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

MARIO CONCILIO NETO

Representação em grafo da cobertura de fluxo de dados e sua aplicação

São Paulo
2021

MARIO CONCILIO NETO

Representação em grafo da cobertura de fluxo de dados e sua aplicação

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

Orientador: Prof. Dr. Marcos Lordello Chaim

São Paulo

2021

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

Concilio Neto, Mario
Representação em grafo da cobertura de fluxo de dados e sua aplicação / Mario Concilio Neto; orientador, Marcos Lordello Chaim. -- São Paulo, 2021.
72 p: il.

Dissertacao (Mestrado em Ciencias) - Programa de Pós-Graduação em Sistemas de Informação, Escola de Artes, Ciências e Humanidades, Universidade de São Paulo, 2021.
Versão corrigida

1. Teste e avaliação de software. 2. Teste de fluxo de dados. 3. Cobertura de fluxo de dados. 4. Representação em grafo. 5. Algoritmos e estruturas de dados. I. Chaim, Marcos Lordello, orient. II. Título.

Dissertação de autoria de Mario Concilio Neto, sob o título **"Representação em grafo da cobertura de fluxo de dados e sua aplicação"**, 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 08 de Outubro de 2021 pela comissão julgadora constituída pelos doutores:

Prof. Dr. Marcos Lordello Chaim
Universidade de São Paulo
Presidente

Profa. Dra. Simone do Rocio Senger de Souza
Universidade de São Paulo

Prof. Dr. Vinicius Humberto Serapilha Durelli
Universidade Federal de São João del Rei

Para Silvia, Mario, Ligia e Monique.

"One, remember to look up at the stars and not down at your feet. Two, never give up work. Work gives you meaning and purpose and life is empty without it. Three, if you are lucky enough to find love, remember it is there and don't throw it away."

(Stephen Hawking)

Resumo

CONCILIO NETO, Mario. **Representação em grafo da cobertura de fluxo de dados e sua aplicação** 2021. 72 f. Dissertação (Mestrado em Ciências) – Escola de Artes, Ciências e Humanidades, Universidade de São Paulo, São Paulo, 2021.

O teste de fluxo de dados ajuda os testadores a projetar testes eficazes, exigindo que os testes executem sequências de comandos a partir de definições de variáveis para um ou mais subsequentes usos. Essas associações definição-uso são derivadas de grafos que modelam o comportamento do software. Um "grafo de fluxo" incluindo apenas caminhos que cobrem associações definição-uso, e não outros fluxos de controle, já foi definido em outro trabalho. Embora esses grafos de fluxo tenham várias vantagens sobre os grafos anteriores, quando calculados, eles omitem alguns caminhos válidos, que são necessários no uso dos grafos para descobrir relacionamentos de subsunção e geração de dados de teste. Essas omissões levam a erros nos resultados, por exemplo, no cálculo da relação de subsunção entre associações definição-uso. Este trabalho estende as soluções anteriores, introduzindo um grafo que representa todos os caminhos que cobrem uma dada associação definição-uso. A dissertação apresenta dados experimentais mostrando que este grafo pode ser gerado a um custo razoável e aplicado de forma eficiente para a descoberta da relação de subsunção de requisitos de fluxo de dados. Outras aplicações para as *graphduas* incluem a geração de dados de entrada e a análise de viabilidade de requisitos de teste de fluxo de dados.

Palavras-chaves: Teste de software, Teste de fluxo de dados, Cobertura de fluxo de dados, Representação em grafo, Estruturas de dados.

Abstract

CONCILIO NETO, Mario. **Graph representation for data-flow coverage and its application.** 2021. 72 p. Dissertation (Master of Science) – School of Arts, Sciences and Humanities, University of São Paulo, São Paulo, 2021.

Data flow testing helps testers design effective tests by requiring the tests to execute sequences of statements from definitions of variables to one or more subsequent uses. These def-use associations are derived from graphs that model software behavior. A "flow graph" that only includes paths that cover def-use associations, and not other control flows, has been defined elsewhere. Although these flow graphs have several advantages over previous graphs, as computed, they omit some valid paths, which are needed to use the graphs to discover subsumption relationships and generate test data. These omissions lead to errors in the results, for example, in the calculation of the subsumption relationship among def-use associations. This work extends previous solutions by introducing a graph that represents all paths that cover a given def-use association. The dissertation presents empirical data showing that this graph can be generated at reasonable cost and efficiently applied for data flow subsumption discovery. We envision other applications for *graphduas* such as input data generation and feasibility analysis of data-flow testing requirements.

Keywords: Software testing, Data-flow testing, Data-flow coverage, Graph representation, Data structures.

Lista de figuras

Lista de algoritmos

Lista de tabelas

Lista de abreviaturas e siglas

ADU	Associação definição-uso
AS	Algoritmo de subsunção
c.r.a	com respeito a
LOC	Linhas de código
TFC	Teste baseado em fluxo de controle
TFD	Teste baseado em fluxo de dados

Sumário

1 Introdução

O desenvolvimento de sistemas de software envolve uma série de atividades nas quais a probabilidade de ocorrência de enganos e introdução de defeitos é enorme. De acordo com Delamaro, Maldonado e Jino (2016) e Ammann e Offutt (2008), defeito é um passo, processo ou definição de dados incorretos e engano é toda ação humana que conduz a um defeito. A existência de um defeito pode ocasionar um estado errôneo durante a execução do programa e tal estado pode levar a uma falha, que é um comportamento observável inesperado perante os requisitos do sistema.

Defeitos podem ser introduzidos logo no início do processo, quando os objetivos podem estar erroneamente ou imperfeitamente especificados, ou em fases do desenvolvimento posteriores. O objetivo da atividade de teste é detectar a presença de tais defeitos, sendo um elemento crítico de garantia de qualidade. É uma atividade importante de verificação e validação para revelar falhas em programas, tendo um papel importante no desenvolvimento de sistemas (JIANG *et al.*, 2018). Representa a última revisão de especificação, projeto e codificação.

Casos de teste são definidos e submetidos ao programa a fim de identificar comportamentos não especificados. Eles exercitam *requisitos de teste* que constituem aspectos relevantes do software a serem verificados pelas técnicas de teste. Por exemplo, pode-se considerar importante que todos os comandos do programa sejam executados ao menos uma vez ou que as funções especificadas para o software sejam verificadas. Em ambos os casos, tem-se requisitos que devem ser exercitados. As técnicas são classificadas em *estruturais*, *funcionais* e *baseadas em defeitos*, de acordo com a origem da informação utilizada para estabelecer os requisitos de teste (DELAMARO; MALDONADO; JINO, 2016).

A técnica estrutural utiliza a implementação para determinar requisitos que devem ser exercitados pelos casos de teste. Já a técnica funcional estabelece os requisitos com base na especificação do software. Por sua vez, a técnica baseada em defeitos visa derivar casos de teste que mostrem a presença ou a ausência de defeitos mais comuns. Uma técnica de teste bastante disseminada na indústria é o teste estrutural, sendo utilizada principalmente para avaliar a qualidade de um conjunto de casos de teste.

O teste estrutural estabelece critérios para determinação dos requisitos de teste. Os requisitos podem ser determinados de duas maneiras: utilizando o fluxo de controle ou o fluxo de dados de um programa. Utilizando fluxo de controle, os requisitos de teste são os nós (blocos de comandos executados sequencialmente) e arcos de um programa (alterações do fluxo de execução devido a comandos de controle de fluxo); um exemplo de critério de teste é o critério *todos-arcos*. Um conjunto de casos de teste para satisfazer esse critério deve conter casos de testes de tal forma que todas saídas dos comandos de controle de fluxo condicional (e.g., *if, while, for, case*, etc) sejam exercitados ao menos uma vez. Note-se que basta conhecer o fluxo de controle do programa para determinar os requisitos de teste.

Já utilizando fluxo de dados, os requisitos de teste são as *associações definição-uso* (ADU). Basicamente, elas requerem 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 sem a sua redefinição. O critério de teste típico deste caso é o critério *todos-usos* (RAPPS; WEYUKER, 1985); ele requer que toda ADU seja executada ao menos uma vez.

Estudos (FRANKL; IAKOUNENKO, 1998; HUTCHINS *et al.*, 1994) demonstram que o *teste baseado em fluxo de dados* (TFD) é tão eficaz quanto o *teste baseado em fluxo de controle* (TFC) do programa. Por eficácia, entenda-se a capacidade de um critério detectar a presença de defeitos. Por outro lado, Hemmati (2015) mostrou que a cobertura de associações definição-uso é capaz de detectar 79% das falhas não detectadas pelos critérios de fluxo de controle em aplicações de grande porte.

No entanto, quarenta anos após sua introdução, dificilmente encontramos TFD em uso em ambientes industriais. Essa situação ocorre porque TFD é caro. Ele tende a exigir que muitos requisitos de teste sejam exercitados durante o teste, o que implica um maior esforço para gerar casos de teste e rastrear ADUs em tempo de execução. Para destacar os números envolvidos no TFD, considere os dados mostrados na Tabela 1. A linha 3 mostra o programa Apache Commons-Math versão 3.2 de 52.731 linhas de código (LOC) escritas em Java. Para testá-lo com o critério todos-usos deve-se exercer 89.678 ADUs. O número de requisitos de fluxo de dados a serem verificadas é mais de 50% maior que o número de linhas de código (LOC) e quase 5 vezes maior que o número de arcos. Os programas da Tabela 1 são bastante diferentes e direcionados para diferentes domínios; no

entanto, as relações entre o número de ADUs e o número de LOC e arcos se comportam aproximadamente da mesma maneira.

Tabela 1 – Tamanho do programa e número de ADUs necessárias

Programa	LOC	Classes	Métodos	ADUs	Arcos	ADUs / LOC	ADUs / Arcos
Commons-Lang	13.743	143	2.281	20.594	7.395	1,5	2,8
Commons-Math 2.1	26.347	428	3.995	42.901	8.745	1,6	4,9
Commons-Math 3.2	52.731	845	6.886	89.678	18.576	1,7	4,8
HSQLDB	116.618	439	8.365	122.987	34.722	1,1	3,5
JFreeChart	66.895	520	8.313	77.957	21.165	1,2	3,7
JTopas	6.184	41	475	3.773	1.233	0,6	3,1
Scimark2	672	10	61	1.220	218	1,8	5,6
Weka r5178	209.482	2.068	20.727	314.369	69.420	1,5	4,5
Weka r10042	184.454	2.257	19.694	271.443	64.760	1,5	4,2
XML-Security	17.929	317	2.353	16.821	6.273	0,9	2,7

Fonte – Araujo e Chaim (2014)

1.1 Motivação

Várias abordagens para reduzir o custo de teste de fluxo de dados (TFD) foram propostas, algumas explorando a relação de subsunção entre ADUs. Marré e Bertolino (2003) propuseram o relacionamento de subsunção entre requisitos (nó, arcos, ADUs, caminhos) a serem testados em um programa em geral. Um requisito de teste TR_2 é subsumido por um requisito TR_1 se todo caminho completo que cobre TR_1 também cobre TR_2. O subconjunto mínimo dos requisitos de teste que inclui todos os outros requisitos necessários é chamado *conjunto de abrangência* e seus elementos são referidos como requisitos *irrestritos*. Uma vantagem desse relacionamento é que um testador pode se concentrar no desenvolvimento de testes para exercitar requisitos irrestritos, pois, ao fazer isso, o exercício do restante dos requisitos é uma consequência. Outro possível uso desse relacionamento é reduzir o esforço para rastrear requisitos de teste em tempo de execução.

Outras maneiras de reduzir o custo do TFD são por meio da geração automática de dados de entrada para exercitar ADUs ou pela verificação da não existência de dados de entrada que as exercitem (análise de viabilidade). Jiang *et al.* (2018) e Vivanti *et al.* (2013) utilizaram algoritmos meta-heurísticos para encontrar dados de entrada a fim de cobrir ADUs. Su *et al.* (2015), por sua vez, usam execução simbólica e verificação de modelos para gerar dados de entrada e analisar a viabilidade das ADUs.

Para facilitar a busca por subsunção entre ADUs (MARRÉ; BERTOLINO, 2003) e explorar caminhos para execução simbólica (Su *et al.*, 2015), pesquisadores propuseram representações baseadas em grafos e estratégias para codificar caminhos que exercitam (cobrem) ADUs (MARRÉ; BERTOLINO, 2003; MARRÉ, 1997; JIANG *et al.*, 2018; Su *et al.*, 2015).

Contudo, essas representações e estratégias possuem problemas. Os grafos $G*$, criados por Marré (1997), Marré e Bertolino (2003), para representar os caminhos que cobrem uma ADU, não incluem sub-caminhos que podem bloquear a subsunção de ADUs. As regras de subsunção de ADUs criada por Jiang *et al.* (2018) possui a mesma deficiência. Finalmente, a estratégia de Su *et al.* (2015) de codificação de caminhos que cobrem uma ADU somente levam em conta restrições de fluxo de controle e não incluem restrições de fluxo de dados de modo que ela não elimina caminhos inválidos que possuem redefinição de variáveis.

1.1.1 Justificativa

As técnicas para redução de custo do teste de fluxo de dados como a subsunção de ADUs e a geração automática de dados de entrada para ADUs baseadas nessas representações e estratégias possuem limitações que podem levar a resultados incorretos ou desempenho inferiores. Portanto, é necessário a criação de representações corretas e eficientes para a cobertura de fluxo de dados.

Esta dissertação apresenta uma representação em grafo da cobertura de fluxo de dados chamada *graphdua(D)* que inclui todos os caminhos válidos cobrindo uma ADU D. Os *graphdua* são uma extensão da proposta de representação da cobertura de fluxo de dados $G*$, proposta por Marré (1997). Eles corrigem os defeitos de representações anteriores; além disso, o custo de calcular os *graphduas* é teoricamente e empiricamente aceitável para sistemas escaláveis.

Os *graphduas* possuem três aplicações imediatas: (1) permitem o cálculo correto da relação de subsunção entre ADUs; (2) possibilitam que técnicas de análise de viabilidade de requisitos de teste, criadas para fluxo de controle, sejam aplicadas para fluxo de dados; e (3) permitem que funções de *fitness* desenvolvidas para algoritmos meta-heurísticos de geração de dados de teste sejam utilizadas no contexto de fluxo de dados.

1.2 Objetivo

O objetivo principal desse projeto é apresentar uma nova representação em grafo — chamada *graphdua* — que descreve todos os caminhos que cobrem um requisito de teste de fluxo de dados. Outros objetivos são (1) mostrar que o custo de criação da nova representação é factível para programas utilizados na indústria e (2) mostrar sua aplicação para descoberta da relação de subsunção.

1.3 Hipóteses

Esta dissertação avaliou duas hipóteses:

1. A nova representação da cobertura de requisitos de fluxo de dados, *graphdua(D)*, contém todos o caminhos que cobrem um particular requisito de fluxo de dados, isto é, uma ADU D.
   Essa hipótese foi verificada de duas maneiras: (1) por meio de prova matemática e (2) por meio da sua utilização em uma aplicação, especificamente, para o cálculo da subsunção entre requisitos de fluxo da dados. A correção do cálculo subsunção de fluxo de dados utilizando *graphduas* indica que elas incluem todos os caminhos de cobertura de fluxo de dados.
2. O custo para calcular os *grapduas* é aceitável.
   Verificou-se que o custo para calcular os *grapduas* é aceitável, tanto teoricamente, utilizando técnicas de análise assintótica de algoritmos, como na prática, com sua utilização para calcular a relação de subsunção em programas programas similares aos encontrados na indústria.

1.4 Avaliação

A avaliação foi feita utilizando técnicas de análise de algoritmos para determinar a complexidade assintótica do algoritmo implementado para calcular os *graphduas*. Para avaliar a escalabilidade do cálculo dos *graphduas* para programas de grande porte, foi realizado um experimento utilizando os programas do "benchmark" *Defects4J* (JUST; JALALI; ERNST, 2014) no qual a relação de subsunção foi calculada utilizando os

graphduas. O cálculo da relação de subsunção foi também utilizada para determinar o tempo necessário para encontrar as *graphduas* e a sua correção.

1.5 Contribuições

Esta dissertação contribui para a área de teste de software ao desenvolver uma representação correta para a cobertura de fluxo de dados que soluciona problemas encontrados em trabalhos anteriores e que pode ser calculada de forma escalável para programas reais.

Além disso, a nova estrutura de dados permite calcular corretamente a relação de subsunção entre requisitos de teste de fluxo de dados, o que poderá reduzir o custo desse tipo de teste. Outras aplicações das *graphduas* incluem a geração de dados de teste e análise de viabilidade de requisitos de fluxo de dados. Essas aplicações são também discutidas neste trabalho.

1.6 Organização do documento

Este capítulo apresentou a motivação, justificativa, hipótese e contribuições dessa dissertação de mestrado. No próximo capítulo os conceitos de teste de software são detalhados. No capítulo 3, é apresentada a revisão bibliográfica que descreve as estratégias para a representação da cobertura de requisitos de teste de fluxo de dados. O capítulo 4 introduz a representação em forma de grafo da cobertura de fluxo de dados, chamada *graphdua*, proposta neste trabalho. O capítulo 5 apresenta a aplicação da estrutura para o cálculo da relação de subsunção entre requisitos de fluxo de dados e um experimento para avaliar a escalabilidade, correção e utilidade das *graphduas*. O capítulo 6 apresentas as conclusões e trabalhos futuros.

2 Conceitos básicos

Neste capítulo são apresentados os conceitos de teste de software que fundamentam a descrição da proposta de pesquisa de mestrado.

2.1 Técnicas de teste

É possível testar um programa de duas maneiras: *ad hoc* ou *sistemática*. No primeiro modo, o testador explora o programa sem nenhum planejamento e documentação; os casos de teste são derivados a partir da sua intuição. Já no teste sistemático, os casos de teste são derivados utilizando uma técnica de teste de forma que aspectos importantes da especificação ou da implementação do programa são exercitados pelos casos de teste. As técnicas de teste podem ainda ter como finalidade detectar os defeitos mais comuns. Por isso, elas são classificadas em *funcionais*, *estruturais* e *baseadas em defeitos*.

