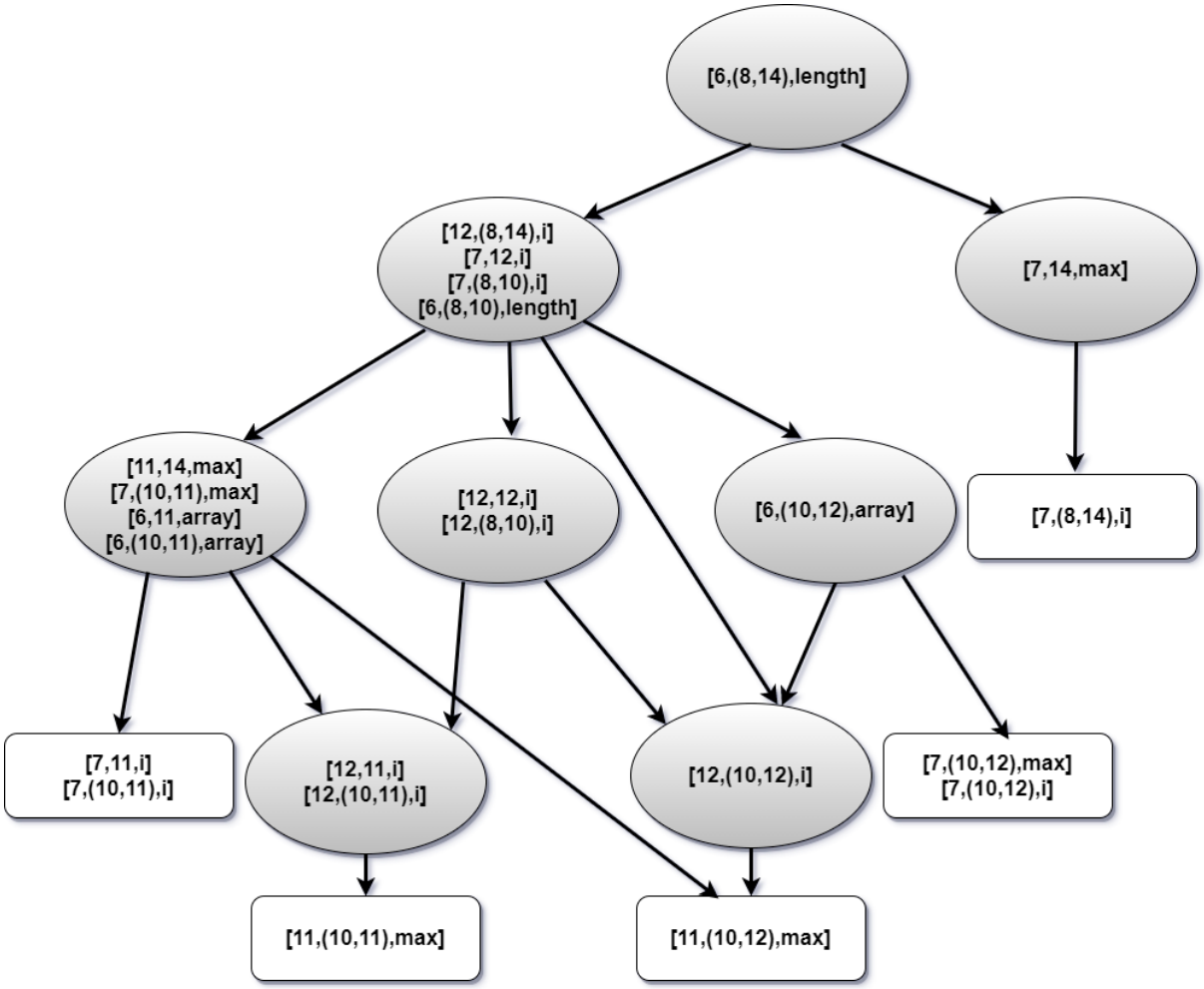


Fonte: Dennis Lopes, 2022

2.4.3 Subsunção de associações definição uso

Marré e Bertolino (1996) também introduziram o conceito da subsunção entre associações definição uso (ADUs). Além disso, as autoras apresentaram um algoritmo para determinar a relação subsunção entre as ADUs de um programa (MARRé; BERTOLINO, 2003). O grafo da relação de subsunção simplificado para as ADUs no programa Max é apresentada na Figura 8.

Figura 8 – Subsunção de ADUs do programa Max



Fonte: Chaim et al., 2022

A relação de subsunção entre ADUs é tal que uma ADU D_1 subsume outra ADU D_2 se todo teste que cobrir D_1 também cobrir D_2. A relação é definida formalmente a seguir (CHAIM; BARAL; OFFUTT; CONCILIO; ARAúJO, 2021).

Definição 2 *$D_1(d_1, u_1, X_1)$ subsume $D_1(d_2, u_2, X_2)$ ($D_2 \rightarrow D_1$), se em todo caminho completo que contém um caminho livre de definição com relação à variável X_1 entre d_1 e u_1 também contém um caminho livre de definição com relação à variável X_2 entre d_2 e u_2.*

Pela observação da Figura 8 podemos depreender que o *conjunto de abrangência* de ADUs é composto por cinco elementos, isto é, uma ADU para cada folha. Esses cinco elementos garantem a cobertura completa de todas as ADUs do programa Max. As folhas do grafo contêm as ADUs não-limitadas; no entanto, como há duas folhas que possuem duas ADUs, existem quatro possíveis conjuntos de abrangência. Isso porque basta escolher

uma ADU das folhas com duas para o conjunto de abrangência. Uma da possibilidades seria o conjunto {(7,11,i), (11,(10,11),max), (11,(10,12),max), (7,(10,12),max), (7,(8,14),i)}.

2.5 Considerações finais

Como visto anteriormente, as técnicas de localização de defeitos baseadas em espectros utilizam um *ranking* que é baseado na quantidade de vezes em que determinada instrução, dentro de um espectro, passa ou falha em um conjunto de testes. Com base nessa informação diferentes métricas calculam a probabilidade de uma determinada instrução ser a responsável pelo defeito permitindo assim que o desenvolvedor priorize as linhas de código que serão alvo de suas primeiras análises com o objetivo de encontrar e corrigir o defeito no menor tempo possível. Daniel e Sim (2013) sugerem que, em alguns casos, os espectros contêm informação ambígua, duplicada ou com ruídos que podem deteriorar o desempenho das métricas de associação.

Nesse trabalho, iremos verificar se, por meio da subsunção de espectros, em particular espectros de fluxo de dados, é possível eliminar informações duplicadas ou ambíguas que podem levar à deterioração das métricas de localização de defeitos. Também será avaliado se há degradação significativa dos resultados da localização de defeitos, ao se calcular as métricas de localização de defeitos, tendo como base apenas os testes que exercitam os caminhos que contém os requisitos de teste não-limitados (*unconstrained*), ou seja, os requisitos que fazem parte do conjunto de abrangência (*spanning tree*).

3 Revisão Bibliográfica

Neste capítulo, serão discutidas técnicas para redução de ruídos na localização de defeitos. Várias técnicas procuram "melhorar"os *rankings* gerados pelas métricas de associação das técncias de localização de defeitos baseadas em espectro (*Spectrum-based Fault Localization* – SBFL). Elas procuram apurar os resultados das métricas de associação adicionando, por exemplo, informações sobre dependência de controle e de dados (XIALON *et al.*, 2013; SOREMEKUN; KIRSCHNER; BöHME; ZELLER, 2021) ou adicionado mais casos de teste (CAMPOS; ABREU; FRASER; D'AMORIM, 2013). No entanto, pode-se entender que essas técnicas "melhoram" os resultados dos *rankings* ao reduzir o ruído associado aos espectros utilizados para localizar defeitos.

Por meio de uma busca não-sistemática obtivemos apenas um trabalho, (DANIEL; SIM, 2013), que utiliza o termo redução ruídos na localização de defeitos. Isso ocorre porque os espectros que podem levar a uma classificação menos eficaz não são vistos como *ruídos* a serem eliminados. Nesse sentido, essa pesquisa propõe um olhar diferente sobre as técnicas que procuram aumentar a precisão dos resultados da SBFL.

3.1 Seleção dos trabalhos

SBFL tem sido amplamente explorada há aproximadamente 20 anos. Vários aspectos foram estudados, por exemplo, as diferentes métricas de associação e sua eficácia (NAISH; LEE; RAMAMOHANARAO, 2010; MA; ZHANG; LU; WANG, 2014; XIE; CHEN; KUO; BAOWEN, 2013), a localização de defeitos em programas com múltiplos defeitos (WONG; GAO; LI; ABREU; WOTAWA, 2016), o impacto da correção coincidente em técnicas de SBFL, entre outros.

Uma linha de investigação tem sido o desenvolvimento de técnicas para aumentar a precisão das métricas de associação. Algumas abordagens utilizam informações de dependência de controle e de dados (XIALON *et al.*, 2013; SOREMEKUN; KIRSCHNER; BöHME; ZELLER, 2021) enquanto outras utilizam alterações nos conjuntos de casos de teste para calcular as métricas de associação (ROYCHOWDHURY; KHURSHID, 2012; DANIEL; SIM, 2013; CAMPOS; ABREU; FRASER; D'AMORIM, 2013). A seguir, discutiremos trabalhos que utilizam essas abordagens para redução de ruídos.

3.2 Redução de ruídos por meio de dependências de fluxo de dados e controle

Alguns autores como Soremekun, Kirschner, Böhme e Zeller (2021) e Xialon *et al.* (2013) argumentam que o fatiamento de programas pode melhorar as técnicas de localização de defeitos.

Xialon *et al.* (2013) argumentam que a eficiência e a efetividade da localização de defeitos diminui com o aumento dos defeitos e que a causa raiz desse problema é o impacto cumulativo das mesmas instruções (ou linhas de código) em defeitos distintos. Sua técnica utiliza as dependências no programa para gerar um *score* de suspeição de comandos com base no fatiamento estático dos testes que falharam e no fatiamento dinâmico dos testes que passaram.

Como citado no parágrafo anterior existem duas categorias de fatiamento de programas: *fatiamento estático* e *fatiamento dinâmico*. O primeiro extrai as dependências das entidades do programa sem a sua execução, enquanto que o segundo extrai as dependências em tempo de execução. O fatiamento estático tem uma grande complexidade de tempo produzindo falsos positivos, enquanto que o fatiamento dinâmico apresenta complexidade de espaço podendo gerar falsos negativos (XIALON *et al.*, 2013).

Xialon *et al.* (2013) misturam o fatiamento dinâmico e o fatiamento estático assumindo que, para os testes que passaram, o fatiamento dinâmico pode ser desconsiderado nos cálculos pois apenas as entidades que causam o defeito são o gatilhos para o defeito. Assim o fatiamento dinâmico é realizado apenas para o conjunto de instruções relacionadas com a saída incorreta do programa diminuindo a complexidade de espaço para o cálculo das dependências em tempo de execução. A partir de uma visão híbrida que é a intersecção do fatiamento dinâmico com o estático é calculada a suspeição de cada instrução gerando um relatório final. Seu método foi capaz de melhorar e efetividade da localização de defeitos significantemente, mas a um custo de tempo também significantemente maior. No estudo empírico o custo da técnica variou de 1.15 e 1.8 vezes o custo da técnica de SBFL tradicional baseada em espectros.

Com base nesses dados, Soremekun, Kirschner, Böhme e Zeller (2021) propõem uma abordagem que combina a localização de defeitos baseada em métricas de associação, com o fatiamento dinâmico realizado pelo desenvolvedor durante o processo de depuração manual. Nesse processo, o desenvolvedor restringe a sua atenção às partes relevantes para

a revelação do defeito presente no código. No experimento, o desenvolvedor inicialmente atua nas localidades com melhor *ranking* nas métricas de associação e então, após visitar um número N de localidades, muda para a estratégia de fatiamento dinâmico excluindo as localizações já exploradas anteriormente.

A abordagem híbrida mostrou-se mais eficiente do que as abordagens individuais (com o uso de técnicas de depuração *ad hoc*), além de diminuir o número de linhas de código a serem investigadas pelo desenvolvedor em cinco vezes, sendo que em média o desenvolvedor precisou investigar 20 linhas de código. Seu trabalho também indica que o desenvolvedor é mais efetivo se mudar para o fatiamento dinâmico logo após investigar os cinco primeiros resultados do *ranking* das métricas de associação (N=5).

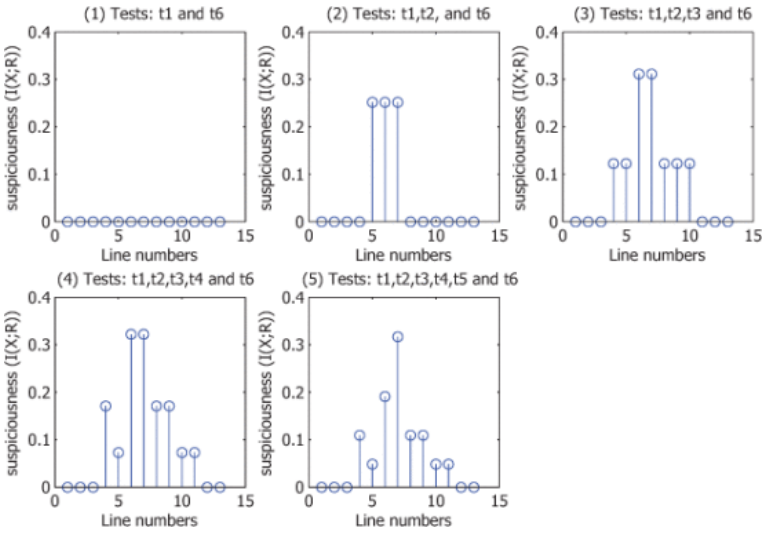
3.3 Redução de ruídos por meio do gerenciamento da entropia

O trabalho de Roychowdhury e Khurshid (2012) utiliza a seleção de características (*features*), um conceito fundamental do aprendizado de máquina, para calcular a suspeição de espectros. A seleção de características é feita com o cálculo da informação mútua, que mede o quanto o conhecimento de uma variável reduz a incerteza sobre a outra.

