UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

JAYNE GUERRA CECONELLO

Ferramenta de auxílio à análise de anomalias em códigos Orientados a Objeto

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Érika Fernandes Cota

Porto Alegre
2016

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitor de Graduação: Prof. Sérgio Roberto Kieling Franco
Diretor do Instituto de Informática: Prof. Luís da Cunha Lamb
Coordenador do Curso de Ciência da Computação: Prof. Raul Fernando Weber
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

AGRADECIMENTOS

Gostaria de agradecer aos meus pais, por me apoiarem constantemente e fazer com que minha graduação fosse possível. Também ao meu irmão, por ser meu melhor amigo em todos os momentos em que precisei de alguém para discutir os problemas (até mesmo os problemas de código).

Gostaria de agradecer ao meu namorado por sua compreensão e apoio durante os momentos em que os estudos tiveram que ser priorizados.

Gostaria de agradecer aos meus colegas pela companhia durante todo o curso, desejo muito sucesso a todos vocês.

Gostaria de agradecer aos meus amigos por compreenderem meus momentos de ausência, que não foram poucos.

Gostaria de agradecer a minha orientadora, Érika Fernandes Cota, por sua dedicação, atenção e apoio quanto aos desafios enfrentados ao longo do desenvolvimento deste trabalho.

Gostaria de agradecer a todos os professores que eu tive na UFRGS por todo conhecimento adquirido, proveniente da qualidade do ensino proporcionada através da dedicação investida.

RESUMO

Com o crescimento significativo da adoção de softwares para a realização das mais comuns tarefas diárias, é cada vez mais evidente a importância do investimento em técnicas e ferramentas que influem na qualidade dos mesmos, a fim de evitar riscos. Muitas vezes o impacto de uma falha pode causar grandes prejuízos, principalmente quando se trata de um sistema crítico.

No intuito de influir na qualidade do software, são criados testes que exploram estruturas e funcionalidades do código. No contexto empresarial, códigos são dinâmicos e são integrados continuamente, sendo inviável a aplicação de testes estruturais contínuos sem a automatização de alguns passos da criação dos mesmos.

A proposta deste trabalho consiste no desenvolvimento de uma ferramenta que implementa critérios de análise estática de código baseando-se nos relacionamentos interclasse existentes, visando à exploração de características comuns à programação orientada a objetos, com o intuito de aprimorar a qualidade dos casos de teste gerados.

Palavras-chave: Teste de software. Automatização de teste. Teste de software orientado a objetos.

Tool to aid anomaly analysis for Object Oriented codes

ABSTRACT

With the significant increase of the adoption of software to perform the most common daily activities, it is increasingly evident the importance of investing on techniques and tools that influence their quality, in order to avoid risks. In many situations, the impact of a failure can cause large losses, especially considering a critical system.

In order to improve the software quality, tests are created to explore the code structure and features. In the business context, codes are dynamic and continuously integrated, making it impossible the continuous application of structural tests without having some steps of the test creation automated.

The goal of this work consists in the development of a tool that implements static code analysis criteria based on the existing interclass relationships, aiming to the exploration of common characteristics of object oriented programming, in order to improve the quality of the generated test cases.

Keywords: Software testing. Testing automation. Object-Oriented software testing.

.

LISTA DE FIGURAS

Figura 1.1 – Fluxograma de desenvolvimento contínuo 13
Figura 2.1 – Exemplo de ITU 21
Figura 2.2 – Exemplo de SDA 22
Figura 2.3 – Exemplo de SDA (2) 22
Figura 2.4 – Exemplo de SDIH 23
Figura 2.5 – Exemplo de IISD 25
Figura 2.6 – Exemplo de ACB1 26
Figura 2.7 – Exemplo de ACB2 27
Figura 2.8 – Exemplo de IC 28
Figura 2.9 – Exemplo de SVA 29
Figura 2.10 – Exemplo de grafo *Yo-Yo*: hierarquia de classes 31
Figura 2.11 – Exemplo de grafo *Yo-Yo*: diagrama 32
Figura 2.12 – Linguagens suportadas 34
Figura 2.13 – Critérios de medida de cobertura 35
Figura 2.14 – Linguagens suportadas (2) 35
Figura 2.15 – Critérios de medida de cobertura (2) 36
Figura 3.1 – Representação da proposta 37
Figura 3.2 – Classes do Protótipo 40
Figura 3.3 – Representação da estrutura básica do arquivo .ll 43
Figura 3.4 – Representação do fluxo de execução 44
Figura 3.5 – Representação do fluxo de execução – Primeira varredura 45
Figura 3.6 – Classes no arquivo .ll 46
Figura 3.7 – Expressão regular utilizada na identificação de classes 46
Figura 3.8 – Expressão regular utilizada na identificação de métodos 47
Figura 3.9 – Expressão regular aplicada na identificação de métodos 47
Figura 3.10 – Exemplo de nome de método 48
Figura 3.11 – Expressão regular utilizada na identificação de atributos 48
Figura 3.12 – Expressão regular aplicada na identificação de atributos 49
Figura 3.13 – Expressão regular aplicada na identificação de atributos (2) 49
Figura 3.14 – Expressão regular aplicada na identificação de atributos (3) 49
Figura 3.15 – Representação do fluxo de execução – Segunda varredura 50
Figura 3.16 – Representação do fluxo de execução – Segunda varredura – Parte 1 51
Figura 3.17 – Exemplo de herança 52
Figura 3.18 – Declaração de tabela virtual 53
Figura 3.19 – Exemplo de grafo Yo-Yo 53
Figura 3.20 – Exemplo de tabela virtual extraída 54
Figura 3.21 – Exemplo de chamada polimórfica 55
Figura 3.22 – Representação do fluxo de execução – Segunda varredura – Parte 2 56
Figura 3.23 – Exemplo de chamada polimórfica (2) 57
Figura 3.24 – Expressão regular utilizada na identificação do primeiro caso 57
Figura 3.25 – Expressões regulares utilizadas na identificação de chamadas polimórficas 58
Figura 3.26 – Expressão regular aplicada na identificação de chamadas polimórficas 58
Figura 3.27 – Expressão regular utilizada na identificação de objetos 58
Figura 3.28 – Expressão regular aplicada na identificação de objetos 58
Figura 3.29 – Expressões regulares utilizada na identificação de definições, usos e chamadas 60
Figura 3.30 – Expressão regular aplicada na identificação de definições, usos e chamadas 60
Figura 3.31 – Exemplo de bitcast 61
Figura 3.32 – Expressão regular utilizada na identificação de bitcasts 61
Figura 3.33 – Exemplo do arquivo de saída Classes.txt 62
Figura 3.34 – Exemplo do arquivo de saída Classes.txt (2) 63
Figura 3.35 – Exemplo do arquivo de saída Tabela de Polimorfismo.txt 63
Figura 3.36 – Exemplo do arquivo de saída Tabela de Polimorfismo.txt 64
Figura 3.37 – Exemplo do arquivo de saída YoYo.txt 65
Figura 3.38 – Exemplo do arquivo de saída YoYo.txt (2) 66

Figura 3.39 – Padrão de arquivo de saída 67
Figura 3.40 – Padrão de arquivo de saída sem resultados 67
Figura 3.41 – Exemplo do arquivo de saída ITU.txt 68
Figura 3.42 – Exemplo do arquivo de saída SDA.txt 69
Figura 3.43 – Exemplo do arquivo de saída SDIH.txt 70
Figura 3.44 – Exemplo do arquivo de saída SDI.txt 72
Figura 3.45 – Exemplo do arquivo de saída ACB1.txt 73
Figura 3.46 – Exemplo do arquivo de saída ACB2.txt 74
Figura 3.47 – Exemplo do arquivo de saída IC.txt 75
Figura 3.48 – Exemplo de fluxo 76
Figura 3.49 – Exemplo de declaração de classes 78
Figura 4.1 – Resultados: Classes.txt 80
Figura 4.2 – Resultados: Fluxo.txt 81
Figura 4.3 – Resultados: YoYo.txt 82
Figura 4.4 – Resultados: ITU.txt 82
Figura 4.5 – Resultados: IC.txt 84
Figura 4.6 – Instrumentos.cpp – Diagrama de Classes 85
Figura 4.7 – Resultados: Classes.txt (Instrumentos.cpp) 86
Figura 4.8 – Resultados: Tabela de Polimorfismo.txt (Instrumentos.cpp) 87
Figura 4.9 – Resultados: Fluxo.txt (Instrumentos.cpp) 87
Figura 4.10 – Resultados: YoYo.txt (Instrumentos.cpp) 88
Figura 4.11 – Resultados: SDA.txt (Instrumentos.cpp) 89
Figura 4.12 – Resultados: SDIH.txt (Instrumentos.cpp) 90
Figura 4.13 – Resultados: SDI.txt (Instrumentos.cpp) 91
Figura 4.14 – Resultados: ACB1.txt (Instrumentos.cpp) 92
Figura 4.15 – Resultados: ACB2.txt (Instrumentos.cpp) 93
Figura 4.16 – Resultados: IC.txt (Instrumentos.cpp) 94

LISTA DE TABELAS

Tabela 2.1 – Falhas e anomalias provenientes do uso de herança e polimorfismo 20
Tabela 2.2 – Exemplo de grafo *Yo-Yo*: chamadas interclasse 32

LISTA DE ABREVIATURAS E SIGLAS

ITU	*Inconsistent Type Use*
SDA	*State Definition Anomaly*
SDIH	*State Definition Inconsistency*
SDI	*State Defined Incorrectly*
IISD	*Indirect Inconsistent State Definition*
ACB1	*Anomalous Construction Behavior (1)*
ACB2	*Anomalous Construction Behavior (2)*
IC	*Incomplete Construction*
SVA	*State Visibility Anomaly*
LLVM IR	LLVM *intermediate representation*

SUMÁRIO

1 INTRODUÇÃO **12**
1.1 A Importância da Execução de Testes **12**
1.2 Testes Automatizados **12**
1.3 Especificidades da Orientação a Objetos **14**
1.4 Ferramentas **15**
1.5 Proposta **15**
1.6 Resultados **16**
1.7 Organização do texto **16**
2. APRESENTAÇÃO DE CONCEITOS **17**
2.1 Fases de Teste **17**
2.2 Orientação a Objetos **17**
2.2.1 Encapsulamento 17
2.2.2 Herança 18
2.2.3 Polimorfismo 18
2.2.4 Acoplamento Dinâmico 18
2.3 Definições e Usos **19**
2.4 Desafios da Orientação a Objetos **19**
2.4.1 ITU - *Inconsistent type use* 20
2.4.2 SDA - *State definition anomaly* 21
2.4.3 SDIH - *State definition inconsistency* 23
2.4.4 SDI - *State defined incorrectly* 24
2.4.5 IISD - *Indirect inconsistent state definition* 24
2.4.6 ACB1 - *Anomalous construction behavior* (1) 25
2.4.7 ACB2 - *Anomalous construction behavior* (2) 26
2.4.8 IC - *Incomplete construction* 27
2.4.9 SVA - *State visibility anomaly* 29
2.5 Grafo *Yo-Yo* **30**
2.6 LLVM **33**
2.7 Trabalhos Relacionados **34**
3 IDENTIFICAÇÃO DE POTENCIAIS ANOMALIAS EM CÓDIGOS ORIENTADOS A OBJETOS PARA AUXILIO NA GERAÇÃO DE TESTES **37**
3.1 Estruturas internas **38**
3.2 Leitura do arquivo .ll **41**
3.2.1 Primeira varredura 44
3.2.2 Segunda varredura 49
3.3 Arquivos gerados **61**
3.3.1 Arquivo Classes.txt 61
3.3.2 Arquivo Tabela de Polimorfismo.txt 63
3.3.3 Arquivo Fluxo.txt 63
3.3.4 Arquivo YoYo.txt 64
3.3.5 Arquivos específicos 66
3.3.5.1 Arquivo ITU.txt 68
3.3.5.2 Arquivo SDA.txt 69
3.3.5.3 Arquivo SDIH.txt 70
3.3.5.4 Arquivo SDI.txt 71
3.3.5.5 Arquivo ACB1.txt 72
3.3.5.6 Arquivo ACB2.txt 74
3.3.5.7 Arquivo IC.txt 75
3.4 Expansão do código **76**
3.4.1 Inclusão de análise de pares DU 76
3.4.2 Inclusão de todos os métodos presentes no arquivo de entrada 77

4 RESULTADOS ... **79**
4.1 Aplicação do protótipo em seu próprio código ... **79**
4.1.1 Saída para o arquivo Classes.txt ... 79
4.1.2 Saída para o arquivo Tabela de Polimorfismo.txt ... 80
4.1.3 Saída para o arquivo Fluxo.txt ... 80
4.1.4 Saída para o arquivo YoYo.txt ... 81
4.1.5 Saída para o arquivo ITU.txt ... 82
4.1.6 Saída para o arquivo SDIH.txt ... 83
4.1.7 Saída para o arquivo IC.txt ... 83
4.2 Aplicação do protótipo no código Instrumentos.cpp ... 84
4.2.1 Saída para o arquivo Classes.txt (Instrumentos.cpp) ... 86
4.2.2 Saída para o arquivo Tabela de Polimorfismo.txt (Instrumentos.cpp) ... 86
4.2.3 Saída para o arquivo Fluxo.txt (Instrumentos.cpp) ... 87
4.2.4 Saída para o arquivo YoYo.txt (Instrumentos.cpp) ... 88
4.2.5 Saída para o arquivo ITU.txt (Instrumentos.cpp) ... 88
4.2.6 Saída para o arquivo SDA.txt (Instrumentos.cpp) ... 88
4.2.7 Saída para o arquivo SDIH.txt (Instrumentos.cpp) ... 89
4.2.8 Saída para o arquivo SDI.txt (Instrumentos.cpp) ... 90
4.2.9 Saída para o arquivo ACB1.txt (Instrumentos.cpp) ... 91
4.2.10 Saída para o arquivo ACB2.txt (Instrumentos.cpp) ... 92
4.2.11 Saída para o arquivo IC.txt (Instrumentos.cpp) ... 93
4.3 Análise dos resultados ... 94
5 CONCLUSÃO ... **96**
REFERÊNCIAS ... **97**

1 INTRODUÇÃO

A etapa de testes é uma fase essencial e contínua no processo de desenvolvimento de software. Esta etapa visa a identificação os problemas contidos no código antes que este seja usado por clientes reais e o impacto destes problemas possa acarretar a perda de qualidade do produto desenvolvido.

As seções seguintes contextualizam as metodologias de testes no cotidiano, relacionando-as às boas práticas induzidas pela engenharia de software. Também são abordados os problemas encontrados na adoção destas boas práticas quanto às peculiaridades das linguagens de programação orientadas a objetos. Também é apresentada a solução proposta para os problemas citados e as ferramentas envolvidas neste processo, seguidas pela descrição do objetivo dos resultados obtidos através da execução do protótipo e uma breve visão geral da organização do texto.

1.1 A Importância da Execução de Testes

O desenvolvimento de um software é um processo que requer diversas etapas de elaboração. Para cada umas destas etapas existem metodologias que podem se aplicadas, a fim de agregar qualidade ao software produzido. A engenharia de software estabelece modelos que abrangem diversos setores de desenvolvimento, convencionando a criação do software e visando a redução da complexidade de manutenção e a qualidade do produto.

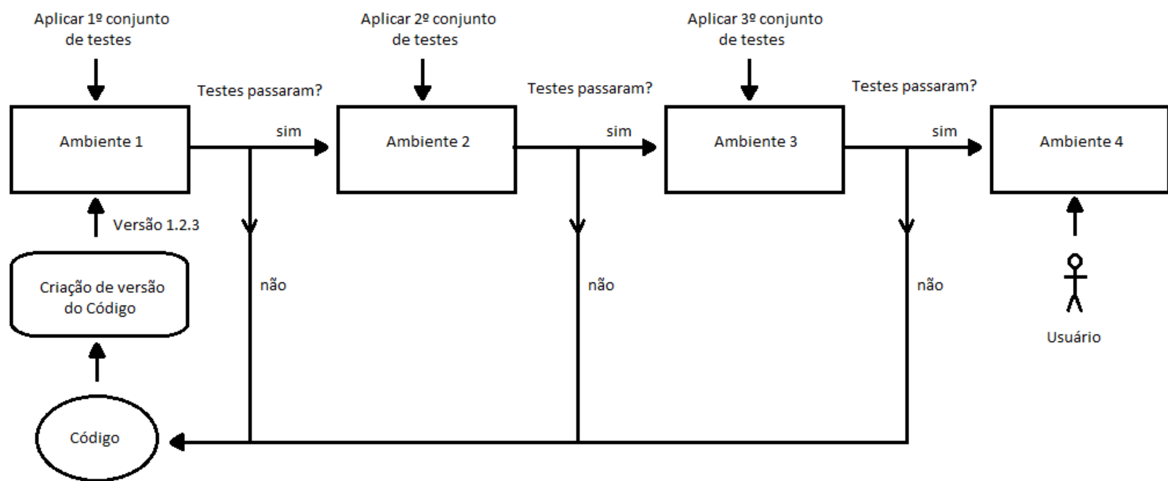
Muitos softwares comerciais são aplicações críticas, tais como os que realizam a gerência de informações bancárias e salariais de seus clientes. Tendo em vista a importância da minimização de possíveis falhas de desenvolvimento nestes softwares, a realização de diversas etapas de teste é indispensável. O teste de software é um dos grandes alicerces da engenharia de software. O teste de software busca revelar falhas do produto ao longo do ciclo de desenvolvimento, a fim de possibilitar que a correção da falha seja feita o mais cedo possível, minimizando seu impacto e seu custo.

1.2 Testes Automatizados

O teste de software é um processo que requer tempo e processamento, tendo como objetivo revelar o maior número possível de falhas em tempo passível de correção, para que a falha cause o menor impacto possível ao projeto. Atualmente, muitas empresas utilizam

sistemas de promoções contínuas entre ambientes de testes de acordo com o nível de teste o qual uma determinada versão do código se encontra.

Figura 1.1 – Fluxograma de desenvolvimento contínuo



Fonte: o autor

Como pode ser observado na Figura 1.1, quando uma versão do código é criada, este código é implantado em um ambiente de teste inicial (Ambiente 1) que tende a ser instável, ou seja, tente a receber novas versões do código de forma mais frequente. Neste primeiro ambiente geralmente são testadas funcionalidades básicas como login e exibição de determinadas interfaces, por exemplo. Se o código não apresenta falhas nos testes realizados, ele está apto a ser promovido para um ambiente mais estável (Ambiente 2), e assim sucessivamente até a versão do código alcançar o último nível de ambiente (Ambiente 4), onde o cliente final faz uso da aplicação. Esse processo é conhecido como entrega contínua (*continuous delivery*). Toda vez que uma falha é encontrada, a versão do código testada é marcada como inválida e uma nova versão do código deve ser criada contendo a correção da falha.

Em uma empresa que faz uso da metodologia de entrega contínua, a descoberta de uma falha nos ambientes de teste iniciais permite que os desenvolvedores possam efetuar a correção antes que essa falha atinja ambientes mais estáveis, economizando o tempo que seria perdido durante os testes nos demais ambientes e evitando que a falha possa chegar até o usuário, o que poderia causar prejuízo financeiro para a empresa em questão.

Neste cenário, a aplicação de testes manuais para a verificação do código em cada um dos ambientes é inviável, visto que cada etapa de teste demanda uma quantidade significativa de tempo e podem existir diversos ambientes de mesmo nível com diferentes códigos a serem testados. Como os conjuntos de testes tendem a ser instáveis, sofrendo atualização para a

inclusão de testes de novas funcionalidades esporadicamente, e devem ser executados frequentemente, ou seja, toda vez que uma nova versão do código é criada, faz-se necessária a execução de testes automatizados.

Tendo a importância da automatização de testes em evidência, é importante definir como esses conjuntos de testes serão criados. A automatização do processo de criação de testes é uma tarefa que aproxima a teoria da prática. Atualmente, do ponto de vista comercial, o tempo gasto com melhorias que não são notadas pelo cliente não é tão significativo quanto as melhorias visíveis, as quais possam atrair mais clientes. Todavia, a qualidade do software influi diretamente na imagem da empresa no mercado e, portanto a realização de testes não deve ser desvalorizada, porém aperfeiçoada. As teorias referentes à geração de conjuntos de testes baseados em estruturas de fluxo podem ser aplicadas de forma automatizada em aplicações reais, fazendo com que exista uma economia de tempo na etapa de criação de testes sem pecar na qualidade dos mesmos.