2.1.1 Técnica funcional

A técnica de teste funcional ou caixa preta consiste em verificar se o software desenvolvido atende aos requisitos especificados pelo usuário, analisando as respostas do sistema quando um conjunto de entradas válidas e inválidas é fornecido ao programa. De acordo com Sommerville *et al.* (2008), o sistema é tratado como uma caixa preta na qual apenas as entradas e as saídas são visíveis ao testador.

No teste funcional, não há necessidade de conhecimento do código fonte, pois o teste concentra-se nas funcionalidades do sistema e não na sua implementação. Exemplos de técnicas funcionais são: *Particionamento de Equivalência*, *Análise de Valores Limites*, *Grafos de Causa e Efeito* e *Categorização-Particionamento* (MYERS, 2004; OSTRAND; BALCER, 1988).

2.1.2 Técnica baseada em defeitos

A técnica baseada em defeitos estabelece os requisitos de teste explorando os defeitos mais comuns. Os critérios *Análise de Mutantes* (DEMILLO; LIPTON; SAYWARD, 1978) e *Semeadura de Defeitos* (BUDD, 1981) são exemplos de técnicas baseadas em defeitos.

O teste de mutação é uma técnica baseada em defeitos que é usada para medir a eficácia de um conjunto de testes (DELAMARO; MALDONADO; JINO, 2016). É introduzido uma pequena alteração no programa original usando operadores de mutação, criando assim um mutante que representa uma versão defeituosa do programa.

A seguir, casos de testes são executados a fim de "matar"os mutantes gerados. Se o programa mutante produzir uma saída diferente da saída do programa original contra o mesmo caso de teste, então o mutante é dito *morto* por tal caso de teste. Cada caso de teste é executado em cada mutante e é monitorado se ele fornece uma saída diferente do programa original.

Um mutante que não é morto por nenhum caso de teste permanece *vivo*. O mutante vivo pode ser morto aprimorando o conjunto de testes e adicionando mais casos de teste que podem matar os mutantes remanescentes (FAROOQ; NADEEM, 2017). No entanto, um mutante pode ser funcionalmente igual ao programa original; neste caso, novos casos de teste não irão produzir saídas distintas. Esses mutantes são chamados de *equivalentes* e, em geral, devem ser verificados "manualmente" pelos testadores.

2.1.3 Técnica estrutural

A técnica de teste estrutural ou caixa branca utiliza a implementação do programa para determinar o que deve ser testado, i.e., os requisitos de teste. Para tanto, critérios baseados em cobertura de código foram definidos para estabelecer esses requisitos. Eles têm por objetivo definir trechos de código que devem ser exercitados ou cobertos pelos casos de teste. Exemplos de critérios são: executar ao menos uma vez todas as linhas do programa ou as saídas de comandos condicionais (e.g. *if*, *while*, *for*).

Este projeto de pesquisa visa desenvolver algoritmos para tornar mais eficaz e eficiente o teste estrutural de programas, em particular o teste baseado em fluxo de dados. Por isso, essa técnica é descrita em detalhes a seguir.

2.2 Conceitos de teste estrutural

A técnica de teste estrutural requer que os casos de teste selecionados cubram determinados caminhos do programa considerados relevantes para que o testador aumente sua confiança em relação à correção do programa.

Os caminhos selecionados definem requisitos de teste que constituem um critério de adequação. Formalmente, o conjunto de casos de teste T é considerado adequado de acordo com um critério de teste C se o conjunto de requisitos de teste (TR_C) estabelecido por C é coberto pelos casos de teste T.

A medida de cobertura ($MC_C(T)$) é utilizada para avaliar o grau de adequação do conjunto T com respeito a (c.r.a.) um critério C. Seja $TR_C(t)$ o conjunto de requisitos de teste estabelecidos pelo critério C cobertos por um caso de teste $t \in T$. A medida $MC_C(T)$ é dada pela fórmula 1, onde $\mid C \mid$ representa o número de elementos de um conjunto C:

$$MC_C(T) = \frac{\mid \bigcup TR_C(t) \mid \text{ tal que } t \in T}{\mid TR_C \mid} \tag{1}$$

2.2.1 Grafo de fluxo de controle

Seja P um programa mapeado para um grafo de fluxo de controle $G(N, E, s, e)$ onde N é o conjunto de nós representando blocos de comandos de tal modo que, ao executar o primeiro comando, todos os comandos subsequentes são executados; s é o nó de entrada; e é o nó de saída; e E é o conjunto de arcos (n', n) de tal modo que $n' \neq n$, que representa uma possível transferência de controle entre os nós n' e n.

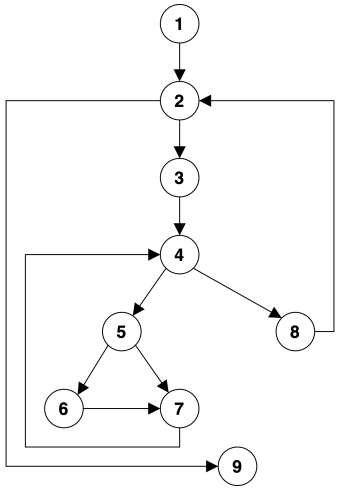
A Figura 1 apresenta um programa que ordena um *array* de inteiros (originalmente apresentado por Marré e Bertolino (1996)). O número inserido nos comentários refere-se ao nó ao qual cada comando está associado. A Figura 2 apresenta o grafo de fluxo de controle obtido do programa exemplo da Figura 1.

Um *caminho* é uma sequência de nós $(n_i, ..., n_k, n_{k+1}, ..., n_j)$, onde $i \leq k < j$, tal que $(n_k, n_{k+1}) \in E$. Um nó n_k é dito *predecessor* do nó n_{k+1} se existe um arco (n_k, n_{k+1}) em E enquanto que n_{k+1} é dito ser *sucessor* de n_k. Um caminho $(n_i, ..., n_k, n_{k+1}, ..., n_j)$ é *completo* se n_i é igual a s (o nó de entrada) e n_j igual a e (o nó de saída).

Figura 1 – Exemplo programa *Sort*

```
/*1*/ void sort(int a[], int n) {
/*1*/     int sortupto, maxpos, mymax, index;
/*1*/     sortupto = 1;
/*1*/     maxpos = 1;
/*2*/     while(sortupto < n) {
/*3*/         mymax = a[sortupto];
/*3*/         index = sortupto +1;
/*4*/         while(index <= n) {
/*5*/             if(a[index] > mymax) {
/*6*/                 mymax = a[index];
/*6*/                 maxpos=index;
/*6*/             }
/*7*/             index++;
/*7*/         }
/*8*/         index = a[sortupto];
/*8*/         a[sortupto]=mymax;
/*8*/         a[maxpos]=index;
/*8*/         sortupto++;
/*8*/     }
/*9*/ }
```

Figura 2 – Grafo de fluxo de controle do programa *Sort*



Fonte – Concilio *et al.* (2021)

O subgrafo $SG(n_i, n_j)$, obtido do grafo $G(N, E, s, e)$, representa todos os caminhos a partir do nó n_i ao nó n_j de G. Seja n_i e n_j dois nós de $G(N, E, s, e)$ de tal maneira que n_i alcança n_j, ou seja, existe pelo menos um caminho de n_i a n_j. O subgrafo $SG(n_i, n_j)$ de G entre n_i e n_j é dado pelo grafo dirigido $G'(N', E', s', e')$[1] onde:

1. n_i e $n_j \in N'$ de tal maneira que o nó de entrada $s' = n_i$ e o nó de saída $e' = n_j$;
2. se existe um caminho $(n_i, n_{i+1}, ..., n_k, ..., n_{j-1}, n_j)$ de G, onde $i < k \leq j$, então $n_k \in N'$ e $(n_{k-1}, n_k) \in E'$.

Um subgrafo contém todos os caminhos de n_i a n_j tal que n_i e n_j ocorre apenas uma vez como nó de entrada e nó de saída, respectivamente.

2.2.2 Critérios de teste

A técnica de teste estrutural utiliza a implementação para determinar requisitos que devem ser cobertos pelos casos de teste e baseia-se em critérios de teste para determinação desses requisitos. Os requisitos podem ser determinados de duas maneiras: utilizando o

[1] Nesta dissertação, um subgrafo é um grafo dirigido (e não um grafo de fluxo de controle) de tal maneira que os arcos não são necessariamente associados com comandos de fluxo de controle (e.g., *if*, *while*, *for*) como em um grafo de fluxo

fluxo de controle ou o fluxo de dados de um programa. Os conceitos de fluxo de controle e de dados em programas são descritos a seguir.

2.2.3 Critérios baseados em fluxo de controle

O teste de fluxo de controle[2] está associado aos nós e arcos do grafo de fluxo de controle do programa. A seguir, são definidos critérios baseados em fluxo de controle.

- *Critério todos-nós:* exige que cada nó do grafo de fluxo de controle do programa seja executado ao menos uma vez.
- *Critério todos-arcos:* exige que cada arco do grafo de fluxo de controle do programa seja executado ao menos uma vez.
- *Critério todos-caminhos:* exige que cada caminho, dado por uma sequência finita de nós do grafo, seja executado pelo menos uma vez.

Esses critérios se baseiam na premissa de que a confiança na correção do programa aumenta se todos os nós, i.e., todos os comandos, todas as transferências de fluxo ou todos os caminhos são testados ao menos uma vez. O critério todos-caminhos pode levar a um conjunto infinito de requisitos de teste se o programa contiver laços e, por isso, é pouco utilizado na prática.

2.2.4 Critérios baseados em fluxo de dados

O teste de fluxo de dados[3] requer que os casos de teste selecionados executem caminhos em um programa entre cada ponto que um valor é atribuído a uma variável e seu subsequente uso. Quando uma variável recebe um novo valor, é dito que uma *definição* ocorreu; o *uso* de uma variável ocorre quando seu valor é referenciado. Uma distinção é feita entre uma variável referenciada para o cálculo de um valor e para o cálculo de um predicado. Quando referenciada em um cálculo de predicado, é chamada *p-uso* e é associada a arcos; caso contrário, é chamada *c-uso* e associada aos nós.

[2] Por simplicidade, quando nos referimos a "teste de fluxo de controle"estamos tratando do teste estrutural baseado em análise de fluxo de controle

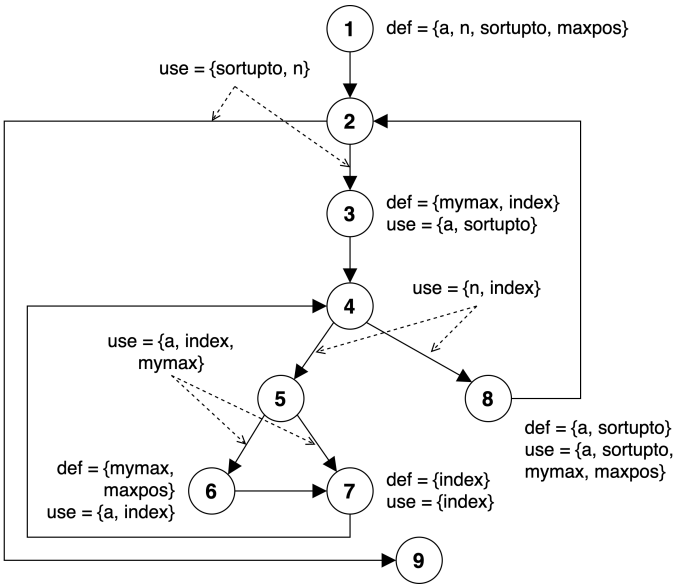
[3] Por simplicidade, quando nos referimos a "teste de fluxo de dados"estamos tratando do teste estrutural baseado em análise de fluxo de dados.

A Figura 3 descreve o grafo de fluxo de controle do programa *Sort* anotado com o conjunto de definições, c-usos, e p-usos associados com nós e arcos. Um *caminho livre de definição* c.r.a. uma variável X é um caminho onde X não é redefinida em nenhum nó no caminho, exceto possivelmente no primeiro e no último nó.

É possível determinar um subgrafo de tal modo que contenha apenas caminhos livre de definição c.r.a. X entre dois nós n_i e n_j. O subgrafo de caminho livre de definição $SG(n_i, n_j, X)$ c.r.a. X de G entre os nós n_i e n_j é dado pelo grafo dirigido $G'(N', E', s', e')$ onde:

1. n_i e $n_j \in N'$ de tal maneira que o nó de entrada $s' = n_i$ e o nó de saída $e' = n_j$;
2. se existe um caminho livre de definição $(n_i, n_{i+1}, ..., n_k, ..., n_{j-1}, n_j)$ c.r.a. variável X, onde $i < k \leq j$, então $n_k \in N'$ e $(n_{k-1}, n_k) \in E'$.

Figura 3 – Grafo de fluxo de controle anotado do programa *Sort*



Fonte – Concilio *et al.* (2021)

Tipicamente, os critérios de teste de fluxo de dados exigem que as ADUs sejam cobertas. A tupla $D = (d, u, X)$, chamada *ADU c-uso*, representa um requisito de teste de fluxo de dados envolvendo uma definição no nó d e um c-uso no nó u da variável X de tal modo que existe um caminho livre de definição c.r.a. X de d a u. Da mesma forma, a tupla $D = (d, (u', u), X)$, chamada *ADU p-uso*, representa a associação entre uma definição e um p-uso da variável X. Nesse caso, o caminho livre de definição $(d, ..., u', u)$ c.r.a. X deve existir.

O critério de teste de fluxo de dados mais utilizado, *todos-usos*, requer que o conjunto de caminhos executado pelos casos de teste de um conjunto de teste T inclua um caminho livre de definição para cada ADU (d, u, X) ou $(d, (u', u), X)$ de um programa P (RAPPS; WEYUKER, 1985). Um conjunto de testes com tal propriedade é dito *adequado* para o critério todos-usos de um programa P se todas as ADUs requeridas forem cobertas. A Tabela 2 mostra os requisitos de testes do programa exemplo para o critério todos-usos.

Tabela 2 – Requisitos de teste para o critério todos-usos para o programa *Sort*

Todos-usos			
(1,3, a)	(1,(2,3), sortupto)	(3,(4,8), index)	(6,8, mymax)
(1,(5,6), a)	(1,(2,9), sortupto)	(3,(5,6), index)	(7,(4,5), index)
(1,(5,7), a)	(1,3, sortupto)	(3,(5,7), index)	(7,(4,8), index)
(1,6, a)	(1,8, sortupto)	(3,6, index)	(7,(5,6), index)
(1,8, a)	(1,8, maxpos)	(3,7, index)	(7,(5,7), index)
(1,(2,3), n)	(3,(5,6), mymax)	(3,8, index)	(7,6, index)
(1,(2,9), n)	(3,(5,7), mymax)	(6,8, maxpos)	(7,7, index)
(1,(4,5), n)	(3,8, mymax)	(6,(5,6), mymax)	(7,8, index)
(1,(4,8), n)	(3,(4,5), index)	(6,(5,7), mymax)	(8,3, a)
(8,(5,6), a)	(8,(5,7), a)	**(8,6, a)**	(8,8, a)
(8,(2,3), sortupto)	(8,(2,9), sortupto)	(8,3, sortupto)	(8,8, sortupto)

Fonte – Concilio *et al.* (2021)

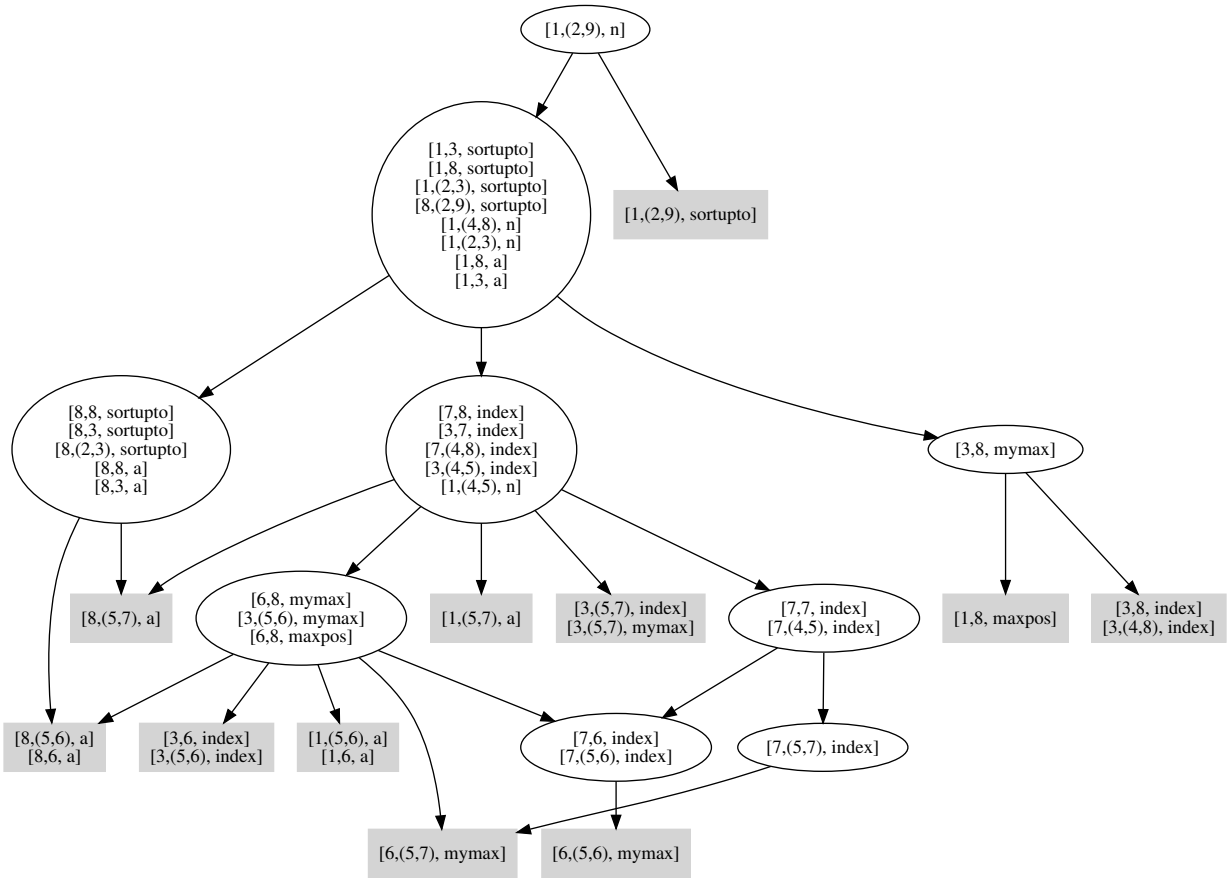
2.2.5 Relação de subsunção entre requisitos de fluxo de dados

Os requisitos de teste estão relacionados. Na Figura 2, sempre que o nó 7 é coberto, os nós 1, 2, 3, 4, 5, 8 e 9 também o são. Baseado em observações como esta, a relação de subsunção entre requisitos (nós, arcos e ADUs) a serem testados no programa foi concebida. Um requisito de teste RT_1 (e.g., um arco) subsume uma requisito RT_2 (outro arco) se todo caminho completo que cobre RT_1 cobre RT_2 também. O subconjunto mínimo de requisitos que subsumem todas os outros requisitos requeridos é chamada *conjunto de abrangência* e seus elementos são chamados *irrestritos* (MARRÉ; BERTOLINO, 2003). Uma vez identificado o conjunto de abrangência, o testador pode focar apenas nos requisitos irrestritos para desenvolver casos de teste e, dessa maneira, reduzir os custos da atividade de teste.