Eles propõem o cálculo da informação mútua por meio de entropias. A entropia é uma medida numérica, representada por números reais, que associa uma quantidade de "incerteza"a cada esquema de probabilidades (NETO, 2006). Para uma boa inferência estatística devemos escolher a distribuição com mínimo viés e, portanto, máxima entropia (NETO, 2006). Roychowdhury e Khurshid (2012) observaram que geralmente, na localização de defeito baseada em métricas de associação (Ochiai, Tarantula, etc), as linhas defeituosas se encontravam na vizinhança das linhas de código mais entrópicas.

A Figura 9, retirada do trabalho de Roychowdhury e Khurshid (2012), mostra que no cálculo de suspeição baseado na informação mútua, os maiores valores convergem para a linha onde há maior probabilidade do defeito ser encontrado. Observe-se ainda que à medida que mais casos de teste são considerados há um aumento no desempenho da localização de defeitos por essa abordagem.

Figura 9 – Casos de teste x suspeição por informação mútua



Fonte: Roychowdhury e Khurshid (2012)

Devido à complexidade do cálculo de informação mútua com o uso de entropias generalizadas, Roychowdhury e Khurshid (2012) usaram uma versão linear do cálculo que aproxima o valor da informação mútua ao considerar que os atributos são independentes uns dos outras. Para avaliação da sua técnica, além da informação mútua, eles calcularam também uma métrica denominada *code-to-examine* que indica o número de linhas a serem examinadas, partindo da linha com maior valor de informação mútua e seguindo com as linhas com menor valor, de maneira decrescente até que a linha com defeito seja encontrada.

Os experimentos realizados com o uso do *Siemens Test Suite* (HUTCHINS; FOSTER; GORADIA; OSTRAND, 1994) demonstraram que o cálculo de suspeição com o uso de informação mútua por meio de entropias genéricas foi capaz de localizar os defeitos em 90% das versões defeituosas examinando apenas 20% de posições do *ranking*. Esse desempenho foi melhor do que outras abordagens, como por exemplo, Tarantula (ROYCHOWDHURY; KHURSHID, 2012).

Campos, Abreu, Fraser e d'Amorim (2013) também utilizam o conceito de entropia para melhorar o desempenho da localização de defeitos. Em sua técnica o cálculo da entropia de um conjunto de espectros suspeitos é usado para gerar novos casos de teste que, de maneira iterativa, melhoram a acurácia da localização de defeitos ao diminuírem a incerteza dos espectros do *ranking* de suspeição.

Eles estenderam a ferramenta EVOSUITE (FRASER; ARCURI, 2011) de geração de testes baseados em busca e alteraram o algoritmo genérico da EVOSUITE para usar

o cálculo da entropia em sua função objetivo (*fitness*). Desta forma, a geração de novos casos de testes transforma-se em um problema de minimização em busca da solução ótima.

Importante destacar que no cálculo de probabilidades do espectro eles utilizam o Teorema de Bayes de forma que, para cada execução de teste, as probabilidades anteriores (*a priori*) são consideradas no cálculo de probabilidades. Ao final é gerado um *ranking* decrescente com a probabilidade de que os espectros expliquem o defeito.

A entropia é calculada por meio da densidade da matriz de cobertura, uma métrica que pode ser usada como representante da entropia do espectro e que representa a porcentagem média de componentes cobertos pelos casos de testes.

Campos, Abreu, Fraser e d'Amorim (2013) realizaram o experimento em seis defeitos de quatro grandes projetos *open source*, além de um exemplo didático de software de *vending machine*. No experimento eles reduziram, na média, 49% da entropia do *ranking* de suspeição e essa redução da entropia ocasionou uma redução média de 91% do número de candidatos a serem inspecionados para localizar o defeito. Eles compararam o resultado da sua estratégia de geração de testes com a geração aleatória de testes e mostraram que a melhora no desempenho não se deu apenas devido ao aumento de testes, mas também à assertividade da técnica de geração de testes baseada na entropia.

3.4 Redução de ruídos por eliminação de informação redundante

O estudo de Daniel e Sim (2013) foi o único artigo encontrado que utiliza o termo *redução de ruídos* aplicados a SBFL. A abordagem também manipula os testes a serem utilizados no cálculo dos *rankings* das métricas de associação por meio da eliminação de informação redundante.

O artigo baseia-se em observações feitas nos programas do *benchmark Siemens Test Suite* (HUTCHINS; FOSTER; GORADIA; OSTRAND, 1994). Em muitas versões dos programas desse *benchmark*, eles identificaram casos de teste que, apesar de distintos, apresentavam o mesmo registro de execução dos comandos, ou seja, o mesmo espectro. A partir dessa observação eles intuíram que esses conjuntos de casos de teste poderiam trazer ruídos ao cálculo das métricas de localização de defeitos. Esses conjuntos de casos de teste foram divididos em três categorias:

1. Pares de casos de teste que compartilham o mesmo espectro mas que conduzem a resultados distintos. Nesse caso, como um teste falha e o outro passa, o cálculo da métrica de localização de defeitos tem como entrada uma informação ambígua.
2. Conjuntos de casos de teste que falham e que possuem o mesmo espectro. Nesse caso, o resultado do cálculo da métrica de localização de defeitos é deteriorado pela informação de entrada do cálculo duplicada.
3. Conjuntos de casos de teste que passam e que possuem o mesmo espectro. De maneira análoga ao item anterior, há deterioração das métricas devido à duplicação dos dados de entrada do cálculo.

Com base nessa categorização, Daniel e Sim (2013) propõem seis esquemas de remoção de casos de teste para eliminação do ruído no cálculo das métricas de localização de defeitos:

- Esquema 1: Para cada caso de teste que falha, remoção de todos os casos de testes que passam e que possuem o mesmo espectro do caso de teste que falhou.
- Esquema 2: Para cada caso de teste que passa, remoção de todos os casos de testes que falham e que possuem o mesmo espectro do caso de teste que passou.
- Esquema 3: Para cada conjunto de casos de teste que falham e que passam com o mesmo espectro, eliminação de todos os casos de teste do conjunto.
- Esquema 4: Para conjuntos de casos de teste que falham ou conjuntos de casos de testes que passam, e que possuem o mesmo espectro, eliminação de todos os casos de teste, com exceção de um caso para cada um dos conjuntos
- Esquema 5: Aplicação do esquema 4 e depois do esquema 1.
- Esquema 6: Aplicação do esquema 4 e depois do esquema 2.
- Esquema 5: Aplicação do esquema 4 e depois do esquema 3.

Os esquemas foram aplicados em 62 versões defeituosas da *Siemens Test Suite* e os resultados de 31 métricas de localização de defeitos foram analisados e comparados a partir do número de posições do *ranking* que foram analisadas no código até que o defeito pudesse ser localizado. Apesar do estudo focar apenas em versões com um único defeito, foi possível identificar uma melhora de desempenho para a maioria das métricas com o uso do esquema 5. Os resultados obtidos também serviram para a elaboração de uma tabela na

qual os autores sugerem os melhores esquemas para cada uma das métricas de associação utilizadas no experimento.

3.5 Ruídos nas métricas de associação

O trabalho de Denmat, Ducassé e Ridoux (2005) propõe uma releitura do trabalho seminal de Jones, Harrold e Stasko (2002a), que propôs a utilização de métricas de associação (em particular, a métrica Tarantula) para localização de defeitos sobre a ótica da mineração de dados. Jones, Harrold e Stasko (2002a) propõem uma checagem cruzada das instruções do programa cobertas pelos casos de teste que falham e pelos casos de teste que passam de maneira a obter as instruções mais suspeitas por meio do calculo de um indicador denominado denominado JHS[1]:

$$JHS(i) = \left(\frac{\%P(i)}{\%P(i)+F(i)}\right)$$

Esse indicador JHS é uma variação das métricas usadas em mineração de dados para caracterizar regras e associação entre dados. As regras de associação indicam a existência de associações, ou a correlação entre os atributos, implicando que ou eles frequentemente aparecem juntos numa transação, ou que a variação na frequência de observação de um atributo num conjunto de transações ocorre sempre com uma variação do segundo atributo nesse mesmo conjunto de transações (SILVA; PEREZ; BOSCARIOLI, 2016).

Denmat, Ducassé e Ridoux (2005) demonstram no trabalho as associações entre a métrica JHS e as métricas da mineração. Eles sugerem a utilização da métrica *lift* para o cálculo da suspeição de uma instrução ou linha de código. Essa métrica representaria a atração ou repulsão entre a execução de uma determinada linha e a falha.

Para entender a métrica *lift*, é necessário conhecer mais duas métricas de associação: suporte e confiança. O suporte é a frequência com que os itens aparecem juntos em transações individuais e a confiança expressa a noção de importância e confiabilidade de uma regra, dada a possibilidade de sua ocorrência (SILVA; PEREZ; BOSCARIOLI, 2016).

A *lift* é então um índice estatístico utilizado para definir o grau de interesse de uma regra de associação indicando o quão mais frequente se torna B, quando A ocorre. No caso específico o quanto a instrução que leva ao defeito se torna mais frequente quando a falha ocorre. A *lift* é caracterizada pela seguinte fórmula:

[1] Esse indicador é a métrica de associação Tarantula proposta por Jones, Harrold e Stasko (2002a)

$$lift(X \rightarrow Y) = \left(\frac{conf(X \rightarrow Y)}{\sup(Y)}\right)$$

Um valor maior do que 1 indicaria a atração entre os casos que falham e os testes que executam determinada instrução. Valores menores que 1 indicariam a repulsão.

O uso das métricas de associação permitem que Denmat, Ducassé e Ridoux (2005) elencam formalmente algumas hipóteses implícitas na abordagem de Jones, Harrold e Stasko (2002a):

- Hipótese 1: Se uma execução leva a uma falha então deve existir pelo menos uma instrução que contém o defeito;
- Hipótese 2: A suspeição de uma determinada instrução pode ser interpretada separadamente do restante do programa (independência dos indicadores); e
- Hipótese 3: A execução de uma instrução com falha leva, na maioria das vezes, a um defeito.

Jones, Harrold e Stasko (2002a) validaram essas hipóteses de maneira empírica por meio de programa mutantes, mas a interpretação dada por Denmat, Ducassé e Ridoux (2005) permite o uso em trabalhos futuros das regras de associação considerando a dependência entre os espectros e, eventualmente, diminuindo o ruído ocasionado por essas relações.

Essas hipóteses explicam formalmente algumas limitações da abordagem de Jones, Harrold e Stasko (2002a):

- o defeito do programa em análise não pode ser um erro no fluxo de controle;
- o mais preciso elemento que pode ser localizado é um bloco básico do programa;
- instruções com defeitos, mas que não necessariamente causam o defeito, não podem ser localizadas por esse método.

Denmat, Ducassé e Ridoux (2005), apesar das limitações apresentadas, consideram o uso das regras de associação promissor pois os componentes da regra de associação podem ser expandidos com a associação de outros elementos (por exemplo podemos supor que o defeito não reside em uma única instrução ou associar chamadas, exceções ou vazamentos de memória).

3.6 Considerações finais

Os trabalhos apresentados proporcionaram um panorama das técnicas para reduzir os ruídos na localização de defeitos baseado em espectros (*Spectrum-based Fault Localization* – SBFL). Estes são apenas alguns exemplos de abordagens de melhoria dos *rankings* para localização de defeitos baseada em métricas de associação e não pretende ser uma lista exaustiva delas.

Daniel e Sim (2013) e Denmat, Ducassé e Ridoux (2005) citam explicitamente a interferência de ambiguidades e duplicidades no cálculo da métricas de associação (suspeição) utilizando espectros. Existem ainda trabalhos que tentam melhorar o desempenho da localização de defeitos combinando técnicas sem, de maneira explicíta, atacarem o enviesamento ocasionado pelos "ruídos". Xialon *et al.* (2013) redefinem as entradas para o cálculo das métricas de associação substituindo espectro de cobertura de código por espectros obtidos do fatiamento estático e dinâmico do programa. Soremekun, Kirschner, Böhme e Zeller (2021), por sua vez, combinam os *rankings* obtidos das métricas de associações com informações de dependência de controle e de dados para refinar a busca pelo defeito.

Xialon *et al.* (2013) sugerem o uso do fatiamento, tanto dinâmico como estático, a um custo maior, para apurar as técnicas SBFL. Porém, a técnica proposta por estes autores não abordam duplicidades e ambiguidades que também podem afetar os cálculos de suspeição baseados no espectro do fatiamento. Soremekun, Kirschner, Böhme e Zeller (2021) também não abordam as ambiguidades e duplicidades das métricas de associação; porém, mostram significativo aumento de desempenho na localização às custas de um processo híbrido e que pressupõem que o desenvolvedor transpasse manualmente o fluxo de controle e de dados dos espectros mais suspeitos indicados pela métricas de associação.

Neste trabalho, é investigado se a relação entre os espectros utilizados para localização de defeito, isto é, a relação de subsunção, pode ser utilizada para melhorar os *rankings* obtidos com métricas de associação. Apesar da influência dessa relação ter sido observada na literatura há bastante tempo por Denmat et al. (DENMAT; DUCASSé; RIDOUX, 2005), que seja do nosso conhecimento, esse efeito não foi investigado adequadamente. Assim, este trabalho visa preencher essa lacuna, investigando o efeito da subsunção em SBFL por meio de um experimento que utiliza programas com defeitos

reais, similares a outros desenvolvidos na indústria e com uma abordagem inédita ao usar dados de subsunção para restringir o número de associações definição uso utilizadas como entrada para o cálculo das métricas de associação.

Analisamos o efeito da subsunção de fluxo de dados nos *rankings* obtidos com a métrica *Ochiai* pois está entre as mais efetivas na localização de defeitos baseada em espectros de fluxos de dados (PEARSON *et al.*, 2017).

4 Feramentas e benchmark

Este capítulo apresenta as ferramentas e o *benchmark* utilizados para avaliar o impacto da relação de subsunção de fluxo de dados na localização de defeitos baseada em espectros. O capítulo é concluído com os procedimentos realizados para coletar os dados necessários para a avaliação.