Com base no código criado pelo desenvolvedor, é possível gerar estruturas que expõem visualmente pontos onde possam existir defeitos. Estas estruturas permitem a exploração de relacionamentos entre os componentes do código. Estes relacionamentos podem ser explorados na etapa de criação do conjunto de testes, a fim de identificar relações anômalas entre componentes.

1.3 Especificidades da Orientação a Objetos

A Orientação a Objetos é um paradigma de projeto e desenvolvimento de software que proporciona a existência de diferentes relacionamentos entre os dados, através do uso de herança e polimorfismo, os quais são descritos no Capítulo 2. Estes relacionamentos mostram-se complexos quando analisados por ferramentas de teste que buscam gerar estruturas para a etapa de teste de integração dos componentes do código.

Por exemplo, estes relacionamentos possibilitam ao desenvolvedor que diferentes métodos de mesmo nome e diferentes classes possam ser acionados dependendo do tipo do objeto que desencadeou a chamada. Esta flexibilidade torna a etapa de teste de integração complexa, pois a identificação do método que foi, de fato, acionado não é trivial.

A partir das definições de falhas e anomalias provenientes do uso de herança e polimorfismo, citadas por Ammann e Offutt (2008) e descritas no Capítulo 2, este trabalho tem como objetivo a construção automatizada de estruturas que promovam a análise de

integração de componentes que exploram as características relacionadas a orientação a objetos.

1.4 Ferramentas

Atualmente a quantidade de ferramentas disponíveis capazes de gerar estruturas de relacionamento interclasse não satisfaz a demanda, sendo, geralmente, específicas para determinadas linguagens, como exemplificado na Seção 2.7.

Na abordagem proposta, uma representação intermediária do código que pode ser compartilhada por outras linguagens de programação é usada na geração de estruturas de relacionamento entre classes e métodos. A proposta visa abranger demais linguagens sem limitar-se a um formato específico do código fonte. Para isso, a representação intermediária previamente mencionada foi obtida através do uso do compilador LLVM, Clang, que suporta as linguagens C, C++, Objective-C e Objective-C++ e utiliza LLVM como *back-end* (CLANG: A C LANGUAGE FAMILY FRONTEND FOR LLVM, 2015). Através da análise do arquivo LLVM obtido, foram geradas as estruturas de relacionamentos interclasse.

LLVM é um projeto compreendido por uma coleção de compiladores modulares e reusáveis e um conjunto de ferramentas tecnológicas. Mais detalhes sobre o LLVM são apresentados no Capítulo 2.

1.5 Proposta

Como mencionado nos tópicos anteriores, a complexidade do relacionamento entre métodos e objetos na programação orientada a objetos justifica a necessidade da existência de uma ferramenta que esclareça e explore estas correlações a fim de facilitar a exploração das mesmas na etapa de criação de testes. É neste contexto que a ferramenta proposta se faz presente.

A ferramenta desenvolvida neste trabalho gera estruturas de relacionamentos entre métodos e classes a partir da representação intermediária do código em LLVM (arquivo .ll). A partir do arquivo .ll é possível gerar estruturas de relacionamentos entre métodos e a partir das mesmas identificar possíveis falhas e anomalias provenientes do uso de herança e polimorfismo citadas por Ammann e Offutt (2008). O protótipo desenvolvido recebe o arquivo .ll como entrada e gera arquivos que tem por objetivo indicar ao usuário em quais pontos de relacionamento interclasse existe o risco de conter as anomalias e falhas descritas.

A ferramenta foi desenvolvida em C++ e seu código fonte está disponível no repositório GIT https://bitbucket.org/Jayne_GC/prototipo.git.

1.6 Resultados

Os resultados obtidos através da execução do protótipo buscam aumentar a qualidade dos testes desenvolvidos, explorando características únicas do uso de orientação a objetos. O resultado da execução do protótipo deve servir como um guia de quais pontos do código devem ser explorados com maior ênfase na etapa de criação de novos testes, a fim de evitar as falhas e anomalias descritas por Ammann e Offutt (2008).

Os resultados apresentados no Capítulo 4 apresentam os arquivos gerados para dois códigos distintos. O primeiro código utilizado foi o código da própria ferramenta. Nesta execução foi possível observar a existência de algumas anomalias relacionadas a herança. O outro código utilizado pertence a uma ferramenta de classificação de instrumentos musicais. Neste caso foram identificadas estruturas anômalas relacionadas a utilização de polimorfismo e herança.

1.7 Organização do texto

Esta monografia está organizada em cinco Capítulos. O Capítulo 2 resume os principais conceitos teóricos utilizados na construção do protótipo. O Capítulo 3 apresenta a proposta e os detalhes relacionados ao desenvolvimento do protótipo. O Capítulo 4 possui a análise dos resultados obtidos através da execução do protótipo desenvolvido. As conclusões e os trabalhos futuros são discutidos no Capítulo 5.

2 APRESENTAÇÃO DE CONCEITOS

2.1 Fases de Teste

O desenvolvimento de software é composto por diversas etapas, tais como especificação de requisitos, definição de artefatos e criação do código fonte. Cada uma destas etapas de desenvolvimento requer diferentes abordagens de teste. O foco proposto consiste no teste de integração, que abrange a comunicação entre os componentes do software no intuito de explorar a modelagem das dependências contidas no software testado. Os testes de integração para sistemas orientados a objetos visam a identificação de características como: conflitos de funcionalidade entre classes e sobreposição, sequência incorreta de unidades e associação polimórfica incorreta.

2.2 Orientação a Objetos

A orientação a objetos surgiu no intuito de prover novos recursos ao desenvolvedor através de novos mecanismos para manipulação e armazenamento de dados. Os dados são agrupados por meio de atributos em estruturas denominadas classes. Estes dados são manipulados por procedimentos (métodos) pertencentes às classes. Esta estruturação permite a proteção dos dados, fazendo com que o acesso aos mesmos seja restrito através do encapsulamento definido. Tendo em vista a modelagem conceitual de problemas a serem implementados, em alguns casos é possível notar que a flexibilidade de modelagem proporcionada pela utilização dos conceitos de classes e objetos é consideravelmente mais simples do que a modelagem baseada em termos de dados e funções. A seguir são descritas algumas das principais características provenientes da orientação a objetos.

2.2.1 Encapsulamento

Encapsulamento é o nome dado à característica que permite ao objeto a proteção de seus dados, impedindo que os demais objetos possam ter acesso aos mesmos. O encapsulamento gerencia a restrição dos relacionamentos entre classes semanticamente diferentes, podendo impedir o compartilhamento de métodos e atributos. O encapsulamento é uma importante ferramenta na busca por qualidade, pois permite isolar componentes a fim de

evitar interdependências entre classes, exceto por meio de suas interfaces (DELAMARO; MALDONADO; JINO, 2007).

2.2.2 Herança

A orientação a objetos permite a definição de classes em função de classes já existentes, este relacionamento é obtido por meio de herança (DELAMARO; MALDONADO; JINO, 2007). Através do uso de herança, as classes são arranjadas em hierarquias de especialização, onde as classes mais especializadas (classes filho ou subclasses) herdam todos os métodos e atributos das classes mais genéricas (classes pai ou superclasses). Assim, as subclasses podem estender as características herdadas das superclasses, promovendo o reuso de componentes já desenvolvidos. O reuso é uma prática que promove facilidade na manutenção e evolução do software, entre outros pontos como a redução do tempo gasto durante o desenvolvimento e a legibilidade do código desenvolvido.

2.2.3 Polimorfismo

O polimorfismo permite que uma mesma construção de linguagem possa assumir diferentes comportamentos, referindo-se a diferentes tipos de objetos ou atributos (DELAMARO; MALDONADO; JINO, 2007). Isso ocorre, por exemplo, quando duas ou mais subclasses distintas (classes filhas de uma mesma superclasse) possuem métodos de mesmo nome, porém cada método possui implementação distinta, e a superclasse (classe pai) possui a definição abstrata dos métodos.

2.2.4 Acoplamento Dinâmico

No acoplamento dinâmico uma única chamada de método pode ser dinamicamente ligada a diferentes métodos, sendo assim, o método que será executado em um determinado trecho do código depende do tipo do objeto que acarretou a chamada no momento da execução (PEZZÈ; YOUNG, 2008). O acoplamento dinâmico ocorre em tempo de execução e altera o fluxo de execução do software e, consequentemente, seu comportamento.

2.3 Definições e Usos

Os conceitos de definição e uso são bastante utilizados na análise de fluxo de dados. O ponto do programa onde um valor é produzido é chamado de ponto de definição. O ponto do programa onde um valor é acessado é chamado de ponto de uso.

As definições ocorrem, em geral, em todos os comandos ou expressões que alteram o valor de uma variável. Quando, em uma linha de código, uma variável é declarada, inicializada ou recebe um valor por parâmetro, podemos dizer que esta linha possui uma definição para esta variável. Os usos ocorrem em todos os comandos que extraem o valor de uma variável, carregando-o na memória a fim de ser utilizado em um contexto específico. Os usos podem ocorrer nos seguintes cenários: expressões, comandos condicionais, passagem de parâmetros, comandos *return*, entre outros (PEZZÈ; YOUNG, 2008).

No contexto da orientação a objetos, o comportamento descrito na explicação dos conceitos de definição e uso de variáveis pode ser aplicado a atributos. Os atributos também podem ser chamados de variáveis de estado do objeto. Quando uma definição ou uso ocorre em relação a um atributo, podemos dizer que seu objeto também sofreu esta mesma ação, visto que, o atributo faz parte do objeto.

2.4 Desafios da Orientação a Objetos

As linguagens orientadas a objetos proporcionam novos graus de abstração através de novos modos de integração dos componentes do código. Devido à complexidade das relações entre estes componentes, é necessário que exista uma ênfase maior nos testes de integração do que nos testes de unidade (AMMANN; OFFUTT, 2008).

O uso de herança, polimorfismo e acoplamento dinâmico provê ao desenvolvedor novos rumos de codificação, proporcionando, em alguns casos, estruturas que se adaptem melhor ao código desenvolvido. Porém, a utilização destes recursos pode acarretar em problemas de difícil detecção e correção. No quadro abaixo, encontram-se as falhas e anomalias provenientes do uso de herança e polimorfismo, definidas por Ammann e Offutt (2008).

Existem nove falhas ou anomalias provenientes do uso de herança e polimorfismo, apresentadas na Tabela 2.1. Estas falhas e anomalias foram utilizadas como referência para a

criação do protótipo proposto, que busca identificar estruturas anômalas que possam influir negativamente na qualidade do código desenvolvido. As mesmas são descritas nas subseções seguintes.

Tabela 2.1 – Falhas e anomalias provenientes do uso de herança e polimorfismo

Acronym	***Fault/Anomaly***
ITU	*Inconsistent type use (context swapping)*
SDA	*State definition anomaly (possible post-condition violation)*
SDIH	*State definition inconsistency (due to state variable hiding)*
SDI	*State defined incorrectly (possible post condition violation)*
IISD	*Indirect inconsistent state definition*
ACB1	*Anomalous construction behavior* (1)
ACB2	*Anomalous construction behavior* (2)
IC	*Incomplete construction*
SVA	*State visibility anomaly*

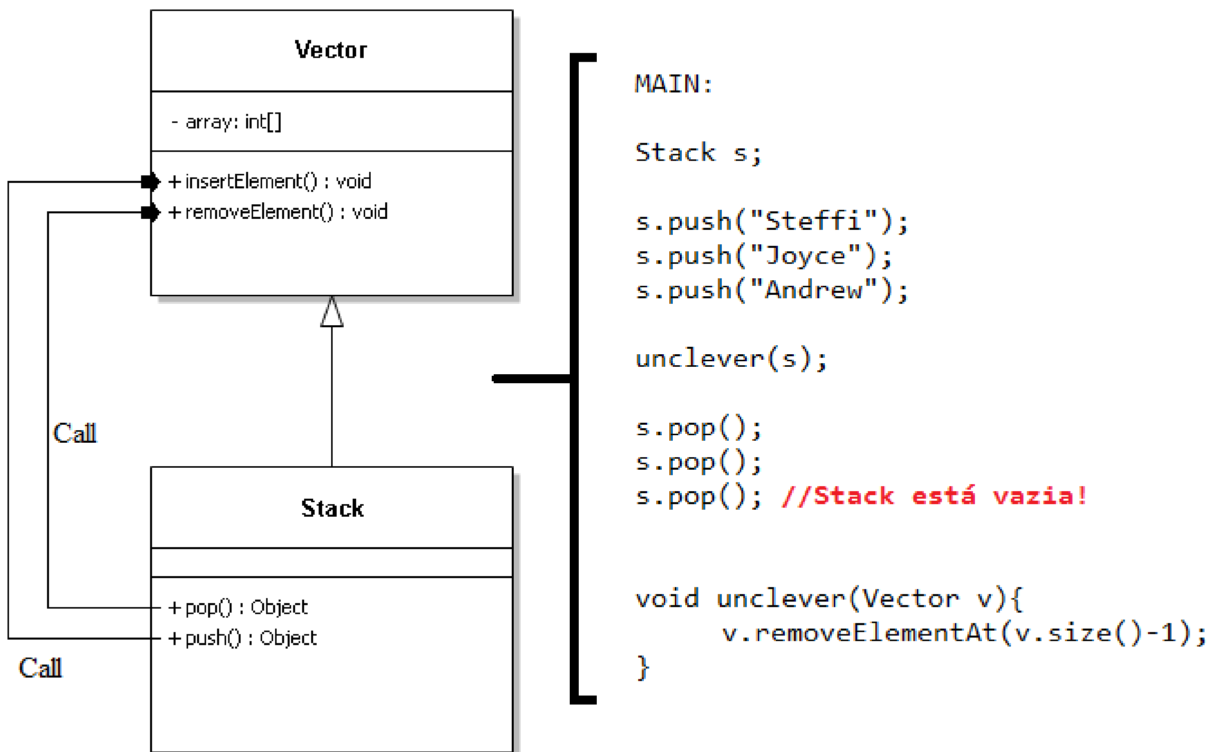
Fonte: Adaptado de AMMANN & OFFUTT (2008, p.240)

2.4.1 ITU - *Inconsistent type use*

O nome ITU significa uso de tipo inconsistente. A ITU é uma anomalia que pode ocorrer quando o relacionamento de herança é usado em contextos nos quais não existe uma relação de subtipo entre as classes pai e filha. Um exemplo deste fenômeno é apresentado na Figura 2.1.

No exemplo apresentado na Figura 2.1, a classe “Stack” é uma subclasse de “Vector”, embora uma pilha não seja de fato um vetor. Portanto os métodos da classe “Stack” não podem sobrescrever inteiramente os métodos da sua subclasse. Neste exemplo os métodos da classe Vector são utilizados pelos métodos da classe Stack a fim de executar movimentos de inclusão e remoção de elementos apenas no topo da pilha, porém os métodos da classe Vector podem remover ou incluir objetos em quaisquer posições se chamados individualmente.

Figura 2.1 – Exemplo de ITU



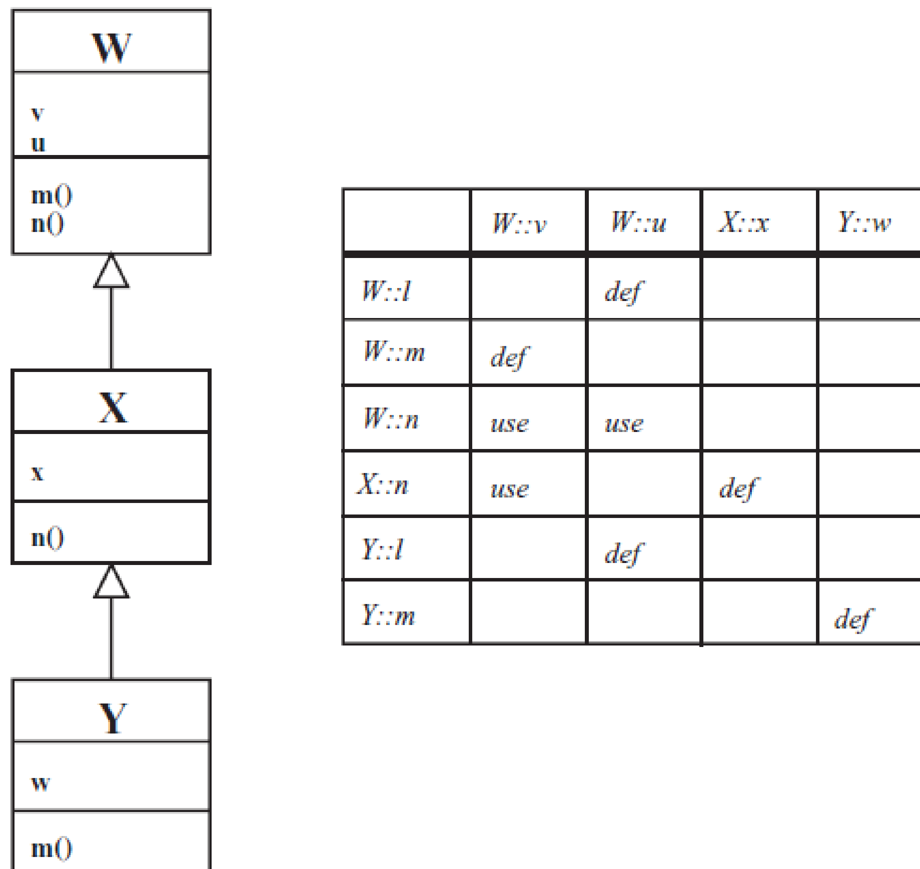
Fonte: Adaptado de AMMANN & OFFUTT (2013, p.17)

A anomalia é exemplificada no fragmento de código presente na Figura 2.1. É possível notar que quando um método da classe herdada é chamado por um objeto da classe herdeira, isto faz com que o estado deste objeto torne-se incoerente e apresente anomalia quando o mesmo é novamente usado como um objeto da subclasse.

2.4.2 SDA - *State definition anomaly*

O nome SDA significa anomalia de definição de estado. Ocorre quando as interações de estado de um descendente não são consistentes com as do ancestral, ou seja, quando os métodos da classe herdeira que sobrescrevem métodos da classe pai não garantem as mesmas definições para as variáveis de estado dos métodos da classe herdada.

Figura 2.2 – Exemplo de SDA



	W::v	*W::u*	*X::x*	*Y::w*
W::l		*def*		
W::m	*def*			
W::n	*use*	*use*		
X::n	*use*		*def*	
Y::l		*def*		
Y::m				*def*

Fonte: AMMANN & OFFUTT (2008, p.243)

Considere a hierarquia de classes apresentada na Figura 2.2. A classe W possui dois métodos, m() e n(), e dois atributos, v e u. O método W::m() define a variável de estado v e o método W::n() utiliza as variáveis u e v. A classe X sobrescreve o método n() de W, porém, só utiliza a variável v. A classe Y sobrescreve o método m() de W, porém não possui a definição de v.

Figura 2.3 – Exemplo de SDA (2)

```
MAIN:

Y objeto

objeto.m()
objeto.n() //v não foi definido
```

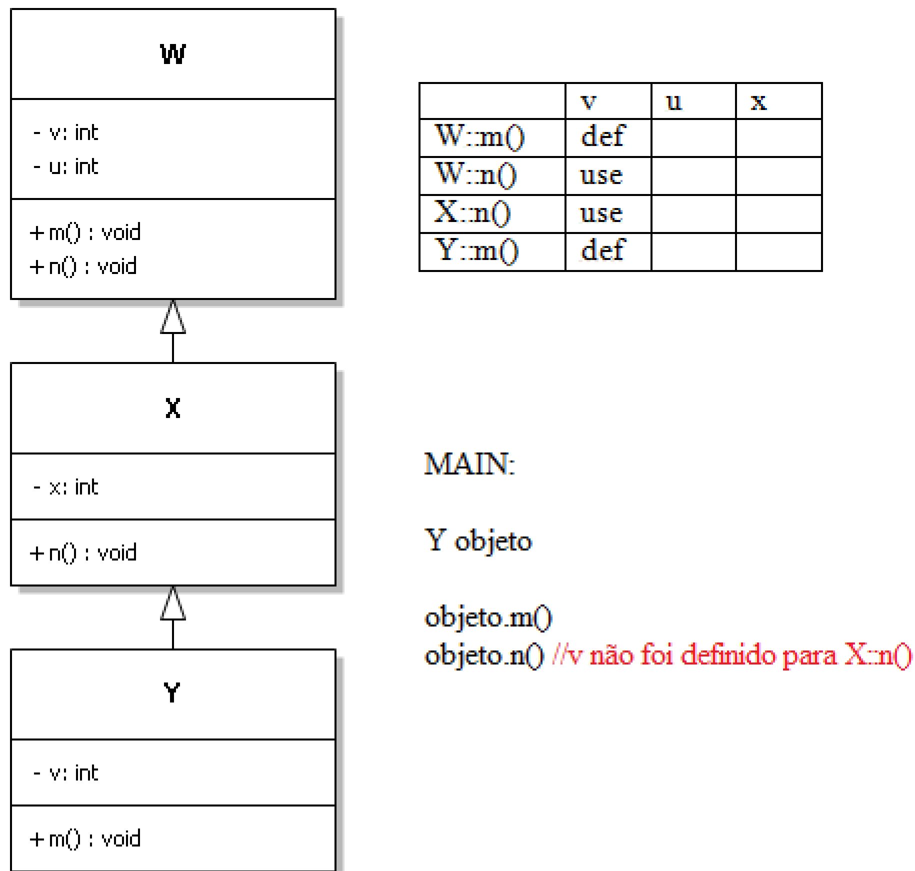
Fonte: O Autor

Quando um objeto do tipo Y utiliza os métodos m() e n(), nesta ordem (como apresentado na Figura 2.3), o método Y::m() é chamado, o qual não possui a definição de v, e então o método X::n() é chamado, o qual utiliza esta mesma variável de estado v. A variável v não foi devidamente definida por Y::m() anteriormente, pois o método Y::m() não é coerente ao método W::m() quanto às variáveis de estado. Sendo assim, uma anomalia de fluxo dados existe e pode causar falhas ao software quando executado.