A relação de subsunção entre ADUs é formalmente definida a seguir. Dado $D_1 = (d_1, u_1, X_1)$ e $D_2 = (d_2, u_2, X_2)$, D_1 subsume D_2 se todo caminho completo que

contém um caminho livre de definição c.r.a. X_1 entre d_1 e u_1 também contém um caminho livre de definição c.r.a. X_2 entre d_2 e u_2. A notação $D_2 \rightarrow D_1$ indica que a ADU D_2 é subsumida pela ADU D_1. No programa exemplo, sempre que $D_1 = (1, (5, 7), a)$ é coberta, $D_2 = (3, 7, index)$ também é; portanto $(3, 7, index) \rightarrow (1, (5, 7), a)$. Por outro lado, a cobertura de $D_2 = (3, 7, index)$ não implica na cobertura de $D_1 = (1, (5, 7), a)$. Consequentemente, a relação de subsunção não é simétrica; ou seja, $D_2 \rightarrow D_1$ não implica $D_1 \rightarrow D_2$.

Figura 4 – Grafo reduzido da relação de subsunção entre ADUs do programa *Sort*



Fonte – Concilio *et al.* (2021)

A Figura 4 descreve um grafo no qual a subsunção entre as ADUs do programa exemplo são representadas. Utilizando a relação de subsunção, um conjunto mínimo de ADUs, chamado *conjunto de abrangência*, pode ser determinado tal que ao cobrir as ADUs pertencentes a este conjunto implique cobrir todas as ADUs exigidas pelo critério de todos-usos. As folhas do grafo na Figura 4, representadas por retângulos com fundo cinza, constituem o conjunto de abrangência das ADUs do programa exemplo. Uma ADU pertencente a tal grupo é uma *ADU irrestrita*, ou *ADU livre*. O conjunto de abrangência

não é único pois podem existir duas ADUs de tal modo que $D_2 \rightarrow D_1$ e $D_1 \rightarrow D_2$ em uma folha. Neste caso, uma delas deve ser incluída no conjunto de abrangência.

Cinco de onze folhas na Figura 4 possuem duas ADUs. Então existem dezesseis conjuntos diferentes de ADUs irrestritas para o programa *Sort*; cada uma com onze ADUs diferentes sendo um possível conjunto de ADUs irrestritas: $\Big\{(1,(2,9),sortupto)$, $(3,(5,7),mymax)$, $(8,(5,7),a)$, $(3,8,index)$, $(6,(5,7),mymax)$, $(6,(5,6),mymax)$, $(1,(5,6),a)$, $(3,6,index)$, $(8,6,a)$, $(1,8,maxpos)\Big\}$.

2.3 Considerações finais

Neste capítulo apresentamos os conceitos de teste de software necessários para o entedimento da representação em grafo da cobertura de fluxo de dados proposta nesta dissertação. Por isso, foi dada mais ênfase aos conceitos de teste estrutural haja vista que ela visa reduzir os custos do teste de fluxo de dados.

Em particular, foi apresentada a relação de subsunção de requisitos de teste de fluxo de dados, isto é, ADUs. Ela pode ser utilizada para reduzir os custos de teste uma vez que o testador pode focar nas ADUs que subsumem todos as demais para elaborar casos de teste, por exemplo. A representação em grafo da cobertura de fluxo é particularmente útil para o cálculo da relação de subsunção entre ADUs.

No próximo capítulo será apresentada a revisão bibliográfica que descreve as diferentes abordagens identificadas na literatura para representar a cobertura de fluxo de dados.

3 Revisão bibliográfica

Neste capítulo são discutidas as abordagens criadas para representar os caminhos que cobrem ADUs, bem como as suas limitações. Que seja de nosso conhecimento, quatro abordagens foram desenvolvidas. Três dessas estratégias apoiam o cálculo da relação de subsunção entre ADUs (MARRÉ; BERTOLINO, 1996; MARRÉ, 1997; JIANG *et al.*, 2018) e uma delas apoia a geração de dados de teste que cobrem ADUs a partir de execução simbólica.

Marré e Bertolino propuseram duas estratégias: a primeira delas (M&B I) é baseada no alinhamento de nós e em restrições sobre os caminhos que conectam esses nós (MARRÉ; BERTOLINO, 1996); e a segunda abordagem (M&B II) é baseada em um grafo que representa os caminhos que podem cobrir uma dada ADU (MARRÉ, 1997; MARRÉ; BERTOLINO, 2003). Su et al. (Su *et al.*, 2015) e Jiang e al. (JIANG *et al.*, 2018) também utilizam o alinhamento de nós para determinar caminhos que cobrem uma ADU.

A seguir são descritas sucintamente essas abordagens.

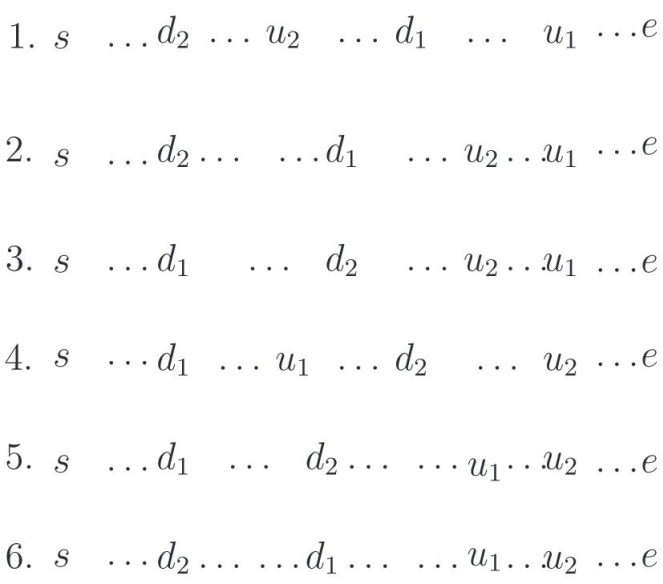
3.1 M&B I

Marré e Bertolino (1996) definiram a relação de subsunção entre ADUs e propuseram um algoritmo (referenciado como algoritmo M&B I) para determiná-la. Sem perda de generalidade, assume-se a seguir apenas ADUs do tipo c-uso[1]. As autoras definiram regras que estabelecem condições sob as quais, sempre que $D_1 = (d_1, u_1, X_1)$ é coberta por um caso de teste, $D_2 = (d_2, u_2, X_2)$ também o é. Ou seja, elas propuseram uma estratégia para codificar os caminhos que cobrem a ADU D_1 e as condições em que a ADU D_2 é coberta nesses caminhos. A Figura 5 exibe esquematicamente essas regras para a subsunção da ADU D_2 pela ADU D_1.

As regras são baseadas nos possíveis alinhamentos dos nós s (nó de entrada), d_1, d_2, u_1, u_2 e e (nó de saída), em restrições sobre os caminhos que conectam esses nós e nas definições permitidas para ocorrer nos nós d_1 e u_1.

[1] De fato, Marré e Bertolino (1996) utilizam um grafo de fluxo no qual comandos sequenciais são associadas a arcos e comandos de transferência condicional a nós. Essa notação não diferencia entre ADUs c-uso e p-uso.

Figura 5 – Regras de subsunção para $D_1 = (d_1, u_1, X_1)$ e $D_2 = (d_2, u_2, X_2)$.



Fonte – Mario Concilio Neto, 2021

Como exemplo de uma regra de subsunção, considere a Regra 2. A grosso modo, para satisfazê-la, os nós s, d_2, e d_1 devem estar alinhados, assim como os nós d_1, u_2, e u_1. Três nós n_i, n_j, e n_k estão *alinhados* se todo o caminho do nó n_i para n_k contém n_j, denotado por $AL(n_i, n_j, n_k)$ (MARRÉ; BERTOLINO, 1996). Tal requisito visa impor a ordem da Regra 2. Além disso, todos os caminhos conectando d_2 e d_1 devem ser livres de definição c.r.a. X_2, e os caminhos conectando d_1 e u_2 que são livres de definição c.r.a. X_1 devem ser livres de definição c.r.a. X_2 também. O requisito final para a subsunção de D_2 por D_1 é que não seja permitida a definição de X_2 no nó d_1.

Supostamente se cumpridas essas condições, todo caminho completo que cobre D_1 também deve cobrir D_2. A análise da Regra 2 indica que, para sua aplicação, existem condições que precisam ser verificadas. Elas são descritas a seguir:

- *Alinhamento de nós.* Essa condição verifica se três nós n_i, n_j e n_m estão alinhados — indicada por $AL(n_i, n_j, n_k)$. No caso da Regra 2, as condições $AL(s, d_2, d_1)$ e $AL(d_1, u_2, u_1)$ devem ser válidas.
- *Todos os caminhos livres de definição c.r.a. X.* Essa condição verifica se todo caminho de um nó n_i para o nó n_j é livre de definição c.r.a X — denotada $TodosLivreDef(n_i, n_j, X)$. Em relação à Regra 2, a condição $TodosLivreDef(d_2, d_1, X_2)$ deve ser verdadeira.

- *Todo caminho livre simultaneamente de definição c.r.a. X e Y*. A condição verifica se todo caminho de n_i para o nó n_j que é livre de definição c.r.a. X também é livre de definição c.r.a. Y — denotada $TodoLivreDefSim(n_i, n_j, X, Y)$. Considerando a Regra 2, $TodoLivreDefSim(d_1, u_2, X_1, X_2)$ deve ser mantida.

A descrição formal de todas as seis regras para a subsunção de $D_1 = (d_1, u_1, X_1)$ por $D_2 = (d_2, u_2, X_2)$ é apresentada a seguir.

1. $TodosLivreDef(d_2, u_2, X_2)$ E $AL(s, u_2, d_1)$ E $AL(s, d_2, u_2)$; OU
2. a) $X_1 \neq X_2$ E $TodosLivreDef(d_2, d_1, X_2)$ E $TodoLivreDefSim(d_1, u_2, X_1, X_2)$ E X_2 não é definida em d_1 E $AL(s, d_2, d_1)$ E $AL(d_1, u_2, u_1)$; OU
   b) $X_1 = X_2$ E $d_1 = d_2$ E $AL(d_1, u_2, u_1)$; OU
3. a) $X_1 \neq X_2$ E $TodoLivreDefSim(d_2, u_2, X_1, X_2)$ E $AL(d_1, d_2, u_1)$ E $AL(d_2, u_2, u_1)$; OU
   b) $X_1 = X_2$ E $d_1 = d_2$ E $AL(d_1, u_2, u_1)$; OU
4. $TodosLivreDef(d_2, u_2, X_2)$ E $AL(u_1, d_2, e)$ E $AL(d_2, u_2, e)$; OU
5. a) $X_1 \neq X_2$ E $TodoLivreDefSim(d_2, u_1, X_1, X_2)$ E $TodosLivreDef(u_1, u_2, X_2)$ E X_2 não é definida em u_1 E $AL(d_1, d_2, u_1)$ E $AL(u_1, u_2, e)$; OU
   b) $X_1 = X_2$ E $d_1 = d_2$ E X_2 não é definida em u_1 E $TodosLivreDef(u_1, u_2, X_2)$ E $AL(u_1, u_2, e)$; OU
6. a) $X_1 \neq X_2$ E $TodosLivreDef(d_2, d_1, X_2)$ E $TodoLivreDefSim(d_1, u_1, X_1, X_2)$ E X_2 não é definida em d_1 E $TodosLivreDef(u_1, u_2, X_2)$ E X_2 não é definida em u_1 E $AL(e_0, d_2, d_1)$ E $AL(u_1, u_2, e)$; OU
   b) $X_1 = X_2$ E $d_1 = d_2$ E X_2 não é definida em u_1 E $TodosLivreDef(u_1, u_2, X_2)$ E $AL(u_1, u_2, e)$.

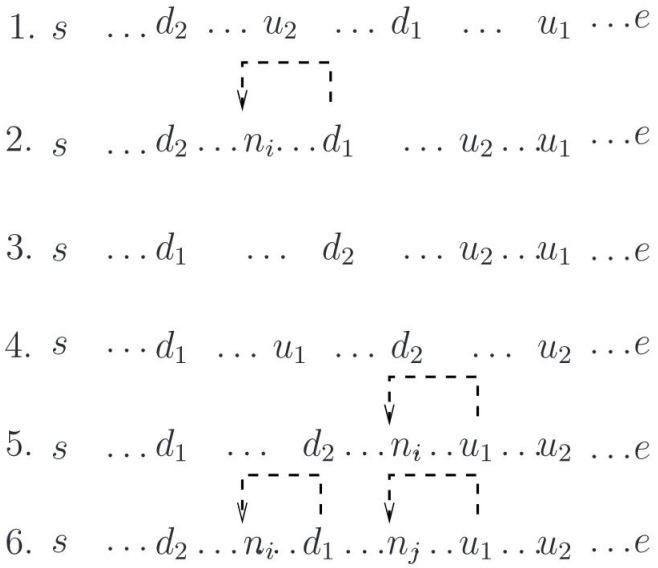
Marré e Bertolino (1996) mostram que $AL(n_i, n_j, n_m)$, $TodosLivreDef(n_i, n_j, X)$, e $TodoLivreDefSim(n_i, n_j, X, Y)$ podem ser verificados a um custo $O(E)$ onde E é o número de arcos do grafo de fluxo do programa. Portanto, o custo para calcular o alinhamento de nós e as restrições nos caminhos é $O(E)$. Como cada ADU é verificada em relação a todas as outras ADUs, o custo total para a verificação de subsunção de ADU é $O(D^2E)$ onde D é o número de ADUs.

As regras descritas acima, no entanto, não levam em consideração caminhos que podem impossibilitar o estabelecimento do relacionamento de subsunção entre ADUs.

Considere a subsunção de $D_2 = (1, (5, 7), index)$ por $D_1 = (3, (5, 7), mymax)$ do programa de exemplo *Sort* (Figura 1). Todas as condições da Regra 2(a) são satisfeitas. Contudo, o caminho $(1, 2, 3, 4, 8, 2, 3, 4, 5, 7)$ cobre D_1 mas não D_2. Nesse caminho, d_2 (no exemplo, nó 3) ocorre duas vezes, mas tal possibilidade não é considerada pela condição $TodoLivreDefSim(d_1, u_2, X_1, X_2)$ da Regra 2(a).

A Figura 6 inclui os caminhos de retorno ausentes como setas tracejadas para trás para as Regras 2, 5 e 6. Para a Regra 2, elas representam todos os caminhos do nó d_1 para o nó n_i onde n_i é um nó pertencente a um caminho que conecta d_2 e d_1. Esses caminhos impedem o relacionamento de subsunção de ADUs sempre que houver uma redefinição de X_2 em qualquer nó entre d_1 e o nó n_i.

Figura 6 – Regras de subsunção para D_1 e D_2 com caminhos de retorno



Fonte – Mario Concilio Neto, 2021

Portanto, a representação de caminhos que cobrem ADUs proposta no algoritmo M&B I não incluem os caminhos de retorno descritos na Figura 6 e, portanto, é incompleta.

3.2 M&B II

Marré e Bertolino (2003) descrevem sucintamente outro algoritmo para encontrar a relação de subsunção entre ADUs, referido como algoritmo M&B II. Ele baseia-se em um grafo intermediário chamado $G*$ cujo objetivo é representar todos os caminhos que cobrem uma ADU $D_1 = (d_1, u_1, X_1)$. Para tanto, ele leva em consideração os caminhos do nó de entrada (s) para o nó de definição (d_1), caminhos livre de definição c.r.a. X_1 de d_1

para o nó u_1, e caminhos de u_1 para o nó de saída (e). O algoritmo de subsunção M&B II verifica se todos os caminhos de $G*$ também incluem d_2 e u_2. Por fim, para descobrir se todos os caminhos que cobrem D_1 também cobrem D_2, ele verifica se nenhum nó n_i entre d_2 e u_2 contém uma definição de X_2.

Segundo as autoras (MARRÉ, 1997), o custo para calcular o grafo $G*$ é $O(E)$. O algoritmo de subsunção M&B II requer, para cada ADU para a qual o $G*$ é criado, que todas as outras ADUs sejam verificadas se são cobertas em $G*$. Essa verificação para cada ADU pode custar também $O(E)$. Então, o custo do cálculo da relação de subsunção das ADUs com esse algoritmo é $O(D^2E^2)$.

O problema do grafo $G*$ é que ele também não inclui os caminhos de retorno contidos na Figura 6. Ou seja, ele não contempla caminhos $d_1 \ldots d_1$ e $u_1 \ldots u_1$ que podem bloquear a relação de subsunção entre ADUs. Portanto, $G*$ falha na representação dos caminhos que cobrem uma ADU $D_1 = (d_1, u_1, X_1)$.