4.1 Jaguar

Jaguar (*JAva coveraGe faUlt locAlization Ranking*) (RIBEIRO, 2016) é uma ferramenta que implementa a localização de defeitos baseada em espectros para depuração de programas escritos em Java. Ela implementa dez métricas de associação, a saber, DRT (CHAIM; MALDONADO; JINO, 2004), Jaccard (CHEN; KICIMAN; FRATKIN; FOX; BREWER, 2002), Kulczynski2 (NAISH; LEE; RAMAMOHANARAO, 2009), McCon (NAISH; LEE; RAMAMOHANARAO, 2009), Minus (XU; ZHANG; CHAN; TSE; LI, 2013), Ochiai (ABREU; ZOETEWEIJ; GOLSTEIJN; GEMUND, 2009), Op (NAISH; LEE; RAMAMOHANARAO, 2011), Tarantula (JONES; HARROLD; STASKO, 2002b), Wong3 (DEBROY; WONG, 2011) e Zoltar (GONZALEZ, 2007), para calcular a suspeição de linhas ou associações definição uso (ADUs) do programa.

Para coletar as informações de cobertura do código, Jaguar utiliza bibliotecas das ferramentas JaCoCo[1] e BA-DUA (ARAUJO; CHAIM, 2014). Para cada teste escrito utilizando a biblioteca JUnit[2], Jaguar invoca a JaCoCo para gerar a cobertura (espectro) de linhas ou a BA-DUA para gerar a cobertura (espectro) de ADUs.

Jaguar foi inicialmente desenvolvida como um *plugin* do ambiente integrado de desenvolvimento de software *Eclipse*[3] e o seu código fonte está disponível no endereço: ⟨https://github.com/saeg/jaguar⟩. Sua última *release* disponibilizou um utilitário para uso da ferramenta em linha de comando e que gera o *ranking* de suspeição no formato XML para as dez métricas de associação citadas anteriormente, porém somente para a cobertura baseada em fluxo de controle (JaCoCo). Nesse trabalho de pesquisa, será utilizada uma versão modificada dessa *release*[4] que, a despeito de não calcular o *ranking* em formato

[1] ⟨https://www.jacoco.org/⟩
[2] ⟨https://www.JUnit.org/⟩
[3] ⟨https://www.eclipse.org/⟩
[4] ⟨https://github.com/marioconcilio/jaguar-df⟩

XML, disponibiliza as informações de cobertura de ADUs para cada um casos de teste executados. A partir dessa informação é possível extrair os coeficientes para cálculo de quaisquer métricas baseadas em dados de cobertura, isto é, espectros. A seguir será descrito o resultado da execução da nova *release* da ferramenta para o programa Max.

O primeiro passo para usar a ferramenta é a compilação do código fonte Java com o parâmetro *-g*. Esse parâmetro gera informação de depuração como número de linhas e variáveis locais. Após a execução da Jaguar linha de comando, são gerados no diretório `.jaguar` da raiz do projeto dois arquivos <*nome da classe*>`.spectra` e <*nome da classe*>`.matrix` para cada classe do programa.

O arquivo <*nome da classe*>`.spectra` lista todas as ADUs da classe; a ordem em que as ADUs são listadas é importante para extração dos dados de cobertura a partir do arquivo <*nome da classe*>`.matrix`. Na Listagem 4.1 podemos ver o arquivo `br.usp.each.sfl.Max.matrix` gerado para a classe do programa Max.

Listing 4.1 – Arquivo br.usp.each.sfl.Max.spectra

```
br.usp.each.sfl.Max#calculaMax:(6,(10,11), array)
br.usp.each.sfl.Max#calculaMax:(6,(10,12), array)
br.usp.each.sfl.Max#calculaMax:(6,11, array)
br.usp.each.sfl.Max#calculaMax:(6,(8,10), length)
br.usp.each.sfl.Max#calculaMax:(6,(8,14), length)
br.usp.each.sfl.Max#calculaMax:(6,14, out)
br.usp.each.sfl.Max#calculaMax:(7,(8,10), i)
br.usp.each.sfl.Max#calculaMax:(7,(8,14), i)
br.usp.each.sfl.Max#calculaMax:(7,(10,11), i)
br.usp.each.sfl.Max#calculaMax:(7,(10,12), i)
br.usp.each.sfl.Max#calculaMax:(7,12, i)
br.usp.each.sfl.Max#calculaMax:(7,11, i)
br.usp.each.sfl.Max#calculaMax:(7,14, max)
br.usp.each.sfl.Max#calculaMax:(7,15, max)
br.usp.each.sfl.Max#calculaMax:(7,(10,11), max)
br.usp.each.sfl.Max#calculaMax:(7,(10,12), max)
br.usp.each.sfl.Max#calculaMax:(11,14, max)
br.usp.each.sfl.Max#calculaMax:(11,15, max)
br.usp.each.sfl.Max#calculaMax:(11,(10,11), max)
br.usp.each.sfl.Max#calculaMax:(11,(10,12), max)
br.usp.each.sfl.Max#calculaMax:(12,(8,10), i)
```

```
br.usp.each.sfl.Max#calculaMax:(12,(8,14), i)
br.usp.each.sfl.Max#calculaMax:(12,(10,11), i)
br.usp.each.sfl.Max#calculaMax:(12,(10,12), i)
br.usp.each.sfl.Max#calculaMax:(12,12, i)
br.usp.each.sfl.Max#calculaMax:(12,11, i)
```

Fonte: Dennis Lopes, 2021

O arquivo <*nome da classe*>`.matrix` lista em cada linha os resultados de cobertura das ADUs para cada caso de teste, isto é, cada método de teste escrito usando a bibliotea JUnit. Cada linha do arquivo apresenta os resultados de um caso de teste específico e as colunas representam, na ordem em que foram apresentadas no arquivo <*nome da classe*>`.spectra`, cada uma das ADUs. O valor `1` indica que a ADU foi coberta e o valor `0` indica que a ADU não foi coberta. A última coluna de cada linha indica se o caso de teste foi executado com sucesso (`+`) ou com falha (`-`).

Listing 4.2 – Arquivo br.usp.each.sfl.Max.matrix

```
1 1 1 1 1 1 1 0 0 1 1 0 0 0 1 1 1 1 0 0 1 1 1 1 1 1 +
1 1 1 1 1 1 1 0 0 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 -
1 1 1 1 1 1 1 0 0 1 1 0 0 0 1 1 1 1 1 0 1 1 1 0 1 1 +
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -
1 1 1 1 1 1 1 0 0 1 1 0 0 0 1 1 1 1 0 1 1 1 1 1 1 1 +
1 1 1 1 1 1 1 0 0 1 1 0 0 0 1 1 1 1 0 1 1 1 1 1 1 1 +
```

Fonte: Dennis Lopes, 2021

Esses dados permitem obter os coeficientes utilizados no cálculo das métricas de localização de defeitos. O *ranking* de suspeição das ADUs é calculado posteriormente com o uso de um programa em Phyton que irá consumir os arquivos <*nome da classe*>`.spectra` e <*nome da classe*>`.matrix`.

4.2 SAtool

SATool (*Subsumption Algorithm Tool*) (CHAIM; BARAL; OFFUTT; CONCILIO; ARAúJO, 2021) é uma ferramenta utilizada para o cálculo da subsunção de ADUs por meio da análise estática de programas escritos na linguagem Java. Sua entrada são os arquivos bytecode do programa Java em análise e sua saída é, para cada classe do programa, um par de arquivos no formato JSON.

O primeiro arquivo gerado, que pode ser observado na Listagem 4.3, lista os métodos da classe e, para cada método, o seu conjunto de ADUs. Para o programa Max foram identificadas 26 ADUs (identificados de 0 a 25).

Listing 4.3 – SAToool - ADUs

```
{
"Class" : "br.usp.each.sfl.Max",
"Methods" : [{ "Name" : "max" ,
"Duas" : 26,
"0" :  "(6,(10,11), array)",
"1" :  "(6,(10,12), array)",
"2" :  "(6,11, array)",
"3" :  "(6,(8,10), length)",
"4" :  "(6,(8,14), length)",
"5" :  "(6,14, out)",
"6" :  "(7,(8,10), i)",
"7" :  "(7,(8,14), i)",
"8" :  "(7,(10,11), i)",
"9" :  "(7,(10,12), i)",
"10" :  "(7,12, i)",
"11" :  "(7,11, i)",
"12" :  "(7,14, max)",
"13" :  "(7,15, max)",
"14" :  "(7,(10,11), max)",
"15" :  "(7,(10,12), max)",
"16" :  "(11,14, max)",
"17" :  "(11,15, max)",
"18" :  "(11,(10,11), max)",
"19" :  "(11,(10,12), max)",
"20" :  "(12,(8,10), i)",
"21" :  "(12,(8,14), i)",
"22" :  "(12,(10,11), i)",
"23" :  "(12,(10,12), i)",
"24" :  "(12,12, i)",
"25" :  "(12,11, i)"}]
}
```

Fonte: Dennis Lopes, 2022

O segundo arquivo, descrito na Listagem 4.4, lista o nome da classe e seus métodos. Para cada método da classe, são apresentadas a quantidade de ADUs (no arquivo referenciada na *tag* `Duas`) e a quantidade de folhas no grafo de subsunção reduzido, isto é, o número de retângulos da Figura 8 (no arquivo referenciada na *tag* `Subsumers`).

Cada folha do grafo de subsunção reduzido recebe um rótulo, por exemplo, “0” para a primeira folha, “1” para a segunda e assim por diante como apresentado na Listagem 4.4. Os identificadores das ADUs não-limitadas (*unconstrained*) são representados pelas listas associadas a esses rótulos. Na sequência, são apresentados rótulos que começam com a letra S aos quais são associados listas de identificadores de ADUs subsumidas. Por exemplo, as ADUs não-limitadas associados ao rótulo “0” subsumem as ADUs associadas ao rótulo “S0”, o que inclui as próprias ADUs não-limitadas, ou seja, as ADU subsumidoras subsumem a si próprias.

Listing 4.4 – SAToool - Subsunção

```
{
"Class" : "br.usp.each.sfl.Max",
"Methods" : [{ "Name" : "max" ,
"Duas" : "26" ,
"Subsumers" : 5,
"0" : [ 11, 8], "S0" : [0, 2, 3, 4, 5, 6, 8, 10, 11, 14, 16, 17, 21 ],
"1" : [ 18], "S1" : [0, 2, 3, 4, 5, 6, 10, 14, 16, 17, 18, 20, 21, 22, 24, 25 ],
"2" : [ 19], "S2" : [0, 1, 2, 3, 4, 5, 6, 10, 14, 16, 17, 19, 20, 21, 23, 24 ],
"3" : [ 15, 9], "S3" : [1, 3, 4, 5, 6, 9, 10, 15, 21 ],
"4" : [ 7], "S4" : [4, 5, 7, 12, 13 ]
}]
}
```

Fonte: Dennis Lopes, 2022

Observe-se na Listagem 4.4 que as ADUs de identificadores 11 e 8 (rótulo “0”), ou seja, as ADUs (7,11, i) e (7,(10,11), i), subsumem as ADUs 0, 2, 3, 4, 5, 6, 8, 10, 11, 14, 16, 17 e 21 (rótulo “S0”) que correspondem, respectivamente, as ADUs (6,(10,11), array) , (6,11, array), (6,(8,10), length), (6,(8,14), length), (6,14, out), (7,(8,10), i), (7,(10,11), i), (7,12, i), (7,11, i), (7,(10,11), max), (11,14, max), (11,15, max), (12,(8,14), i). Os identificadores no arquivo da Listagem 4.4 referencem às ADUs na Listagem 4.3 do arquivo `br.usp.each.sfl.Max.duas.json` gerado pela ferramenta SATool.

4.3 Ranking de ADUs não-limitadas e subsumidas

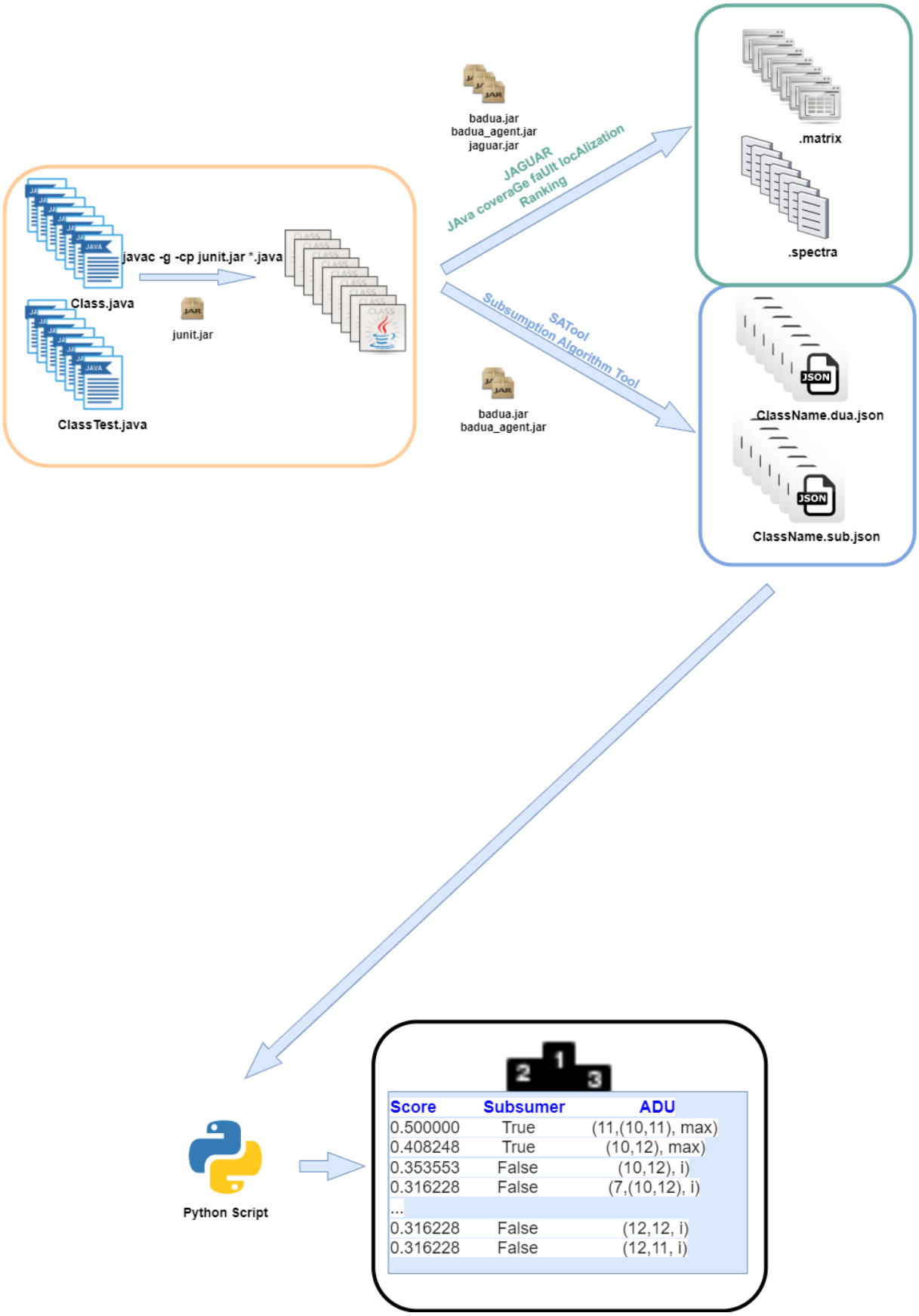
A partir dos arquivos bytecode gerados pela compilação dos arquivos Java com informações de depuração, o processo de cálculo do *ranking* de ADUs não-limitadas e subsumidas segue dois processos paralelos: o cálculo do par de arquivos *<nome da*