2.4.3 SDIH - *State definition inconsistency*

O nome SDIH significa inconsistência de definição de estado causada por variáveis de estado escondidas. Isso ocorre quando a inclusão de uma variável local em uma classe herdeira sobrescreve a variável da classe herdada, deixando esta escondida, podendo assim causar uma anomalia de fluxo de dados. A Figura 2.4 apresenta um exemplo desta anomalia.

Figura 2.4 – Exemplo de SDIH



Fonte: O Autor

Na Figura 2.4, a classe W possui os atributos u e v e os métodos m() e n(). O método W::m() define a variável v e n() faz uso da variável v. A classe X possui o método n() que sobrescreve W::n() e também faz uso da variável v. A classe Y possui o método m(), que sobrescreve W::m() e também define a variável v. A classe Y também possui uma variável de estado v, que esconde a variável v definida por W para objetos do tipo Y.

Quando o trecho de código da Figura 2.4 é executado, a chamada do método Y::m() definirá a variável de estado v de Y. Quando o método n() for chamado, o método X::n(), que faz uso da variável v de W, pode apresentar inconsistência ou falhas de fluxo de dados, visto que a variável v de W não foi previamente definida.

2.4.4 SDI - *State defined incorrectly*

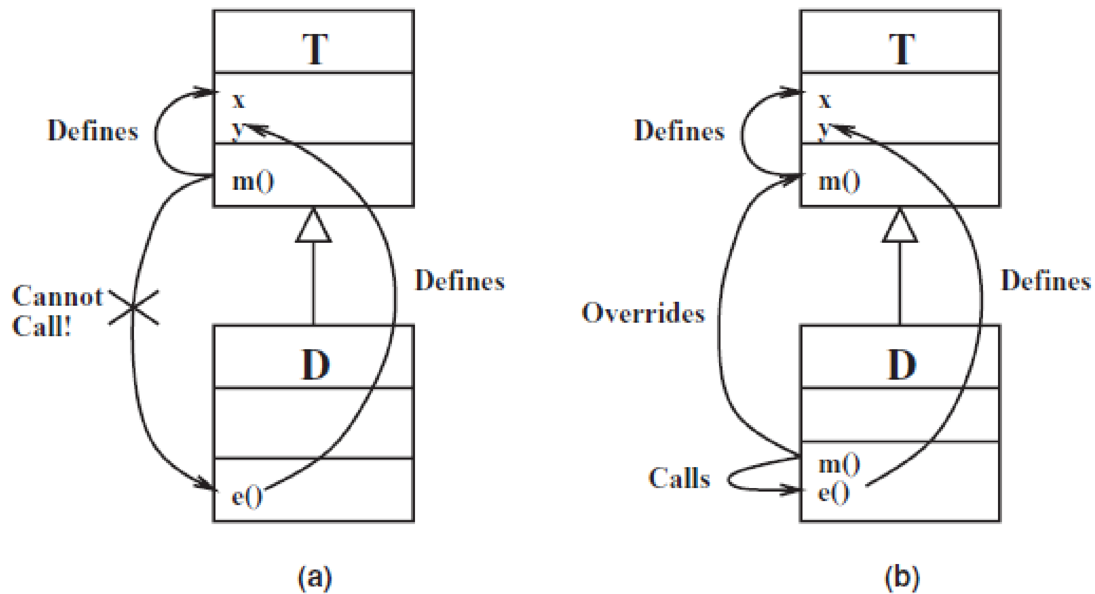
O nome SDI significa definição incorreta de estado. Esta anomalia ocorre quando o método herdeiro define a mesma variável de estado que o método antecessor define, porém a computação feita pelo método herdeiro não é semanticamente correspondente à computação do método sobrescrito em relação à mesma variável de estado. Sendo assim, o comportamento externo do descendente será diferente do comportamento do ancestral. Esta não pode ser considerada uma anomalia de fluxo de dados, más é uma anomalia de comportamento em potencial. Do ponto de vista de teste, é importante verificar se o comportamento do método herdeiro está coerente com a relação de subtipo definida na hierarquia.

2.4.5 IISD - *Indirect inconsistent state definition*

O nome ISSD significa falha de definição de estado indireta inconsistente. Esta falha pode ocorrer quando uma classe descendente adiciona um método que define uma variável de estado herdada, podendo assim causar anomalia de fluxo de dados.

Considere o diagrama de classes apresentado na Figura 2.5(a) onde a classe D é uma especialização da classe T. A classe D possui apenas um método, D::e(). Este método possui uma definição da variável de estado y, pertencente à classe T. A classe T possui duas variáveis de estado, x e y, e um método T::m(). O método T::m() possui a definição da variável de estado x e é possível notar que este método não possui visibilidade para chamar o método D::e(), visto que, D::e() encontra-se em uma classe herdeira.

Figura 2.5 – Exemplo de IISD



Fonte: AMMANN & OFFUTT (2008, p.244)

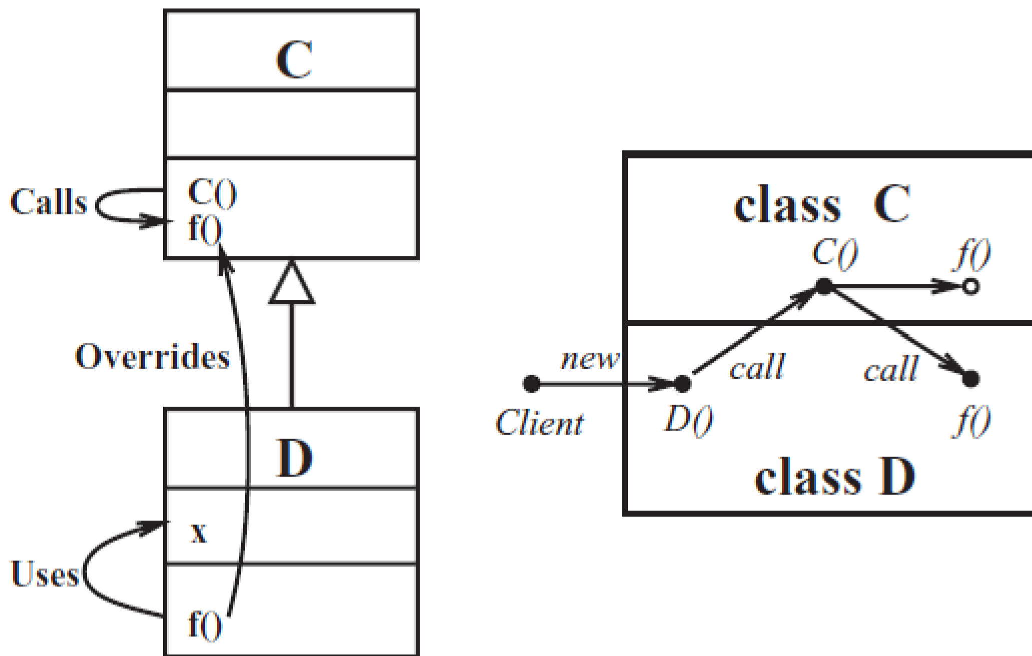
Se um novo método m() for definido na classe D, o método m() da classe T será sobrescrito, como representado na Figura 2.5(b). Diferente do caso anterior, agora D::e() poderá ser chamado por D::m(), visto que ambos fazem parte da mesma classe. Logo, uma execução do método m() da classe D acarretaria na execução do método e() que causaria alterações na variável de estado y da classe herdada. Como o método m() executado não é semanticamente equivalente ao método sobrescrito (método m() da classe T), deve-se verificar se o estado indicado por y é, de fato, consistente, uma vez que sua alteração foi feita em um escopo diferente de sua definição original.

2.4.6 ACB1 - *Anomalous construction behavior* (1)

O nome ACB1 significa comportamento anômalo de construção. O ACB1 é uma anomalia de fluxo de dados que pode ocorrer quando o construtor de uma classe ancestral executa um método polimórfico na classe herdeira, conforme ilustrado na Figura 2.6.

Conforme o exemplo apresentado, uma superclasse C possui um construtor C() que chama uma função polimórfica f(). Como f() é polimórfico, um método descendente de C é capaz de sobrescrevê-lo, como é feito por D::f() na Figura 2.6.

Figura 2.6 – Exemplo de ACB1



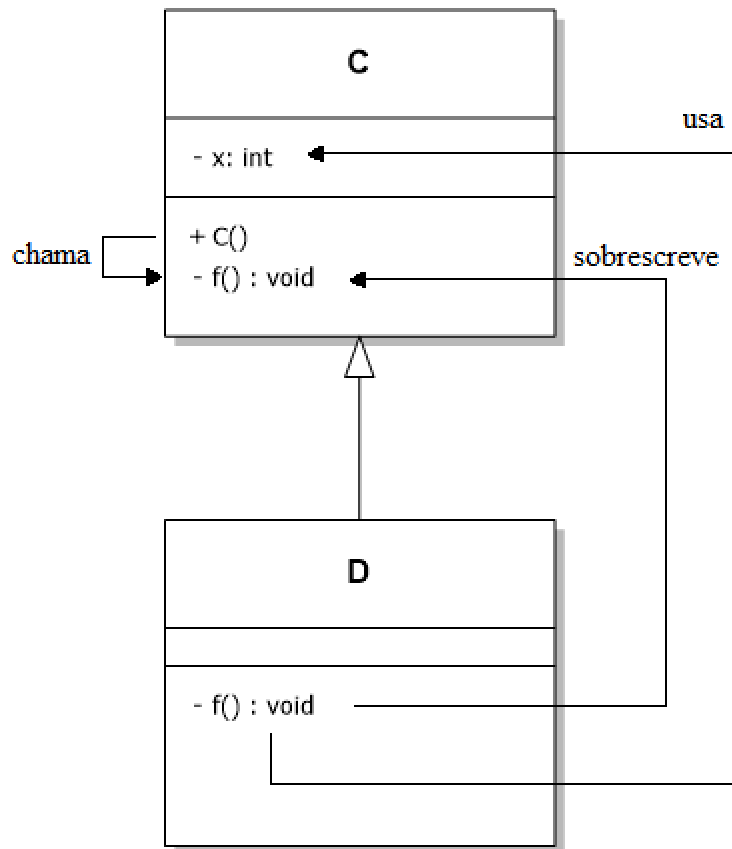
Fonte: AMMANN & OFFUTT (2008, p.245)

Dependendo da linguagem de programação utilizada, a criação de um objeto do tipo D poderá seguir o fluxo apresentado à direita na Figura 2.6, onde o construtor de C é executado antes do construtor de D. Neste cenário, a versão de f() chamada dentro de C() será aquela definida em D. Se D::f() faz uso de um atributo de D, a falha aparece, pois este atributo ainda não foi definido, visto que o construtor de D ainda não foi executado.

2.4.7 ACB2 - *Anomalous construction behavior* (2)

Ambos ACB2 e ACB1 são referentes à anomalias de fluxo de dados causadas por chamadas de métodos polimórficos feitas através de construtores. A ACB2 é uma anomalia de fluxo de dados que ocorre quando um método polimórfico é chamado por um construtor em uma subclasse e faz uso de variáveis de estado de sua superclasse, sem que os mesmos tenham sido previamente definidos, pois o método que o faz foi sobrescrito pelo método chamado.

Figura 2.7 – Exemplo de ACB2



Fonte: O Autor

No exemplo apresentado na Figura 2.7, as duas classes C e D possuem o método polimórfico f() e o construtor de C faz uso do método f(). Uma anomalia de fluxo de dados pode ocorrer se o método f() pertencente a D utiliza a variável de estado x, herdada de C, e esta não foi previamente definida em f() de C. Isto também depende do conjunto de variáveis usadas por f(), da ordem na qual as variáveis de estado de C são construídas e a ordem na qual f() é chamada no construtor de C.

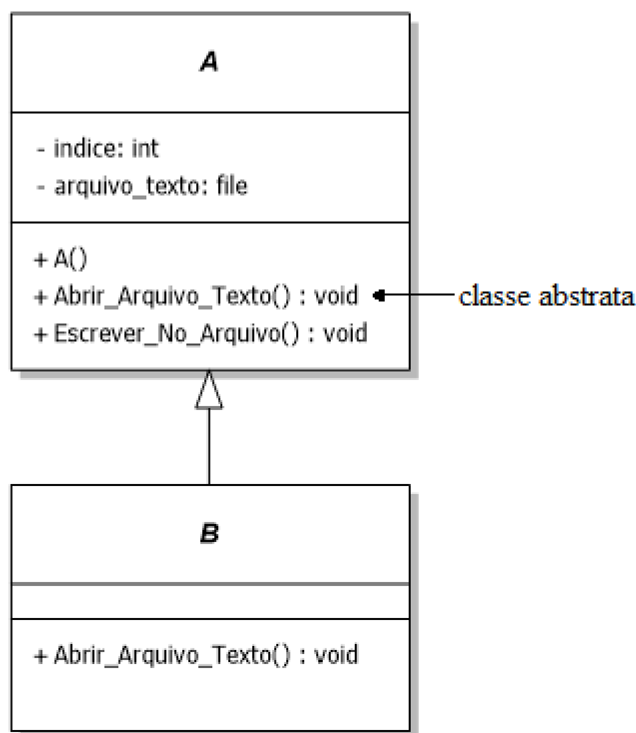
Portanto a execução de métodos polimórficos a partir de construtores não é uma prática segura, pois pode introduzir anomalias de fluxo de dados no processo de construção, como exemplificados em ambos os tópicos ACB1 e ACB2.

2.4.8 IC - *Incomplete construction*

O nome IC refere-se à falha de construção incompleta. Isso ocorre quando o estado inicial do objeto não é completamente definido. Em algumas linguagens de programação, os

valores das variáveis de estado de uma classe, antes de sua construção, são indefinidos. As variáveis de estado devem ser propriamente inicializadas durante a execução dos construtores das classes, fazendo com que, após sua execução, todas as variáveis de estado estejam bem definidas. Porém, quando a inicialização de uma variável é esquecida, isto pode acarretar em uma anomalia de fluxo de dados entre o construtor e cada método que faça uso desta variável antes que ela seja devidamente definida.

Figura 2.8 – Exemplo de IC



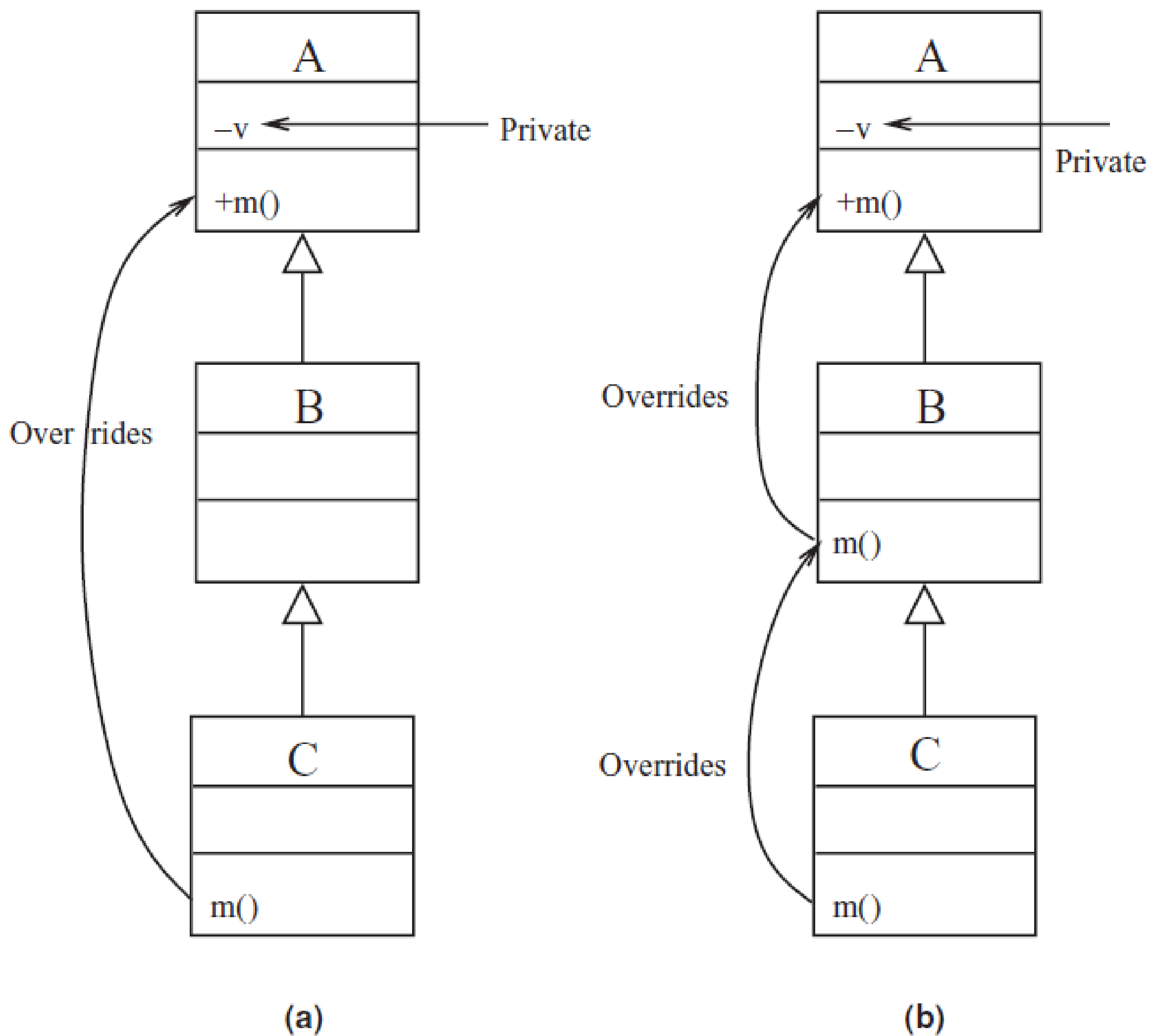
Fonte: O Autor

Na Figura 2.8 são apresentadas duas classes, A e B. O construtor de A não possui definição para a variável “arquivo_texto”. Considere que o método “Abrir_Arquivo_Texto()” possui uma definição para a variável “arquivo_texto”. Se este método for executado logo após o construtor de A e não houver usos desta variável até então, a execução não apresentará anomalia de fluxo de dados. Porém, supondo que o método “Escrever_No_Arquivo()” faça uso da variável “arquivo_texto” e se, após a execução do construtor de A, o método “Escrever_No_Arquivo()” for executado, então ocorrerá anomalia de fluxo de dados, visto que o valor de “arquivo_texto” encontra-se indefinido.

2.4.9 SVA - *State visibility anomaly*

O nome SVA se refere à anomalia de visibilidade de estado. Isto pode ocorrer quando uma classe ancestral possui uma variável de estado privada. A existência de uma variável privada em uma classe ancestral faz com que seus métodos devam ser acionados por suas classes filhas para que elas sejam capazes de modificar a variável. Todo esse processo pode causar anomalias de fluxo de dados, como mostrado na Figura 2.9.