3.3 Algoritmo de Jiang et al.

Jiang *et al.* (2018) também apresentam um algoritmo baseado em regras para subsunção de ADUs. Eles restringem os caminhos que cobrem uma ADU específica aplicando o conceito de *ordem def-uso*, introduzido por Santelices e Harrold (2007). A ADU $D = (d, u, X)$ está em uma ordem def-uso se uma das seguintes condições se verificar: (1) o nó d não é alcançável a partir do nó u; (2) o nó d domina o nó u; ou (3) o nó u pós-domina o nó d. Um nó n_i domina um nó n_j se todos os caminhos do nó inicial até n_j incluem n_i. Por outro lado, n_j pós-domina um nó n_i se qualquer caminho de n_i para o nó de saída incluir n_j. Um nó n é dominado por si mesmo, mas não pós-domina a si mesmo (JIANG *et al.*, 2018).

Para determinar a relação de subsunção entre ADUs, Jiang *et al.* (2018) utilizam também o conceito de *dependência de controle* (SANTELICES; HARROLD, 2007). Um nó n_i é dependente de controle de n_k, se e somente se, existir um caminho direto P de n_i para n_k e qualquer n em P (excluindo n_i e n_k) é pós-dominado por n_k e n_i não é pós-dominado por n_k. Se n_k é dependente de controle de n_i, então n_i possui dois arcos de saída: um que leva à execução de n_k e outro que não. DC(n_k) representa o arco do qual n_k é dependente de controle.

Usando esses dois conceitos, Jiang *et al.* (2018) definem a subsunção de ADUs da seguinte maneira. Uma ADU $D_2 = (d_2, u_2, X_2)$ é subsumida por $D_1 = (d_1, u_1, X_1)$ se, e somente se, as três condições a seguir forem atendidas.

1. D_1 e D_2 devem satisfazer a ordem def-uso;
2. $\mathrm{DC}(d_1) \bigcup \mathrm{DC}(u_1) \supseteq \mathrm{DC}(d_2) \bigcup \mathrm{DC}(u_2)$; e
3. Há um caminho entre d_1 e u_1 que não contém uma definição de X_1 e há um caminho entre d_2 e u_2 que não contém uma definição de X_2.

Como resultado, uma ADU D está em ordem def-uso se, sempre que D for coberto, o nó d tem garantia de ocorrer antes do nó u. No entanto, se a ordem def-uso for obtida pelas condições (2) ou (3), o nó de uso u pode alcançar o nó def d. Nesse caso, os mesmos caminhos que $G*$ não inclui também não são incluídos pela ordem def-uso. Considere a ADU (3,6, mymax) do programa Sort. Ela está na ordem def-uso devido à condição (1) (nó de definição 3 domina o nó de uso 6), mas o nó 6 alcança o nó 3. Logo, há um caminho do nó 3 ao nó 3 que não é codificado pela ordem def-uso que pode bloquear a subsunção de ADUs, por exemplo, a subsunção de $D_2 = (1, (5, 7), index)$ por $D_1 = (3, (5, 7), mymax)$.

Os autores não discutem a complexidade de seu algoritmo para calcular a subsunção de ADUs. No entanto, a relação de dominância, necessária para determinar os conjuntos $\mathrm{DC}(n_i)$, em grafos redutíveis (HECHT, 1977) é O(|E|); a acessibilidade dos nós, necessária para verificar a ordem def-uso, no pior caso, precisa visitar todas os arcos. Portanto, o custo para verificar a subsunção de uma ADU D_1 contra outra ADU D_2 é O(|E|). Como cada ADU é verificada em relação a todas as outras, o custo total é $\mathrm{O}(|\mathrm{D}^2\ ||\mathrm{E}|)$.

3.4 Codificação de caminhos para cobrir ADUs de Su et al.

Su *et al.* (2015) usam análise estática para reduzir a explosão de caminhos analisados por meio de execução simbólica (SE) para gerar entradas de teste ou para detectar se as ADUs são não-executáveis. Eles usam a análise de dominância para encontrar um conjunto de *pontos de corte* para orientar a exploração do caminho. Dada uma ADU $D = (d, u, X)$, seus pontos de corte são uma sequência de pontos de controle críticos $c_1, \ldots, c_i, \ldots, c_n$ que devem ser percorridos em sequência por qualquer caminho que cobre D.

Ou seja, a sequência $c_1, \ldots, d, \ldots, c_i, \ldots, u, \ldots, c_n$ sempre ocorre em qualquer caminho que cobre D. Os autores desenvolveram heurísticas baseadas em pontos de corte

para selecionar os caminhos a serem executados simbolicamente para determinar os dados de entrada que cobrem a ADU D ou para mostrar que ela é não executável.

A relação de dominância utiliza apenas informações de fluxo de controle. Assim, embora os pontos de corte estejam presentes em todo caminho que cobre D, os nós que ocorrem entre d e c_i $(d, \ldots, c_i)$ e entre c_i e u $(c_i, \ldots, u)$ devem ser verificados contra possíveis redefinições da variável X. Su *et al.* (2015) verifica se há redefinições durante a exploração de caminhos para execução simbólica. O custo para encontrar os pontos de corte é O(|E|) onde E é o número de arcos do grafo de fluxo de controle do programa.

A representação proposta neste trabalho prescinde dessa verificação porque inclui apenas caminhos livre de definição c.r.a variável X. Além disso, possui o mesmo custo para ser calculada.

3.5 Considerações finais

Neste capítulo foram apresentadas as diferentes abordagem identificadas na literatura para representar os caminhos que cobrem um associação definição uso (ADU). Todas as abordagens avaliadas são incompletas para representar corretamente esses caminhos.

As abordagens de Marré e Bertolino (MARRÉ; BERTOLINO, 1996; MARRÉ, 1997; MARRÉ; BERTOLINO, 2003) e a de Jiang et al. (JIANG *et al.*, 2018) são incompletas porque não incluem caminhos que podem cobrir uma dada ADU mas bloquear a subsunção de outras ADUs. A falha em incluir esses caminhos pode levar ao cálculo incorreto da relação de subsunção. O pontos de cortes introduzidos por Su *et al.* (2015), por sua vez, permitem caminhos com redefinição de variáveis e, portanto, incluem caminhos que não cobrem uma dada ADU.

No próximo capítulo será introduzida uma representação baseada em grafo para representar a cobertura de ADUs que é completa e que resolve os problemas identificados nas abordagens anteriores.

4 Uma nova estrutura em grafo para representar cobertura de fluxo de dados

Neste capítulo é introduzida uma estrutura de dados, chamada *graphdua*, que representa na forma de um grafo todos os caminhos que cobrem uma particular ADU D. Além disso, são apresentados os algoritmos utilizados para encontrá-la, bem como a análise de custo dos algoritmos e uma prova da correção da *graphdua*.

4.1 Estrutura graphdua

Suponhamos que desejamos determinar todos os caminhos completos que cobrem a ADU $(8, 6, a)$ (em negrito na Tabela 2), requerida para testar o programa *Sort* (Figura 1). Mostraremos informalmente a seguir como $graphdua(8, 6, a)$ — que codifica todos esses caminhos — é determinada.

4.1.1 Alcançando a definição

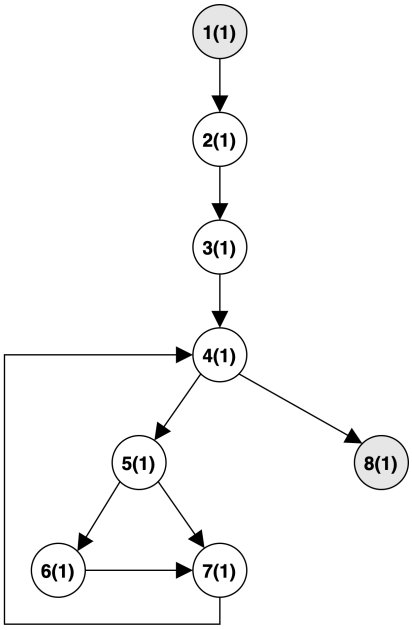
Para cobrir $(8, 6, a)$, os dados de entrada devem inicialmente levar o programa a alcançar o nó de definição, ou seja, nó 8. Vários caminhos alcançam o nó de definição 8.

Ao examinar a Figura 2, pode-se observar que (1,2,3,4,8) é o caminho mais curto para alcançar o nó 8. Outros caminhos também levam ao nó 8 como (1,2,3,4,5,6,7,4,8) ou (1,2,3,4,5,7,4,8), apenas para mencionar dois caminhos com uma única interação do laço. A Figura 7 apresenta um subgrafo do fluxograma original, no qual todos os caminhos do nó inicial 1 até o nó de definição 8 são representados. No subgrafo da Figura 7, referido como subgrafo *SG1*, os nós são identificados por um par $n(1)$ onde 1 identifica o subgrafo *SG1* e n é o nó original do fluxograma do programa exemplo.

4.1.2 Ao redor da definição

O nó de definição 8 foi alcançado. Portanto, o próximo passo é alcançar o nó de uso, nó 6, percorrendo um caminho livre de definição c.r.a. variável a do nó 8 para o nó 6. Digamos, porém, que antes de alcançar o nó de uso, o caminho (8,2,3,4,5,7,4,5,7,4,8) é percorrido pelo caso de teste. Nesse caso de teste, o nó de definição é visitado duas vezes

Figura 7 – *SG1*



Fonte – Concilio *et al.* (2021)

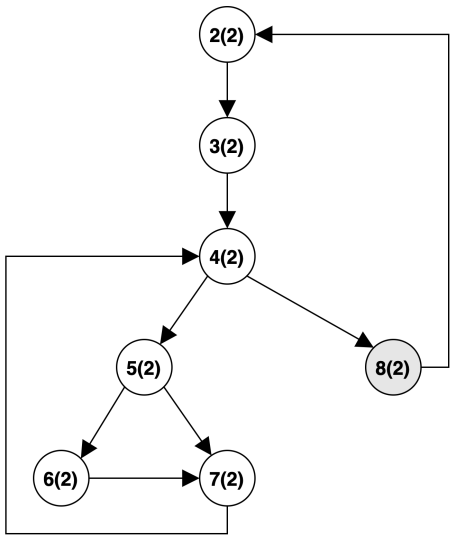
antes de o nó de uso ser alcançado por um caminho livre de definição. E, dependendo do caso de teste, o nó 8 pode ser visitado várias vezes antes de atingir o nó de uso.

A Figura 8 mostra o subgrafo *SG2*. O objetivo é representar os caminhos do nó de definição para o nó de definição. Para esse propósito, o nó 8(2), nó de definição, deve ser tanto o nó de entrada como o nó de saída do *SG2* (indicado em cinza na Figura 8). A ideia fundamental do subgrafo *SG2* é representar os caminhos que começam e terminam no nó de definição. Nesses caminhos, uma redefinição da variável da ADU (no exemplo, variável a) pode ocorrer. Como o nó de saída do *SG2* é o nó de definição, a redefinição não tem impacto na cobertura da ADU. Note-se que o nó de uso (nó 6(2)) também está incluído no *SG2*. Embora contra-intuitiva, sua presença no *SG2* pode representar caminhos nos quais a ADU $(8, 6, a)$ é coberta duas ou mais vezes.

4.1.3 Alcançando o uso

Como consideramos que a ADU $(8, 6, a)$ acabará sendo coberta, existe pelo menos um caminho livre de definição c.r.a. a do nó 8 ao nó 6 que será percorrido. Portanto, devemos encontrar o subgrafo contendo todos os caminhos livres de definição c.r.a. a que atingem o nó de uso. A Figura 9 descreve o subgrafo *SG3*, que descreve todos os caminhos

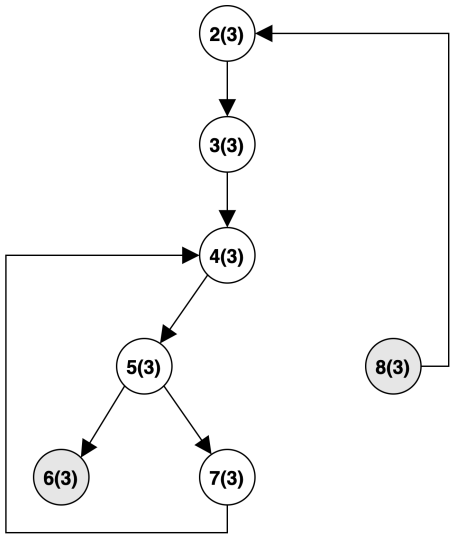
Figura 8 – *SG2*



Fonte – Concilio *et al.* (2021)

livres de definição c.r.a. variável a do nó 8 ao nó 6 da ADU $(8, 6, a)$. Observe que o nó de definição, nó 8(3), é o nó de entrada e o nó de uso, 6(3), é o nó de saída de *SG3*.

Figura 9 – *SG3*



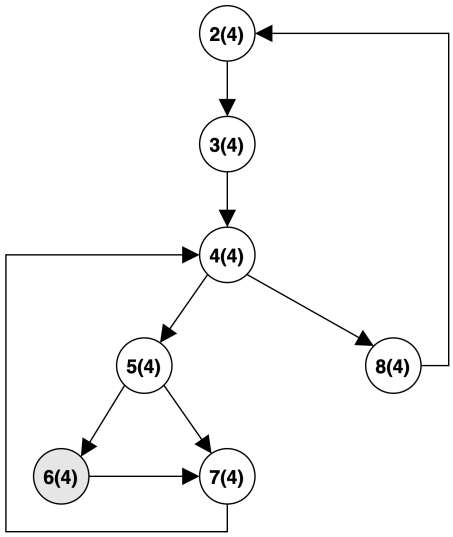
Fonte – Concilio *et al.* (2021)

4.1.4 Ao redor do uso

Ao atravessar um dos caminhos de *SG3*, a ADU $(8, 6, a)$ é coberta. Agora a execução do caso de teste deve ser concluída. Ou seja, devemos determinar um subgrafo que inclua todos os caminhos do nó de uso para o nó de saída. Digamos que o caminho

(6,5,7,4,5,7,4,8,2,3,4,5,6) seja percorrido pelo caso de teste antes de atingir o nó de saída. De maneira semelhante aos caminhos descritos no *SG2*, o nó de uso pode ser visitado várias vezes antes de a execução terminar no nó de saída. O subgrafo *SG4* apresentado na Figura 10 descreve todos os caminhos que começam no nó 6 e terminam no nó 6 (indicado em cinza na Figura 10). O nó de entrada e de saída do SG4 é o nó 6(4), que é o nó de uso.

Figura 10 – *SG4*



Fonte – Concilio *et al.* (2021)

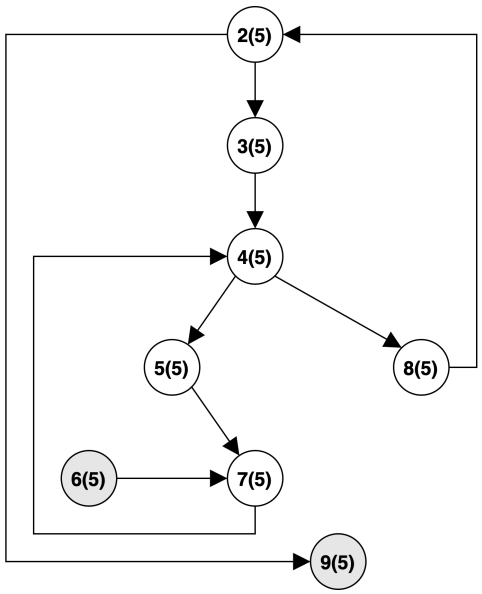
4.1.5 Alcançando o nó de saída

Em algum momento, a execução termina, isto é, existem caminhos do nó de uso para o nó de saída. A Figura 11 descreve todos os caminhos que começam no nó 6 e terminam no nó de saída. O nó de entrada é o nó 6(5) e o nó de saída 9(5).

4.1.6 Graphdua(8,6 a)

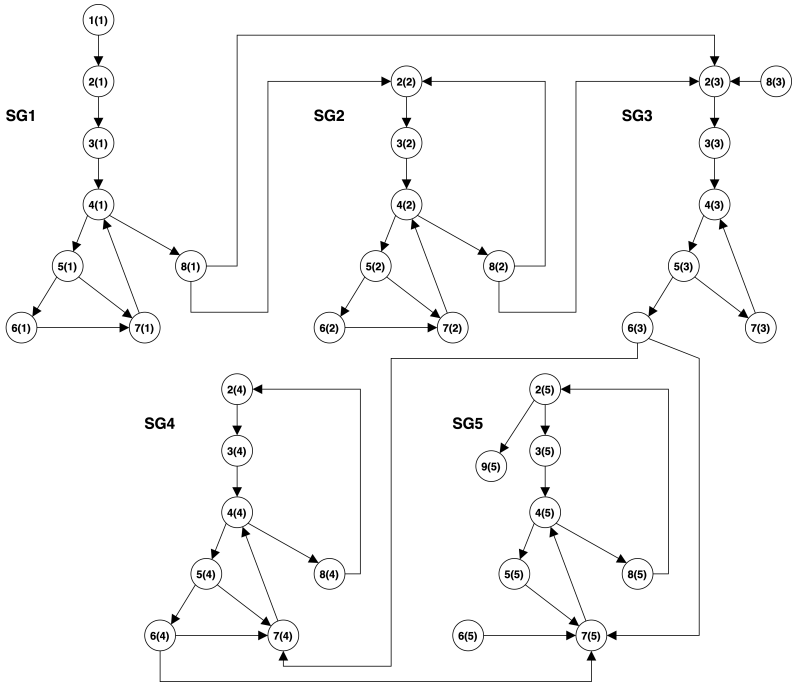
A representação gráfica da cobertura de ADUs consiste em conectar *SG1*, *SG2*, *SG3*, *SG4* e *SG5*. Assim, conectando os subgrafos das Figuras 7, 8, 9, 10 e 11, é obtido um grafo que compreende todos os caminhos completos que cobrem a ADU $(8, 6, a)$. A Figura 12 apresenta este grafo, chamado $graphdua(8, 6, a)$, que resulta da combinação de todos os subgrafos obtidos para a ADU $(8, 6, a)$.

Figura 11 – *SG5*



Fonte – Concilio *et al.* (2021)

Figura 12 – $graphdua(8, 6, a)$ — Grafo de todos os caminhos completos que satisfazem a ADU $(8, 6, a)$.