classe>`.matrix` e <*nome da classe*>`.spectra` para cada uma das classes pela ferramenta Jaguar e o cálculo de subsunção a partir da análise estática do bytecode pela ferramenta SATool que gera o par de arquivos JSON com a lista de ADUs e a informação de subsunção. Esses arquivos são a entrada de um programa desenvolvido em Python que a partir desses arquivos gera o *ranking* de ADUs não limitadas e subsumidas para qualquer medida de associação implementada pela Jaguar.

Um sumário desse processo de geração dos *rankings* pode ser observado na Figura 10. Todo o processo de geração do ranking, com seus dados de entrada, dados de saída e scripts utilizados pode ser encontrado no repositório github:

https://github.com/dennislopes/df-experiments

Figura 10 – Cálculo *ranking* de ADUs não-limitadas e subsumidas



Fonte: Dennis Lopes, 2021

Tabela 10 – Cálculo do *ranking* de ADUs para o programa Max

Ochiai	**Não-limitada?**	**DUA**
0.71	False	(12,(10,12), i)
0.71	False	(7,14, max)
0.35	True	(7,(10,12), i)
0.35	True	(7,(10,12), max)
0.35	False	(6,(8,10), length)
0.35	False	(6,(8,14), length)
0.35	False	(6,(10,12), array)
0.35	False	(7,(8,10), i)
0.35	False	(7,12, i)
0.35	False	(12,(8,10), i)
0.35	False	(12,(8,14), i)
0.35	False	(12,12, i)
0.00	True	(11,(10,11), max)
0.00	True	(11,(10,12), max)
0.00	True	(7,(10,11), i)
0.00	True	(7,(8,14), i)
0.00	False	(6,(10,11), array)
0.00	False	(6,11, array)
0.00	True	(7,11, i)
0.00	False	(7,(10,11), max)
0.00	False	(11,14, max)
0.00	False	(12,(10,11), i)
0.00	False	(12,11, i)

Fonte: Dennis Lopes, 2022

A Tabela 10 exibe o *ranking Ochiai* para o programa Max, gerado no formato CSV a partir do processo descrito. Observe-se que a segunda coluna indica se a ADU é não limitada (*True*) ou subsumida (*False*).

4.4 Repositório Defect4J

Defects4J é um repositório de defeitos e um arcabouço de execução de testes que possibilita a reprodução e o isolamento de defeitos reais encontrados em projetos de software (JUST; JALALI; ERNST, 2014). Cada defeito catalogado no banco de dados pode ser reproduzido a partir da instanciaçao de uma versão específica presente no histórico da ferramenta de controle de versão.

A reprodutibilidade de experimentos é facilitada por uma abstração que permite instanciar, a partir da ferramenta de controle de versão, a versão do programa com o

defeito e também a versão do programa em que esse defeito foi corrigido. A ferramenta disponibiliza ainda uma *suite* de teste que inclui pelo menos um caso de teste que expõe o defeito. Esse caso de teste falha na versão defeituosa do software e passa na versão na qual o defeito corrigido.

Cada um dos defeitos possui as seguintes propriedades:

- A versão defeituosa foi salva em uma ferramenta de controle de defeitos e identificada na mensagem do `commit` que corrigiu o defeito;
- O defeito foi corrigido em um `commit` único;
- O defeito é minimizado, ou seja, os mantenedores do Defects4j removeram mudanças irrelevantes do `commit`;
- O defeito foi corrigido pela modificação do código fonte;
- Existe pelo menos um teste que falha antes da correção e que passa após sua correção (JUST; JALALI; ERNST, 2014).

A versão atual da ferramenta, disponível no endereço ⟨https://github.com/rjust/defects4j⟩, contém 835 defeitos distribuidos em 17 projetos *open source*. A distribuição e os nomes dos projetos estão na tabela 11.

Tabela 11 – Lista de programas e quantidade de defeitos (Defects4J)

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

Fonte: Dennis Lopes, 2021

4.5 Considerações finais

Este capítulo apresentou as ferramentas Jaguar e SATool que foram utilizadas para coletar os espectros e as relações de subsunção dos programas do repositório Defects4J. Foi apresentado ainda como os dados dessas ferramentas são utilizados para gerar os *rankings* das associações definição uso (ADU) não-limitadas e subsumidas para os diferentes defeitos da base de dados Defects4J. No próximo capítulo, são apresentados o desenho experimental, os resultados e a discussão do experimento realizado para avaliar o impacto da relação de subsunção de fluxo de dados na localização de defeitos.

5 Avaliação Experimental

Este capítulo apresenta o experimento realizado para avaliar o impacto da relação de subsunção de fluxo de dados na localização de defeitos baseada em espectros. São apresentadas inicialmente as questões de pesquisa, os defeitos do arcabouço *Defects4J* utilizados, as métricas coletadas e os resultados. Conclui-se o capítulo com a análise das ameaças à validade e uma discussão sobre os resultados.

5.1 Questões de pesquisa

O objetivo do experimento é comparar a eficácia na localização de defeitos do *ranking* obtido por meio do cálculo das métricas de associação com o uso apenas das ADUs não-limitadas versus o *ranking* obtido com o uso do conjunto total de ADUs. A hipótese de pesquisa a ser validada é descrita a seguir:

Hipótese: *O conjunto de ADUs não-limitadas, por ser um conjunto menor de ADUs, pode levar o desenvolvedor a investigar menos código sem diminuir significativamente o número de defeitos encontrados.*

As questões de pesquisa apresentadas a seguir visam avaliar essa hipótese.

QP1 *Qual a perda em termos de localização de defeitos?*

Ao utilizar apenas as ADUs não-limitadas, um defeito pode não ser localizado porque as ADUs subsumidas são desconsideradas. Logo, QP1 avalia o quão frequente é a situação em que o defeito está associado apenas a ADUs subsumidas, levando a perdas com o uso de ADUs não-limitadas.

QP2 *Qual a perda em termos de ranqueamento?*

Outra situação de perda ocorre quando as ADUs subsumidas são posicionadas no *ranking* antes das ADUs não-limitadas. QP2 avalia a situação em que as ADUs não-limitadas incluem o defeito mas podem levar o desenvolvedor a investigar mais posições no *ranking* de suspeição.

QP3 *Qual a diminuição em termos de linhas inspecionadas?*

QP3 compara o número de linhas investigadas utilizando ADUs não-limitadas e todas as ADUs para verificar se ocorre diminuição na quantidade de código investigado.

5.2 Defeitos utilizados no experimento

Devido às características inerentes às ferramentas utilizadas, existem algumas limitações quanto aos defeitos constantes no repositório Defects4J que puderam ser utilizados. As razões para a eliminação de alguns defeitos são detalhadas a seguir:

1. Ocorrência de exceções que podem levar à perda de informações de localização. A ferramenta Jaguar, utilizada para gerar dados de cobertura de fluxo de dados, utiliza a ferramenta BA-DUA (ARAUJO; CHAIM, 2014) que, por sua vez, não gera dados de cobertura para exceções não tratadas nos métodos. Defeitos localizados nesses métodos, portanto, não podem ser utilizados no experimento.
2. Falta de cobertura de defeitos localizados em métodos de um só nó, isto é, sem comandos de mudança de fluxo como *if*, *while* ou *for*. BA-DUA não coleta cobertura para esses métodos pois eles não possuem ADUs.
3. Limite do tamanho do *bytecode* Java para cada método. Como a *Java Virtual Machine* limita que o *bytecode* para cada método em uma classe não pode exceder a 64K bytes, se o *bytecode* somado à instrumentação adicionada pela ferramenta de cobertura ultrapassar os 64K, o método não será instrumentado e a cobertura não será gerada.

Assim, de um total de 835 defeitos do Defects4J, foram considerados para o experimento 781 (93,53% do total de defeitos). A Tabela 12 especifica o total de defeitos do Defects4J e o número de defeitos utilizados no experimento para cada um dos programas. Ela exclui os defeitos devidos aos itens (1) e (3); na análise dos dados, são excluídos os defeitos devido ao item (2).

Tabela 12 – Número de defeitos utilizados no experimento

Programa	**Defects4J**	**Utilizados**
Chart	26	23
Cli	39	39
Closure	174	172
Codec	18	18
Collections	4	4
Compress	47	46
Csv	16	16
Gson	18	17
JacksonCore	26	22
JacksonDatabind	112	109
JacksonXml	6	6
Jsoup	93	93
JxPath	22	22
Lang	64	29
Math	106	102
Mockito	38	38
Time	26	25

Fonte: Dennis Lopes, 2022

5.3 Métricas coletadas

Métricas de associação atribuem para cada ADU um valor de suspeição, obtido por meio das ADUs exercitadas pelos casos de teste que falham e pelos casos de teste que passam (ver Tabela 5). Em nosso experimento, a métrica de associação utilizada foi Ochiai, que atribui valores que variam entre 0 e 1.

A partir dos valores de suspeição atribuídos por Ochiai, as ADUs foram listadas por ordem de suspeição para cada versão defeituosa, isto é, foi criado um *ranking* de suspeição das ADUs. Em seguida, foram verificadas se linhas associadas ao defeito, extraídas do arcabouço Defects4J, fazem parte do código associado às ADUs do *ranking*.

A Tabela 13 apresenta o *ranking* obtido para a versão defeituosa Chart 18b. A primeira linha defeituosa de Chart 18b a aparecer no *ranking*, linha 333, faz parte da ADU `(333,337, this)`, do método `removeValue`. Essa ADU possui o *score* 1,0, o primeiro *score* do *ranking*. Observe-se ainda que cada *score* possui várias ADUs empatadas e que essas ADUs podem ser não-limitadas ou subsumidas.

Em nosso experimento, verifica-se a posição no *ranking* da primeira ADU que contém, em suas linhas, o número da linha defeituosa. Essa posição determina uma

Tabela 13 – Exemplo de ranking para determinação de linhas – Chart versão 18b

Rank.	***Score***	**Não-Limitada?**	**Método**	**ADU**
1	1,0	Sim	removeValue	(333,337, this)
1	1,0	Sim	removeValue	(333,(334,337), index)
1	1,0	Sim	removeValue	(333,337, index)
2	0,87	Sim	removeValue	(333,(334,335), index)
2	0,87	Sim	removeValue	(316,(318,321), this)
2	0,87	Sim	removeValue	(316,(318,321), index)
2	0,87	Sim	removeValue	(316,(318,321), this.keys)
3	0,71	Não	removeColumn	(455,460, this)
3	0,71	Não	removeColumn	(455,460, columnKey)
3	0,71	Sim	removeColumn	(455,458, columnKey)
3	0,71	Não	removeColumn	(455,460, this.columnKeys)
3	0,71	Sim	removeColumn	(455,(456,457), iterator)
3	0,71	Não	removeColumn	(455,(456,460), iterator)
3	0,71	Sim	removeColumn	(455,457, iterator)

Fonte: Dennis Lopes, 2022

lista de ADUs (não-limitadas e subsumidas) que, por hipótese, seriam verificadas pelo desenvolvedor até que ele selecionasse essa ADU que localiza o defeito.

A segunda verificação utilizou apenas o subconjunto de ADUs não-limitadas e comparou se o *score* da ADU não-limitada que localiza o defeito é igual ou melhor que o *score* da ADU subsumida que localiza o defeito ao tomarmos, nesse caso, apenas o subconjunto de ADUs subsumidas. A contagem total de ADUs pode ser vista na Tabela 14. Essa tabela subdivide as ADUs em dois grupos: grupo principal e o grupo "linha".

Para o programa Cli, 16 defeitos foram localizados utilizando ADUs não-limitadas, sendo que dois desses também foram encontrados por ADUs subsumidas. O grupo "linha" representa os casos em que o defeito é encontrado apenas por esse tipo de ADU, ou seja, se encontramos o defeito em uma ADU subsumida', não existe nenhuma ADU não-limitada que localiza esse defeito. O contrário vale para a ADU não-limitada'. Novamente para o programa Cli, dez defeitos são localizados apenas por ADUs não-limitadas e cinco por apenas ADUs subsumidas. A coluna *Não encontrado* refere-se aos defeitos excluídos devido ao item (2) das razões para exclusão de defeitos (Seção 5.2).