Figura 2.9 – Exemplo de SVA



Fonte: AMMANN & OFFUTT (2008, p.247)

Seja uma superclasse A contendo uma variável de estado privada v e um método m(), que possui uma definição para a variável v. A superclasse A possui uma subclasse B, e esta possui uma subclasse C. A classe C possui uma definição de m(), que sobrescreve o método m() da classe A, como apresentado na Figura 2.9(a). Visto que a variável de estado v é

privada, o método m() de C não pode interagir propriamente com a mesma, portanto o método m() de C deve acionar o método m() de A no intuito de modificar a variável v.

Agora suponha que seja incluído o método m() em B, como apresentado na Figura 2.9(b). Nesta situação, a chamada super.m() presente no método C::m() aciona, na verdade, o método B::m(), que, por sua vez, deveria chamar o método A::m(). Portanto o método m() de C não possui mais controle direto sobre o fluxo de dados. A anomalia ocorre porque C::m() não pode mais chamar A::m() diretamente, a não ser que o faça utilizando o nome da classe. Da mesma forma, para manter o comportamento coerente, o método m() de B deve fazer a chamada ao método sobrescrito A::m().

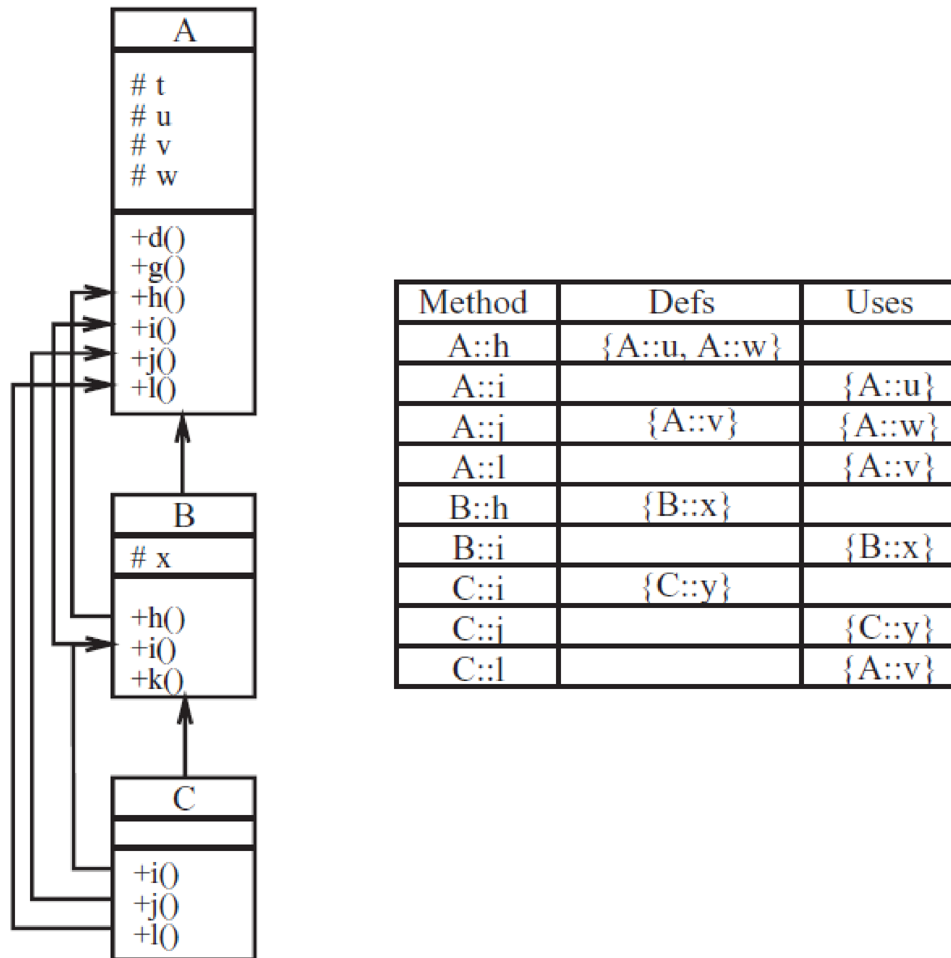
Em geral, quando possuímos variáveis de estado privadas, a única forma de evitar que anomalias de fluxo de dados como esta ocorram é fazendo com que todos os métodos que sobrescrevem métodos de seus ancestrais possuam chamadas para os métodos sobrescritos. Esta anomalia também depende da linguagem de programação utilizada, visto que, algumas linguagens não possuem palavras-chave para acionar métodos de classes pertencentes à sua hierarquia.

2.5 Grafo *Yo-Yo*

Como já mencionado, a orientação a objetos proporciona novos rumos de modelagem para a codificação. Porém, tendo em vista o crescimento da complexidade de integração entre os componentes do software, muitas vezes a identificação da versão do método que será executada não é uma tarefa trivial. O uso de polimorfismo faz com que diferentes métodos sejam acionados dependendo do objeto utilizado. A execução pode variar para cima e para baixo entre as camadas de herança. Esta variação é chamada de efeito *yo-yo*, nomenclatura utilizada por Amman;Offut (2008). Baseado neste conceito, Amman;Offut (2008) apresenta a definição de um grafo de fluxo, conhecido como grafo *yo-yo*. Este grafo visa representar a interação entre os métodos herdados e sobrescritos para cada descendente.

No grafo *yo-yo* as chamadas de método para método são representadas por flechas vindas do método que executa a chamada para o método chamado. Cada classe possui um nível no grafo *yo-yo* que contém as chamadas feitas por um objeto do tipo desta classe. No grafo existem dois tipos de flechas, as flechas em negrito são as chamadas que serão efetuadas e as flechas claras são chamadas que não podem ser feitas, pois estas foram sobrescritas por métodos polimórficos.

Figura 2.10 – Exemplo de grafo *Yo-Yo*: hierarquia de classes



Method	Defs	Uses
A::h	{A::u, A::w}	
A::i		{A::u}
A::j	{A::v}	{A::w}
A::l		{A::v}
B::h	{B::x}	
B::i		{B::x}
C::i	{C::y}	
C::j		{C::y}
C::l		{A::v}

Fonte: AMMANN & OFFUTT (2008, p.238)

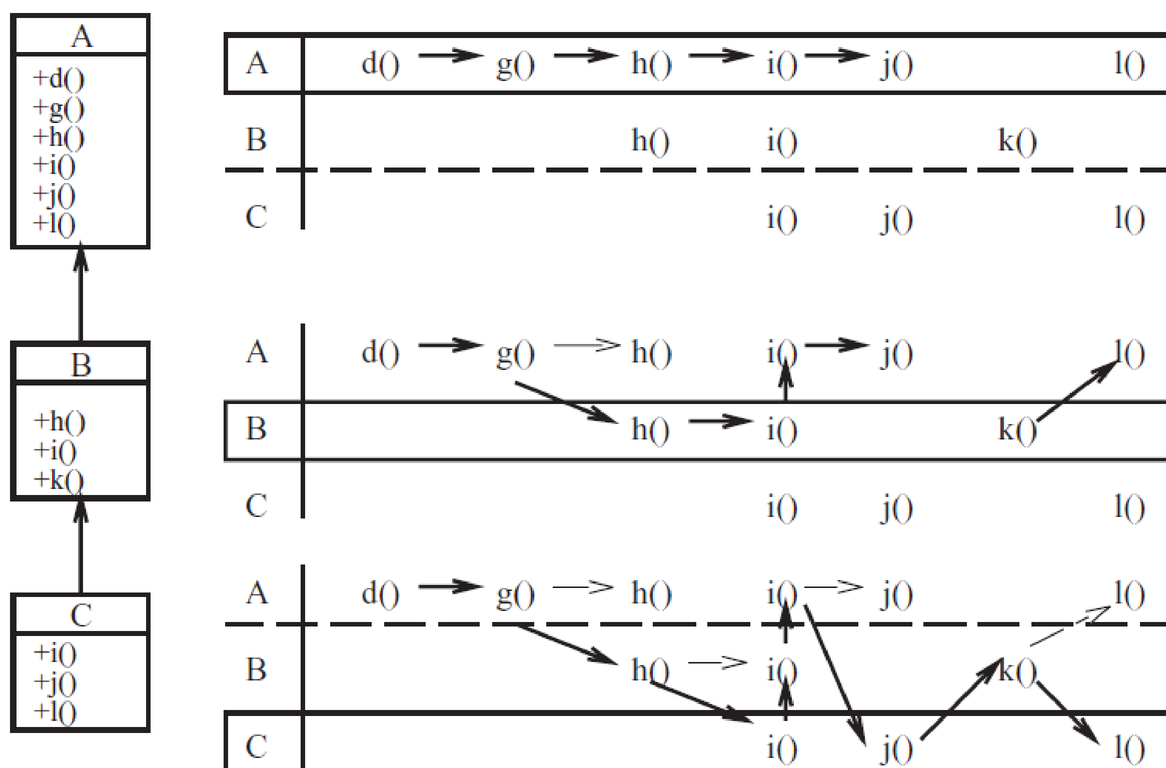
Para exemplificar a construção de um grafo *yo-yo*, será considerada a hierarquia de classes apresentada na Figura 2.10. Nela existem: três classes (A, B e C), as chamadas entre seus métodos e uma tabela que indica as definições e usos das variáveis de estado por cada método. O grafo *yo-yo* correspondente é representado na Figura 2.11, tendo em vista que o relacionamento de chamadas feitas entre os métodos está representado na Tabela 2.2. É importante notar as diferenças entre os níveis (diferentes classes). Cada tipo de objeto possui uma visibilidade diferente dos demais em relação aos métodos chamados durante a execução. Muitas vezes esta percepção não é clara aos desenvolvedores e testadores, fazendo-se útil o uso de diagramas como o grafo *yo-yo*, no intuito de evitar anomalias tais quais as descritas nos tópicos anteriores.

Tabela 2.2 – Exemplo de grafo *Yo-Yo*: chamadas interclasse

Classe	Método	Chamadas
A	d()	g()
	g()	h()
	h()	i()
	i()	j()
B	h()	i()
	i()	**A**::i()
	k()	l()
C	i()	**B**::i()
	j()	k()

Fonte: O Autor

Figura 2.11 – Exemplo de grafo *Yo-Yo*: diagrama



Fonte: AMMANN & OFFUTT (2008, p.239)

O grafo *yo-yo* apresentado na Figura 2.11 ilustra a sequência de chamadas que serão desencadeadas a partir da chamada do método d() para objetos dos tipos de A, B e C, baseado nos relacionamentos entre métodos exibidos na Tabela 2.2. A primeira camada (nível) do grafo representa a chamada do método d() a partir de um objeto do tipo da classe A. Neste caso é possível notar que a sequência é simples e direta. A segunda camada representa a

sequência de chamadas para um objeto do tipo B. Neste caso existem métodos polimórficos que serão executados ao invés dos métodos da classe A, como é o caso do método h(). A terceira camada, por fim, apresenta a sequência de chamadas para um objeto do tipo C. Nesta camada a complexidade das conexões devido à existência de métodos polimórficos é mais evidente. A existência destas conexões complexas requer maior esforço na etapa de criação de testes. Isto enaltece a importância da criação de ferramentas que auxiliem neste processo.

2.6 LLVM

LLVM é uma infraestrutura de compilação compreendida por uma coleção de compiladores modulares. Apesar do nome (*Low Level Virtual Machine*), o projeto não é especificamente relacionado a máquinas virtuais tradicionais, porém pode fornecer bibliotecas úteis na construção das mesmas. O LLVM começou como um projeto de pesquisa na Universidade de Illinois, no intuito de oferecer uma infraestrutura para pesquisa que facilitasse o desenvolvimento de análise e transformação de código intermediário. Desde então, LLVM cresceu e possui diversos subprojetos e está presente em diversas pesquisas acadêmicas (The LLVM *Compiler Infrastructure*, 2016).

O subprojeto LLVM Core possui bibliotecas de otimização e suporte à geração de código. Estas bibliotecas são construídas em torno de uma representação de código conhecida como representação intermediária LLVM (LLVM IR) (The LLVM *Compiler Infrastructure*, 2016). No protótipo desenvolvido neste trabalho, a identificação das estruturas do código pertinentes em teste ocorre a partir da análise da representação intermediária LLVM IR.

O subprojeto Clang consiste em um *front-end* para as linguagens C, C++, Objective-C e Objective-C++ (CLANG: A C LANGUAGE FAMILY FRONTEND FOR LLVM, 2016). Neste projeto, Clang foi utilizado na compilação e conversão de arquivos .cpp (código em C++) em arquivos .ll (LLVM) a fim de gerar exemplos de entrada para o protótipo desenvolvido.

A representação intermediária LLVM apresenta algumas peculiaridades quanto ao formato do código gerado após a compilação. O formato deste código distancia-se da linguagem C++ e aproximando-se de uma linguagem mais genérica. Sendo assim, a identificação dos padrões das instruções presentes nesta representação intermediária foi um dos principais fatores de análise na construção do protótipo desenvolvido. A etapa de

identificação de padrões tal como os algoritmos envolvidos neste processo são discorridos no Capítulo 3.

2.7 Trabalhos Relacionados

A fim de identificar trabalhos relacionados à ferramenta desenvolvida neste trabalho, foram pesquisadas ferramentas de teste que possuem aspectos relacionados à cobertura de código. A ferramenta desenvolvida analisa relacionamentos entre classes, tais como herança e polimorfismo, e aceita arquivos .ll como entrada, o que possibilita a analise de programas independente da linguagem de programação utilizada na sua implementação original.

A publicação "*A Survey of Coverage Based Testing Tools*", de 2006, apresenta uma comparação entre dezessete ferramentas de teste baseadas em cobertura de código. O objetivo da análise feita nesta publicação foi comparar estas ferramentas a ferramenta eXVantage. Entre as características comparadas então: as linguagens de programação às quais estas ferramentas se aplicam e os níveis de cobertura disponibilizados.

A Figura 2.12 apresenta as linguagens de programação aceitas por cada ferramenta. Nesta imagem é possível notar que todas as ferramentas estão presas a linguagens de programação especificas.

Figura 2.12 – Linguagens suportadas

Tool Name:	C++/C	Java	Other
Agitar [3]		X	
Bullseye [5]	X		
Clover [6]		X	.net
Cobertura [7]		X	
CodeTest [8]	X		
Dynamic [9]	X		
EMMA [21]		X	
eXVantage [4]	X	X	
gcov [10]	X		
Insure++ [11]	X		
Intel [12]	X		
JTest [15]		X	.net
JCover [13]		X	
Koalog [14]		X	
PurifyPlus [16]	X	X	Basic, .net
Semantic Designs (SD) [17]	X	X	C#, PHP, COBOL, PARLANSE
TCAT [18]	X	X	

Fonte: YANG, LI, & WEISS (2009, p.2)

A Figura 2.13 apresenta a extensão da cobertura proporcionada por algumas das ferramentas analisadas. Como pode ser observado, algumas das ferramentas possuem análise de cobertura para métodos e classes, porem não há identificação de relações polimórficas ou da existência de herança.

Figura 2.13 – Critérios de medida de cobertura

	Line	Decision	Method	Class
Agitar [3]	X	X	X	X
Bullseye [5]		X	X	
Clover [6]	X	X	X	
Cobertura [7]	X	X		
CodeTest [8]	X	X		
Dynamic [9]	X	X	X	
EMMA [21]	X		X	X
eXVantage [4]	X	X	X	
gcov [10]	X			
Insure++ [11]	X			
JCover [13]	X	X	X	X
JTest [15]	X	X		
PurifyPlus [16]	X		X	
SD [17]	X	X	X	X
TCAT [18]	X	X	X	X

Fonte: YANG, LI, & WEISS (2009, p.3)

Em outra pesquisa publicada a respeito de ferramentas para teste baseadas em cobertura de código, são comparadas cinco outras ferramentas: EvoSuite, JaCoCo, AURORA, DCC e OCCF. Uma das tabelas disponíveis na pesquisa "*The Study of Various Code Coverage Tools*", publicada em 2014, é exibida na Figura 2.14. Nesta figura são comparadas as linguagens suportadas por estas ferramentas. Assim como no caso anterior, as ferramentas estão relacionadas a linguagens específicas, exceto pela OCCF.

Figura 2.14 – Linguagens suportadas (2)

Tool Name	C/C++	Java	Other
EvoSuite	--	✓	--
JaCoCo	--	✓	--
AURORA	--	✓	--
DCC	✓	--	--
OCCF	✓	✓	✓

Fonte: SHELKE & NAGPURE (2014, p.2)

A Figura 2.15 exibe os critérios de cobertura abrangidos por cada ferramenta. Como é possível notar, são analisados apenas critérios de cobertura específicos, como: cobertura de linhas (*Statement*/*Line*), cobertura de decisões (*Branch*/*Decision*) e cobertura de métodos (*Method*/*Function*). Portanto estas ferramentas diferem da proposta apresentada neste trabalho quanto à identificação de estruturas anômalas resultantes do uso de herança e polimorfismo.

Figura 2.15 – Critérios de medida de cobertura (2)

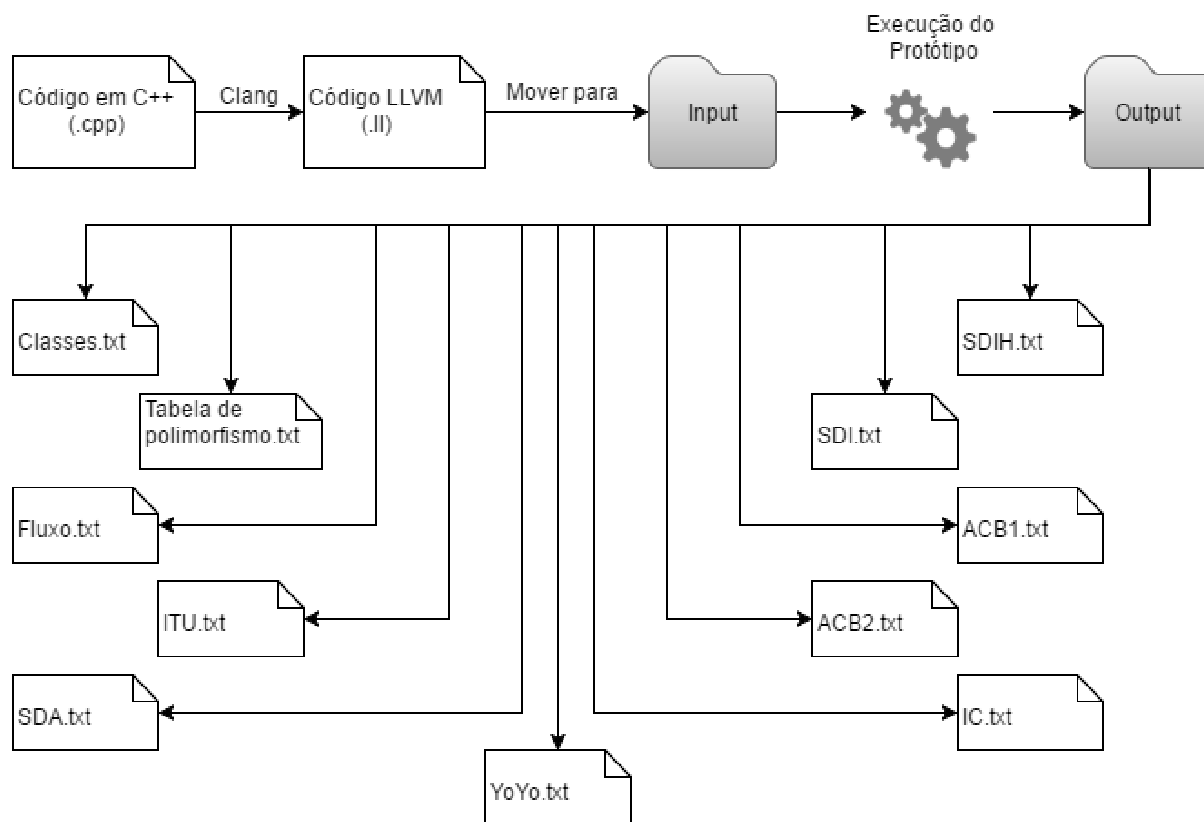
Tool name	Statement/ Line	Branch/ Decision	Method/ Function
Evosuite	--	✓	--
JaCoCo	✓	✓	✓
AURORA	✓	--	--
DCC	✓	✓	✓
OCCF	✓	✓	--

Fonte: SHELKE & NAGPURE (2014, p.3)

3 IDENTIFICAÇÃO DE POTENCIAIS ANOMALIAS EM CÓDIGOS ORIENTADOS A OBJETOS PARA AUXILIO NA GERAÇÃO DE TESTES

A proposta deste trabalho consiste na criação de uma ferramenta capaz de analisar arquivos LLVM (.ll) a fim de gerar as estruturas básicas de um código orientado a objetos para análise de possíveis estruturas anômalas e, com isso, influenciar na criação de testes baseados nos relacionamentos interclasse. As estruturas básicas geradas consistem em classes, métodos, atributos e os relacionamentos existentes entre eles. O protótipo foi desenvolvido em C++ no intuito de usá-lo como entrada de sua própria execução a fim de gerar estruturas de teste para o mesmo. A Figura 3.1 ilustra o uso da ferramenta e os arquivos de entrada e saída envolvidos.