Fonte – Concilio *et al.* (2021)

O nó de saída de *SG1* é conectado aos sucessores do nó de entrada de *SG2* e *SG3*. Isso ocorre porque um caminho que cobre $(8, 6, a)$, após atingir o nó de definição percorrendo *SG1*, pode ir diretamente para o nó de uso de *SG3* ou pode percorrer *SG2* antes de percorrer *SG3*. O nó de saída de *SG1* não se conecta ao nó de entrada de *SG2* e de *SG3* porque eles são o mesmo nó no grafo de fluxo original (no caso, nó 8).

SG2, por sua vez, se conecta aos sucessores do nó inicial de *SG3*. As conexões de *SG3* são semelhantes às de *SG1*. Ele se conecta a *SG4* se o caminho incluir um percurso pelo nó de uso; em seguida, ele se conecta a *SG5* para alcançar o nó de saída do programa. Analogamente a *SG2*, *SG4* se conecta a *SG5*.

Como um nó n de G pode ocorrer mais de uma vez no *graphdua*, utilizamos o par $n(id)$ para identificar unicamente um nó do *graphdua*, sendo que id identifica o subgrafo e n refere-se ao nó do grafo de fluxo original. Como resultado, percorrendo $graphdua(8, 6, a)$ e imprimindo o primeiro elemento do par de identificação, obteremos qualquer caminho completo que cobre $(8, 6, a)$.

Observe na Figura 12 que uma *graphdua* pode possuir nós inalcançáveis, como 8(3) e 6(5). Como o nó de saída de *SG2*, nó 8(2), está conectado ao sucessor do nó de entrada de *SG3*, 2(3), e não há arco de *SG3* atingindo seu nó de entrada 8(3), ele se torna inalcançável. O mesmo ocorre quando o *SG4* está conectado a *SG5* e 6(5) se torna inalcançável. Assim, quando $graphdua(8, 6, a)$ é percorrido, eles não são impressos.

A seguir, apresentamos a definição formal de *graphdua* e mostramos como ela é construída.

4.2 Graphdua — definição formal e construção

A ideia fundamental do $graphdua(D)$ é gerar um grafo que descreve todos os caminhos que cobrem uma determinada ADU c-uso $D = (d, u, X)$[1]. A seguir é apresentada a definição formal da representação em forma de grafo da cobertura de fluxo de dados, isto é, de um $graphdua(D)$.

4.2.1 Definição

Seja $D = (d, u, X)$ uma ADU requerida para testar um programa P de acordo com o critério todos-usos. Um $graphdua(D)$ é dado pela tupla $G_D = (N_D, E_D, s_D, e_D)$ tal que:

1. N_D é o conjunto de nós $n(I)$ onde n é um nó do grafo de fluxo $G(N, E, s, e)$ de P e I é o identificador da instância de n em G_D;

[1] A construção da graphdua para uma ADU p-uso $(d, (u', u), X)$ pode ser reduzida ao problema de encontrar a graphdua para ADUs c-uso. Na seção 4.2.4, discutimos como isso é realizado.

2. E_D é o conjunto de arcos $(m(I), n(I'))$ onde (m, n) é um arco de G e I e I' são identificadores das instâncias dos nós m e n em G_D;
3. $s_D = s(I)$ e $e_D = e(I')$ são, respectivamente, os nós de entrada e saída de G_D, onde s e e são os nós de entrada e saída de G e I e I' são identificadores das instâncias dos nós s e e em G_D; e
4. qualquer caminho que cubra D pode ser obtido percorrendo $graphdua(D)$ e imprimindo o valor n para cada nó $n(I) \in N_D$ visitado.

4.2.2 Encontrando os subgrafos

Conforme discutido no início desse capítulo, o *graphdua* é construído encontrando subgrafos e conectando-os. Bertolino e Marré (1994) propuseram um algoritmo para determinar o subgrafo $SG(n_i, n_j)$ entre os nós n_i e n_j de um grafo de fluxo G ao qual nos referimos por **encontrarSubGrafo**[2]. Os subgrafos *SG1*, *SG2*, *SG4* e *SG5* que compõem o *graphdua* são criados utilizando **encontrarSubGrafo**. Esses subgrafos contêm todos os caminhos de um nó n_i para o nó n_j.

No entanto, quando uma ADU (d, u, X) é coberta, deve existir pelo menos um caminho do nó de definição d para o nó de uso u, de modo que a variável X não seja redefinida, isto é, um caminho livre de definição de d a u c.r.a. X. *SG3* é o subgrafo que contém esses caminhos.

As mesmas autoras (MARRÉ; BERTOLINO, 1996) propuseram uma variante do algoritmo para determinar subgrafos que compreendem caminhos livre de definição ao qual chamaremos **encontrarLivreDefSubGrafo**[3]. Essencialmente, **encontrarLivreDefSubGrafo** é o mesmo algoritmo **encontrarSubGrafo** com uma única alteração.

O algoritmo **encontrarLivreDefSubGrafo** verifica se existe uma definição da variável X sempre que um nó é visitado, exceto o primeiro (n_i) e o último (n_j). Nesse caso, o nó não pode fazer parte de *SG3*, pois o subgrafo só pode conter caminhos livre de definição c.r.a. X. O custo dos algoritmos **encontrarSubGrafo** e **encontrarLivreDefSubGrafo** é $O(E)$ (BERTOLINO; MARRÉ, 1994). Utilizamos os seguintes parâmetros para invocá-los em nosso algoritmo para criar $graphdua(D)$.

encontrarSubGrafo :

[2] O algoritmo **encontrarSubGrafo** proposto por Bertolino e Marré (1994) está descrito no apêndice A.
[3] O apêndice A contém a descrição de **encontrarLivreDefSubGrafo**.

entrada – Grafo de fluxo $G(N, E, s, e)$; nós n_i e $n_j \in N$; e o identificador do subgrafo I.

saída – $SGI(n_i, n_j)$ representado por um grafo $G_I(N_I, E_I, s(I), e(I))$ que contém todos os caminhos de n_i a n_j.

encontrarLivreDefSubGrafo :

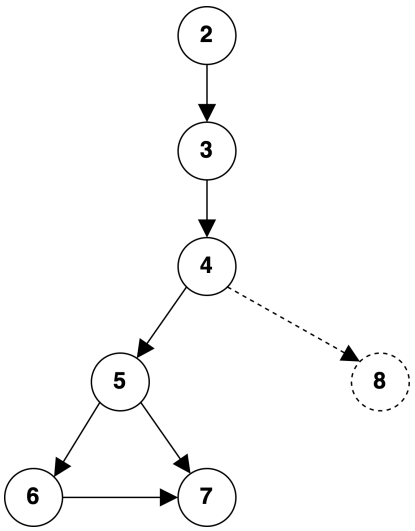
entrada – Grafo de fluxo $G(N, E, s, e)$; nós n_i e $n_j \in N$; variável X; conjunto $defs(n)$ de todos os nós $n \in N$; e o identificador de subgrafo I.

saída – $SGI(n_i, n_j)$ representado por um grafo $G_I(N_I, E_I, s(I), e(I))$ que contém todos os caminhos livres de definição de n_i a n_j c.r.a. variável X.

Contudo, ao utilizar **encontrarSubGrafo** e **encontrarLivreDefSubGrafo**, como definidos pelas autoras para gerar *graphduas*, verificamos que eles podem incluir caminhos que não são partes do subgrafo. Esses caminhos são denominados nesta dissertação *caminhos pendentes*.

Por exemplo, se **encontrarSubGrafo** for utilizado para determinar o subgrafo $SG(2, 7)$ do grafo de fluxo do programa Sort (Figura 3), o subgrafo da Figura 13 será gerado. O arco e nó pontilhados representam um *caminho pendente* e não fazem parte do subgrafo $SG(2, 7)$.

Figura 13 – Subgrafo com caminho pendente



Fonte – Concilio *et al.* (2021)

O Algoritmo 1, chamado **limparCaminhosPendentes**, foi desenvolvido para eliminar os caminhos pendentes que podem ser gerados por **encontrarSubGrafo** e

encontrarLivreDefSubGrafo. Ele recebe como parâmetros o subgrafo e uma lista de nós para serem verificados; i.e., nós que não possuem sucessores e não são o nó de saída. No exemplo do subgrafo $SG(2,7)$, o algoritmo recebe o subgrafo da Figura 13 e o nó 8 porque esse nó não é o nó de saída do subgrafo (no exemplo, o nó 7) e não possui sucessores.

O Algoritmo 1 também custa $O(E)$ pois visita no máximo três vezes todos os arcos do grafo de fluxo original. Como resultado, os custos de **encontrarSubGrafo** e **encontrarLivreDefSubGrafo** não são alterados ao adicionarmos a limpeza de caminhos pendentes nesses algoritmos.

Algoritmo 1 Algoritmo limparCaminhosPendentes

Input Subgrafo $SG(N_{SG}, E_{SG}, s_{SG}, e_{SG})$; e conjunto $NosVerificar$ contendo os nós a serem verificados

Output Subgrafo SG contendo todos os caminhos de n_i até n_j

1: **function** LIMPARCAMINHOSPENDENTES($SG, NosVerificados$)
2: $W \leftarrow e_{SG}$ ▷ Visita os nós a partir do nó de saída
3: $PredVisitados \leftarrow \emptyset$
4: **while** $W \neq \emptyset$ **do**
5: $n \leftarrow$ **removerNo**(W)
6: **foreach** $n_{pred} \in Predecessores(n)$ **do**
7: **if** $n_{pred} \notin PredVisitados$ **then**
8: $PredVisitados \leftarrow PredVisitados \bigcup n_{pred}$
9: $VerificaVisitados \leftarrow \emptyset$ ▷ Visita os nós a partir dos nós a serem verificados
10: **while** $NosVerificar \neq \emptyset$ **do**
11: $n \leftarrow$ **removerNo**($NosVerificar$)
12: **foreach** $n_{pred} \in Predecessores(n)$ **do**
13: **if** $n_{pred} \notin VerificaVisitados$ **then**
14: $VerificaVisitados \leftarrow VerificaVisitados \bigcup n_{pred}$
15: $LimparNos \leftarrow VerificaVisitados - PredVisitados$ ▷ Limpa caminhos pendentes
16: **while** $LimparNos \neq \emptyset$ **do**
17: $n \leftarrow$ **removerNo**($LimparNos$)
18: **foreach** $n_{pred} \in Predecessores(n)$ **do**
19: $E_{SG} \leftarrow E_{SG} - (n_{pred}, n)$
20: **foreach** $n_{suc} \in Sucessores(n)$ **do**
21: $E_{SG} \leftarrow E_{SG} - (n, n_{suc})$
22: $N_{SG} \leftarrow N_{SG} - n$
23: **return** $SG(N_{SG}, E_{SG}, s_{SG}, e_{SG})$

Fonte – Concilio *et al.* (2021)

4.2.3 Encontrando a graphdua

O Algoritmo 2, chamado **encontrarGraphdua**, encontra a *graphdua* de uma ADU $D = (d, u, X)$. Seus parâmetros de entrada são um grafo de fluxo G, uma ADU D e o conjunto de definições ($defs(n)$) que ocorrem em todos os nós n de G. A saída é o $graphdua(D)$.

Inicialmente, as linhas 2 a 6 invocam **encontrarSubGrafo** e **encontrarLivreDefSubGrafo** para determinar *SG1*, *SG2*, *SG3*, *SG4* e *SG5*. O único subgrafo que existe para todas as ADUs é *SG3*; os demais podem ou não existir, dependendo das características da ADU. Por exemplo, a ADU $(1, (2, 9), n)$ possui apenas *SG3*, pois o nó de entrada é igual ao nó de definição e o nó de destino do arco é igual ao nó de saída. Logo, não existem *SG1*, *SG2*, *SG4* e *SG5*. O algoritmo **encontrarSubGrafo** retorna nulo sempre que um subgrafo não puder ser encontrado. Depois de criados, os subgrafos são conectados.

As linhas 7 e 8 calculam os conjuntos de nós N_D e de arcos E_D do $graphdua(D)$. Eles são calculados por meio da união, respectivamente, dos nós e dos arcos de todos os subgrafos. Subgrafos nulos possuem conjuntos vazios de nós e arcos.

As linhas de 9 a 14 e de 17 a 18 conectam *SG1* a outros subgrafos. A linha 9 verifica se *SG1* existe e, caso ele exista, a linha 10 atribui o nó de entrada de *SG1*, $s(1)$, ao nó de entrada de $graphdua(D)$; as linhas 11 a 12 conectam o último nó de *SG1*, o nó de definição $d(1)$, aos sucessores do nó de entrada de *SG3*, $d(3)$. Eles fazem isso adicionando novos arcos a E_D que conectam os nós de *SG1* aos nós de *SG3*. Quando *SG1* não existe, o nó de entrada de *SG3*, $d(3)$, é o nó inicial de $graphdua(D)$ (linha 14). As linhas 16 e 17 conectam *SG1* a *SG2*, se *SG2* existir. As linhas 18 a 19, quando *SG2* não é nulo, conectam *SG2* a *SG3*.

Analogamente, as linhas 20 a 24 conectam *SG4* a *SG3* (linhas 21 e 22) e a *SG5* (linhas 23 e 24), quando *SG4* é diferente de nulo. Finalmente, as linhas 25 a 28, se *SG5* não for nulo, fazem o nó de saída de *SG5* igual ao nó de saída de $graphdua(D)$ (linha 26) e conectam *SG3* a *SG5* (linhas 27 e 28). Quando *SG5* é nulo, o nó de saída do $graphdua(D)$ é o nó de saída do *SG3* (linha 30).

Algoritmo 2 Algoritmo encontrarGraphdua

Input Grafo de fluxo $G(N, E, s, e)$; ADU $D = (d, u, X)$; e conjunto $defs(n)$ para cada nó $n \in N$.

Output $graph(D)$ dado pelo grafo $G_D(N_D, E_D, s_D, e_D)$ que contém todos os caminhos que cobrem a ADU D

1: **function** ENCONTRARGRAPHDUA($G, D, defs(n)$)
2: $SG1 \leftarrow$ **encontrarSubGrafo**$(G, s, d, 1)$
3: $SG2 \leftarrow$ **encontrarSubGrafo**$(G, d, d, 2)$
4: $SG3 \leftarrow$ **encontrarLivreDefSubGrafo**$(G, d, u, X, defs(n), 3)$
5: $SG4 \leftarrow$ **encontrarSubGrafo**$(G, u, u, 4)$
6: $SG5 \leftarrow$ **encontrarSubGrafo**$(G, u, e, 5)$
7: $N_D \leftarrow N_1 \bigcup N_2 \bigcup N_3 \bigcup N_4 \bigcup N_5$
8: $E_D \leftarrow E_1 \bigcup E_2 \bigcup E_3 \bigcup E_4 \bigcup E_5$
9: **if** $SG1 \neq null$ **then**
10: $s_D \leftarrow s(1)$
11: **foreach** $n_{suc} \in Sucessores(d(3))$ **do**
12: $E_D \leftarrow E_D \bigcup (d(1), n_{suc})$
13: **else**
14: $s_D \leftarrow d(3)$
15: **if** $SG2 \neq null$ **then**
16: **foreach** $n_{suc} \in Sucessores(d(2))$ **do**
17: $E_D \leftarrow E_D \bigcup (d(1), n_{suc})$
18: **foreach** $n_{suc} \in Sucessores(d(3))$ **do**
19: $E_D \leftarrow E_D \bigcup (d(2), n_{suc})$
20: **if** $SG4 \neq null$ **then**
21: **foreach** $n_{suc} \in Sucessores(u(4))$ **do**
22: $E_D \leftarrow E_D \bigcup (u(3), n_{suc})$
23: **foreach** $n_{suc} \in Sucessores(u(5))$ **do**
24: $E_D \leftarrow E_D \bigcup (u(4), n_{suc})$
25: **if** $SG5 \neq null$ **then**
26: $e_D \leftarrow u(5)$
27: **foreach** $n_{suc} \in Sucessores(u(5))$ **do**
28: $E_D \leftarrow E_D \bigcup (u(3), n_{suc})$
29: **else**
30: $e_D \leftarrow u(3)$
31: **return** $G_D(N_D, E_D, s(I), e(I'))$

Fonte – Concilio *et al.* (2021)

4.2.4 Geração de subgrafos livres de definição para ADUs p-uso

Para ADUs p-uso como $(d, (u', u), X)$ subgrafos *SG1*, *SG2*, *SG4* e *SG5* podem ser calculados usando **encontrarSubGrafo** da mesma maneira que são calculadas para ADUs c-uso. No entanto, o subgrafo *SG3* das ADUs p-uso exige tratamento especial

quando o nó de destino u possui mais de um predecessor. Considere a ADU $(8,(5,7),a)$ e o grafo de fluxo de controle anotado (Figura 3) do programa *Sort* (Figura 1). Todos os caminhos do nó 8 até o arco (5,7) são livres de definição c.r.a. variável a. No entanto, o nó 7 possui dois predecessores: nós 5 e 6. Portanto, ele pode ser alcançado através de 5 e 6. Se for precedido por 6, o nó 7 ocorrerá pelo menos duas vezes no caminho em direção ao arco (5,7), uma vez que ele é sempre o último nó.

O algoritmo de Bertolino e Marré (1994), porém, assume que o nó final (neste caso o nó 7) ocorrerá apenas uma vez nos caminhos contidos pelo subgrafo gerado[4]. Para superar essa restrição, modificamos o grafo de fluxo original para calcular o *SG3* das ADUs p-uso. A Figura 15(a) mostra parte de um grafo de fluxo anotado no qual existe um arco (u',u) com um p-uso da variável X e um nó d com uma definição de X. Conforme mostrado na Figura, o nó u possui vários predecessores. Suponhamos que exista um caminho livre de definição de d para (u',u) e, como resultado, uma ADU p-uso $(d,(u',u),X)$.

Figura 14 – Modificação do grafo de fluxo para encontrar *SG3* para ADUs p-uso.