Nos resultados, diversas ADUs possuem o mesmo *ranking* e a cada um desses *rankings* foi atribuído a um inteiro no intervalo [1, ∞], onde o maior *ranking* recebeu o

Tabela 14 – Localização do defeito por tipo de ADU

Programa	**Não-Limitada**	**Não-Limitada'**	**Subsumida**	**Subsumida'**	**Não-Encontrada**
Chart	8	4	0	1	10
Cli	16	10	2	5	6
Closure	105	22	16	6	23
Codec	7	0	2	2	7
Collections	0	0	0	0	4
Compress	26	5	3	4	8
Csv	5	2	0	0	9
Gson	5	5	0	1	6
JacksonCore	11	2	2	2	5
JacksonDatabind	60	12	20	6	11
JacksonXml	3	0	1	0	2
Jsoup	47	10	8	9	19
JxPath	15	1	6	0	0
Lang	17	1	0	4	7
Math	44	8	6	12	32
Mockito	19	4	5	3	7
Time	11	5	0	2	7

Fonte: Dennis Lopes, 2022

valor 1 e os *rankings* posteriores, em ordem crescente, receberam os valores 2, 3, 4 ... Esses dados foram utilizados para calcular as métricas a seguir.

5.3.1 Assertividade do conjunto de ADUs não-limitadas

A partir da análise de cada versão dos programas do Defects4J, determinou-se a assertividade do conjunto de ADUs não-limitadas na localização de defeitos. Esse dados permitiram o cálculo percentual da assertividade, ou seja, o quanto o subconjunto de ADUs não-limitadas foi assertivo na localização do defeitos. A assertividade é dada pela fórmula a seguir:

$$Assertividade = \frac{DefeitosConjuntoADUsNaoLimitadas}{TotalDefeitosADUs}$$

Ou seja, o percentual de defeitos localizados apenas pelo conjunto de ADUs de interesse (ADUs não-limitadas) em relação ao total de defeitos localizados pelo conjunto total de ADUs (soma de ADUs não-limitadas e ADUs subsumidas).

5.3.2 Número de linhas a inspecionar

Uma associação definição-uso (*i,j,var*) ou (i,(*j,k*),*var*) indica que existe ao menos um caminho livre de definição do nó *i*, no qual ocorre a definição da variável *var*, até o nó *j* ou ramo (*i,j*), onde há o uso de *var*. Assim *i*, *j* e *k* são as linhas onde o par definição-uso ocorre.

Com base nessa definição, foi comparada a quantidade de linhas a ser inspecionadas pelo desenvolvedor quando ele utiliza apenas o conjunto de ADUs não-limitadas e quando ele utiliza o conjunto completo de ADUs (ADUs não-limitadas e ADUs subsumidas). Para exemplificar, temos o conjunto de ADUs da Tabela 13. A coluna *Não-Limitada?* indica se a ADU é subsumida (Não) ou não-limitada (Sim).

Imagine-se que um desenvolvedor decida investigar as ADUs com os três maiores *scores* da Tabela 13: *scores* 1, 0,87 e 0,71. Nesse caso, o conjunto de linhas a serem inspecionadas pelo desenvolvedor para a localização de defeitos é 333, 334, 335, 337, 316, 318, 321, 455, 456, 457 e 458, quando utilizado o conjunto de ADUs não-limitadas. Quando utilizamos todas as ADUs é acrescentada uma linha a mais, a linha 460. Observe-se que no exemplo dado o ganho foi marginal e que os dois conjuntos não são totalmente disjuntos.

Considerando apenas o conjunto disjunto, representado apenas pelas linhas determinadas pelas ADUs subsumidas, foi calculado o seu percentual em relação ao conjunto total de ADUs. Esse é o percentual de linhas que será descartado pelo desenvolvedor em sua depuração quando utilizando o subconjunto de ADUs não-limitadas ao invés do conjunto total de ADUs. A Tabela 19 apresenta o número médio de linhas a serem inspecionadas pelo programador quando utilizando esses dois conjuntos: ADUs não-limitadas (*ADUs NL*) e todas ADUs (*ADUs todas*). Nesse caso, para o cálculo desse percentual foi considerado que o desenvolvedor investigaria todo o *ranking*.

5.4 Subconjuntos Top N Score

O uso de uma ferramenta de localização de defeitos pode sugerir, a partir do seu *ranking*, muitos pontos de partida promissores para a depuração. Mesmo que a ferramenta não identifique a localização exata do defeito, a exibição de pontos de entrada de código relevantes pode ajudar na compreensão do programa (PARNIN; ORSO, 2011).

Com base nessa percepção, foram selecionados subconjuntos do *ranking* geral com apenas as primeiras N posições. Note-se que em uma mesma posição do *ranking* pode haver ADUs com igual probabilidade de serem selecionadas pelo desenvolvedor para análise. Esses subconjuntos selecionados contêm as ADUs com os N maiores scores, denominados *Subconjuntos Top N Score*.

Assim sendo, os subconjuntos Top N Score são os subconjuntos para os quais selecionamos do ranking geral apenas as associações definição-uso possuem os n primeiros scores, ou seja, o subconjunto de ADUs que possuem apenas o score mais alto do ranking (Top 1), o subconjunto de ADUs que possuem apenas os dois scores mais altos do ranking (Top 2) e assim sucessivamente.

Considerando a Tabela 13, o subconjunto *Top 3 Score* inclui as ADUs que possuem os *scores* 1,0, 0,87 e 0,71 e, no caso da seleção de todas as ADUs, esse subconjunto inclui 14 ADUs. Já no caso das ADUs não-limitadas, o *Top 3 Score* também inclui as ADUs que possuem os scores 1,0, 0,87 e 0,71, porém, são excluídas as quatro ADUs subsumidas e o conjunto final possui 10 ADUs.

Esses conjuntos foram utilizados como entrada para os *scripts* de cálculo das métricas descritas anteriormente. Os resultados permitem avaliar o impacto das ADUs não-limitadas na localização de defeitos quando apenas um subconjunto do *ranking* é investigado. No caso dos conjuntos *Subconjuntos Top N Score*, entretanto, consideramos apenas as ADUs que possuem *score* maior ou igual ao *score* da ADU que revela o defeito. Dessa forma, em um *ranking* Top 10, se o defeito é revelado no terceiro *score*, apenas os três primeiros *score* são considerados no cálculos.

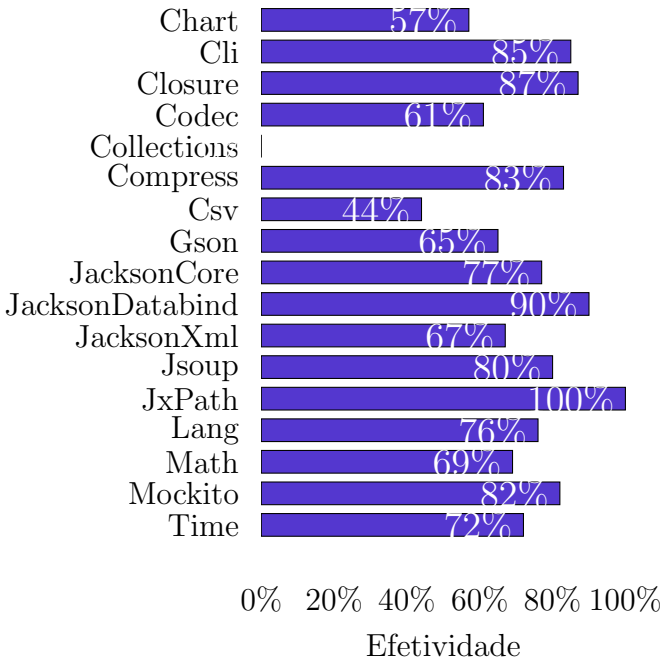
5.5 Resultados

Nesta seção, são apresentados os resultados do experimento organizados pelas questões de pesquisa QP1, QP2 e QP3.

QP1: Qual a perda em termos de localização de defeitos?

A Figura 11 apresenta a efetividade quando se utiliza todas as ADUs para localizar os defeitos selecionados do Defects4J, isto é, a porcentagem de defeitos que são encontrados

Figura 11 – Efetividade do total de ADUs



Fonte: Dennis Lopes, 2022

utilizando todas ADUs. Observe-se que, para o programa Chart, 57% dos 23 defeitos utilizados são localizados utilizando espectros de fluxo de dados (ADUs). Logo, ADUs não encontram 10 defeitos, seja porque o defeito está localizado em um método de um único nó, seja porque se trata de um defeito de omissão para o qual não há código associado. Essa métrica, portanto, mostra a efetividade, como um todo, da técnica de localização de defeitos baseada em espectros de fluxo de dados.

Considerando esses defeitos localizados pela técnica SBFL, na Tabela 15 é apresentada a assertividade da estratégia de localização de defeitos com o uso de apenas ADUs não-limitadas. Exemplificando para o programa Cli, temos que 33 defeitos são localizados utilizando todas as ADUs, sendo que destes, 26 são localizados utilizando apenas o subconjunto de ADUs não-limitadas. Em média, considerando todos os programas do *Defects4J* utilizados no experimento, 76% dos defeitos foram localizados pelos *rankings* gerados pelos conjuntos de ADUs não-limitadas.

Esses dados representam situações em que o *ranking* de ADUs não-limitadas inclui ao menos uma linha defeituosa e onde o *score* da ADU que localiza o defeito nesse *ranking* é igual ou maior do que o *score* da ADU que localiza o defeito no *ranking* de todas as ADUs. Eles não compreendem situações em que a ADU não-limitada, mesmo com um *score* menor do que uma ADU subsumida, ainda é capaz de localizar o defeito. A Figura 12

Tabela 15 – Assertividade das ADUs não-limitadas usando todo o ranking

Programa	**# Defeitos Encontrados**	**# Defeitos ADUs NL**	**Assertividade**
Chart	13	12	92%
Cli	33	26	79%
Closure	149	127	85%
Codec	11	7	64%
Collections	0	0	0%
Compress	38	31	82%
Csv	7	7	100%
Gson	11	10	91%
JacksonCore	17	13	76%
JacksonDatabind	98	72	73%
JacksonXml	4	3	75%
Jsoup	74	57	77%
JxPath	22	16	73%
Lang	22	18	82%
Math	70	52	74%
Mockito	31	23	74%
Time	18	16	89%
Média Geral			**76**%

Fonte: Dennis Lopes, 2022

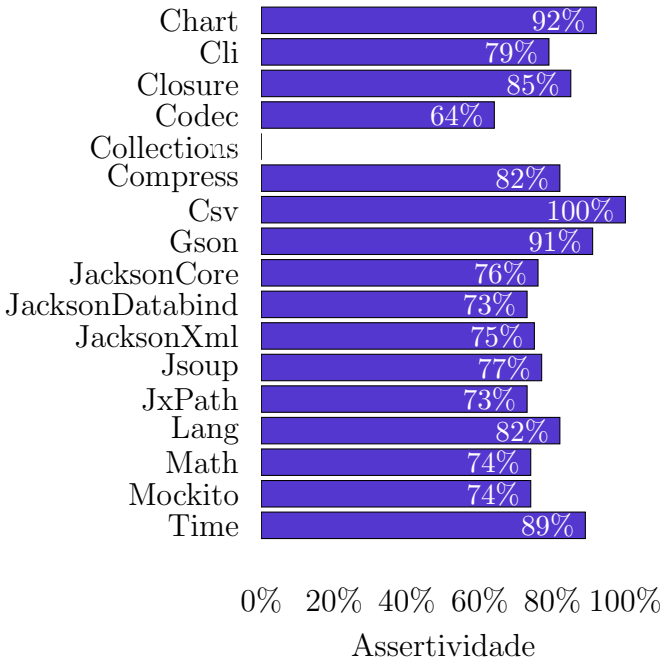
também apresenta os valores de assertividade para cada programa. No caso do programa Cli, a figura mostra que 79% dos defeitos encontrados pelas ADUs são encontrados pelas ADUs não-limitadas.

Sobre a perspectiva dos *Top N Scores*, os resultados são ainda mais promissores. A Figura 13 mostra o número de defeitos localizados em cada um dos *Top N scores*, respectivamente, Top 10 (371 defeitos), Top 5 (323 defeitos), Top 3 (275 defeitos), Top 2 (246 defeitos) e Top 1 (192 defeitos). O gráfico de tendência sobreposto mostra que quanto menos ADUs são consideradas, maior a assertividade obtida do *ranking* obtido com o subconjunto de ADUs não-limitadas. A assertividade foi calculada tomando-se a porcentagem de defeitos encontrados pelo *ranking* de ADUs não-limitadas frente ao número de defeitos encontrados pelo *ranking* gerado pelo total de ADUs (não-limitadas e subsumidas).