Figura 3.1 – Representação da proposta



Fonte: O Autor

Como representado na Figura 3.1, a entrada esperada para o protótipo desenvolvido é um arquivo .ll contendo código LLVM IR gerado a partir de um código fonte em C++ pelo Clang. Como mencionado no Capítulo 2, Clang foi utilizado neste trabalho para a conversão de arquivos C++ para o formato desejado. Após a conversão dos arquivos, estes devem ser movidos para a pasta “Input” do protótipo. Arquivos de outras linguagens de programação

também podem ser usados, desde que exista um compilador que transforme estes arquivos em arquivos .ll, e que a geração do código LLVM IR feita por este compilador siga os mesmo padrões utilizados pelo Clang. Note que apenas um arquivo de entrada é previsto.

Após a execução do protótipo, alguns arquivos serão gerados a fim de informar ao usuário os pontos de possíveis anomalias encontrados. Estes arquivos estarão dentro de uma pasta denominada “Output”. Os arquivos Classes.txt, Fluxo.txt, YoYo.txt e Tabela de Polimorfismo.txt foram criados no intuito de dar visibilidade às estruturas identificadas pelo protótipo. Os demais arquivos gerados possuem guias para a identificação de estruturas anômalas que se enquadram nas anomalias descritas por AMMANN & OFFUTT (2008, p.240), discorridas no Capítulo 2.

Nas seções seguintes são apresentados esclarecimentos sobre as peculiaridades do código LLVM (arquivo .ll) e os padrões utilizados na identificação das estruturas. Também são discutidos detalhes referentes à implementação do protótipo e as possibilidades de expansão da ferramenta. Ao longo deste capítulo são exibidas exemplificações dos arquivos de saída gerados e suas funcionalidades.

3.1 Estruturas internas

A fim de armazenar as informações necessárias para a análise de estruturas anômalas, foram criadas as estruturas representadas na Figura 3.2. Na Figura 3.2 são exibidas todas as classes do protótipo e seus atributos. Os métodos de cada classe não foram inclusos na imagem no intuito de manter o foco nos pontos principais do diagrama, evitando poluí-lo. A seguir são descritas todas as classes e os relacionamentos entre as mesmas.

Para cada classe presente no arquivo de entrada (.ll), um objeto do tipo Classe será criado durante a execução do protótipo. Cada Classe pode possuir: um nome, uma lista de métodos, uma lista de atributos, uma lista de classes herdadas, e uma linha da tabela virtual (utilizada em chamadas polimórficas). A tabela virtual é uma tabela que possui todos os métodos polimórficos entre si. Cada linha da tabela virtual possui os métodos que serão chamados para um objeto de um determinado tipo. Cada coluna desta tabela pode ser referenciada em chamadas polimórficas, o que determina o método chamado é a classe do objeto que acarretou a chamada. A partir da dupla, linha e coluna, é possível determinar o método chamado. Mais detalhes sobre a tabela virtual são apresentados ao longo da seção 3.2.2.

A lista de atributos de uma classe é composta por uma lista de objetos do tipo "Atributo". Um objeto do tipo "Atributo" possui: um nome, uma identificação (ID), o nome da classe a qual o atributo pertence e uma lista de outros nomes com os quais o mesmo atributo aparece ao longo do código. No arquivo de entrada lido, alguns atributos tendem a aparecer com variações de sua nomenclatura ao longo do código. Isso acontece quando há bifurcações no fluxo, por exemplo, no uso de loops ou condicionais. Visto que os atributos são identificados ao longo do código, os atributos que não são usados ou definidos pelo código não são exibidos no arquivo LLVM, sendo assim, impossível identificá-los. O atributo de identificação (ID) é utilizado na diferenciação entre variáveis de mesmo nome, porém de classes diferentes. Esta situação é exemplificada na seção 3.2.1.

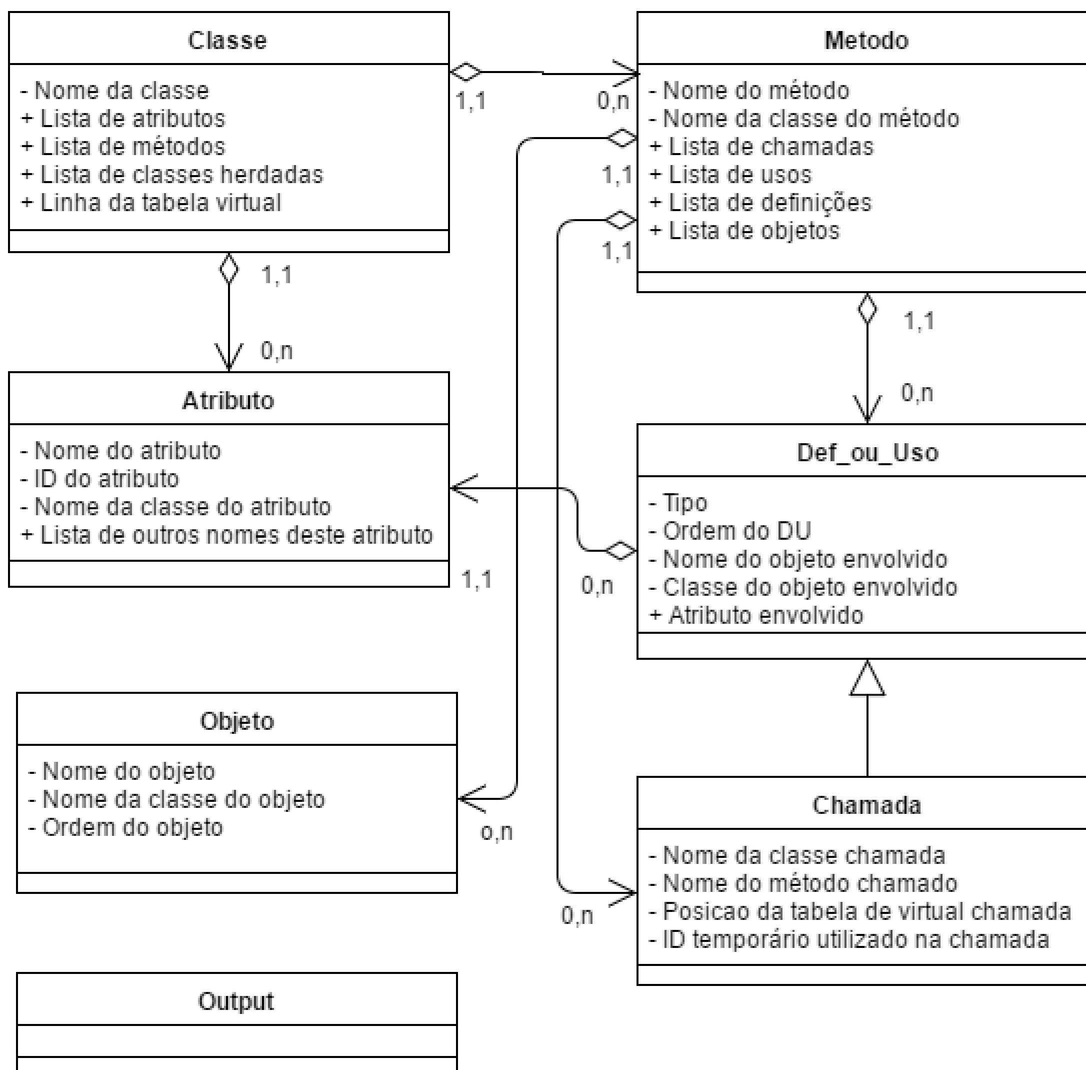
A lista de métodos de uma classe consiste em uma lista de objetos do tipo "Metodo". Para cada método pertencente às classes criadas pelo usuário no arquivo de entrada, um objeto do tipo "Metodo" será criado. Cada método possui: o nome do método, o nome da classe a qual pertence, uma lista de chamadas, uma lista de definições, uma lista de usos e uma lista de objetos definidos em seu escopo. Estas listas são detalhadas ao longo desta seção.

A lista de chamadas contidas na classe "Metodo" refere-se a uma lista de objetos do tipo "Chamada". Um objeto do tipo "Chamada" é criado toda vez que uma chamada para outro método é identificada dentro do escopo do método. Geralmente as chamadas possuem um método alvo e um objeto que acarretou esta chamada. As chamadas polimórficas não possuem apenas um método alvo, porém apontam para uma coluna da tabela virtual que possui métodos que são polimórficos entre si.

Um objeto do tipo "Chamada" é composto por: um caractere de tipo, um número que indica a ordem de aparição no código, o nome e a classe do objeto envolvido nesta chamada, o nome e a classe do método chamado e a posição da tabela virtual chamada (no caso de chamada polimórfica). O caractere de tipo é um atributo herdado da classe "Def_ou_Uso" e é utilizado para identificar se o objeto é uma definição (D), um uso (U), uma chamada usual (C) ou uma chamada polimórfica (P). Este caractere é utilizado a fim de promover a reutilização de alguns métodos ao longo do código do protótipo. O número de ordem indica em qual linha esta chamada aparece dentro do escopo do método. Este número foi criado no intuito de identificar se uma definição ocorre antes de um determinado uso ou de uma chamada. A ordem pode ser usada a fim de identificar os pares de definição e uso em trabalhos futuros, como abordado na seção 3.4. O objeto que acarretou a chamada é armazenado juntamente com sua respectiva classe. O método chamado também deve ser armazenado a fim de registrar o fluxo de execução e o relacionamento entre os métodos. Se a chamada registrada for uma

chamada polimórfica, então o método chamado é indeterminado. Portanto existe um atributo na classe "Chamada" que registra a coluna da tabela virtual a qual esta chamada referencia. O método que será chamado é determinado dinamicamente através do tipo do objeto que acarretou a chamada e a classe deste objeto é o que determina a linha da tabela virtual que deverá ser usada.

Figura 3.2 – Classes do Protótipo



Fonte: O Autor

Assim como a lista de chamadas, as listas de definições e usos estão diretamente relacionadas ao escopo do método. A lista de definições é uma lista de objetos do tipo "Def_ou_Uso". Para cada definição de atributo encontrada ao longo do método, um objeto do tipo "Def_ou_Uso" é criado e inserido na lista de definições do método correspondente. Assim como a lista de definições, a lista de usos também é uma lista de objetos do tipo

"Def_ou_Uso" e recebe um novo objeto toda vez que um uso de atributo ou objeto é encontrado no escopo do método.

Um objeto da classe "Def_ou_Uso" possui: um caractere de tipo, um número que indica a ordem de aparição no código, o nome e a classe do objeto envolvido nesta definição ou uso e o atributo definido ou utilizado neste contexto. O tipo e a ordem seguem a mesma definição apresentada anteriormente dentro do contexto da classe "Chamada". Uma definição ou uso pode fazer alusão apenas a um objeto ou diretamente ao atributo do objeto, portanto o campo atributo foi criado no intuito de registrar esta ação.

A lista de objetos presente na classe "Metodo" é uma lista de objetos do tipo "Objeto". A classe Objeto possui: o nome do objeto, a classe do objeto e a ordem em que ele aparece definido dentro do escopo do método ao qual pertence. Todos os objetos declarados no escopo de um método específico são registrados na lista de seu respectivo método. Cada definição ou uso encontrado no escopo deste mesmo método é associado com a lista de objetos declarados, a fim de identificar quando este objeto é alterado ou utilizado.

Embora não exista um relacionamento de subtipo entre as classes, classe "Chamada" é herdeira de classe "Def_ou_Uso". Este relacionamento foi definido porque uma chamada requer muitos dos atributos e métodos que também estão presentes em uma definição ou uso, portanto com uma relação de herança entre as classes, estes métodos e atributos podem ser compartilhados e não precisam ser redefinidos dentro de um novo escopo. A classe "Chamada" é uma especialização, pois além de todos os atributos pertencentes à classe "Def_ou_Uso", ainda são necessários atributos que armazenam a identificação dos métodos chamados ou posições da tabela virtual referenciadas.

A classe "Output" foi criada no intuito de separar os métodos que geram os arquivos de saída dos demais. Portanto esta classe não possui relacionamento com as demais e nem atributos.

3.2 Leitura do arquivo .ll

O formato do arquivo .ll possui algumas peculiaridades, visto que encontra-se em uma linguagem intermediária. A linguagem apresentada é semelhante à linguagem de baixo nível Assembly, e, portanto, sua leitura não é intuitiva. Outra peculiaridade do LLVM é o seu foco na otimização do código. Por exemplo, quando um determinado método não é acionado durante a execução do código compilado, o compilador ignorará sua declaração e o mesmo não será exibido no arquivo .ll.

O primeiro desafio na etapa de identificação das estruturas do programa foi a necessidade de validação dos padrões encontrados. Quando o código é compilado, o arquivo na linguagem intermediária exibirá não apenas os métodos criados pelo desenvolvedor, mas também todos os outros utilizados por bibliotecas importadas. Neste trabalho o foco do teste serão as classes e métodos declarados no código de entrada. É possível distinguir os métodos criados pelo desenvolvedor do código alvo dos demais métodos através da análise do padrão do nome da classe dentro da linha que possui a definição do método. A necessidade do uso de validações como esta influi negativamente no tempo de processamento do arquivo de entrada e faz com que sejam necessárias duas varreduras no código de entrada.

A ordem de exibição dos componentes do código no arquivo LLVM pode ser observada na Figura 3.3. A Figura 3.3 possui legendas em vermelho para identificação, em alto nível, das informações que aquele trecho do código abrange. As primeiras linhas do arquivo possuem o nome do arquivo C++ que deu origem ao código e as informações da plataforma alvo de execução. Após estas especificações, são exibidas todas as classes utilizadas pelo código e, logo após, todos os nomes de métodos declarados ao longo do código, porém desprovidos de detalhes. As linhas seguintes são declarações de constantes. Nas constantes estarão presentes todos os conteúdos de saídas textuais utilizados ao longo do código, como os parâmetros contidos nas instruções “cout” e “printf”, por exemplo. Estas constantes estão respectivamente associadas a um identificador único. Justamente com as constantes, estão as declarações de todas as linhas da tabela virtual, que são posteriormente combinadas a fim de analisar os relacionamentos polimórficos. Após as constantes, encontram-se todas as definições de métodos utilizados ao longo do código. Dentro da definição do método estão: todas as declarações de objetos pertencentes ao método, todas as definições e usos de objetos e atributos e todas as chamadas realizadas para os demais métodos. E, por fim, encontram-se as especificações da versão do Clang utilizada na geração no arquivo analisado.

Figura 3.3 – Representação da estrutura básica do arquivo .ll

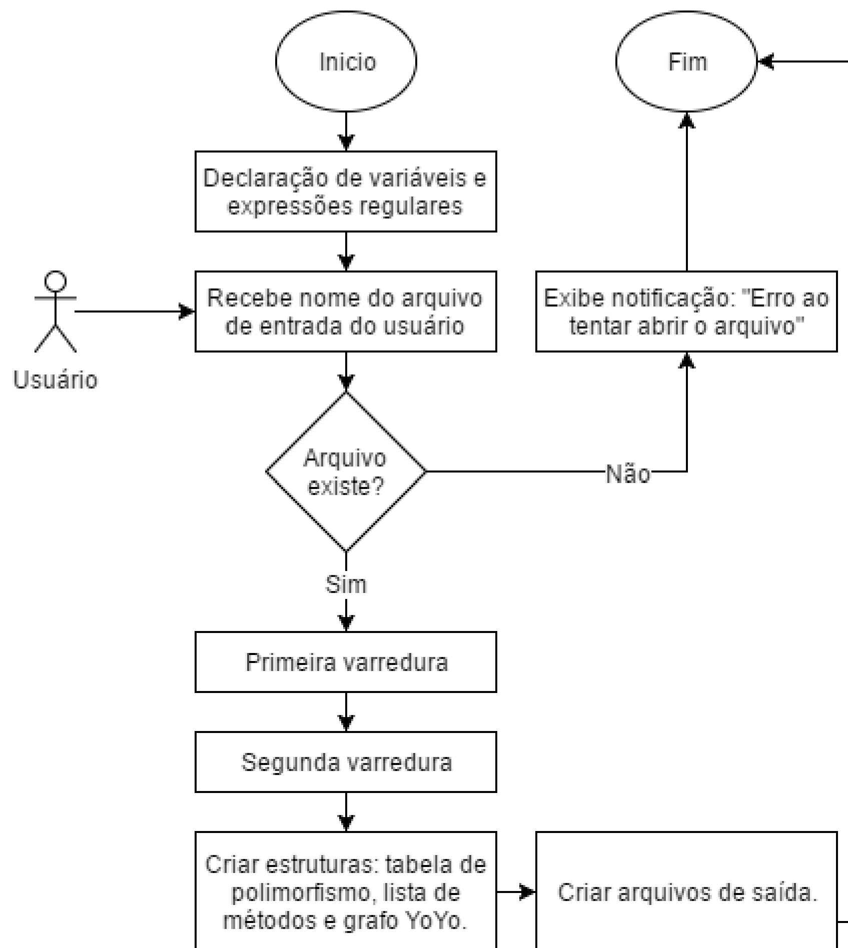
```
Nome do arquivo original
; ModuleID = 'SDA.cpp'

Target
target datalayout = "e-m:w-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-w64-windows-gnu"

Declaração de todas as classes
%"class.std::num_put" = type { %"class.std::locale::facet.base", [4 x i8] }
%class.Y = type { %class.X, i32 }
%class.X = type { %class.W, i32 }
...

Declaração de todos os métodos, sem tipo ou classe
$_ZN1Y1mEv = comdat any
...

Declaração de constantes e tabelas virtuais
@_ZTV1X = linkonce_odr unnamed_addr constant [4 x i8*] [i8* null, i8* bitcast ({ i8*, i8

Declaração dos métodos
define linkonce_odr void @_ZN1W1mEv(%class.W* %this) unnamed_addr #5 comdat align 2 {
entry:
  %this.addr = alloca %class.W*, align 8
  store %class.W* %this, %class.W** %this.addr, align 8
  %this1 = load %class.W*, %class.W** %this.addr
  %v = getelementptr inbounds %class.W, %class.W* %this1, i32 0, i32 2
  store i32 1, i32* %v, align 4
  ret void
}
...

Informações sobre a versão do Clang utilizada
!0 = !{i32 1, !"PIC Level", i32 2}
!1 = !{!"clang version 3.7.0 (tags/RELEASE_370/final)"}
```

Fonte: O Autor

A Figura 3.4 apresenta uma visão geral do fluxo de execução do protótipo. No inicio são declaradas todas as expressões regulares que serão utilizadas na identificação de padrões no processamento do arquivo .ll, esta etapa é descrita nas seções 3.2.1 e 3.2.2. Após a declaração de variáveis, o código solicita ao usuário que informe o nome do arquivo de entrada, sem extensão. Este arquivo deve estar presente dentro da pasta input. Por exemplo, se o arquivo de entrada chama-se “arquivo.ll”, é esperado que o mesmo esteja dentro da pasta “Input” do protótipo e que o usuário informe apenas o nome “arquivo” à aplicação. Após a identificação do arquivo de entrada, o código verifica se o arquivo foi encontrado. Caso não o seja, será exibida a mensagem “Erro ao tentar abrir o arquivo” e a execução será encerrada.

Caso o arquivo de entrada exista, o próximo passo é a primeira varredura no código de entrada. Como mencionado anteriormente, são necessárias duas varreduras no código de entrada a fim de coletar as informações necessárias para a análise de estruturas. Após o armazenamento das estruturas nas respectivas classes, são criadas estruturas auxiliares para percorrer os dados obtidos. Estas estruturas auxiliares consistem em uma lista contendo todos os métodos lidos e uma matriz contendo a tabela virtual. A partir da obtenção de todas as estruturas internas necessárias, os arquivos de saída serão gerados e a execução chegará ao fim.