(a) (b)

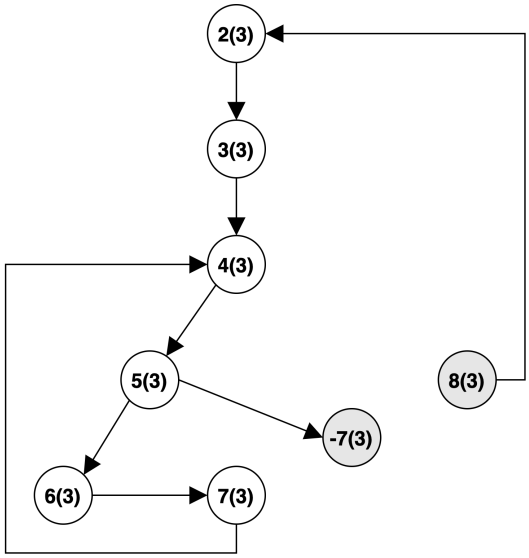
Fonte – Concilio *et al.* (2021)

O grafo de fluxo original é modificado da seguinte forma: o arco (u',u) é removido; e um novo nó $-u$ (menos u) é adicionado, além de um arco $(u',-u)$. Optamos por chamar o novo nó $-u$ para diferenciá-lo do nó u que pode ocorrer mais de uma vez no caminho para o arco (u',u). Embora tenha um identificador diferente $(-u)$, ele tem o mesmo código e, consequentemente, é anotado com os mesmos conjuntos de definições, c-usos e p-usos do nó u. Seu objetivo é permitir a aplicação de **encontrarLivreDefSubGrafo** em ADUs p-uso. A Figura 15(b) mostra o novo grafo de fluxo resultante das modificações. Como resultado, **encontrarLivreDefSubGrafo** pode ser aplicado tendo como parâmetros d para o primeiro nó (n_i) e $-u$ para o último nó (n_j) do subgrafo. Ao fazer isso, o problema

[4] De fato, as autoras utilizam um grafo de fluxo no qual instruções sequenciais são associadas a arcos e instruções de transferência condicional a nós. Essa notação não diferencia entre ADUs c-uso e p-uso.

de encontrar *SG3* para ADUs p-uso é reduzido ao de encontrar *SG3* para ADUs c-uso. A Figura 15 mostra o *SG3* gerado para a ADU $(8, (5, 7), a)$.

Figura 15 – *SG3* — Subgrafo de caminhos livre de definição c.r.a. variável a que começam no nó 8 e terminam no arco (5,7).



Fonte – Concilio *et al.* (2021)

4.2.5 Análise de custo

Como mostrado por Marré e Bertolino (1996), o custo para determinar o subgrafo $SG(n_i, n_j)$ de um grafo G é $O(E)$ onde E é o tamanho do conjunto de arcos E de G. Como **encontrarGraphdua** determina no máximo cinco subgrafos, o custo para encontrar todos os subgrafos também é $O(E)$. Mesmo adicionando o algoritmo **limparCaminhosPendentes** depois do cálculo de cada subgrafo, esse custo não é alterado uma vez que **limparCaminhosPendentes** também é $O(E)$ (ver Seção 4.2.2).

O algoritmo **encontrarGraphdua** (Algoritmo 2) estabelece conexões entre *SG1* e *SG2* e *SG3*, entre *SG2* e *SG3*, entre *SG3* e *SG4* e *SG5*, e, finalmente, entre *SG4* e *SG5*. Assim, no máximo, serão estabelecidas seis conexões entre grafos. Cada conexão consiste em criar novos arcos entre o nó de saída do subgrafo precedente com os sucessores do nó de entrada do próximo subgrafo. Como os novos arcos são limitados ao número de arcos de E de G, o custo para conectar dois subgrafos é $O(E)$. Desta maneira, o custo de **encontrarGraphdua** é igualmente $O(E)$.

Portanto, o custo combinado de encontrar subgrafos e conectá-los é $O(E)$, o que torna o custo para encontrar o *graphdua*(D) também $O(E)$.

4.2.6 Prova de correção

A seguir, provamos que *graphdua*(D) compreende todos os caminhos que cobrem uma ADU D.

Teorema 1. *Seja $\pi = (s, \ldots, d, \ldots, u, \ldots, e)$ um caminho completo que cobre uma ADU $D = (d, u, X)$ requerida para testar um programa P e graphdua(D) a representação em grafo da cobertura de D gerada utilizando o Algoritmo 2. Então existe um percorrimento de graphdua(D) que gera π imprimindo o primeiro elemento de cada nó visitado do graphdua(D) durante o percorrimento.*

Demonstração. Iremos provar o teorema por contradição. Suponhamos que exista um caminho completo $\pi' = (s, n_1, \ldots, n_{k-1}, d, n_{k+1}, \ldots, n_{m-1}, u, n_{m+1}, \ldots, n_{p-1}, e)$, onde $s = n_0$, $d = n_k$, $u = n_m$, e $e = n_p$, que cobre D, mas não pode ser obtido percorrendo *graphdua(D)*.

Como π' cobre D, ele pode ser dividido em três caminhos: $\pi' = (s, \ldots, n_{k-1}, d)$, $(n_{k+1}, \ldots, n_{m-1}, u)$, $(n_{m+1}, \ldots, n_{p-1}, e)$. Esses caminhos são tratados nos casos 1, 2 e 3 a seguir.

Caso 1. $(s, n_1, \ldots, n_{k-1}, d)$ é um caminho entre o nó s e o nó d. Os subgrafos componentes *SG1* e *SG2* de *graphdua*(D) contêm todos os caminhos que conectam o nó de entrada s ao nó de definição d sem e com a repetição do nó d. Portanto, percorrendo *SG1* e depois *SG2* obteremos o caminho $(s, n_1, \ldots, n_{k-1}, d)$.

Caso 2. $(n_{k+1}, \ldots, n_{m-1}, u)$ é um caminho livre de definição c.r.a. variável X entre o nó d (não incluso) e o nó u. Por definição, *SG3* contém todos os caminhos livres de definição c.r.a. X de d a u. Os componentes *SG1* e *SG2* são conectados aos sucessores do nó $d(3)$ de *SG3* no *graphdua*(D). Consequentemente, o caminho livre de definição $(n_{k+1}, \ldots, n_{m-1}, u)$ c.r.a. X pode ser obtido atravessando o componente *SG3* de *graphdua*(D).

Caso 3. $(n_{m+1}, \ldots, n_{p-1}, e)$ é um caminho entre o nó u (não incluso) e o nó e. Os subgrafos *SG4* e *SG5* de *graphdua*(D) contêm todos os caminhos desde o nó de uso u até o nó de saída e sem e com a repetição do nó u. O componente *SG3* de *graphdua*(D)

é conectado aos sucessores de $u(4)$ e $u(5)$ de, respectivamente, *SG4* e *SG5*. Portanto, obteremos $(n_{m+1}, \ldots, n_{p-1}, j)$ percorrendo os componentes *SG4* e *SG5* de $graphdua(D)$.

Portanto, todos os caminhos que compõem π' podem ser obtidos percorrendo $graphdua(D)$, o que significa que π' pode ser obtido a partir de $graphdua(D)$. É uma contradição. Assim sendo, um caminho que cobre D mas que não pode ser gerado por $graphdua(D)$ não existe. □

4.3 Considerações finais

Apresentamos neste capítulo a estrutura *graphdua* por meio de um exemplo, definimo-la formalmente, apresentamos os algoritmos para calculá-la e a análise de seus custos. O capítulo foi concluído apresentando uma prova de que a *graphdua(D)* representa todos os caminhos que cobrem a ADU D.

No próximo capítulo a aplicação da estrutura *graphdua* para cálculo da relação de subsunção entre ADUs será apresentada.

5 Aplicação da estrutura graphdua no cálculo da relação de subsunção de ADUs

Neste capítulo é apresentada a aplicação da estrutura de dados *graphdua* para encontrar a relação de subsunção entre associações definição-uso (ADU). Inicialmente, apresentamos a subsunção local ADU-nó; em seguida, mostramos como ela é utilizada juntamente com a estrutura *graphdua* para calcular a subsunção de ADUs. O capítulo é concluído com a apresentação de uma avaliação experimental cujo objetivo é avaliar se as estruturas *graphduas* podem ser calculadas em escala e se a sua aplicação para encontrar a subsunção de fluxo de dados leva à redução dos custos do teste.

5.1 Subsunção local ADU-nó

A subsunção ADU-nó identifica as ADUs que são cobertas quando um nó específico do grafo é visitado. Formalmente, uma ADU $D = (d, u, X)$ ou $D = (d, (u', u), X)$ é uma *ADU-subsumida* pelo nó n se D é coberta em todos os caminhos de teste que visitam o nó n e alcançam o nó de saída. O conjunto de ADUs subsumidas pelo nó n é o conjunto de todas as ADUs que são cobertas por todos os caminhos de teste que visitam n.

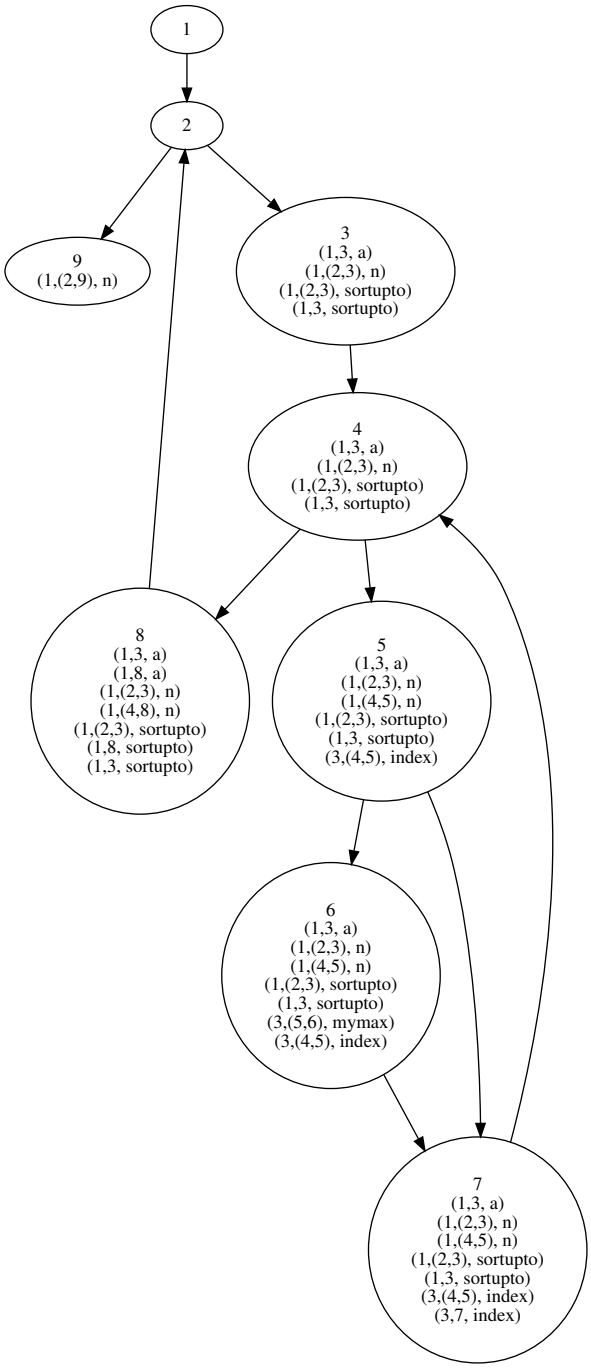
Há diferença entre *subsunção ADU-nó local* — conjunto de ADUs cobertas por todos os caminhos que **alcançam** n; e *subsunção ADU-nó global* — conjunto de ADUs cobertas por todos os caminhos de teste que alcançam n e continuam até o nó de saída. O conjunto de ADUs globalmente subsumidas inclui ADUs que são ADU-nós subsumidas por nós contidos em todos os caminhos do nó n ao nó de saída.

A subsunção local de ADUs em um nó particular n foi introduzida para tratar as situações de *interrupção da execução*, i.e., exceções ou outras maneiras de abortar um programa (CHAIM; BARAL; OFFUTT, 2021; CHAIM *et al.*, 2021).

Figura 16 mostra os conjuntos de subsunção de ADUs por nós para o programa Sort (Figuras 1 e 2). Cada nó contém as ADUs subsumidas localmente. Por exemplo, se o nó 5 é alcançado, a definição de *sortupto* no nó 1 e seu uso no nó 3 são garantidamente cobertos.

Caso o programa não termine precocemente, o nó 5 é pós-dominado pelos nós 7, 8, 2 e 6. Quando eles são visitados por um caminho de teste, o conjunto de ADUs que são globalmente subsumidas pelo nó 5 inclui as seis ADUs listadas no nó 5, mais a ADU

Figura 16 – Subsunção ADU-nó local para o programa Sort



Fonte – Concilio *et al.* (2021)

$(3, 7, index)$ do nó 7 e as ADU $(1, 8, a)$, $(1, (4, 8), n)$ e $(1, 8, sortupto)$ do nó 8. Portanto, o nó 5 subsume localmente seis ADUs e subsume globalmente dez ADUs.

Se todos os nós de Sort forem visitados ao menos uma vez, um total de 13 ADUs de um total de 44 ADUs serão cobertas. Deste modo, a cobertura de nós resultaria em uma cobertura de fluxo de dados de 27%.

Chaim, Baral e Offutt (2021) desenvolveram o *Algoritmo de Subsunção* (AS) para calcular subsunção local de ADUs. O custo de AS para calcular a relação de subsunção local ADU-nó é ≈O(|N|) onde N é o número de nós. A descrição detalhada de AS, o cálculo de sua complexidade e a prova de correção podem ser obtidas nas referências (CHAIM; BARAL; OFFUTT, 2021; CHAIM *et al.*, 2021).

5.2 Encontrando a subsunção entre requisitos de fluxo de dados

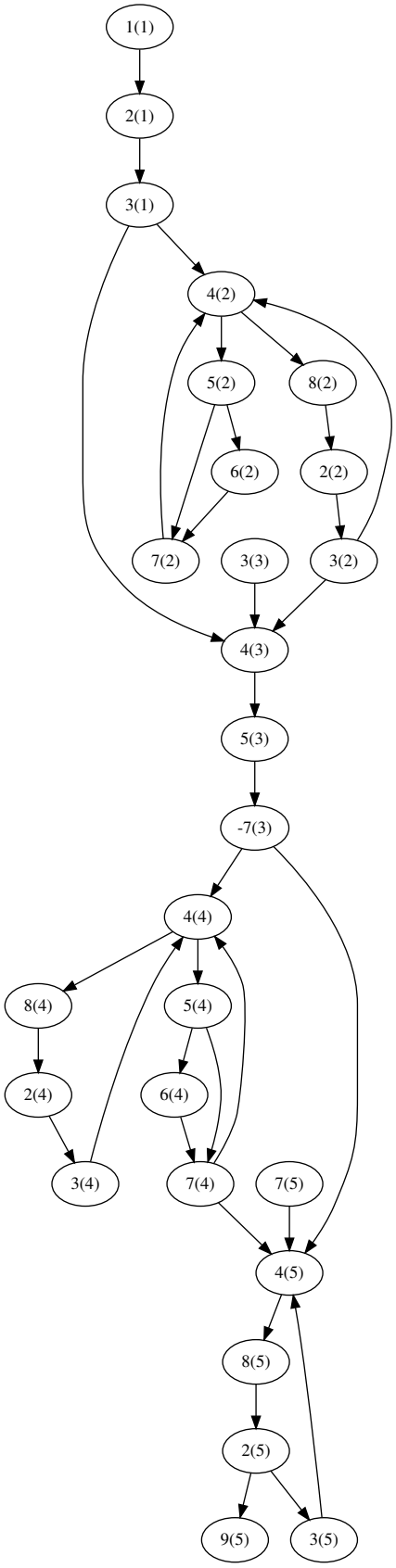
Para cada ADU D_1, a subsunção entre requisitos de fluxo de dados, isto é, subsunção ADU-ADU, associa um conjunto de ADUs D_2 que são cobertas em todos os caminhos de teste que cobrem uma outra ADU D_1. O Algoritmo de Subsunção (AS) é aplicado nos *graph(D)* de cada ADU exigida pelo critério todos-usos para encontrar a subsunção ADU-ADU.

Dada uma ADU $D_1 = (d_1, u_1, X_1)$, a estrutura $graphdua(D_1)$ é calculada e, em seguida, o AS é executado no grafo resultante, isto é, $graphdua(D_1)$. A saída de AS são conjuntos *Covered* associados a cada nó de $graphdua(D_1)$; *Covered(n(I))* contém as ADUs que são cobertas sempre que o nó $n(I)$ é atingido por caminhos contidos em $graphdua(D_1)$. A Figura 17 mostra o *graphdua* gerado para a ADU (3,(5,7), mymax). AS receberá como entrada *graphdua(3,(5,7), mymax))* e devolverá os conjuntos *Covered* associado a cada um dos nós da Figura.

O conjunto $Covered(e_{D_1})$, onde e_{D_1} é o nó de saída de $graphdua(D_1)$, contém o conjunto de ADUs que são cobertas em caminhos que saem do nó de entrada até o nó de saída de $graphdua(D_1)$, ou seja, as ADUs D_2 que são subsumidas em qualquer caminho cobrindo D_1. No $graphdua(3, (5, 7), mymax)$ (Figura 17), o conjunto *Covered(9(5))* contém todas as ADUs subsumidas pela ADU (3,(5,7), mymax).

No que diz respeito ao custo, $graphdua(D_1)$ custa para ser calculada, onde |E| é o número de arcos (MARRÉ, 1997); por isso, seu custo é $O(|N|)$, assumindo que o número de arcos é proporcional ao número de nós, o que ocorre para a maioria dos programas. O custo de AS é determinado pelo número de nós do grafo no qual ele é aplicado. Uma *graphdua* não contém mais do que cinco vezes o número de nós do grafo de fluxo do programa, logo, o número de nós de uma *graphdua* é $O(|N|)$. Como resultado, executar AS em uma *graphdua* custa $\approx O(|N|)$. Consequentemente, calcular as ADUs subsumidas por

Figura 17 – *Graphdua* gerado para a ADU $(3, (5, 7), mymax)$



Fonte – Concilio *et al.* (2021)

D_1 é $\approx O(|\mathrm{N}|)$. Se U é o conjunto de todas as ADUs necessárias para testar um programa, a subsunção ADU-ADU irá custar $\approx O(|\mathrm{U}||\mathrm{N}|)$ para ser calculada.