A Tabela 16 mostra ainda que:

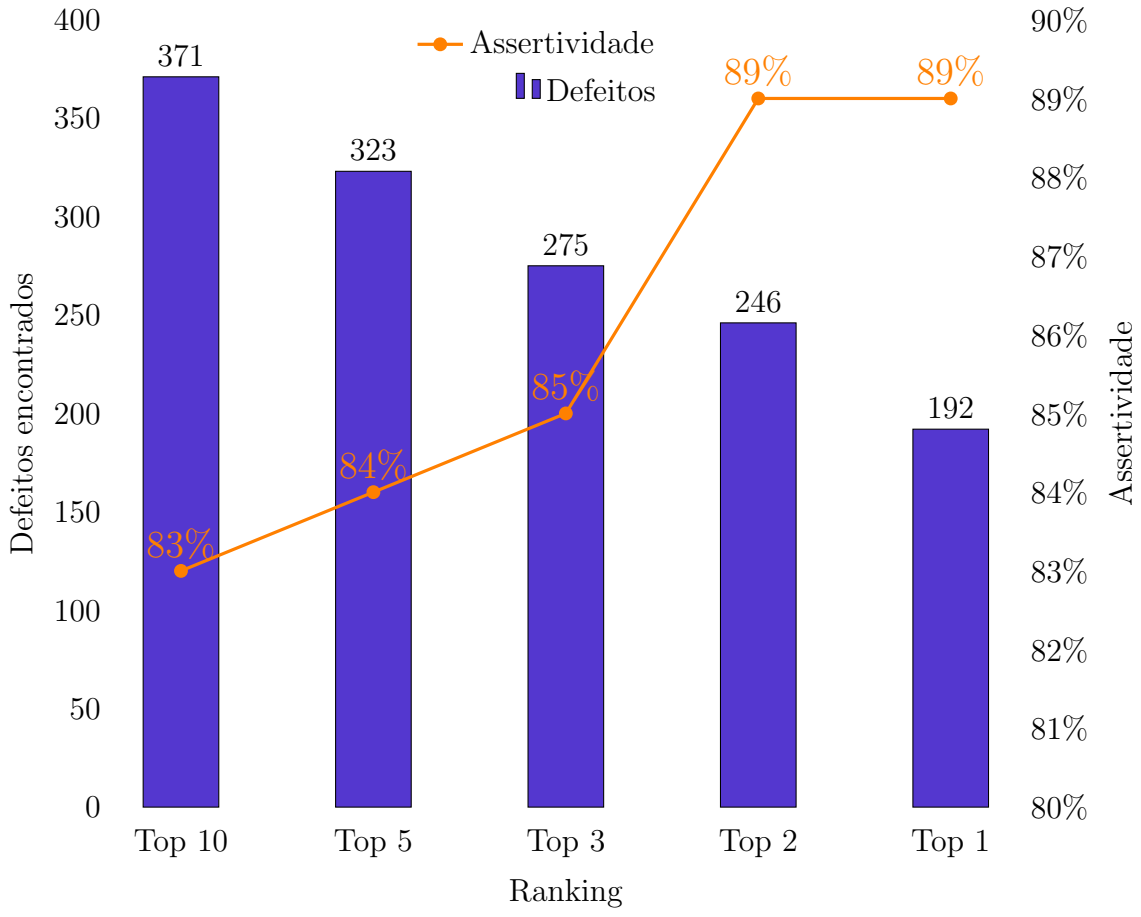
- Para o *ranking* Top 10 Score, não houve perda em 3 dos 17 projetos, com média de assertividade de 83% e 86% de mediana.

Figura 12 – Assertividade das ADUs não-limitadas



Fonte: Dennis Lopes, 2022

Figura 13 – Top N Score - Assertividade x Número de Defeitos



Fonte: Dennis Lopes, 2022

Tabela 16 – Assertividade Top N com ADUs não-limitadas

Programa	**Top 1**	**Top 2**	**Top3**	**Top 5**	**Top 10**
Chart	100%	100%	100%	100%	100%
Cli	90%	86%	84%	84%	84%
Closure	96%	96%	94%	94%	95%
Codec	80%	75%	82%	77%	73%
Collections	0%	0%	0%	0%	0%
Compress	91%	93%	89%	85%	85%
Csv	100%	100%	100%	100%	100%
Gson	100%	100%	100%	100%	89%
JacksonCore	100%	90%	83%	86%	87%
JacksonDatabind	85%	83%	82%	82%	82%
JacksonXml	100%	100%	100%	100%	100%
Jsoup	87%	89%	86%	83%	81%
JxPath	100%	91%	82%	84%	86%
Lang	87%	89%	86%	87%	87%
Math	85%	87%	85%	83%	80%
Mockito	100%	100%	86%	81%	81%
Time	100%	100%	100%	100%	93%
Média	**89%**	**87%**	**85%**	**84%**	**83%**
Mediana	**96%**	**91%**	**86%**	**85%**	**86%**

Fonte: Dennis Lopes, 2022

- Para o *ranking* Top 3 Score, não houve perda em 5 dos 17 projetos, com média de assertividade de 85% e 86% de mediana.
- Para o *ranking* Top 1 Score, não houve perda em 8 dos 17 projetos, com média de assertividade de 89% e 96% de mediana.

À medida em que se aumenta o número de *rankings* considerados, diminui-se a assertividade. Parnin e Orso (PARNIN; ORSO, 2011) sugerem que, geralmente, os programadores não investigam muitas posições do *ranking* durante a depuração. Essa característica reforça a necessidade de uma boa assertividade no *Top 1 Score*. Outro estudo, realizado com estudantes de graduação, mostrou que a localização baseada em espectros foi útil apenas quando os defeitos estavam localizados nas primeiras posições do *ranking* (XIE *et al.*, 2016). A assertividade das ADUs não-limitadas para o Top 1 Score é de 89% em média, com mediana de 96%, o que indica sua compatibilidade com os cenários mais comuns de depuração.

O percentual de perda, sumarizado na Tabela 17, é relacionado aos cenários de versões dos programas nos quais é possível localizar o defeito com o uso de uma ADU

Tabela 17 – Percentual de perda

Programa	**# Defeitos ADUs Subsumidas**	**Perda**
Chart	1	8%
Cli	5	15%
Closure	6	4%
Codec	2	18%
Collections	0	0%
Compress	4	11%
Csv	0	0%
Gson	1	9%
JacksonCore	2	12%
JacksonDatabind	6	6%
JacksonXml	0	0%
Jsoup	9	12%
JxPath	0	0%
Lang	4	18%
Math	12	17%
Mockito	3	10%
Time	2	11%
Média		**9**%
Mediana		**10**%

Fonte: Dennis Lopes, 2022

subsumida, mas não é possível encontrar esse mesmo defeito a partir do conjunto de ADUs não-limitadas. Os detalhes desses defeitos aparecem na Tabela 14, na coluna Subsumida'.

QP2: Qual a perda em termos de ranqueamento?

Existem situações nas quais o defeito mesmo tendo sido encontrado primeiro por uma ADU subsumida, este também é encontrado por uma ADU não-limitada porém, em uma posição inferior no *ranking*, isto é, a ADU não-limitada que localiza o defeito aparece depois de uma ADU subsumida que também o localizou. Na Tabela 14, essas ocorrências aparecem na coluna *Subsumida*.

Todos os defeitos indicados na coluna *Subsumida* foram encontrados primeiro por uma ADU subsumida, mas também foram encontrados por uma ADU não-limitada que possui um *score* pior do que *score* da ADU subsumida. Observe-se, entretanto, que há poucos casos em que o defeito aparece melhor ranqueado por ADUs subsumidas para os programas do repositório Defects4J. Considerando os programas do *Defects4J*, isso

Tabela 18 – Perda em termos de ranqueamento

Programa	**# Defeitos ADUs Subsumidas**	**Percentual de Perda**
Chart	0	0%
Cli	2	6%
Closure	16	11%
Codec	2	18%
Collections	0	0%
Compress	3	8%
Csv	0	0%
Gson	0	0%
JacksonCore	2	12%
JacksonDatabind	20	20%
JacksonXml	1	25%
Jsoup	8	11%
JxPath	6	27%
Lang	0	0%
Math	6	9%
Mockito	5	16%
Time	0	0%
Média		**10**%
Mediana		**9**%

Fonte: Dennis Lopes, 2022

aconteceu para mais de 10 versões para apenas dois programas (ver as linhas referentes aos programas Closure e JacksonDatabind na Tabela 14).

Consideramos perda em termos de ranqueamento as situações em que os defeitos são encontrados pelos dois subconjuntos (não-limitadas e subsumidas), mas onde a ADU subsumida tem um melhor *score*. A Tabela 18 mostra a frequência em que essa situação ocorre para os defeitos do arcabouço *Defect4J* considerados. O percentual médio de perda foi de 10% e em 6 dos 17 projetos não houve nenhuma perda.

QP3: Qual a diminuição em termos de linhas inspecionadas?

Espera-se que o uso de um subconjunto menor de ADUs — as ADUs não-limitadas — para localização dos defeitos leve o programador a inspecionar menos linhas de código, de forma a otimizar o seu tempo de depuração.

A Tabela 19 detalha o número médio de linhas a serem inspecionadas ao utilizarmos apenas as ADUs não-limitadas e ao utilizarmos o conjunto total de ADUs, considerando

Tabela 19 – Número médio de linhas inspecionadas

Programa	**# Defeitos**	**ADUs NL**	**ADUs todas**
Chart	23	164,22	228,13
Cli	39	65,26	86,15
Closure	172	803,54	929,33
Codec	18	42,00	58,61
Collections	4	58,50	74,75
Compress	46	113,07	167,57
Csv	16	44,69	66,75
Gson	17	105,59	150,12
JacksonCore	22	135,50	190,18
JacksonDatabind	109	610,83	737,45
JacksonXml	6	123,33	158,50
Jsoup	93	198,28	261,05
JxPath	22	547,23	799,09
Lang	29	42,93	61,00
Math	102	91,39	142,16
Mockito	38	86,66	101,32
Time	25	178,72	229,20

Fonte: Dennis Lopes, 2022

que o desenvolvedor irá investigar todo o *ranking*. Ainda que esses dois subconjuntos tenham intersecção não vazia, houve uma diminuição de linhas investigadas.

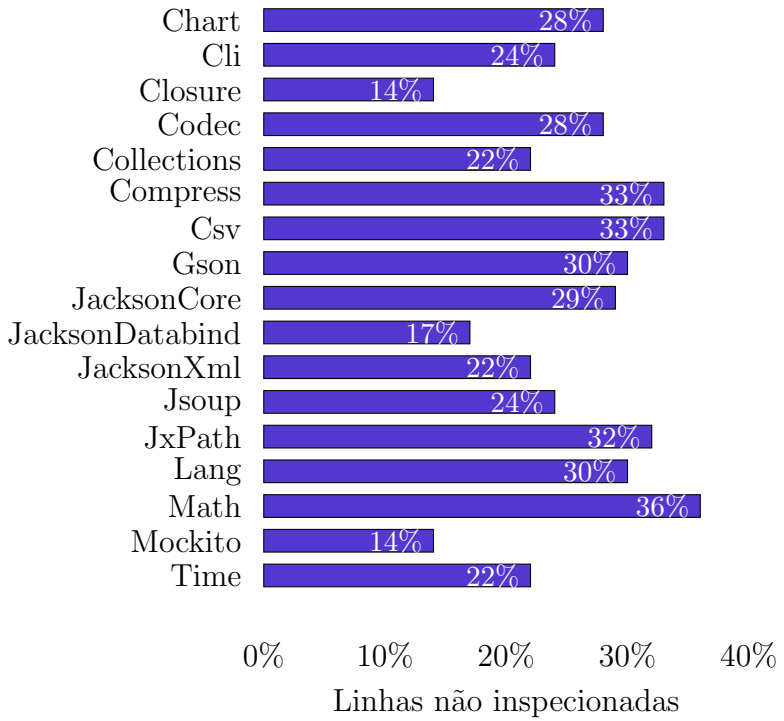
A Figura 14 apresenta, para cada programa, a porcentagem de diminuição do número de linhas investigadas, que variou de 14% a 36%, em média.

Ao consideramos apenas os *Top N rankings*, a porcentagem variou, na média, de 20% (Top 1) a 29% (Top 10). Levando em conta que o programador finaliza a localização do defeito ao encontrá-lo e, dado o corte feito nos *rankings* a partir do *score* que localiza o defeito, esse é o resultado que mais se aproxima da realidade de um processo empírico de depuração. Os detalhes da diminuição média para cada um dos programas estão detalhados na Tabela 20.

5.6 Ameaças à validade

As principais ameaças à validade do experimento apresentado são as relacionadas à validade interna, à validade externa e à validade de conclusão.

Figura 14 – Percentual de linhas não inspecionadas



Fonte: Dennis Lopes, 2022

Tabela 20 – Diminuição de linhas inspecionadas com ADUs não-limitadas – Top N

Programa	**Top 1**	**Top 2**	**Top3**	**Top 5**	**Top 10**
Chart	19%	23%	22%	34%	32%
Cli	20%	22%	23%	23%	30%
Closure	19%	22%	22%	26%	27%
Codec	12%	15%	16%	19%	18%
Collections	0%	26%	24%	22%	22%
Compress	19%	23%	27%	29%	29%
Csv	23%	21%	27%	29%	34%
Gson	28%	33%	38%	34%	38%
JacksonCore	30%	42%	43%	42%	47%
JacksonDatabind	23%	25%	26%	26%	26%
JacksonXml	24%	26%	21%	20%	20%
Jsoup	15%	25%	26%	27%	27%
JxPath	14%	16%	15%	15%	17%
Lang	22%	26%	27%	27%	27%
Math	31%	31%	36%	36%	40%
Mockito	14%	17%	20%	20%	23%
Time	25%	29%	27%	28%	35%
Média	**20%**	**25%**	**26%**	**27%**	**29%**
Mediana	**20%**	**25%**	**26%**	**27%**	**27%**

Fonte: Dennis Lopes, 2022

As ferramentas utilizadas e os *scripts* criados para gerar os dados do experimento são uma ameaça interna à validade. Os resultados foram verificados para algumas versões de programas menores; no entanto, programas com milhares de linhas de código (e.g., Math e Closure) não permitiram essa verificação. Outra ameaça interna é o fato de a ferramenta Jaguar não coletar cobertura quando ocorre uma exceção no método que contém o defeito. Uma nova versão da ferramenta que trata estas situações está em desenvolvimento e os dados serão coletados novamente. Os dados da nova versão poderão permitir a localização de defeitos adicionais.

No experimento apresentado, um defeito foi considerado localizado quando uma das linhas listadas como parte do defeito, segundo o arcabouço *Defect4J*, fazia parte de uma das linhas das ADUs. No entanto, muitos defeitos consistem em *omissão* de código, isto é, o código da versão corrigida incluiu novas linhas de código em determinados pontos do programa. As linhas associadas a esses pontos são considerados como parte do defeito, mas no entanto, raramente essas linhas são visitadas por espectros estruturais (linhas, ramos, ADUs).