Figura 3.4 – Representação do fluxo de execução



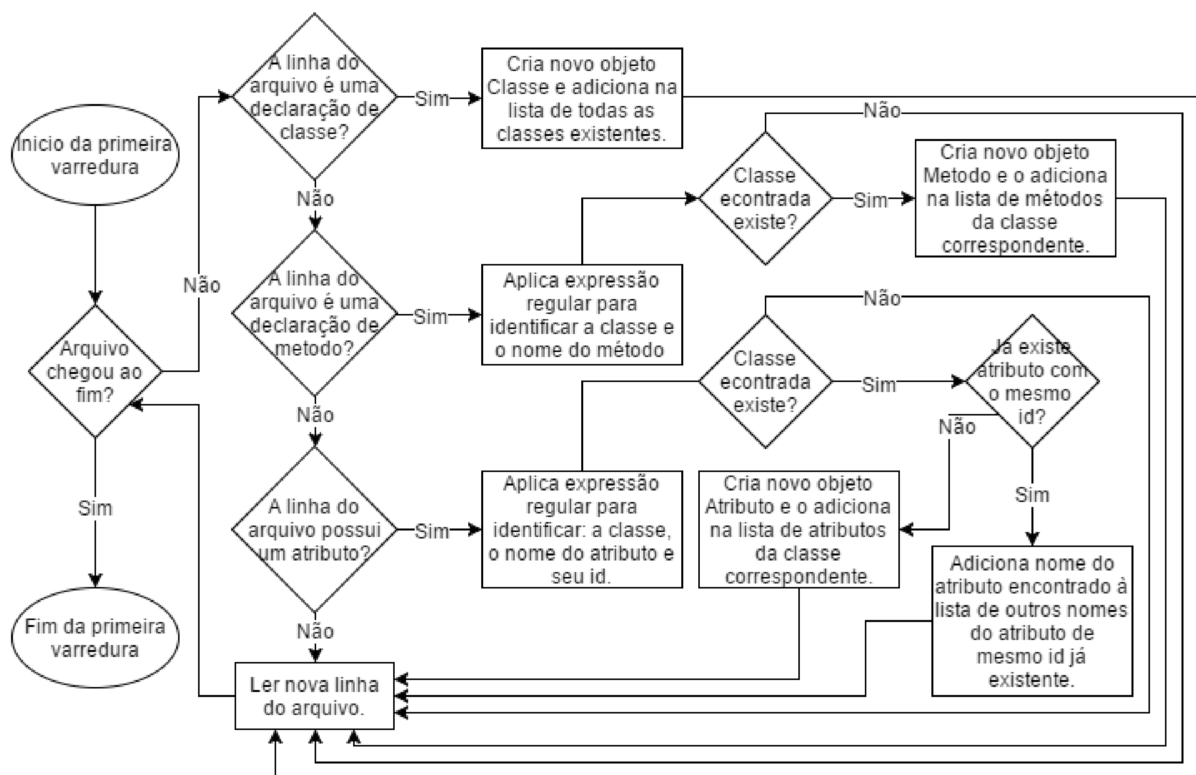
Fonte: O Autor

3.2.1 Primeira varredura

Na primeira varredura são lidas todas as classes, métodos e atributos. Embora a obtenção dos nomes das classes seja um pré-requisito para a identificação dos métodos e atributos, os três podem ser executados na mesma etapa, pois as declarações de classes estão no topo do arquivo de entrada.

O algoritmo referente à primeira varredura encontra-se representado na Figura 3.5. Para cada linha do arquivo de entrada são realizadas três comparações a fim de identificar se esta linha possui uma declaração de classe, uma declaração de método ou um atributo. Se a linha possuir uma declaração de classe, uma nova instância da classe “Classe” será criada e suas informações armazenadas. Se a linha possuir uma declaração de método, o nome da classe deste método será validado quanto ao nome das classes já existentes. Se a classe for válida, então um novo objeto de “Metodo” será criado e vinculado a sua respectiva classe. Caso contrário, o método encontrado é ignorado e a varredura continua, visto que este método não pertence a uma classe definida no arquivo de entrada.

Figura 3.5 – Representação do fluxo de execução – Primeira varredura



Fonte: O Autor

Se a linha de entrada for um carregamento de atributo, o nome de sua classe será validado, assim como é feito para novos métodos. A peculiaridade da identificação de novos atributos está na possibilidade de existência de atributos de mesmo nome ou com nomes temporários dependentes do escopo do código. Portanto, antes de criar um novo objeto “Atributo” é preciso verificar se um atributo com mesmo número de identificação (ID) já existe na classe. Se existir, o atributo encontrado é apenas outra menção a um atributo já

registrado e seu nome será incluso na lista de outros nomes do atributo já existente. Caso contrário, um novo objeto “Atributo” será criado e incluso na lista de atributos de sua respectiva classe.

Esta análise é feita para o arquivo de entrada inteiro, visto que as declarações de métodos e carregamentos de atributos encontram-se distribuídos ao longo do código. A complexidade desta operação é diretamente relacionada ao tamanho do arquivo de entrada. Quanto maior o arquivo de entrada, maior será o impacto no rendimento desta etapa do código.

Algumas expressões regulares são utilizadas no intuito de identificar os padrões presentes no arquivo de entrada. Estas expressões foram definidas através de análise do código. Na Figura 3.6, por exemplo, é possível observar o padrão de declaração de classes pertencentes ao código original. As classes importadas possuem um padrão similar, porém aparecem entre aspas.

Figura 3.6 – Classes no arquivo .ll

```
19 %class.Y = type { %class.X, i32 }
20 %class.X = type { %class.W, i32 }
21 %class.W = type { i32, i32 }
```

Fonte: O Autor

Figura 3.7 – Expressão regular utilizada na identificação de classes

```
regex find_class ("%class\\.[^\\.]* = type.*");
```

Fonte: O Autor

A expressão regular presente na Figura 3.7 foi utilizada na identificação do padrão das classes. O nome da classe encontra-se na primeira parte da linha, antes do símbolo de igualdade. Por exemplo, as classes declaradas na Figura 3.6 são as classes Y, X e W, respectivamente. Após o símbolo de igualdade tem-se a especificação da classe e suas características quanto a herança. No exemplo apresentado, a classe Y é herdeira da classe X e a classe X é herdeira da classe W. Porém essa identificação não é feita durante a primeira varredura, visto que a classe X ainda não foi declarada quando a classe Y declara-se filha de X. Portanto este passo faz parte da segunda varredura juntamente com a fase de identificação de herança múltipla.

Ainda na primeira interação, são lidos e validados os métodos pertencentes a cada classe. A fim de obter as informações necessárias, a análise de cada método foi feita com base na definição do mesmo. Na Figura 3.8 estão as expressões regulares utilizadas na identificação das linhas que possuem definições para métodos.

Figura 3.8 – Expressão regular utilizada na identificação de métodos

```
regex find_method ("^define linkonce_odr .* @_(.*)%class\\.(.*)");
regex find_method_void ("^define void @_(.*)%class\\.(.*)");
```

Fonte: O Autor

A Figura 3.9 exibe um exemplo do resultado da aplicação das expressões regulares disponíveis na Figura 3.8. Os métodos relevantes na criação das estruturas finais do protótipo são os que pertencem às classes identificadas anteriormente e o método “main”. No caso da Figura 3.9, os métodos das linhas 71, 83 e 203 serão ignorados na etapa de validação da estrutura encontrada. Os demais métodos serão inclusos na lista de métodos da classe a qual referenciam. Por exemplo, a classe Y possuirá os métodos _ZN1YC2Ev e _ZN1Y1mEv.

Figura 3.9 – Expressão regular aplicada na identificação de métodos

```
Linha 71: define internal void @__cxx_global_var_init() #0 {
Linha 83: define internal void @__dtor__ZStL8__ioinit() #0 {
Linha 93: define i32 @main() #3 {
Linha 110: define linkonce_odr void @_ZN1YC2Ev(%class.Y* %this) unnamed_addr #4 comdat align 2 {
Linha 123: define linkonce_odr void @_ZN1Y1mEv(%class.Y* %this) unnamed_addr #5 comdat align 2 {
Linha 134: define linkonce_odr void @_ZN1XC2Ev(%class.X* %this) unnamed_addr #4 comdat align 2 {
Linha 147: define linkonce_odr void @_ZN1X1nEv(%class.X* %this) unnamed_addr #3 comdat align 2 {
Linha 165: define linkonce_odr void @_ZN1WC2Ev(%class.W* %this) unnamed_addr #4 comdat align 2 {
Linha 176: define linkonce_odr void @_ZN1W1mEv(%class.W* %this) unnamed_addr #5 comdat align 2 {
Linha 187: define linkonce_odr void @_ZN1W1nEv(%class.W* %this) unnamed_addr #3 comdat align 2 {
Linha 203: define internal void @_GLOBAL__sub_I_SDA.cpp() #0 {
```

Fonte: O Autor

Embora seja possível detectar o nome exato da classe declarada no código a partir do arquivo LLVM, a situação não é equivalente quanto ao nome dos métodos. A nomenclatura dos métodos exibida no arquivo não é totalmente fiel ao nome do método pertencente ao código. Porém, é possível identificar o nome do método existente dentro do valor obtido no arquivo .ll. Por exemplo, considere uma classe chamada “veiculo_rodoviario” e o método “get_rodas” definido nesta classe. Considere ainda que o nome de método exibido na Figura 3.10 provém do arquivo .ll deste mesmo código. A partir da análise do nome exibido é possível deduzir que este método trata-se do mesmo método previamente mencionado e já

conhecido pelo usuário. Os arquivos de saída usam os nomes encontrados no arquivo .ll, mas o usuário pode relacioná-los facilmente ao seu código original.

Figura 3.10 – Exemplo de nome de método

_ZN18veiculo_rodoviario9get_rodasEv

Fonte: O Autor

Assim como os nomes das classes, é possível encontrar o nome do atributo exatamente como se encontra no código original. Entretanto, dentro de alguns contextos condicionais que fazem referência a um mesmo atributo, este pode receber variações em seu nome original. Contudo, a cada referência, ele mantém um indicador único que permite associá-lo com suas variações. Por exemplo, se o atributo “atr_a” é alterado dentro de um comando “if”, a linguagem intermediária poderá representar este mesmo atributo usando o nome “atr_a2”, mas continua referenciando-o através de seu identificador único, como é exemplificado ao longo desta seção.

Assim como no caso dos métodos, apenas os atributos utilizados aparecerão no arquivo .ll. Como os nomes de atributos encontram-se disponíveis em apenas uma parte específica do arquivo, a estratégia utilizada para a obtenção dos mesmos foi através da análise de cada carregamento dos mesmos que precede um uso ou uma definição dentro de métodos arbitrários. Através do estudo do comportamento da linguagem, foi possível detectar este padrão e aplicar a expressão regular presente na Figura 3.11 a fim de encontrar as informações desejadas. No padrão definido, cada linha possui o nome do atributo, a classe à qual pertence e seu identificador dentro desta classe.

Figura 3.11 – Expressão regular utilizada na identificação de atributos

```
regex find_attribute ("^  %(.*)getelementptr inbounds %class(.*),(.*),(.*),(.*)");
```

Fonte: O Autor

O resultado do uso do padrão apresentado na Figura 3.11 é exibido na Figura 3.12. Como mencionado anteriormente, cada atributo possui uma identificação única em relação aos demais dentro de uma mesma classe. Isso nos permite identificar que, no exemplo apresentado na Figura 3.12, o atributo v e o atributo v2 são os mesmos, pois ambos pertencem

à classe W e fazem referência ao atributo que possui o identificador 2 dentro desta classe. O identificador é o último número presente na linha encontrada. No exemplo da Figura 3.13, embora os identificadores sejam os mesmos para ambos os atributos v, sabemos que não se trata do mesmo atributo, pois um deles pertence à classe Y e o outro à classe W.

Figura 3.12 – Expressão regular aplicada na identificação de atributos

```
Linha 128:   %w = getelementptr inbounds %class.Y, %class.Y* %this1, i32 0, i32 1
Linha 153:   %v = getelementptr inbounds %class.W, %class.W* %0, i32 0, i32 2
Linha 155:   %x = getelementptr inbounds %class.X, %class.X* %this1, i32 0, i32 1
Linha 158:   %v2 = getelementptr inbounds %class.W, %class.W* %2, i32 0, i32 2
```

Fonte: O Autor

Figura 3.13 – Expressão regular aplicada na identificação de atributos (2)

```
Linha 70:   %v = getelementptr inbounds %class.Y, %class.Y* %this1, i32 0, i32 1
Linha 82:   %v = getelementptr inbounds %class.W, %class.W* %0, i32 0, i32 1
```

Fonte: O Autor

Supondo que uma classe Y possua os atributos v, v1 e v2, a distinção entre eles ocorrerá unicamente a partir de seu identificador único, como apresentado na Figura 3.14. É possivel notar que v1 não se refere a v, pois seu identificador é 3, e o identificacor de v é 1. O mesmo pode ser observado para v2, visto que seu identificador é 2.

Figura 3.14 – Expressão regular aplicada na identificação de atributos (3)

```
Linha 65:   %v1 = getelementptr inbounds %class.Y, %class.Y* %obj_y, i32 0, i32 3
Linha 88:   %v = getelementptr inbounds %class.Y, %class.Y* %this1, i32 0, i32 1
Linha 112:   %v1 = getelementptr inbounds %class.Y, %class.Y* %this1, i32 0, i32 3
Linha 123:   %v2 = getelementptr inbounds %class.Y, %class.Y* %this1, i32 0, i32 2
```

Fonte: O Autor

Cada atributo identificado é inserido na lista de atributos de sua respectiva classe. Quando um atributo aparece no código com um nome diferente, como foi o caso de v na Figura 3.12, este nome é incluso em uma lista de outros nomes dentro da estrutura atributo.

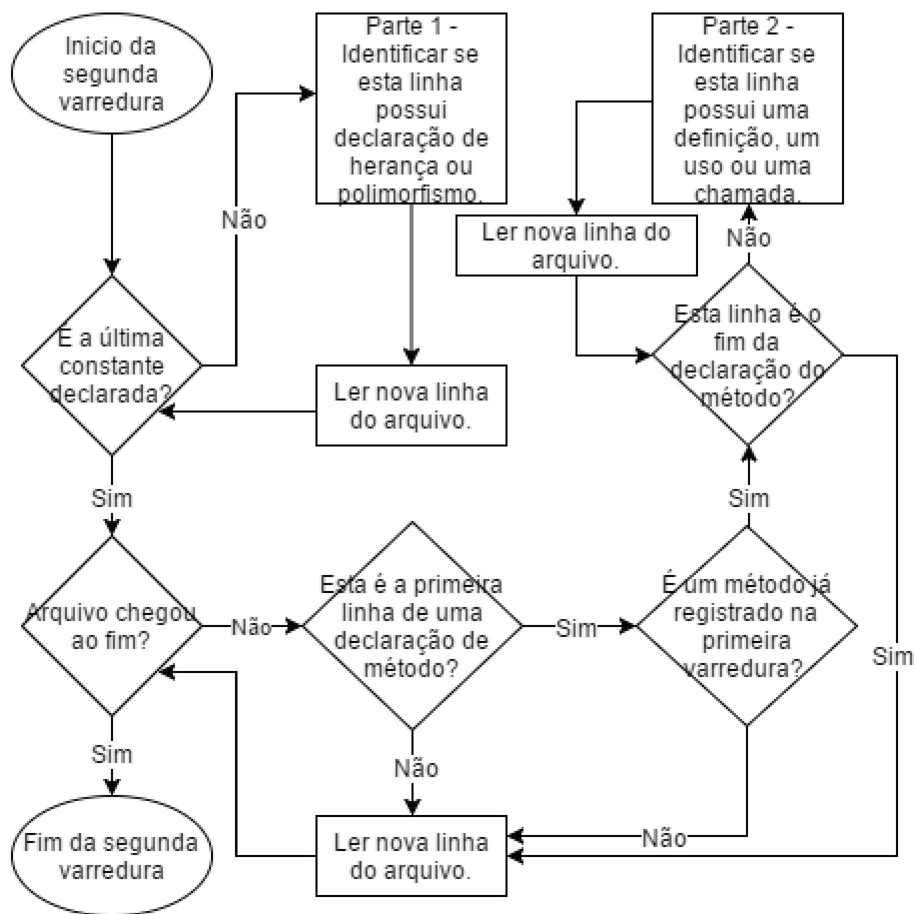
3.2.2 Segunda varredura

Na segunda varredura são identificadas as relações de herança, a tabela de polimorfismo, as declarações de objetos, as definições e usos de atributos e as chamadas feitas pelos métodos. Esta varredura é feita em duas partes, tendo em vista que as informações

necessárias para a identificação de heranças e da tabela virtual estarão contidas nas linhas que precedem as declarações de métodos. Portanto na primeira parte serão lidas as estruturas de herança e polimorfismo e na segunda parte serão lidas as declarações de objetos, as definições, os usos e as chamadas que ocorrem dentro dos métodos.

A Figura 3.15 apresenta uma visão geral dos passos presentes na segunda varredura. Assim como a primeira, cada linha do arquivo será analisada separadamente e a complexidade é dependente do tamanho do arquivo de entrada. A primeira parte da segunda varredura é uma leitura das constantes declaradas. Esta parte chega ao fim quando a última constante é declarada. A partir deste ponto do arquivo de entrada, a etapa de declaração de métodos começa. A segunda parte da segunda varredura analisa o escopo de cada método já registrado. Os métodos que não foram identificados na primeira varredura serão ignorados nesta etapa.

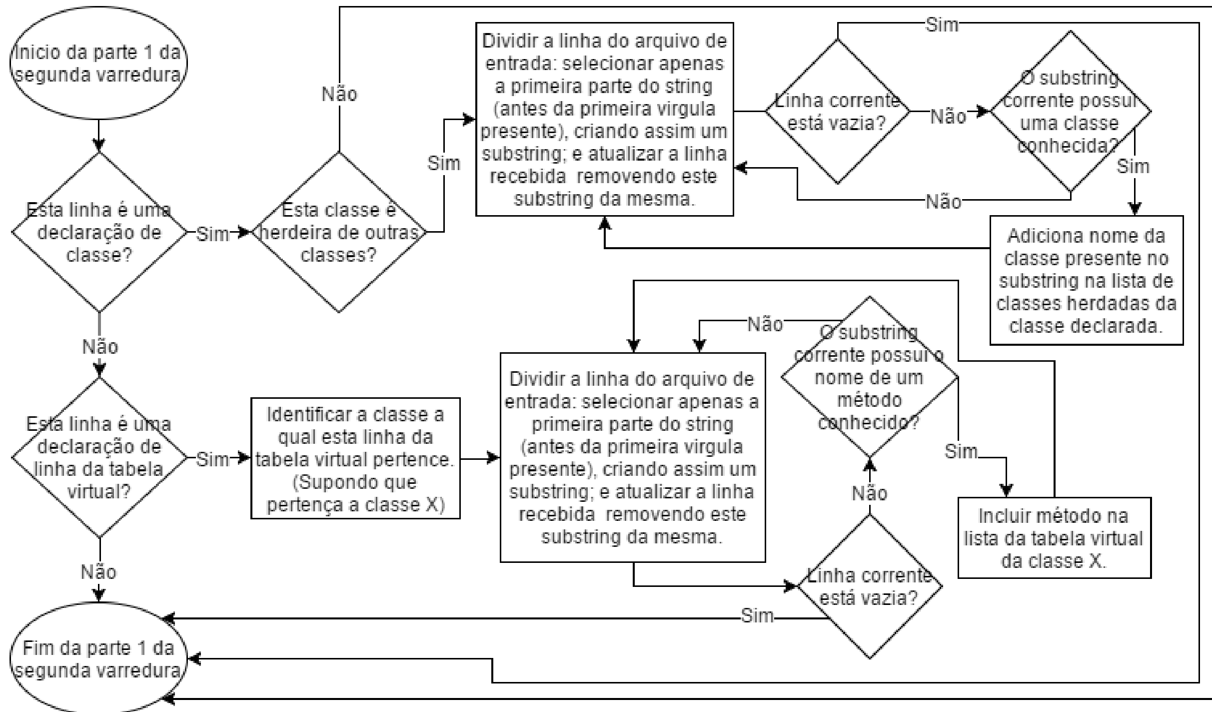
Figura 3.15 – Representação do fluxo de execução – Segunda varredura



Fonte: O Autor

No bloco identificado como “Parte 1” na Figura 3.15 ocorre a análise das linhas do arquivo de entrada quanto a declaração de heranças e definições de linhas da tabela virtual. A Figura 3.16 apresenta o fluxo de execução desta etapa.