No entanto, para calcular as ADUs irrestritas, isto é, as folhas do grafo da Figura 4, a relação de subsunção ADU-ADU não é suficiente. É preciso primeiro criar um grafo que represente as relações de subsunção entre ADUs $D_2 \rightarrow D_1$, chamado de grafo de subsunção, e depois calcular as regiões fortemente conectadas desse grafo. O resultado é desses passos é o grafo da Figura 4. Segundo Marré e Bertolino (1996), o cálculo das ADUs irrestritas custa até $O(|\mathrm{U}|^2)$. Porém, implementando o conjunto de ADUs subsumidas como vetores de bits e percorrendo-os com instruções de máquina esse custo é reduzido.

5.3 Avaliação experimental da aplicação da graphdua para subsunção de fluxo de dados

Nesta seção é apresentada uma avaliação experimental cujo objetivo é avaliar se as estruturas *graphduas* podem ser calculadas em escala e se a sua aplicação para encontrar a subsunção de fluxo de dados leva à redução dos custos do teste. Três questões de pesquisa são investigadas:

QP1: Quanto tempo é necessário para encontrar as *graphduas* de um programa?

QP2: Quanto esforço de teste é economizado usando ADUs irrestritas?

QP3: As *graphduas* representam corretamente os caminhos que cobrem uma ADU?

O restante desta seção apresenta e discute os resultados da avaliação experimental.

5.3.1 Resultados

Para nosso estudo, foram utilizados 17 programas do repositório *Defects4J* (JUST; JALALI; ERNST, 2014), mais o programa de aprendizado de máquina *Weka*. Selecionamos as primeiras versões defeituosas do *Defects4J* (refenciadas como 1b) e a versão 3.8 do *Weka*. As finalidades dos programas variam: manipulação de texto em arquivos comprimidos e binários (*Compress*, *Csv*, *Gson*, *JacksonCore*, *JacksonDataBind*, *JacksonXml* e *JSoup*); análise sintática e compilação (*Cli*, *Closure* e *JXPath*); manipulação de estruturas de dados e utilitários (*Collections* e *Lang*); matemática, estatística e mineração de dados (*Math* e *Weka*); manipulação de data e hora (*Time*); e teste de software (*Mockito*). O tamanho

dos programas também é diversificado: variam de pequenos programas, como *Csv* (929 ADUs), a programas maiores, como *Weka* (337.063 ADUs). A Tabela 3 apresenta métricas associadas aos programas — linhas de código (coluna *LOC*), número de métodos que possuem ADUs (coluna *Métodos com ADUs*) e número de métodos executados pelos testes (coluna *Métodos executados*) — e o total de ADUs requiridas para testar os programas (coluna *ADUs*).

Tabela 3 – Métricas e ADUs requiridas pelos programas

Programa	LOC	Métodos com ADUs	Métodos executados	ADUs
Csv	602	40	37	929
Cli	1107	60	54	1291
Codec	1946	109	92	4446
Jsoup	2046	136	119	1866
JacksonXml (J-Xml)	3084	179	124	3402
Compress	4974	217	116	6286
Gson	3840	226	191	3281
Mockito	7468	400	*	4236
JacksonCore (J-Core)	10.978	563	383	17.653
JxPath	14.699	800	691	20.178
Lang	15.270	1157	*	22.290
Time	19.672	1182	1010	18.160
Collections	18.156	1311	1094	16.937
JacksonDatabind (J-DataBind)	27.274	1737	1266	31.797
Math	54.518	2415	1999	87.603
Chart	68.346	3219	2151	81.847
Closure	61.177	3696	3241	78.068
Weka	216.781	11.315	1964	337.063
Total	531.938	28.758	14.532	737.333

Fonte – Chaim *et al.* (2021)

A Tabela 5.3.1 apresenta os dados experimentais da aplicação do algoritmo para encontrar *graphduas* nos programas do estudo. A Tabela contém o nome de cada programa, o número de ADUs (**#ADUs**), a porcentagem de ADUs irrestritas c.r.a. total de ADUs (**Irr.**), o tempo médio em milissegundos necessário para encontrar todas *graphduas* — medido depois de dez execuções (**TGra.**), o tempo em milissegundos necessário para descobrir as ADUs irrestritas (**TIrr.**), e a proporção entre o tempo para calcular as *graphduas* e o tempo total para determinar as ADUs irrestritas (**Prop.**).

Implementamos e executamos o algoritmo para gerar todas as *graphduas* para cada método que contém pelo menos uma ADU. Muitos métodos não possuem nenhuma ADU

Tabela 4 – Custo de aplicação do algoritmo para encontrar *graphduas*

Programa	**#ADUs**	**Irr.** (%)	**TGra.** (ms)	**TIrr.** (ms)	**Prop.** (%)
Csv	929	31.5	424.5	670.0	63.4
Cli	1291	29.4	533.8	923.2	57.8
JSoup	1866	26.8	599.1	877.3	68.3
Gson	3281	29.7	826.7	1295.8	63.8
J-Xml	3402	26.5	776.0	1237.9	62.7
Mockito	4236	32.1	1121.7	1931.1	58.1
Codec	4446	37.9	2080.9	6778.4	30.7
Compress	6286	28.3	2393.5	5797.4	41.3
Collections	16937	30.0	2456.9	4556.9	53.9
J-Core	17653	28.1	2751.3	6394.3	43.0
Time	18160	31.1	3081.6	5915.2	52.1
JxPath	20178	29.7	3386.5	6494.9	52.1
Lang	22290	32.0	3079.2	6087.2	50.6
J-Databind	31797	26.4	3946.2	6982.3	56.5
Closure	78068	31.8	11913.6	32076.6	37.1
Chart	81847	24.5	10004.4	18420.5	54.3
Math	87603	25.3	27010.4	104246.0	25.9
Weka	337063	27.4	38200.9	98377.5	38.8

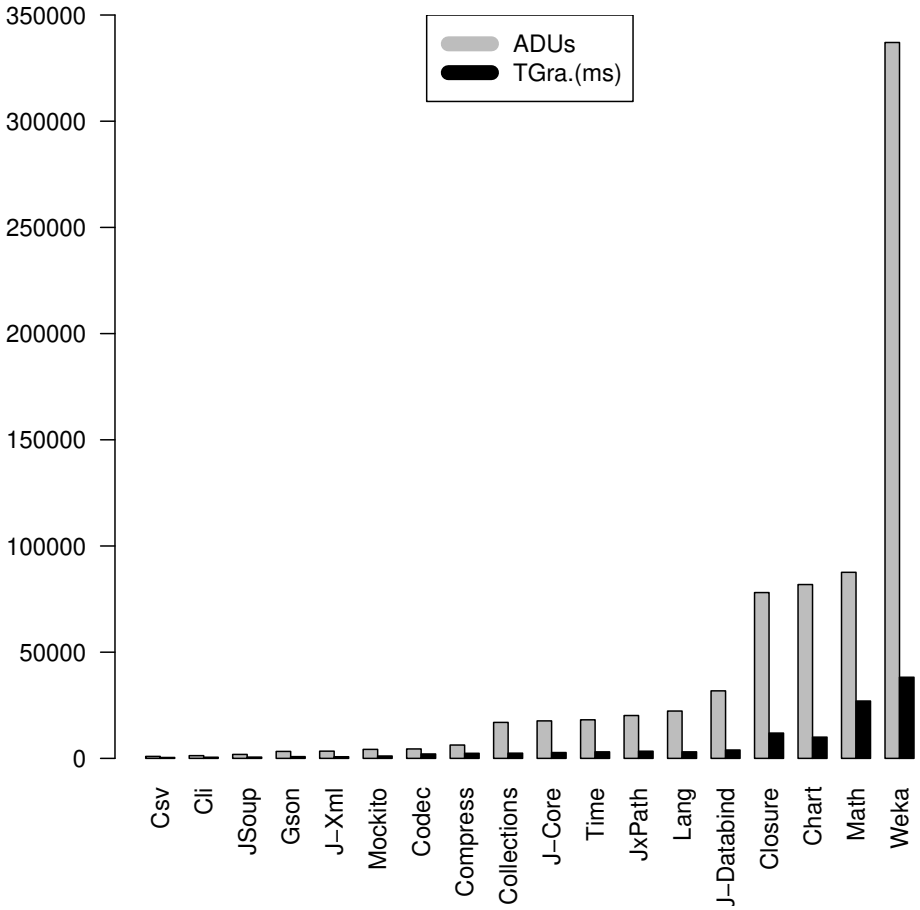
Fonte – Concilio *et al.* (2021)

pois contêm somente definições e usos locais; os dados de tais métodos não foram incluídos na Tabela 5.3.1. Para cada método, foi executado o Algoritmo de Subsunção (AS) em suas *graphduas* para encontrar a relação de subsunção ADU-ADU e, em seguida, as ADUs irrestritas. Todos os dados foram coletados utilizando um MacAir, 1.8 GHz Intel Core i5 Dual-Core, 8 GB 1600 MHz DDR3.

As linhas da Tabela são ordenadas pela quantidade de ADUs. O número de ADUs irrestritas variou de 24.5% para o *Chart* até 37.9% para o *Codec*, com uma média de 29.4%. O tempo para calcular as *graphduas* variou aproximadamente de acordo com o número de ADUs do programa, sendo necessário 38s para o *Weka* e 27s para o *Math*, os programas mais custosos. Todos os outros programas precisaram de menos de 12s para encontrar todas as *graphduas*. A Figura 18 representa o número de ADUs e o tempo necessário para gerar as *graphduas* para cada programa. O custo de geração da *graphdua* na análise de subsunção variou de 25,9% para o *Math* a 68,3% para o *Jsoup*, sendo a média 50,5%.

Além de provar a correção das *graphduas* para representar os caminhos que cobrem uma ADU (ver Seção 4.2.6), utilizamos uma abordagem indireta, semelhante ao teste metamórfico (AMMANN; OFFUTT, 2008), para verificá-la utilizando dados obtidos dos

Figura 18 – Número de ADUs e o tempo para geração das graphduas



Fonte – Concilio *et al.* (2021)

programas objeto da avaliação. Duas propriedades da relação de subsunção ADU-ADU foram verificadas. A verificação dessas propriedades confirmam a propriedade das *graphduas* de representar corretamente cobertura de fluxo de dados.

A relação de subsunção é reflexiva, isto é, uma ADU D subsume a si mesmo: $D \rightarrow D$. A propriedade reflexiva foi verificada em 737.333 ADUs de 28.758 métodos de todos os 18 programas e falhou em 71 ADUs de 24 métodos. As falhas deveram-se a dois problemas que levam a grafos de fluxo de controle mal formados que impossibilitam a aplicação de AS: (1) o nó inicial possui arcos de entrada ou (2) o grafo de fluxo de controle possui auto-laços (n, n).

A cobertura dos testes automatizados que acompanham 16 programas (todos os 18 menos *Lang* e *Mockito*) e as ADUs irrestritas calculadas foram utilizadas para avaliar a relação de subsunção entre ADUs e, indiretamente, a correção da estrutura *graphdua*. A

relação de subsunção implica que as ADUs subsumidas devem ser cobertas quando a ADU subsumidora o é. A propriedade de subsunção das ADUs irrestritas para cada método executado nos testes foi verificada. Ela falhou para 18 (de um total de 14.532) métodos executados. A propriedade de subsunção foi refutada em 17 métodos devido à ocorrência de uma exceção dentro de uma cláusula `try` e para um método devido a uma cláusula `synchronized`, que bloqueou a cobertura das ADUs subsumidas.

5.3.2 Discussão

QP1 avalia o custo (tempo) para gerar as *graphduas* de um programa. Com exceção de *Lang* e *Chart* (ambos em negrito na Tabela 5.3.1), que foram ligeiramente menos custosos apesar de possuírem mais ADUs do que os programas imediatamente anteriores, o tempo (custo) para encontrar as *graphduas* é maior quanto maior o número de ADUs do programa. Essa tendência é mostrada na Figura 18.

Exceções menos evidentes dessa tendência são os programas *Codec* e *Math*. *Weka* tem quase quatro vezes mais ADUs, mas é apenas 1,4 vezes mais custosa do que *Math*. O custo assintótico para encontrar uma *graphdua* é O(|E|) ou O(|N|), assumindo que o número de arcos (E) é linear ao número de nós (N). Assim, programas com métodos mais complexos tendem a serem mais custosos para calcular as *graphduas*. Dois métodos em *Math* possuem 2.197 ADUs e grafos de fluxo de controle com 327 nós, que parecem ser clones um do outro. Eles dominam o custo total para gerar as *graphduas* de *Math*. Situação semelhante ocorre com os programas *Codec* e *Mockito*. *Codec* possui apenas 210 ADUs a mais que *Mockito* mas possui métodos mais complexos, o que faz com que sejam necessários 5,8 s mais para calcular as *graphduas*.

No geral, a maioria das *graphduas* foram calculadas rapidamente, com poucos métodos extremamente complexos sendo exceções. Como resultado, centenas de milhares de *graphduas* são calculadas em dezenas de segundos. Embora representem uma quantidade significativa do custo da análise de subsunção, em média 50% e custando até 68%, a análise de subsunção de ADUs é escalável. Para o programa mais caro para cálculo da subsunção ADU-ADU, *Math*, foi necessário aproximadamente 1min e 40s para encontrar as ADUs irrestritas para todos os métodos. Isso ocorre porque o Algoritmo de Subsunção (AS) é mais sensível à complexidade dos programas (CHAIM *et al.*, 2021).

QP2 visa avaliar a utilidade das *graphduas*. A coluna **Irr.** apresenta o percentual de ADUs irrestritas c.r.a. total de ADUs. Em média, representam 30%, o que significa uma economia de 70% no esforço de teste pois o testador precisaria verificar apenas 30% das ADUs para atingir a cobertura de fluxo de dados. Porém, advertimos o leitor que as relações de subsunção entre ADUs podem ser invalidadas por ADUs inalcançáveis ou devido à interrupção da execução do programa.

QP3 avaliou de forma indireta a correção da estrutura *graphdua* para representar a cobertura de fluxo de dados. A propriedade reflexiva foi verificada usando todos 18 programas objeto e a propriedade de subsunção usando a cobertura de 16 programas. As falhas observadas foram reduzidas, o que permitiu que elas fossem analisadas caso a caso. Em nenhuma delas, a falha nas propriedades deveu-se a uma incorreção na representação da cobertura de fluxo de dados fornecida pelas *graphduas*.

Diferentemente de estudos anteriores (MARRÉ; BERTOLINO, 2003; JIANG *et al.*, 2018), na avaliação experimental apresentada, as ADUs irrestritas foram calculadas levando em consideração todos os caminhos cobrindo uma ADU em particular. Além disso, ela utilizou programas similares aos desenvolvidos na indústria. Assim, os resultados indicam que as *graphduas* podem ser utilizadas em ambientes industriais e são úteis para reduzir o custo dos testes de fluxo de dados.

5.3.3 Ameaças à validade

As principais ameaças à validade da avaliação experimental são originadas de riscos internos e externos. O repositório *Defects4J* foi utilizado para reduzir a ameaça externa, uma vez que seus programas são de código aberto e comparáveis aos desenvolvidos na indústria. Além disso, também foi adicionado o programa *Weka* porque se observou que softwares matemáticos possuem métodos complexos e, por isso, exigem mais do algoritmo que calcula as *graphduas*.

A avaliação experimental também possui ameaças à validade interna. Os algoritmos para encontrar as *graphduas*, para ler os programas compilados em *bytecodes* e obter as informações de controle e fluxo de dados e o Algoritmo de Subsunção (AS) foram implementados por nós. Para reduzir essa ameaça à validade, a subsunção ADU-ADU e as ADUs irrestritas foram verificadas automaticamente usando dados estáticos de 18 programas do estudo e de cobertura para 16 desses programas. Os resultados indicam que

as *graphduas* são calculadas corretamente; no entanto, a implementação realizada utiliza a linguagem *Java* e, em especial, o arcabouço *Java Collections*; ineficiências ocultas da linguagem Java e do arcabouço *Collections* podem afetar os dados de tempo obtidos.

5.4 Considerações finais

O capítulo apresentou a aplicação da estrutura *graphdua* para o cálculo da relação de subsunção entre ADUs. Além disso, um experimento foi realizado para avaliar a escalabilidade, correção e utilidade das *graphduas*.

Os resultados indicam que as *graphduas* são calculadas em dezenas de segundos para programas que possuem centenas de milhares de ADUs. Adicionalmente, o conjunto de ADUs irrestritas, calculadas usando *graphduas*, pode reduzir potencialmente em 70% o número de ADUs a serem verificadas por um testador que utiliza o teste de fluxo de dados. Finalmente, os dados experimentais confirmam que as *graphduas* representam corretamente a cobertura de fluxo de dados.

6 Conclusões

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

6.1 Resumo

Esse trabalho apresentou uma nova representação em grafo, chamada *graphdua*, que descreve todos os caminhos válidos que cobrem uma ADU (RAPPS; WEYUKER, 1985). Também mostramos que o custo de criação da nova representação é factível para programas utilizados na indústria e sua aplicação para descoberta da relação de subsunção.

A estrutra *graphdua* foi concebida como uma extensão da proposta de representação da cobertura de fluxo de dados $G*$ (MARRÉ, 1997), incluindo os caminhos de retorno não encontrados (Figura 6) que podem bloquear a relação de subsunção entre ADUs. A estrutura *graphdua* consiste em encontrar e conectar os subgrafos *SG1*, *SG2*, *SG3*, *SG4* e *SG5*; determinados utilizando os algoritmos **encontrarSubGrafo** e **encontrarLivreDefSubGrafo**, definidos por Bertolino e Marré (1994), e corrigidos utilizando o algoritmo proposto **limparCaminhosPendentes** (algoritmo 1).

Um experimento foi conduzido para avaliar se as *graphduas* podem ser calculadas em escala e se a sua aplicação para encontrar a subsunção de fluxo de dados leva à redução dos custos do teste. Foram utilizados 17 programas do repositório *Defects4J* (JUST; JALALI; ERNST, 2014), mais o programa de aprendizado de máquina *Weka*. As finalidades dos programas, assim como o tamanho, variam: desde pequenos programas, como *Csv* com 929 ADUs, a programas maiores, como *Weka* com 337.063 ADUs.