Pearson *et al.* (2017) consideram um conjunto de linhas *candidatas*, próximas ao código omitido, como parte do defeito. A não utilização das linhas candidatas no experimento é uma ameaça à validade de conclusão pois os resultados são uma indicação subestimada da habilidade das ADUs na localização de defeitos. Outra possível ameaça à validade de conclusão é a análise dos dados utilizando somente estatística descritiva. Isso ocorreu porque a comparação realizada é entre todas ADUs e um subconjunto seu (de ADUs não-limitadas), o que impediu a aplicação de outras técnicas de análise estatística (e.g., teste de hipóteses).

O experimento utilizou os programas e suas versões defeituosas do arcabouço *Defects4J* para reduzir a ameaça à validade externa. Os programas do arcabouço compreendem software de diferentes características e tamanhos, similares àqueles desenvolvidos pelas indústria. Além disso, a maior parte das versões defeituosas foram utilizadas no experimento. Apesar de os programas do arcabouço Defects4J não incluir toda a diversidade possível de softwares, ele inclui uma amostra importante de programas com defeitos reais.

5.7 Considerações finais

Conforme visto na Figura 14, o desenvolvedor depurando um programa irá inspecionar um número de linhas menor se utilizar as ADUs não-limitadas ao invés de todas as ADUs. A diminuição média é de 26%, sendo a menor diminuição observada de 14% do número de linhas em todos os projetos do Defects4J. Esse resultado sugere um menor esforço de depuração, mas essa hipótese demanda um estudo futuro com usuários para sua confirmação.

Outro resultado a ser destacado no experimento foi a alta assertividade do conjunto de ADUs não-limitadas para o *Top 1 Score*: 8 dos 17 projetos tiveram 100% de assertividade apenas com o conjunto de ADUs não-limitadas, conforme observado na Tabela 16. Além disso, a média e a mediana para essa faixa foi de 89% e 96% respectivamente.

Assim, para o cenário mas provável de utilização das ADUs não-limitadas, ocorre uma diminuição do número de linhas a serem investigadas com uma perda pequena da eficácia na localização de defeitos. Nesse sentido, os resultados indicam que as ADUs não-limitadas contribuem para manter o engajamento do desenvolvedor na localização baseada em espectros, visto que diminui o número de linhas a serem investigadas nas primeiras posições do *ranking* com pouca perda de eficácia.

6 Conclusões

Esse capítulo apresenta o resumo dos resultados obtidos, nossas contribuições e os trabalhos futuros.

6.1 Resumo

Os programadores no seu dia a dia são confrontados com indicações sintomáticas de problemas, isto é, falhas, no software ao avaliar os resultados de um teste; porém, a relação entre uma falha e a sua causa interna, isto é, o defeito, pode não ser óbvia. O processo mental que conecta um sintoma a uma causa é a depuração de software (PRESSMAN, 2001).

Dada a execução de uma *suite* de testes, que possui testes que passam e que falham, a técnica de localização de defeitos baseada em espectros utiliza heurísticas para determinar, com base na frequência com que os espectros foram exercitados pela *suite*, quais localizações são mais "suspeitas", ou seja, quais localizações possuem maior probabilidade de estarem erradas e consequentemente associadas ao defeito apresentado (PEARSON *et al.*, 2017).

As técnicas de localização de defeitos baseadas em espectros são promissoras por terem obtido resultados significativos na redução do espaço de busca dos defeitos a um custo em tempo de execução razoável (SOUZA; CHAIM; KON, 2016). Porém, dada a forma com que os *scores* são calculados, com base na frequência com que determinados espectros são estimulados pelos casos de teste, os resultados podem se mostrar enviesados a depender das relações entre os espectros.

Como exemplo, podemos citar a quantidade de vezes em que o fluxo principal de uma aplicação é executado pelos casos de teste. Esse fluxo é executado mais do que qualquer outro fluxo e essa frequência elevada dá-se puramente pela relação de dependência entre os fluxos. Ao trabalharmos com associações definição uso (ADU) essa dependência reflete-se na relação de subsunção entre as ADUs não-limitadas e a ADUs subsumidas. Cada ADU é exercitada ou coberta por caminhos entre a definição e o subsequente uso de uma variável do programa. A relação de subsunção de fluxo de dados, assim como no caso de dependência entre um fluxo principal e os alternativos, estabelece uma relação de dependência entre os caminhos determinados por uma ADU subsumida e os caminhos determinados por uma ADU não-limitada.

Em linhas gerais, as ADUs subsumidas é um subconjunto de ADUs que, em dadas condições, sempre são exercitadas (cobertas) quando um conjunto minimal de ADUs não-limitadas o é. Assim, ao garantir o exercício desse conjunto minimal de ADUs não-limitadas, garante-se a exercício de todas as ADUs em determinadas condições. O uso de apenas ADUs não-limitadas pode, por hipótese, remover os ruídos adicionados ao cálculo das métricas de associação por conta da relação de subsunção. Esta dissertação apresenta os resultados de um experimento realizado para avaliar o impacto da relação de subsunção de fluxo de dados na localização de defeitos baseada em espectros.

O impacto da relação de subsunção de fluxo de dados foi avaliada com o uso do repositório *Defects4J* (JUST; JALALI; ERNST, 2014). O *Defects4J* é um arcabouço de execução de testes que permite a reprodução e o isolamento de defeitos reais encontrados em projetos de software reais. Para cada defeito catalogado no arcabouço, é possível recuperar o ambiente original no qual esse defeito foi encontrado e fazer o uso de conjuntos de testes que incluem pelo menos um caso de teste que expõe o defeito.

Um dos grandes trunfos do arcabouço *Defects4J* é o de fornecer um *benchmark* que é comparável e reproduzível entre diferentes experimentos e também o uso de defeitos reais de software. Os conjuntos de teste fornecidos com os defeitos do arcabouço *Defects4J* foram utilizados para o cálculo de métricas de associação que permitiram a criação dos *rankings* de suspeição usando a técnica de localização de defeitos baseada em espectros (*Spectrum-based Fault localization* — SBFL).

De um total de 835 defeitos catalogados no *Defects4J*, foram utilizados para o experimento 781 defeitos (representando 93,53% do total de defeitos). O ambiente de cada um desses defeitos foi reproduzido e com o uso da ferramenta Jaguar linha de comando foi calculada a frequência de execução das ADUs para cada uma das versões defeituosas. Com a frequência calculada pela Jaguar, utilizou-se *scripts* desenvolvidos em Python para o cálculo do *ranking* baseado na métrica de associação *Ochiai* (ABREU; ZOETEWEIJ; GEMUND, 2007) para todas as ADUs e apenas para as ADUs não-limitadas.

Com o uso do arcabouço do *Defects4J* foi possıvel avaliar a hipótese de que o conjunto de ADUs não-limitadas pode levar o desenvolvedor a investigar menos código, aumentando a sua produtividade, sem no entanto diminuir de maneira substancial o número de defeitos encontrados. As métricas determinadas para essa avaliação foram a assertividade do conjunto de ADUs na localização de defeitos, bem como o percentual de diminuição no numero de linhas investigadas pelo desenvolvedor. A métrica de assertividade

representa o percentual de defeitos que são encontrados primeiro por ADUs não-limitadas e, nessa situação, os *scores* de ADUs subsumidas são irrelevantes para encontrar o defeito uma vez que o desenvolvedor já encontrou o defeito com a ADU não-limitada e encerrou seu processo de depuração. Considerando os programas do *Defects4J* separadamente, a assertividade das ADUs não-limitadas variou de 64% a 100%, sendo 76% a média.

No experimento também foram avaliados os *rankings Top10, Top5, Top3, Top2* e *Top1*. O propósito foi avaliar como a assertividade variava em relação a diferentes segmentos do *ranking* geral, isto é, para o primeiros N, onde N varia de 1,2,3,5 até 10, *scores* do *ranking* geral. Pôde-se observar que a assertividade foi inversamente proporcional ao tamanho do segumento do *ranking*. Esse resultado sugere que quanto menor o segmento do *ranking* utilizado, maior a chance de as ADUs não-limitadas determinadas por esse segmento revelarem o defeito. O resultado variou de 83% para o *Top10*, chegando a 89% para o *Top1*. Esses resultados estão disponíveis na Figura 13.

O percentual de diminuição no número de linhas inspecionadas variou de 20% a 29%, sendo diretamente proporcional ao tamanho do segmento do *ranking* (considerando os segmentos *Top 1, Top 2, Top 3, Top 5* e *Top 10*). Em média, para o *Top 1 ranking* foi obtido uma redução mínima de 20% no número de linhas inspecionadas.

Esses dois últimos resultados, assertividade *Top N* das ADUs não-limitadas e diminuição no número de linhas inspecionadas, indicam que *rankings* menores foram mais assertivos na localização dos defeitos e permitiram com que o desenvolvedor inspecionasse um menor número de linhas de código.

Como já citado anteriormente, o trabalho de Parnin e Orso (PARNIN; ORSO, 2011) sugere que os programadores não investigam muitas posições do *ranking* durante a depuração. Dado esse comportamento do desenvolvedor, o experimento mostrou que o uso das ADUs não-limitadas obteve os melhores resultados associado aos cenários mais realísticos de depuração.

6.2 Contribuições

Esse trabalho trouxe contribuições para o entendimento do impacto das relações de subsunção na localização de defeito baseada em espectros de fluxo de dados. Elas discutidas em detalhes nesta dissertação e publicadas no trabalho *Data flow Subsumption*

and its Impact on Spectrum-based Fault Localization, de autoria de Dennis Lopes Silva, Higor Amario de Souza e Marcos Lordello Chaim, apresentado no *7th Brazilian Symposium on Systematic and Automated Software Testing*, realizado de 3 a 7 de outubro de 2022 em Uberlândia, MG, Brasil.

As principais contribuições dizem respeito ao entendimento da perda de assertividade e da redução do número de linhas investigadas quando são utilizadas as ADUs não-limitadas para localização de defeitos. Essas duas contribuições são resumidas a seguir.

6.2.1 Análise da perda de assertividade dos subconjuntos de ADUs não-limitados

A perda de informação, que leva à não localização de defeitos, refere-se às situações nas quais as métricas de associação com o uso do subconjunto de ADUs não-limitadas é incapaz de encontrar defeitos que são encontrados com o uso de todas as ADUs, ou seja, quando o defeito só é encontrado por meio de uma ADU subsumida.

Os resultados do experimento indicam que o uso das ADUs não-limitadas é promissor na detecção de defeitos. Houve perda de assertividade devido ao uso das ADUs não-limitadas, isto é, alguns defeitos não são localizados utilizando essas ADUs. Porém, essa perda é limitada, em particular, quando consideramos apenas os primeiros *scores* do *ranking* (*TopN*). No geral houve uma perda, em média, de 10% dos defeitos utilizando a estratégia de cálculo que considera apenas as ADUs não-limitadas (Tabela 17).

Também houve perda relacionada ao ranqueamento quando o defeito, tendo sido encontrado por uma ADU não-limitada, também o é por uma ADU subsumida, só que em melhor posição de *ranking*. Essa perda, ainda que não definitiva, já que o defeito também é encontrado por uma ADU não-limitada, variou de 0% a 27% com uma média geral de 10% de perda, conforme se observa na Tabela 18.

Por outro lado, as ADUs não-limitadas são mais convenientes para o cálculo das métricas de associação pois demandam menor tempo de processamento por serem em menor número e requerem que menos código seja examinado pelo desenvolvedor.

6.2.2 Redução das linhas de código examinadas

Os resultados do experimento indicam que o uso de ADUs não-limitadas reducem o número de linhas a serem investigadas pelo desenvolvedor. Conforme apresentado na Figura 14, o desenvolvedor irá inspecionar um número de linhas menor se utilizar as ADUs não-limitadas ao invés de todas as ADUs. A diminuição média é de 26%, sendo a menor diminuição observada de 14% e a maior de 36% para os projetos do *Defects4J*, considerando todo o *ranking*. Considerando os conjuntos *TopN*, a redução do número de linhas a serem investigadas varia de 20% a 29%. Esses resultados indicam um menor esforço de depuração, mas essa hipótese demanda um estudo futuro com usuários para sua confirmação.

6.3 Trabalhos Futuros

Este estudo abre algumas possibilidades para trabalhos futuros que envolvam a relação de subsunção e a localização de defeitos baseada em espectros de fluxo de dados. Em particular, as seguintes linhas de pesquisa podem ser desenvolvidas:

1. Investigar de maneira analítica, a partir dos programs do repositório *Defects4J*, quais são as condições ou estruturas que levam à perda de eficácia das ADUs não-limitadas. Em outras palavras, que condições ou estruturas presentes no programa podem afetar os resultados do *ranking* levando à perda de assertividade ou de ranqueamento?

2. Explorar os resultados de assertividade quando são considerados como critérios de assertividade não apenas a existência da linha defeituosa na definição da ADU, mas também a existência de linhas vizinhas a essa linha defeituosa. Para isso, poderiam ser considerados as linhas candidatas, como no trabalho conduzido por Campos, Abreu, Fraser e d'Amorim (2013).

3. Avaliação dos resultados observados no experimento em um estudo com usuários. As questões de pesquisa a serem investigadas são: (1) os desenvolvedores encontram menos defeitos se utilizarem somente ADUs não-limitadas em comparação com o uso de todas ADUs? (2) Eles levam menos tempo?

Referências[1]

ABREU, R.; ZOETEWEIJ, P.; GEMUND, A. J. V. On the accuracy of spectrum-based fault localization. *Testing: Academic and Industrial Conference Practice and Research Techniques*, IEEE, p. 89–98, 2007. Citado 2 vezes nas páginas 25 e 76.

ABREU, R.; ZOETEWEIJ, P.; GOLSTEIJN, R.; GEMUND, A. J. C. A practical evaluation of spectrum-based fault localization. *The Journal of Systems and Software*, v. 82, p. 1780–1792, 2009. Citado na página 48.

AGRAWAL, R.; IMIELI'NSKI, T.; ARUN, S. Mining association rules between sets of items in large databases. In: . [S.l.: s.n.], 1993. v. 22, p. 207–216. ISBN 0897915925. Citado na página 16.

AMMANN, P.; DELAMARO, M. E.; OFFUTT, J. Establishing theoretical minimal set of mutants. *IEEE Seventh International Conference on Software Testing, Verification and Validation*, IEEE, 2014. Citado na página 15.