Figura 3.16 – Representação do fluxo de execução – Segunda varredura – Parte 1



Fonte: O Autor

A identificação de herança encontra-se nesta iteração devido à necessidade de que todas as classes já estejam registradas para que as mesmas possam ser validadas antes que sejam incluídas na lista de classes herdadas. Quando uma classe é descendente de outra classe, esta informação está disponível juntamente com a declaração da classe. A partir da Figura 3.16, é possível notar que se uma declaração de classe for encontrada e se esta declaração possui herança, a linha de entrada será dividida em *n* partes sendo *n* o número de classes ancestrais declaradas. Cada classe ancestral identificada será validada. Se a classe herdada existir, então esta será inclusa na lista de classes herdadas pela classe declarada. Caso a classe não exista, a relação de herança não é registrada para esta classe.

Por exemplo, supondo que o código apresentado na Figura 3.17 seja a entrada de uma execução, o protótipo desenvolvido identificará uma herança múltipla para a classe “caminhao”, que será descendente de “veiculo_rodoviario”, “veiculo_A” e “veiculo_B”, e assim por diante para as demais classes. As classes que não possuem outras classes em sua

especificação não herdam de outras classes, como é o caso de “veiculo_A”, “veiculo_B” e “veiculo_C”.

Figura 3.17 – Exemplo de herança

```
%class.caminhao = type { %class.veiculo_rodoviario, %class.veiculo_A, %class.veiculo_B, i32 }
%class.veiculo_rodoviario = type { %class.veiculo_C, i32, i32 }
%class.veiculo_C = type { i32 }
%class.veiculo_A = type { i32 }
%class.veiculo_B = type { i32 }
%class.automovel = type { %class.veiculo_rodoviario, i32 }
```

Fonte: O Autor

Cada classe possui uma lista de classes herdadas. Toda vez que um relacionamento de herança é encontrado para uma classe X, a classe herdada será validada e incluída como classe herdeira na lista de X.

Quando a linha corrente não é uma declaração de classe, verifica-se se a mesma possui uma declaração de linha da tabela virtual. A parte inferior da Figura 3.16 mostra os passos de análise e execução desta etapa. Para cada declaração de linha da tabela virtual encontrada, os métodos pertencentes às mesmas serão validados e incluídos na estrutura da respectiva classe.

A tabela de polimorfismo é construída a partir das linhas da tabela que são declaradas no inicio do documento. Sempre que uma chamada polimórfica é feita, está chamada faz referência estaticamente a uma posição específica da tabela virtual, o tipo do objeto que aciona a chamada será o responsável pela especificação de qual método da posição da tabela indicada deverá ser chamado.

No inicio do arquivo .ll existem algumas linhas que podem ser identificadas a partir do padrão “@_ZTV[0-9]<Nome da classe>”. Estas linhas definem a tabela virtual para um objeto do tipo da classe contida no campo nome da classe. Na segunda varredura estas tabelas são salvas na estrutura da classe a qual pertence. É necessário que este passo esteja presente apenas na segunda varredura devido à necessidade de validação dos métodos presentes na tabela. Após a varredura, cada classe que faz parte de uma estrutura hierárquica que faz uso de polimorfismo terá uma linha da tabela salva em sua estrutura. A linha armazenada na estrutura da classe é exemplificada na Figura 3.20, onde cada linha da tabela virtual pertence a uma classe diferente.

As linhas são declaradas no arquivo LLVM como representado na Figura 3.18. Cada linha possui um número arbitrário de métodos e seu posicionamento na sequência em que é exibido é o que define sua posição na tabela virtual. No exemplo apresentado, são exibidas três linhas de declaração, uma para a classe A, uma para B e uma para C. Cada linha possui

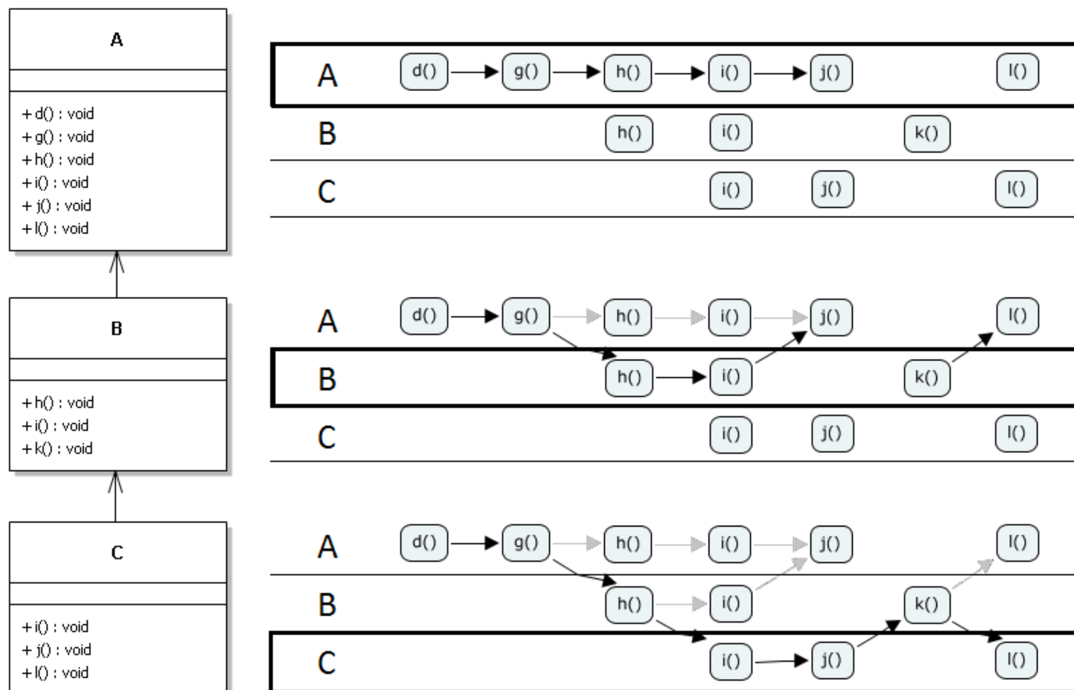
quatro métodos conhecidos. Este exemplo é a implementação de uma adaptação do exemplo de grafo yo-yo apresentado no Capítulo 2, uma releitura do exemplo dado por Amman;Offut (2008) a fim de torná-lo mais simples e proporcionar mais clareza aos pontos desejados neste capitulo. Os relacionamentos entre as classes A, B e C são exibidos na Figura 3.19.

Figura 3.18 – Declaração de tabela virtual

```
2  @_ZTV1A = linkonce_odr unnamed_addr constant [6 x i8*] [i8* null, i8*
   bitcast ({ i8*, i8* }* @_ZTI1A to i8*), i8* bitcast (void (%class.A*)*
   @_ZN1A1hEv to i8*), i8* bitcast (void (%class.A*)* @_ZN1A1iEv to i8*), i8*
   bitcast (i32 (%class.A*)* @_ZN1A1jEv to i8*), i8* bitcast (i32 (%class.A*)*
   @_ZN1A1lEv to i8*)], comdat, align 8
3
4  @_ZTV1B = linkonce_odr unnamed_addr constant [6 x i8*] [i8* null, i8*
   bitcast ({ i8*, i8*, i8* }* @_ZTI1B to i8*), i8* bitcast (void (%class.B*)*
   @_ZN1B1hEv to i8*), i8* bitcast (void (%class.B*)* @_ZN1B1iEv to i8*), i8*
   bitcast (i32 (%class.A*)* @_ZN1A1jEv to i8*), i8* bitcast (i32 (%class.A*)*
   @_ZN1A1lEv to i8*)], comdat, align 8
5
6  @_ZTV1C = linkonce_odr unnamed_addr constant [6 x i8*] [i8* null, i8*
   bitcast ({ i8*, i8*, i8* }* @_ZTI1C to i8*), i8* bitcast (void (%class.B*)*
   @_ZN1B1hEv to i8*), i8* bitcast (void (%class.C*)* @_ZN1C1iEv to i8*), i8*
   bitcast (i32 (%class.C*)* @_ZN1C1jEv to i8*), i8* bitcast (i32 (%class.C*)*
   @_ZN1C1lEv to i8*)], comdat, align 8
```

Fonte: O Autor

Figura 3.19 – Exemplo de grafo Yo-Yo



Fonte: O Autor

Com base nos dados obtidos na Figura 3.18, a tabela virtual da Figura 3.20 é criada. Como pode ser observado através da comparação da tabela gerada com a Figura 3.19, cada coluna da tabela é um método que pode ser sobrescrito, dependendo do caminho tomado pelo fluxo de execução. A tabela gerada pelo protótipo possui a informação de qual linha deve ser executada dependendo do tipo do objeto. Por exemplo, se o método d() é chamado no contexto de um objeto da classe B, a tabela indica qual versão do método h() será chamada. Neste exemplo, o h() de B deve ser executado ao invés do h() de A. Cada coluna da tabela corresponde a um conjunto de métodos polimórficos entre si. Por exemplo, a coluna 1 diz respeito ao método i(). Assim, na definição de todos os métodos h() no código LLVM, haverá uma referência para a coluna 1 da tabela de polimorfismo, visto que, a partir do esquema da Figura 3.19, o método h() deve chamar o método i().

Figura 3.20 – Exemplo de tabela virtual extraída

	0	1	2	3
A	_ZN1A1hEv	_ZN1A1iEv	_ZN1A1jEv	_ZN1A1lEv
B	_ZN1B1hEv	_ZN1B1iEv	_ZN1A1jEv	_ZN1A1lEv
C	_ZN1B1hEv	_ZN1C1iEv	_ZN1C1jEv	_ZN1C1lEv

Fonte: O Autor

A Figura 3.21 exemplifica a chamada polimórfica feita para a posição 1 da tabela virtual pelo método h() da classe B. O método carrega o dado da tabela virtual da coluna 1, representado pelo trecho destacado “%vtable, i64 1”, e escreve na memória da variável temporária %1 e, em seguida, efetua a chamada para o método necessário, o que será decidido em tempo de execução.

Figura 3.21 – Exemplo de chamada polimórfica

```
; Function Attrs: uwtable
define linkonce_odr void @_ZN1B1hEv(%class.B* %this) unnamed_addr #3 comdat align 2 {
entry:
  %this.addr = alloca %class.B*, align 8
  store %class.B* %this, %class.B** %this.addr, align 8
  %this1 = load %class.B*, %class.B** %this.addr
  %call = call dereferenceable(272) %"class.std::basic_ostream"*
  @_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc(%"class.std::basic_ostream"*
  @_ZSt4cout, i8* getelementptr inbounds ([5 x i8], [5 x i8]* @.str.7, i32 0, i32 0))
  %0 = bitcast %class.B* %this1 to void (%class.B*)***
  %vtable = load void (%class.B*)**, void (%class.B*)*** %0
  %vfn = getelementptr inbounds void (%class.B*)*, void (%class.B*)** %vtable, i64 1
  %1 = load void (%class.B*)*, void (%class.B*)** %vfn
  call void %1(%class.B* %this1)
  ret void
}
```

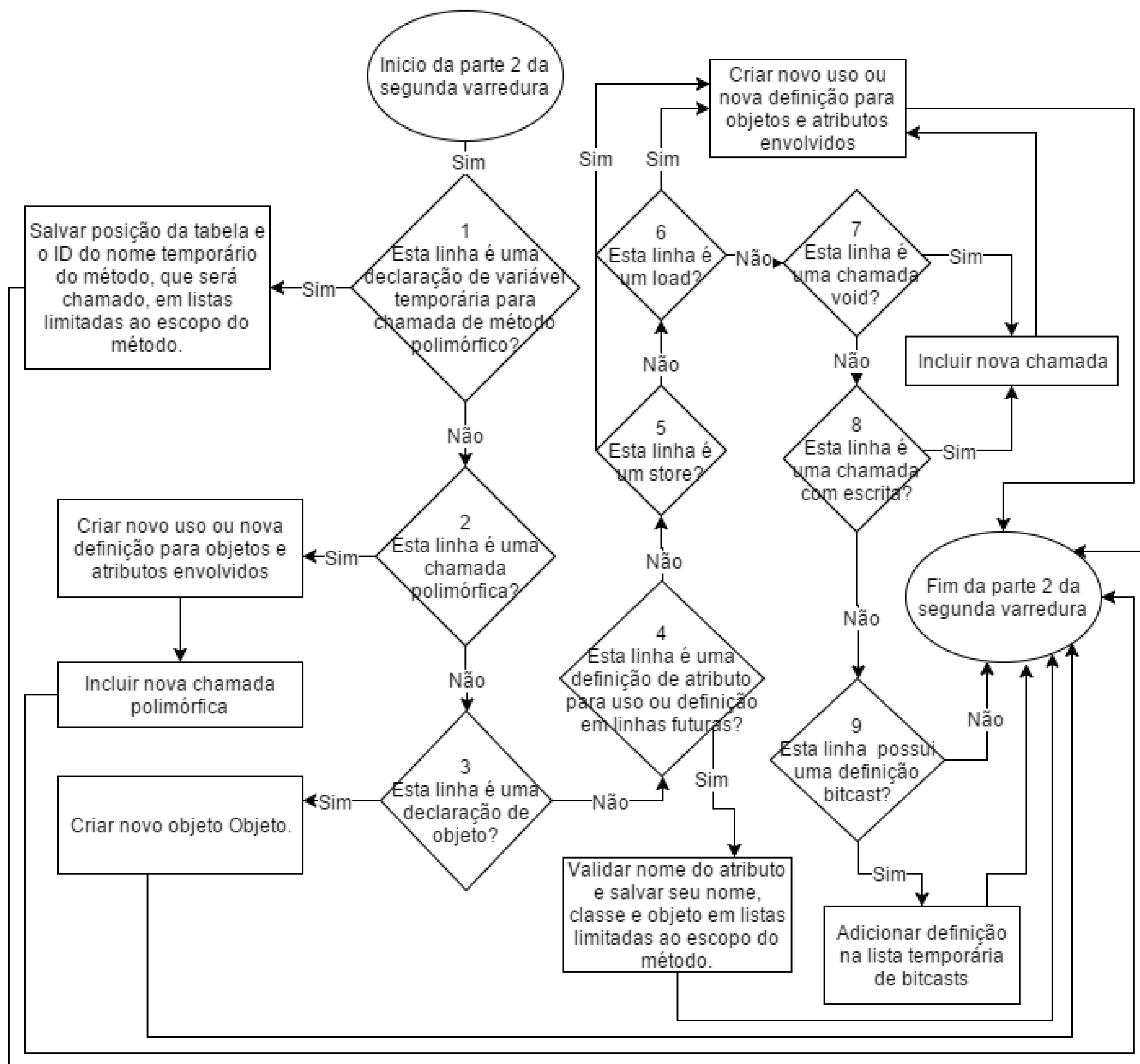
Fonte: O Autor

As informações referentes à tabela virtual lida serão exibidas no arquivo de saída “Tabela de Polimorfismo.txt” a fim de dar visibilidade às estruturas geradas. Esta tabela é utilizada para a análise de possíveis anomalias durante a execução do protótipo.

Ainda na segunda varredura há o bloco identificado como “Parte 2” na Figura 3.15, nele ocorre a análise das linhas do arquivo de entrada quanto aos relacionamentos interclasse relevantes que ocorrem dentro do escopo dos métodos, tais como: declaração de objetos, definições, usos e chamadas. Este é o passo que demanda maior tempo de execução, pois requer a criação e validação de diversas novas estruturas. A Figura 3.22 apresenta o fluxo de execução desta etapa.

Cada linha pertencente ao escopo de um método será submetida à análise apresentada na Figura 3.22. Esta análise possui nove passos, ou seja, a mesma linha do arquivo pode ser enquadrada em um de nove diferentes casos ou nenhum. Cada caso é uma peculiaridade do método que será registrada.

Figura 3.22 – Representação do fluxo de execução – Segunda varredura – Parte 2



Fonte: O Autor

O primeiro caso analisa se a linha corrente possui uma definição de variável temporária (%vfn) para o carregamento de uma posição da tabela virtual. A identificação de chamadas polimórficas é mais complexa, pois requer a análise de mais de uma linha de declaração. Portanto este primeiro caso existe no intuito de armazenar as estruturas temporárias necessárias na identificação de futuras chamadas polimórficas. A linha identificada no primeiro caso acontece antes de chamadas polimórficas e é esta a linha que possui a informação de qual coluna da tabela virtual será chamada. A Figura 3.23 apresenta as quatro linhas envolvidas em uma chamada polimórfica. A linha identificada neste primeiro caso apresenta o mesmo padrão da linha 218. No exemplo exibido, a variável “%vfn” é criada para a posição 2 da tabela virtual e a próxima linha (219) define um nome temporário de

método (%1), que será chamado nas próximas linhas, não necessariamente logo após sua definição. Como os padrões das linhas 218 e 219 são exibidos juntos, sempre que a linha corrente segue este padrão, a próxima linha será lida dentro deste mesmo caso a fim de registrar qual o nome de método temporário definido para esta posição da tabela virtual. Estas informações são armazenadas em listas temporárias. Toda vez que o padrão de chamada polimórfica é identificado em outro caso, estas listas temporárias são percorridas para a análise de qual coluna da tabela virtual esta vinculada com a chamada polimórfica feita. A expressão regular utilizada na identificação do primeiro caso encontra-se disponível na Figura 3.24.

Figura 3.23 – Exemplo de chamada polimórfica (2)

```
217  %vtable = load i32 (%class.A*)**, i32 (%class.A*)*** %0
218  %vfn = getelementptr inbounds i32 (%class.A*)*, i32 (%class.A*)** %vtable, i64 2
219  %1 = load i32 (%class.A*)*, i32 (%class.A*)** %vfn
220  %call2 = call i32 %1(%class.A* %this1)
```

Fonte: O Autor

Figura 3.24 – Expressão regular utilizada na identificação do primeiro caso

```
regex find_vtable_prev_call ("%vfn[0-9]* = getelementptr inbounds (.*)");
```

Fonte: O Autor

O segundo caso é referente às chamadas polimórficas. A Figura 3.23 exemplifica uma parte do arquivo LLVM no qual há uma chamada polimórfica. O padrão identificado neste caso será semelhante ao apresentado na linha 220. As expressões regulares exibidas na Figura 3.25 são utilizadas na identificação desta linha e, dependendo de expressão na qual se enquadram, poderá ser incluído um uso ou uma definição para os atributos ou objetos envolvidos na chamada. Um exemplo da aplicação das expressões regulares definidas pode ser encontrado na Figura 3.26.

As características da chamada polimórfica identificada são encontradas através da análise das listas criadas no primeiro e nono casos. As listas criadas no primeiro caso possuem os nomes de métodos temporários e as posições chamadas. A chamada identificada possuirá o nome do método temporário, que será associado com as listas temporárias a fim de identificar a qual posição da tabela virtual esta chamada faz referência. O nono caso armazena variáveis temporárias para objetos que serão usados nestas chamadas e devem ser associados com a chamada polimórfica a fim de identificar se algum destes objetos está sendo usado.

Assim que todas as informações necessárias são obtidas, a chamada polimórfica é incluída na lista de chamadas do método.

Figura 3.25 – Expressões regulares utilizadas na identificação de chamadas polimórficas

```
Definição + Chamada Polimórfica
regex find_poli_call_call ("^ *%call[0-9]* = call.* %[0-9]*\\((.*)%[^ ]* %[^ ]*(.*)\\)");
regex find_poli_call_invoke ("^ *%call[0-9]* = invoke.* %[0-9]*\\((.*)%[^ ]* %[^ ]*(.*)\\)");

Uso + Chamada Polimórfica
regex find_poli_call_void ("^ *call void %[0-9]*\\((.*)%[^ ]* %[^ ]*(.*)\\)");
regex find_poli_invoke_void ("^ *invoke void %[0-9]*\\((.*)%[^ ]* %[^ ]*(.*)\\)");
```

Fonte: O Autor

Figura 3.26 – Expressão regular aplicada na identificação de chamadas polimórficas

```
Definição + Chamada Polimórfica
%call2 = call i32 %2(%class.A* %0)

Uso + Chamada Polimórfica
call void %1(%class.B* %this1)
invoke void %5(%class.ComCordas* %this1, i32 0)
```

Fonte: O Autor

O terceiro caso identifica se a linha corrente é uma declaração de objeto. A identificação de objetos é feita através do uso da expressão regular apresentada na Figura 3.27. O nome do objeto é seguido por sua classe, como exemplificado na Figura 3.28. Assim que identificado, a classe do objeto é validada e o mesmo é incluído na lista de objetos do método.