O tempo para calcular as *graphduas* variou aproximadamente de acordo com o número de ADUs do programa, sendo necessário 38s e 27s para os programas mais custosos. Todos os demais precisaram de menos de 12s para encontrar todas as *graphduas* (Figura 18). O custo de geração da *graphdua* na análise de subsunção variou de 25,9% a 68,3%, sendo a média 50,5%. E o número de ADUs irrestritas variou de 24.5% até 37.9%, com uma média de 29.4%.

No geral, a maioria das *graphduas* foram calculadas rapidamente, com poucos métodos extremamente complexos sendo exceções. Mesmo representando uma quantidade significativa do custo da análise de subsunção, em média 50% e custando até 68%, a

análise de subsunção de ADUs é escalável. Além disso, o percentual de ADUs irrestritas encontradas representam, em média, 30% das ADUs de um programa, significando uma economia de 70% no esforço de teste.

6.2 Contribuições

Esta pesquisa levou a várias contribuições no contexto de testes de fluxo de dados que são descritos abaixo:

6.2.1 Análise da relação de subsunção entre requisitos de teste

A principal contribuição deste trabalho é uma representação correta para a cobertura de fluxo de dados, solucionando problemas encontrados em trabalhos anteriores e que pode ser calculada de forma escalável para programas reais. Além disso, a nova estrutura de dados permite calcular corretamente a relação de subsunção entre requisitos de teste de fluxo de dados, reduzindo o custo desse tipo de teste.

Porém, a aplicabilidade da estrutura *graphdua* não é restrita apenas ao cálculo da relação de subsunção entre ADUs. A seguir, descremos outras possíveis aplicações que se tornam possíveis de serem implementadas corretamente usando as *graphduas*.

6.2.2 Geração de dados de entrada para testes

Como a estrutura *graphdua* é semelhante a um grafo de fluxo, pode-se adaptar técnicas de geração de dados de entrada baseadas em fluxo de controle para o contexto de fluxo de dados (BERTOLINO; MARRÉ, 1994; FRASER; ARCURI, 2011; BEYER *et al.*, 2004). Por exemplo, a heurística *com menos predicados* — que seleciona caminhos do grafo de fluxo que possuem menos predicados para reduzir a probabilidade de escolher caminhos inviáveis ou não executávies (YATES; MALEVRIS, 1989) — pode ser aplicada a uma *graphdua* de uma ADU a fim de escolher caminhos para percorrê-la para a geração de dados de entrada de teste.

Além disso, a estrutura *graphdua* pode ser associada a técnicas de geração de dados de teste baseadas em meta-heurísticas para cobertura de ADUs. Pode-se calcular o *nível*

de abordagem e a *distância do ramo* (MCMINN, 2004) c.r.a. último nó (o nó de uso) do *SG*3. Tais heurísticas determinam a aptidão dos indivíduos para a cobertura do nó na ferramenta *EvoSuite* (FRASER; ARCURI, 2011).

Estratégias que usam verificação de modelo e execução simbólica para geração de dados de entrada também podem se beneficiar das *graphduas*. Beyer *et al.* (2004) utilizam verificação de modelos e execução simbólica para gerar dados de teste para cobrir locais de programa. A verificação de modelos é utilizada para verificar se um determinado local do programa é alcançável (ou viável). Em seguida, a execução simbólica é utilizada para determinar um vetor de entrada que percorre o caminho. Selecionando locais na *graphdua*, i.e., o nó de uso, pode-se determinar os dados de teste para cobrir a ADU alvo. Nesse sentido, a *graphdua* pode apoiar essa abordagem para o teste de fluxo de dados.

6.2.3 Análise de viabilidade

Como mencionado acima, a verificação de modelos pode determinar se um local do programa é alcançável (BEYER *et al.*, 2004). Por outro lado, caso seja possível determinar que não existe uma entrada que alcance o local, significa que ele é inacessível ou inalcançável. Se o último nó de *SG3* (o nó de uso) de uma *graphdua* consiste em um nó inalcançável, então não há dados de entrada que o percorram. Em outras palavras, a ADU é inviável.

6.3 Trabalhos futuros

O desenvolvimento da estrutura *graphdua* permite explorar trabalhos futuros envolvendo subsunção de requisitos de fluxo de dados, geração de dados de entrada e análise de viabilidade. Em particular, as seguintes linhas de pesquisa podem ser investigadas.

1. Avaliações experimentais do uso da relação de subsunção para a criação de dados de teste.
2. Uso da relação de subsunção de fluxo de dados para melhor ajustar a localização de defeitos eliminando informações redundantes.
3. Uso das *graphduas* para o desenvolvimento de técnicas para geração de dados de entrada.

4. Explorar a execução simbólica dos caminhos gerados pelas *graphduas* para análise de viabilidade de requisitos de fluxo de dados.

Referências

AMMANN, P.; OFFUTT, J. *Introduction to Software Testing*. 1. ed. New York, NY, USA: Cambridge University Press, 2008. ISBN 0521880386, 9780521880381. Citado 2 vezes nas páginas 15 e 58.

ARAUJO, R. P. A.; CHAIM, M. L. Data-flow testing in the large. In: *IEEE International Conference on Software Testing. Verification and Validation*. [S.l.]: IEEE, 2014. p. 81–90. Citado na página 17.

BERTOLINO, A.; MARRÉ, M. Automatic generation of path covers based on the control flow analysis of computer programs. *IEEE Transactions on Software Engineering*, v. 20, n. 12, p. 885–899, Dec 1994. ISSN 0098-5589. Citado 5 vezes nas páginas 43, 48, 63, 64 e 70.

BEYER, D.; CHLIPALA, A. J.; HENZINGER, T. A.; JHALA, R.; MAJUMDAR, R. Generating tests from counterexamples. In: *Proceedings of the 26th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2004. (ICSE '04), p. 326–335. ISBN 0-7695-2163-0. Citado 2 vezes nas páginas 64 e 65.

BUDD, T. A. Mutation analysis: ideas, examples, problems, and prospects. In: . [S.l.: s.n.], 1981. Citado na página 22.

CHAIM, M. L.; BARAL, K.; OFFUTT, J. A data flow analysis framework for data flow subsumption. *CoRR*, abs/2101.05962, 2021. Disponível em: ⟨https://arxiv.org/abs/2101.05962⟩. Citado 2 vezes nas páginas 52 e 54.

CHAIM, M. L.; BARAL, K.; OFFUTT, J.; CONCILIO, M.; ARAUJO, R. P. A. Efficiently finding data flow subsumptions. In: *14th IEEE International Conference on Software Testing, Validation and Verification, ICST 2021, Porto de Galinhas, Brazil*. [S.l.: s.n.], 2021. Citado 4 vezes nas páginas 52, 54, 57 e 60.

CONCILIO, M.; ARAUJO, R. P. A.; CHAIM, M. L.; OFFUTT, J. Graph representation for data flow coverage. In: *45th IEEE Annual Computers, Software, and Applications Conference , COMPSAC 2021*. [S.l.: s.n.], 2021. Citado 17 vezes nas páginas 24, 26, 27, 28, 38, 39, 40, 41, 44, 45, 47, 48, 49, 53, 55, 58 e 59.

DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. *Introdução ao teste de software*. [S.l.]: Elsevier, 2016. Citado 2 vezes nas páginas 15 e 22.

DEMILLO, R. A.; LIPTON, R. J.; SAYWARD, F. G. Hints on test data selection: Help for the practicing programmer. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 11, n. 4, p. 34–41, abr. 1978. ISSN 0018-9162. Disponível em: ⟨https://doi.org/10.1109/C-M.1978.218136⟩. Citado na página 22.

FAROOQ, F.; NADEEM, A. A fault based approach to test case prioritization. In: *2017 International Conference on Frontiers of Information Technology (FIT)*. [S.l.: s.n.], 2017. p. 52–57. Citado na página 22.

FRANKL, P. G.; IAKOUNENKO, O. Further empirical studies of test effectiveness. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 23, n. 6, p. 153–162, nov. 1998. ISSN 0163-5948. Disponível em: ⟨http://doi.acm.org/10.1145/291252.288298⟩. Citado na página 16.

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*. New York, NY, USA: ACM, 2011. (ESEC/FSE '11), p. 416–419. ISBN 978-1-4503-0443-6. Citado 2 vezes nas páginas 64 e 65.

HECHT, M. S. *Flow Analysis of Computer Programs*. New York, NY, USA: Elsevier Science Inc., 1977. ISBN 0444002162. Citado na página 35.

HEMMATI, H. How effective are code coverage criteria? In: *2015 IEEE International Conference on Software Quality, Reliability and Security*. [S.l.: s.n.], 2015. p. 151–156. Citado na página 16.

HUTCHINS, M.; FOSTER, H.; GORADIA, T.; OSTRAND, T. Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In: *Proceedings of 16th International Conference on Software Engineering*. [S.l.: s.n.], 1994. p. 191–200. ISSN 0270-5257. Citado na página 16.

JIANG, S.; CHEN, J.; ZHANG, Y.; QIAN, J.; WANG, R.; XUE, M. Evolutionary approach to generating test data for data flow test. *IET Software*, v. 12, n. 4, p. 318–323, 2018. ISSN 1751-8806. Citado 8 vezes nas páginas 15, 17, 18, 30, 34, 35, 36 e 61.

JUST, R.; JALALI, D.; ERNST, M. D. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, 2014. (ISSTA 2014), p. 437–440. ISBN 978-1-4503-2645-2. Disponível em: ⟨http://doi.acm.org/10.1145/2610384.2628055⟩. Citado 3 vezes nas páginas 19, 56 e 63.

MARRÉ, M. *Program Flow Analysis for Reducing and Estimating the Cost of Test Coverage Criteria*. Tese (Doutorado) — Dep. de Computacion, FCEyN – Universidad de Buenos Aires, Argentina, 1997. Citado 6 vezes nas páginas 18, 30, 34, 36, 54 e 63.

MARRÉ, M.; BERTOLINO, A. Unconstrained duas and their use in achieving all-uses coverage. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 21, n. 3, p. 147–157, maio 1996. ISSN 0163-5948. Disponível em: ⟨http://doi.acm.org/10.1145/226295.226312⟩. Citado 9 vezes nas páginas 23, 30, 31, 32, 36, 43, 49, 56 e 70.

MARRÉ, M.; BERTOLINO, A. Using spanning sets for coverage testing. *IEEE Transactions on Software Engineering*, v. 29, n. 11, p. 974–984, Nov 2003. ISSN 0098-5589. Citado 7 vezes nas páginas 17, 18, 27, 30, 33, 36 e 61.

MCMINN, P. Search-based software test data generation: A survey: Research articles. *Softw. Test. Verif. Reliab.*, John Wiley and Sons Ltd., Chichester, UK, v. 14, n. 2, p. 105–156, jun. 2004. ISSN 0960-0833. Citado na página 65.

MYERS, G. J. *The Art of Software Testing, Second Edition*. 2. ed. Hoboken, N.J: Wiley, 2004. Hardcover. ISBN 0471469122. Citado na página 21.

OSTRAND, T.; BALCER, M. J. The category-partition method for specifying and generating functional tests. *Commun. ACM*, v. 31, p. 676–686, 06 1988. Citado na página 21.

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, April 1985. ISSN 0098-5589. Citado 3 vezes nas páginas 16, 27 e 63.

SANTELICES, R.; HARROLD, M. J. Efficiently monitoring data-flow test coverage. In: *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2007. (ASE '07), p. 343–352. ISBN 978-1-59593-882-4. Disponível em: ⟨http://doi.acm.org/10.1145/1321631.1321682⟩. Citado na página 34.

SOMMERVILLE, I.; MELNIKOFF, S. S. S.; ARAKAKI, R.; BARBOSA, E. d. A.; HIRAMA, K. *Engenharia de software*. São Paulo: Pearson Prentice Hall, 2008. ISBN 9788588639287 8588639289. Disponível em: ⟨http://www.worldcat.org/search?qt=worldcat_org_all&q=9788588639287⟩. Citado na página 21.

Su, T.; Fu, Z.; Pu, G.; He, J.; Su, Z. Combining symbolic execution and model checking for data flow testing. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. [S.l.: s.n.], 2015. v. 1, p. 654–665. ISSN 1558-1225. Citado 5 vezes nas páginas 17, 18, 30, 35 e 36.

Vivanti, M.; Mis, A.; Gorla, A.; Fraser, G. Search-based data-flow test generation. In: *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. [S.l.: s.n.], 2013. p. 370–379. ISSN 2332-6549. Citado na página 17.

YATES, D.; MALEVRIS, N. Reducing the effects of infeasible paths in branch testing. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 14, n. 8, p. 48–54, nov. 1989. ISSN 0163-5948. Citado na página 64.

Apêndice A – Algoritmos para encontrar subgrafos

A seguir, os algoritmos propostos por Marré e Bertolino (BERTOLINO; MARRÉ, 1994; MARRÉ; BERTOLINO, 1996) para encontrar subgrafos, chamados **encontrarSubGrafo** e **encontrarLivreDefSubGrafo**, respectivamente. Ambos já incluem o algoritmo para limpar caminhos pendentes (algoritmo 1).

Algoritmo 3 Algoritmo encontrarSubGrafo

Input Grafo de fluxo $G(N, E, s, e)$; nós n_i; n_j
Output Subgrafo SG contendo todos os caminhos de n_i até n_j

function ENCONTRARSUBGRAFO(G, n_i, n_j)
  $W \leftarrow n_i$
  $Arcos \leftarrow \emptyset$
  $Suc \leftarrow \emptyset$
  $Verificados \leftarrow \emptyset$
  while $W \neq \emptyset$ **do** ▷ Encontra os sucessores de n_i
    $n \leftarrow$ **removerNo**(W)
    foreach $n_{suc} \in Sucessores(n)$ **do**
      if $n_i = n_j$ **or** $n_{suc} \neq n_i$ **then**
        $Arcos \leftarrow Arcos \bigcup (n, n_{suc})$
        if $n_{suc} \notin Suc$ **then**
          $Suc \leftarrow Suc \bigcup n_{suc}$
      else
        $Verificados \leftarrow Verificados \bigcup n$
  $W \leftarrow n_j$ ▷ Encontra os predecessores de n_j
  $Pred \leftarrow \emptyset$
  while $W \neq \emptyset$ **do**
    $n \leftarrow$ **removerNo**(W)
    foreach $n_{pred} \in Predecessores(n)$ **do**
      if $n_i = n_j$ **or** $n_{pred} \neq n_j$ **then**
        if $n_{pred} \notin Pred$ **then**
          $Pred \leftarrow Pred \bigcup n_{pred}$
  $Nos \leftarrow Suc \bigcap Pred$ ▷ Constrói subgrafo
  $N_{SG} \leftarrow n_i$
  $E_{SG} \leftarrow \emptyset$
  $W \leftarrow n_i$
  while $W \neq \emptyset$ **do**
    $n \leftarrow$ **removerNo**(W)
    foreach $n_{suc} \in Sucessores(n)$ **do**
      if $(n, n_{suc}) \in Nos$ **and** $(n, n_{suc}) \in Arcos$ **then**
        $N_{SG} \leftarrow N_{SG} \bigcup n$
        $E_{SG} \leftarrow E_{SG} \bigcup (n, n_{suc})$
        $Arcos \leftarrow Arcos - (n, n_{suc})$
        $W \leftarrow W \bigcup n_{suc}$
  $s_{SG} \leftarrow n_i$ ▷ Define nós de entrada e saída
  $e_{SG} \leftarrow n_j$
  $SG \leftarrow$ **limparCaminhosPendentes**($SG(N_{SG}, E_{SG}, s_{SG}, e_{SG}), Verificados$)
  return SG

Fonte – Mario Concilio Neto (2021)

Algoritmo 4 Algoritmo encontrarLivreDefSubGrafo

Input Grafo de fluxo $G(N, E, s, e)$; nós n_i; n_j; variável X; $Def(n)$: conjunto de variáveis definidas em n

Output Subgrafo SG contendo todos os caminhos de n_i até n_j

```
1: function ENCONTRARLIVREDEFSUBGRAFO(G, n_i, n_j, X, Def(n))
2:     W ← n_i
3:     Arcos ← ∅
4:     Suc ← ∅
5:     Verificados ← ∅
6:     while W ≠ ∅ do                          ▷ Encontra os sucessores de n_i
7:         n ← removerNo(W)
8:         foreach n_suc ∈ Sucessores(n) do
9:             if n_i ≠ n_suc and n_j ≠ n_suc and X ∈ Def(n_suc) then
10:                Verificados ⋃ n
11:                continue
12:            if n_i = n_j or n_suc ≠ n_i then
13:                Arcos ← Arcos ⋃ (n, n_suc)
14:                if n_suc ∉ Suc then
15:                    Suc ← Suc ⋃ n_suc
16:            else
17:                Verificados ← Verificados ⋃ n
18:    W ← n_j                                 ▷ Encontra os predecessores de n_j
19:    Pred ← ∅
20:    while W ≠ ∅ do
21:        n ← removerNo(W)
22:        foreach n_pred ∈ Predecessores(n) do
23:            if n_i ≠ n_pred and n_j ≠ n_pred and X ∈ Def(n_pred) then
24:                continue
25:            if n_i = n_j or n_pred ≠ n_j then
26:                if n_pred ∉ Pred then
27:                    Pred ← Pred ⋃ n_pred
28:    Nos ← Suc ⋂ Pred                        ▷ Constrói subgrafo
29:    N_SG ← n_i
30:    E_SG ← ∅
31:    W ← n_i
32:    while W ≠ ∅ do
33:        n ← removerNo(W)
34:        foreach n_suc ∈ Sucessores(n) do
35:            if (n, n_suc) ∈ Nos and (n, n_suc) ∈ Arcos then
36:                N_SG ← N_SG ⋃ n
37:                E_SG ← E_SG ⋃ (n, n_suc)
38:                Arcos ← Arcos − (n, n_suc)
39:                W ← W ⋃ n_suc
40:    s_SG ← n_i                              ▷ Define nós de entrada e saída
41:    e_SG ← n_j
42:    SG ← limparCaminhosPendentes(SG(N_SG, E_SG, s_SG, e_SG), Verificados)
43:    return SG
```

Fonte – Mario Concilio Neto (2021)