AMMANN, P.; OFFUTT, J. *Introduction to Software Testing*. 2. ed. [S.l.]: Cambridge University Press, 2017. Citado 2 vezes nas páginas 12 e 30.

ARAKI, K.; FURUKAWA, Z.; CHENG, J. A general framework for debugging. *IEEE Software Magazine*, v. 8, n. 3, p. 14–20, 1991. Citado na página 12.

ARAUJO, R. P. A.; CHAIM, M. L. Data-flow testing in the large. *Software Testing, Verification and Validation (ICST)*, IEEE, p. 81–90, 2014. Citado 4 vezes nas páginas 15, 21, 48 e 59.

BARR, A. *The problem with Software - Why smart engineers write bad code*. 1. ed. [S.l.]: MIT Press, 2018. Citado 2 vezes nas páginas 12 e 13.

CAMPOS, J.; ABREU, R.; FRASER, G.; D'AMORIM, M. Entropy-based test generation for improved fault localization. *28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, p. 257–267, 2013. Citado 4 vezes nas páginas 38, 41, 42 e 79.

CHAIM, M. L. *Depuração de Programas Baseada em Informação de Teste Estrutural*. Tese (Doutorado) — Universidade Estadual de Campinas, 2001. Citado na página 13.

CHAIM, M. L.; ARAUJO, R. P. A. An efficient bitwise algorithm for intra-procedural data-flow testing coverage. *Information Processing Letters*, v. 113, n. 8, p. 293–300, 2013. Citado 3 vezes nas páginas 15, 19 e 22.

CHAIM, M. L.; BARAL, K.; OFFUTT, J.; CONCILIO, M.; ARAúJO, R. P. A. Efficiently finding data flow subsumptions. *IEEE Conference on Software Testing, Verification and VAlidation*, 2021. Citado 2 vezes nas páginas 36 e 50.

CHAIM, M. L.; MALDONADO, J. C.; JINO, M. A debugging strategy based on requirements of testing. In: MAINTENANCE, S. E. C. on S.; REEGINEERING (Ed.). [S.l.: s.n.], 2003. p. 160–169. Citado na página 21.

[1] De acordo com a Associação Brasileira de Normas Técnicas. NBR 6023.

CHAIM, M. L.; MALDONADO, J. C.; JINO, M. A debugging strategy based on the requirements of testing. *Journal of Software Maintenance and Evolution: Research and Practice*, v. 16, n. 4-5, p. 277–308, 2004. Citado na página 48.

CHEN, M. Y.; KICIMAN, E.; FRATKIN, E.; FOX, A.; BREWER, E. Pinpoint: Problem determination in large, dynamic internet services. In: SYSTEMS, P. I. C. on D.; NETWORKS (Ed.). [S.l.]: IEEE, 2002. p. 595–604. Citado na página 48.

CLARKE, L. A.; PODGURSKI, A.; RICHARDSON, D. J.; ZEIL, S. J. A comparison of data flow path selection criteria. In: *ICSE*. Washington, DC, USA: IEEE Computer Society Press, 1985. (ICSE '85), p. 244–251. ISBN 0818606207. Citado na página 15.

COLLOFELLO, J. S.; COUSINS, L. Towards automatic software fault localization through decision-to-decision path analysis. *Proc. Nat. Comput. Conf.*, p. 539–544, 1987. Citado na página 14.

DANIEL, P.; SIM, K. Y. Noise reduction for spectrum-based fault localization. *Internationl Journal of Control and Automation (IJCA)*, Springer, v. 6, n. 5, p. 117–126, 2013. Citado 5 vezes nas páginas 37, 38, 42, 43 e 46.

DEBROY, V.; WONG, W. E. On the consensus-based application of fault localization techniques. *IEEE 35th Annual Computer Software and Applications Conference Workshops*, p. 506–511, 2011. Citado na página 48.

DELAMARO, M. E.; CHAIM, M. L.; VINCENZI, A. M. R. *Técnicas e ferramentas de teste de software*. 1. ed. [S.l.]: PUC Rio, 2010. Citado 3 vezes nas páginas 22, 23 e 24.

DENMAT, T.; DUCASSé, M.; RIDOUX, O. Data mining and cross-checking of execution traces: A re-interpretation of jones, harrold and stasko test information visualization. *International Conference on Automated Software Engineering (ASE)*, ACM, 2005. Citado 5 vezes nas páginas 15, 16, 44, 45 e 46.

FRASER, G.; ARCURI, A. Evosuite: Automatic test suite generation for object-oriented software. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. [S.l.]: ACM, 2011. (ESEC/FSE), p. 416–419. ISBN 978-1-4503-0443-6. Citado na página 41.

GONZALEZ, A. *Automatic error detection techniques based on dynamic invariants*. Tese (Doutorado) — Delft University of Technology, 2007. Citado na página 48.

HARROLD, M. J.; ROTHERMEL, G. Performing dataflow testing on classes. In: . [S.l.: s.n.], 1994. v. 19, p. 154–163. ISBN 0897916913. Citado na página 26.

HUTCHINS, M.; FOSTER, H.; GORADIA, T.; OSTRAND, T. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In: *16th International Conference on Software Engineering*. [S.l.: s.n.], 1994. (ICSE), p. 191–200. ISBN 0-8186-5855-X. Citado 2 vezes nas páginas 41 e 42.

JONES, J. A.; HARROLD, M. J.; STASKO, J. Visualization of test information to assist fault localization. In: *Proceedings of the 24th ACM/IEEE International Conference on Software Engineering*. [S.l.]: ACM, 2002. p. 467–477. Citado 3 vezes nas páginas 25, 44 e 45.

JONES, J. A.; HARROLD, M. J.; STASKO, J. Visualization of test information to assist fault localization. In: *Proceedings of the 24th ACM/IEEE International Conference on Software Engineering*. [S.l.: s.n.], 2002. Citado na página 48.

JUST, R.; JALALI, D.; ERNST, M. D. Defects4j: A database of existing faults to enable controlled testing studies for java program. *ISSTA*, IEEE, 2014. Citado 3 vezes nas páginas 55, 56 e 76.

LAWRANCE, J.; BOGART, C.; BURNETT, M.; BELLAMY, R.; RECTOR, K.; FLEMING, S. D. How programmers debug, revisited: An information foraging theory perpective. *IEEE Transactions on Sofware Engineering*, IEEE, v. 39, n. 2, p. 197–215, 2013. Citado na página 12.

LIBLIT, B. *et al.* Scalable statistical bug isolation. In: *Conf. Programm. Language Design Implementation*. [S.l.]: ACM, 2005. p. 15–26. Citado na página 14.

LIU, C. *et al.* Statistical debugging: A hypothesis testing-based approach. *Trans. Soft. Eng.*, IEEE, v. 32, n. 10, p. 831–848, 2006. Citado na página 14.

MA, C.; ZHANG, Y.; LU, Y.; WANG, Q. Uniformly evaluating and comparing ranking metrics for spectral fault localization. *14th International Conference on Quality Software*, p. 315–320, 2014. Citado na página 38.

MAO, X.; LEI, Y.; DAY, Z.; QI, Y.; WANG, C. Slice-based statistical fault localization. *Journal of Systems and Software*, Elsevier Inc, p. 51–62, 2014. Citado na página 21.

MARRé, M.; BERTOLINO, A. Unconstrained duas and their use in achieving all-uses coverage. *International Symposium on Software Testing and Analysis*, ACM, 1996. Citado 2 vezes nas páginas 15 e 35.

MARRé, M.; BERTOLINO, A. Using spanning sets for coverage testing. *IEEE Transactions on Software Engineering*, v. 29, p. 974–984, 2003. Citado 3 vezes nas páginas 33, 34 e 35.

MATHUR, A. P. *Foundations of Software Testing - Second Edition*. West Lafayette, Indiana, EUA: Pearson India, 2013. Citado na página 29.

NAISH, L.; LEE, H.; RAMAMOHANARAO, K. A model for spectra-based software diagnosis. *ACM Trans. Softw. Eng. Methodol.*, v. 20, p. 11, 08 2011. Citado na página 48.

NAISH, L.; LEE, H. J.; RAMAMOHANARAO, K. Spectral debugging with weights and incremental ranking. In: *Asia Pacific Software Engineering Conference, APSEC*. [S.l.: s.n.], 2009. p. 168–175. Citado na página 48.

NAISH, L.; LEE, H. J.; RAMAMOHANARAO, K. Statements versus predicates in spectral bug localization. In: *Asia Pacific Software Engineering Conference, APSEC*. Koa, Hawaii: [s.n.], 2010. p. 375–384. Citado na página 38.

NETO, F. M. *Entropias Generalizadas e os Fundamentos Estatísticos da Termondinâmica*. Dissertação (Mestrado) — Universidade de Brasília, 2006. Citado na página 40.

NIST. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. [S.l.], 2002. Citado na página 12.

PARNIN, C.; ORSO, A. Are automated debugging techniques actually helping programmers? In: *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*. [S.l.: s.n.], 2011. (ISSTA'11), p. 199–209. Citado 3 vezes nas páginas 63, 68 e 77.

PEARSON, S.; CAMPOS, J.; JUST, R.; FRASER, G.; ABREU, R.; ERNST, M. D.; PANG, D.; KELLER, B. Evaluating and improving fault localization. In: *Proceedings of the 39th International Conference on Software Engineering*. [S.l.: s.n.], 2017. (ICSE'17), p. 609–620. Citado 3 vezes nas páginas 47, 73 e 75.

PRESSMAN, R. S. *Software Engineering: a practitioner's approach*. [S.l.]: McGraw-Hill, 2001. Citado na página 75.

RAPPS, S.; WEYUKER, E. J. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11, n. 4, p. 367–375, 1985. Citado 3 vezes nas páginas 14, 15 e 25.

RIBEIRO, H. L. *On the use of control and data-flow in fault localization*. Dissertação (Mestrado) — Universidade de São Paulo, 2016. Citado 2 vezes nas páginas 26 e 48.

RIBEIRO, H. L.; ROBERTO, P. A.; CHAIM, M. L. Evaluating data-flow coverage in spectrum-base fault localization. *arXiv:1906.11715*, ACM, 2019. Citado 2 vezes nas páginas 14 e 15.

ROYCHOWDHURY, S.; KHURSHID, S. A family of generalized entropies and its application to software fault localization. *6th IEEE International Conference Intelligent Systems*, IEEE, p. 368–373, 2012. Citado 3 vezes nas páginas 38, 40 e 41.

SANTELICES, R. *et al.* Lightweight fault-localization using multiple coverage types. *International Conference on Software Enginnering (ICSE)*, IEEE, p. 56–66, 2009. Citado na página 15.

SILVA, L. A.; PEREZ, S. M.; BOSCARIOLI, C. *Introdução à Mineração de Dados Com Aplicações em R*. 4. ed. [S.l.]: Elsevier, 2016. Citado na página 44.

SOMMERVILLE, I. *Software Engineering*. 10. ed. [S.l.]: Pearson Education Limited, 2016. Citado na página 12.

SOREMEKUN, E.; KIRSCHNER, L.; BöHME, M.; ZELLER, A. Locating faults with program slicing: An empirical analysis. *arXiv:201.03008v1*, ACM, 2021. Citado 3 vezes nas páginas 38, 39 e 46.

SOUZA, H. A.; CHAIM, M. L.; KON, F. Spectrum-based software fault localization: A survey of techniques, advances, and challenges. *arXiv preprint arXiv:1607.04347*, 2016. Citado 4 vezes nas páginas 13, 14, 25 e 75.

VESSEY, I. Expertise in debugging computer programs: An analysis of the content of verbal protocols. *International Journal on Man-Machine Studies*, IEEE, v. 23, n. 5, p. 459–494, 1985. Citado na página 13.

WEISER, M. Program slicing. In: *International Conference os Software Engineering*. [S.l.: s.n.], 1981. p. 439–449. Citado na página 14.

WONG, W. E.; GAO, R.; LI, Y.; ABREU, R.; WOTAWA, F. A survey on software fault localization. *11th International Conference on Software Testing, Verification and Validation (ICST)*, 2016. Citado 4 vezes nas páginas 12, 13, 14 e 38.

XIALON, J.; JIANG, S.; CHEN, X.; WANG, X.; ZHANG, Y.; CAO, H. Hsfal: Efective fault localization using hybrid spectrum of full slices and execution slices. *The Journal of Systems and Software*, IEEE, p. 15, 2013. Citado 3 vezes nas páginas 38, 39 e 46.

XIE, X.; CHEN, T. Y.; KUO, F.; BAOWEN, B. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *Association for Computing Machinery*, v. 22, n. 4, 2013. Citado na página 38.

XIE, X.; LIU, Z.; SONG, S.; CHEN, Z.; XUAN, J.; XU, B. Revisit of automatic debugging via human focus-tracking analysis. In: *Proceedings of the 38th International Conference on Software Engineering*. [S.l.: s.n.], 2016. (ICSE'16), p. 808–819. Citado na página 68.

XU, J.; ZHANG, Z.; CHAN, W.; TSE, T.; LI, S. A general noise-reduction framework for fault localization of java programs. *Information and Software Technology*, v. 55, n. 5, p. 880–896, 2013. ISSN 0950-5849. Citado na página 48.

YOU, Y. S.; HUANG, C. Y.; PENG, K. L.; HSU, C. J. Evaluation and analysis of spectrum-based fault localization with modified similarity coefficients for software debugging. *Computer Software and Applications Conference (COMPSAC)*, IEEE, p. 180–189, 2013. Citado na página 21.

ZELLER, A. *Why Programs Fail: A Guide to Systematic Debugging*. [S.l.]: Morgan Kaufmann Publishers Inc, 2005. Citado 3 vezes nas páginas 12, 19 e 21.

ZHAO, L.; WANG, L.; YWIN, X. Context-aware fault localization via control flow analysis. *Journal of Software*, Academy Publisher, v. 6, n. 10, p. 1977–1984, 2011. Citado na página 15.