Figura 3.27 – Expressão regular utilizada na identificação de objetos

```
regex find_alloca ("%[^ ]* = alloca %class\\.[^ ]*,(.*)");
```

Fonte: O Autor

Figura 3.28 – Expressão regular aplicada na identificação de objetos

```
Linha 126:   %obj_a = alloca %class.A, align 8
Linha 127:   %obj_b = alloca %class.B, align 8
Linha 128:   %obj_c = alloca %class.C, align 8
```

Fonte: O Autor

O quarto caso tem por objetivo a identificação dos atributos carregados em memória. Toda vez que um atributo é carregado em memória significa que um uso ou uma definição pode ocorrer nos próximos passos do método. Portanto cada carregamento de atributo é registrado em uma lista de futuros possíveis usos ou definições. Esta estrutura facilita a identificação destes atributos dentro de chamadas, que tendem a ser longas cadeias de caracteres. A expressão regular utilizada neste caso é a mesma utilizada na primeira varredura para a identificação dos atributos existentes no código.

O quinto caso visa identificar casos de "*store*" (escritas na memória) que fazem referência a atributos de classes. Quando identificado este padrão, uma definição deverá ser criada para a respectiva linha do código. O sexto caso identifica casos de "*load*" (carregamento de dados da memória) que fazem referência a atributos de classes. Os casos de "*load*" são considerados usos e devem ser registrados. Os casos cinco e seis possuem computação semelhante, diferindo apenas no caractere que identifica o tipo do objeto "Def_ou_Uso".

O sétimo caso identifica chamadas "void", ou seja, chamadas que não alteram os atributos envolvidos. Toda vez que uma chamada "void" é identificada, um uso deverá ser incluso para cada atributo/objeto envolvido e um objeto da classe "Chamada" também será criado. O oitavo caso identifica chamadas que possuem escritas, ou seja, que podem alterar os atributos envolvidos. Para cada chamada com escrita, é criada uma definição para o objeto envolvido e um novo objeto do tipo "Chamada".

Os casos cinco, seis, sete e oito compartilham estruturas lógicas, visto que todos acarretam na criação de definições ou usos. Com a aplicação das expressões regulares presentes na Figura 3.29, obtemos as definições, usos e chamadas presentes dentro do método. Assim que uma linha do código for compatível com algum destes padrões, o nome do atributo e/ou objeto envolvidos são validados e registrados no método. A fase de validação é de grande importância nesta etapa, pois é através dela que se pode eliminar as chamadas feitas a métodos que não pertencem ao código original ou que não referenciam objetos ou atributos conhecidos. A validação dos atributos e objetos acontece a partir da utilização das estruturas temporárias geradas nos passos quatro e nove, que definem listas de atributos e objetos, respectivamente, já carregados no escopo do método e passíveis de uso ou definição. No caso dos objetos não só a lista gerada no nono caso pode ser utilizada para validá-los, mas como também a lista de todos os objetos já definidos do escopo do método, que é gerada no terceiro caso.

A expressão regular utilizada no quinto caso é representada pelo identificador 1 da Figura 3.29. As expressões regulares das linhas identificadas por 2 e 3 são as utilizadas no oitavo caso. A expressão regular utilizada no sexto caso é representada com o identificador 4. E as expressões identificadas com os números 5 e 6 são usadas na identificação do sétimo caso.

Figura 3.29 – Expressões regulares utilizada na identificação de definições, usos e chamadas

```
  Definição
1 regex find_store (".*store.*");

  Definição + Chamada
2 regex find_call_call (".*%call[0-9]* = call.*");
3 regex find_call_invoke (".*%call[0-9]* = invoke.*");

  Uso
4 regex find_load (".*load.*");

  Uso + Chamada
5 regex find_call_void (".*call void.*");
6 regex find_invoke_void (".*invoke void.*");
```

Fonte: O Autor

A Figura 3.30 apresenta exemplos identificados pelas expressões regulares descritas na Figura 3.29.

Figura 3.30 – Expressão regular aplicada na identificação de definições, usos e chamadas

```
Definição
store %class.Def_ou_Uso* %this, %class.Def_ou_Uso** %this.addr, align 8

Definição + Chamada
%call472 = call i64 @_ZNKSt6vectorI6ClasseSaIS0_EE4sizeEv(%"class.std::vector.13"* %all_classes)
%call151 = invoke zeroext i1 @_ZNSt14basic_if(...)(%"class.std::basic_ifstream"* %arquivo)

Uso
%this1 = load %class.Atributo*, %class.Atributo** %this.addr

Uso + Chamada
call void @_ZNSt6vectorI6ClasseSaIS0_EE5clearEv(%"class.std::vector.13"* %all_classes) #2
invoke void @_ZN6MetodoC2Ev(%class.Metodo* %main_method)
```

Fonte: O Autor

O nono caso identifica se a linha analisada é uma definição temporária de objeto. Quando um objeto ocasiona uma chamada hierárquica, um *bitcast* é feito neste objeto no intuito de projetá-lo a um novo tipo a fim de acarretar a chamada. Na Figura 3.31 é possível

notar que um *bitcast* é feito na criação de um objeto temporário “%0” que na realidade é o objeto “%obj_b”, que pertence a classe B, sendo projetado para o tipo da classe A a fim de fazer uma chamada para o método @_ZN1A1dEv, pertencente à classe A. Cada bitcast encontrado é armazenado em uma lista e analisado quando novas definições, usos, chamadas e chamadas polimórficas são identificadas. A Figura 3.32 possui a expressão regular utilizada na identificação de *bitcasts*.

Figura 3.31 – Exemplo de bitcast

```
136  %0 = bitcast %class.B* %obj_b to %class.A*
137  call void @_ZN1A1dEv(%class.A* %0)
```

Fonte: O Autor

Figura 3.32 – Expressão regular utilizada na identificação de bitcasts

```
regex find_bitcast ("^( *)%[0-9]* = bitcast %class\\.[^ ]* %[^ ]* to %class\\.[^ ]*");
```

Fonte: O Autor

3.3 Arquivos gerados

Como citado anteriormente, a execução do protótipo resulta na criação de diversos arquivos, cada um com informações específicas sobre um grupo de estruturas geradas.

3.3.1 Arquivo Classes.txt

O arquivo Classes.txt contém todas as classes, seus atributos e métodos contidos no código LLVM. Para cada método exibido, também são exibidos os objetos declarados em seu escopo e todas as definições, usos, chamadas e chamadas polimórficas feitas, seguidas de suas respectivas ordens de aparição. Ao fim deste arquivo encontram-se as informações do método main.

Na Figura 3.33 encontra-se um exemplo de classe dentro do arquivo Classes.txt. O nome da Classe encontra-se no canto superior esquerdo. Neste caso, esta é a classe Y, que herda da classe X, como é possível observar no topo do arquivo. Logo abaixo do nome da classe encontram-se os atributos da classe. Neste caso temos que w é um atributo da classe Y. Após os atributos os métodos são exibidos. Cada método possui uma lista de objetos que são declarados dentro de seu escopo, uma lista de definições, uma lista de usos e uma lista de chamadas. Como o foco é o teste interclasses, as definições e usos locais aos métodos que não envolvem atributos ou objetos não foram considerados.

Na estrutura apresentada na Figura 3.33, a existência de um objeto no escopo do método é representada pelo nome do objeto e pela classe a qual pertence. Neste caso, o método _ZN1Y1mEv possui um objeto declarado em seu escopo de nome “this_value”, que é um objeto da classe Y.

Figura 3.33 – Exemplo do arquivo de saída Classes.txt

```
Classe: Y    Filho de: X
    Atributos:
        w
    Metodos:
        _ZN1YC2Ev
            Objetos:
                this_value      Objeto_classe: Y
            Definições:
            Usos:
            Chamadas:
                Tipo: Chamada -  Metodo_Chamado: _ZN1XC2Ev   Metodo_Chamado_Classe: X ...
                                    ...Objeto_nome: this_value      Objeto_classe: Y    Ordem: 6
        _ZN1Y1mEv
            Objetos:
                this_value      Objeto_classe: Y
            Definições:
                Tipo: Definição -  Atributo: w   Objeto_nome: this_value  Objeto_classe: Y   Ordem: 6
            Usos:
            Chamadas:
```

Fonte: O Autor

As definições, usos, chamadas e chamadas polimórficas possuem um identificador no inicio da linha de exibição, chamado “tipo”. As definições e usos exibem o nome do atributo definido ou usado, o nome do objeto envolvido, a classe deste objeto e o número que representa o número da linha do código em que a ocorrência aparece dentro do escopo do método (chamado de “ordem”). As chamadas exibem o método chamado, seguido por sua classe, o objeto envolvido na chamada, a classe deste objeto e a ordem de aparição. Por exemplo, na Figura 3.33 é possível observar que existe uma chamada do método _ZN1YC2Ev para o método _ZN1XC2Ev na sexta linha dentro do escopo do método _ZN1YC2Ev.

As chamadas polimórficas exibem: a posição na tabela de polimorfismo para a qual ela aponta, o nome a e classe do objeto envolvido e a ordem de aparição no escopo do método. No exemplo apresentado na Figura 3.34 é possível notar que o método _ZN1B1iEv possui uma chamada polimórfica para a segunda coluna da tabela virtual, utilizando o objeto “this_value”. A classe do objeto, neste caso, é variável, visto que seu tipo é dinâmico e depende do contexto. A classe B aparece como classe do objeto pelo fato do mesmo ter sido declarado como um objeto deste tipo, porém este objeto pode ser projetado para outro tipo ao

longo do escopo do método. A chamada polimórfica exibida acontece na décima linha dentro do escopo do método.

Figura 3.34 – Exemplo do arquivo de saída Classes.txt (2)

```
_ZN1B1iEv
    Objetos:
        this_value      Objeto_classe: B
    Definições:
        Tipo: Definição -       Objeto_nome: this_value     Objeto_classe: B    Ordem: 10
    Usos:
    Chamadas:
        Tipo: Chamada Polimórfica -  Posicao_na_tabela_de_polimorfismo: 2  ...
                                         ...Objeto_nome: this_value  Objeto_classe: B   Ordem: 10
```

Fonte: O Autor

O arquivo Classes.txt foi criado no intuito de trazer visibilidade para a estrutura interna gerada durante a execução e possibilitar que diferentes análises de código possam ser feitas em cima destes dados, independente da atuação do programa.

3.3.2 Arquivo Tabela de Polimorfismo.txt

Este arquivo exibe a tabela de polimorfismo gerada durante a execução do protótipo. Com base na tabela é possível analisar quais métodos são polimórficos entre si e qual método será chamado quando o objeto de um tipo X for acionado. O intuito da criação deste arquivo foi trazer visibilidade para estes relacionamentos polimórficos independente do restante da computação do protótipo. A Figura 3.35 exemplifica o conteúdo do arquivo, cada linha é referente a uma classe e cada coluna é um conjunto de métodos polimórficos.

Figura 3.35 – Exemplo do arquivo de saída Tabela de Polimorfismo.txt

	0	1	2	3
A	_ZN1A1hEv	_ZN1A1iEv	_ZN1A1jEv	_ZN1A1lEv
B	_ZN1B1hEv	_ZN1B1iEv	_ZN1A1jEv	_ZN1A1lEv
C	_ZN1B1hEv	_ZN1C1iEv	_ZN1C1jEv	_ZN1C1lEv
D	_ZN1D1hEv	_ZN1D1iEv	_ZN1D1jEv	_ZN1D1lEv
E	_ZN1E1hEv	_ZN1E1iEv	_ZN1D1jEv	_ZN1D1lEv
F	_ZN1E1hEv	_ZN1F1iEv	_ZN1F1jEv	_ZN1F1lEv

Fonte: O Autor

3.3.3 Arquivo Fluxo.txt

O arquivo Fluxo.txt simula o fluxo de execução a partir de todos os métodos chamados pelo *main*, supondo que todos tenham sido chamados e levando em conta o tipo de

objeto que acarretou a chamada. Os métodos são exibidos na ordem em que aparecem no código LLVM. Este arquivo foi criado a fim de mostrar ao usuário os relacionamentos de chamada que estão acontecendo a partir do *main* no estado atual. Este arquivo existe para facilitar ao usuário entender as correlações entre os métodos e suas chamadas entre si. A saída gerada pode ser usada ainda para análise de cobertura para um dado conjunto de testes.

O algoritmo usado na geração deste arquivo não analisa condicionais ou loops, ele apenas detecta que o método atual esta chamando outros métodos e os exibe recursivamente. Visto que não há detecção de loops, existe um limite para o número de chamadas em profundidade, que pode ser editado de acordo com a aplicação. No momento limita-se a profundidade a 50 métodos chamados através de recursão. Não há limite para métodos chamados sequencialmente.

A Figura 3.36 exibe um exemplo de arquivo Fluxo.txt. Neste código, o método main chama três outros métodos, _ZN1YC2Ev, _ZN1Y1mEv e _ZN1X1nEv. A classe dos métodos chamados encontra-se entre parênteses ao lado do nome do método. O método ZN1YC2Ev chama _ZN1XC2Ev, e _ZN1XC2Ev chama _ZN1WC2Ev. Este fluxo não significa que todos os métodos que o main pode chamar, serão chamados de fato, mas se forem, os mesmos deverão desencadear as chamadas exibidas.

Figura 3.36 – Exemplo do arquivo de saída Tabela de Polimorfismo.txt

```
-> main ()
        -> _ZN1YC2Ev (Y)
                -> _ZN1XC2Ev (X)
                        -> _ZN1WC2Ev (W)
        -> _ZN1Y1mEv (Y)
        -> _ZN1X1nEv (X)
```

Fonte: O Autor

3.3.4 Arquivo YoYo.txt

Este arquivo possui todas as chamadas em cadeia realizadas a partir de cada método identificado no arquivo de entrada. A partir deste documento o usuário pode analisar o impacto de cada chamada recursivamente e todas as possibilidades de chamadas polimórficas, assim como é feito no grafo *Yo-Yo*. Para cada método exibido, sua respectiva classe é exibida entre parênteses ao seu lado.

O número de tabulações de cada linha indica que as chamadas para estes métodos se encontram dentro do escopo do mesmo método. Por exemplo, na situação representada na

Figura 3.37 a chamada polimórfica para os métodos j() de A e de C, representada por "P (M_Aj M_Cj)", e a chamada para método "M_Bh" ocorrem no escopo do método "M_ma". Como a chamada polimórfica poderá acarretar na execução do método M_Aj ou do método M_Cj, ambos são exibidos como parte do escopo da chamada polimórfica (com uma tabulação extra).

Figura 3.37 – Exemplo do arquivo de saída YoYo.txt

```
-----------------------------------------------------------
Método: M_mA (A)

        -> M_mA (A)
                -> P (M_Aj M_Cj)
                        -> M_Aj (A)
                        -> M_Cj (C)
                                -> M_Bk (B)
                                        -> P (M_Al M_Cl)
                                                -> M_Al (A)
                                                -> M_Cl (C)
                -> M_Bh (B)
```

Fonte: O Autor

A partir da análise do arquivo *YoYo*.txt, é possível observar o fluxo de execução para um objeto de um determinado tipo. Por exemplo, supondo que o usuário deseja saber qual fluxo será executado para um objeto da classe B. Este usuário deverá seguir o fluxo dos métodos polimórficos do tipo B ou herdados por esta classe a cada bifurcação causada por uma chamada polimórfica. As chamadas polimórficas são exibidas como exemplificado na Figura 3.38. Neste exemplo o método _ZN1A1hEv (método h() de A) chama o método polimórfico i(). Todos os métodos polimórficos i() são exibidos entre parênteses ao lado da letra P. Para cada método i() passível de execução, todas as suas futuras chamadas são exibidas, recursivamente, até que não existam mais chamadas possíveis ou que um loop seja encontrado (após 50 iterações).

Figura 3.38 – Exemplo do arquivo de saída YoYo.txt (2)

```
-------------------------------------------------------------
Método: _ZN1A1hEv (A)

        -> _ZN1A1hEv (A)
                -> P (_ZN1A1iEv _ZN1B1iEv _ZN1C1iEv)
                        -> _ZN1A1iEv (A)
                                -> P (_ZN1A1jEv _ZN1C1jEv)
                                        -> _ZN1A1jEv (A)
                                        -> _ZN1C1jEv (C)
                                                -> _ZN1B1kEv (B)
                        -> _ZN1B1iEv (B)
                                -> P (_ZN1A1jEv _ZN1C1jEv)
                                        -> _ZN1A1jEv (A)
                                        -> _ZN1C1jEv (C)
                                                -> _ZN1B1kEv (B)
                        -> _ZN1C1iEv (C)
                                -> P (_ZN1A1jEv _ZN1C1jEv)
                                        -> _ZN1A1jEv (A)
                                        -> _ZN1C1jEv (C)
                                                -> _ZN1B1kEv (B)
```

Fonte: O Autor

3.3.5 Arquivos específicos

Os arquivos referentes às anomalias descritas por Amman;Offut (2008) foram criados no intuito de indicar ao usuário a quais aspectos do código ele deve atentar-se e quais estruturas do código devem ser testadas com mais cuidado a fim de encontrar estas possíveis falhas ou anomalias. Os arquivos referentes às anomalias possuem o mesmo formato apresentado na Figura 3.39. Primeiro é exibida a sigla da anomalia descrita por Amman;Offut (2008) e uma tradução do seu nome. Em seguida há uma breve descrição sobre do que se trata esta possível anomalia, seguida pela descrição da saída apresentada pelo protótipo. Na próxima linha há uma descrição de como esta anomalia pode ser verificada a partir das informações fornecidas neste documento. As informações variam dependendo da anomalia que se deseja explorar. Após as descrições, encontram-se as informações providas pelo protótipo que buscam auxiliar na detecção da anomalia.

Figura 3.39 – Padrão de arquivo de saída

```
2   SIGLA - TRADUÇÃO DO NOME DA ANOMALIA
3   É uma anomalia que pode ocorrer quando ...
4   SAIDA: Descrição da saída exibida neste documento.
5   TESTE: Como utilizar as informações deste documento a fim de detectar a
    anomalia no código.
6
7   -----------------------------------------------------------------------
8
9   CASO 1
10  -----------------------------------------------------------------------
11
12  CASO 2
```

Fonte: O Autor

Quando não existem, no código, estruturas que possam acarretar uma dada anomalia, o arquivo de saída correspondente será semelhante ao apresentado na imagem Figura 3.40. Um exemplo disso é quando a anomalia diz respeito a relacionamentos entre classes descendentes, mas o arquivo de entrada não faz uso de herança. Neste caso não é possível gerar estruturas que auxiliam no teste da anomalia, visto que ela nunca irá ocorrer.

Figura 3.40 – Padrão de arquivo de saída sem resultados

```
2   SIGLA - TRADUÇÃO DO NOME DA ANOMALIA
3   É uma anomalia que pode ocorrer quando ...
4   SAIDA: Descrição da saída exibida neste documento.
5   TESTE: Como utilizar as informações deste documento a fim de detectar a
    anomalia no código.
6
7   -----------------------------------------------------------------------
8   Nenhum caso encontrado no código analisado!
9
```

Fonte: O Autor

Foram desenvolvidos arquivos de saída para sete das nove anomalias descritas no Capítulo 2. As anomalias IISD e SVA dependem de um histórico de execução do protótipo para o mesmo arquivo de entrada para que sejam devidamente avaliadas, visto que ambas são acarretadas por alterações de estruturas internas que mudam a visibilidade de métodos e atributos, ou seja, podem acontecer em etapas de evolução ou manutenção do sistema em teste. Ambas podem ser incluídas em uma possível expansão do protótipo, a qual mantenha um histórico dos resultados e execute a análise baseada nas alterações entre versões do mesmo código.

3.3.5.1 Arquivo ITU.txt

O arquivo ITU.txt exibe informações que podem auxiliar o usuário do protótipo a desenvolver testes voltados à identificação de anomalias de uso de tipos inconsistentes. A Figura 3.41 mostra que o arquivo possui uma breve descrição do problema, da saída gerada pelo protótipo e dicas de como podem ser criados testes a fim de detectar a anomalia descrita.

Figura 3.41 – Exemplo do arquivo de saída ITU.txt

```
2   ITU - USO DE TIPO INCONSISTENTE
3   É uma anomalia que pode ocorrer quando o relacionamento de herança é
    usado e contextos nos quais não existe uma relação de subtipo entre as
    classes pai e filha.
4   SAÍDA: Para cada classe existente, são exibidos todos os seus métodos que
    efetuam chamadas para métodos de classes herdadas.
5   TESTE: Para cada chamada entre métodos descrita abaixo, verificar se o
    objeto da classe herdeira é usado no contexto da classe herdada.
6
7   --------------------------------------------------------------------------
8   Chamadas a partir dos métodos da Classe: Y
9       Método: _ZN1YC2Ev
10          Chamadas:
11              Tipo: Chamada -     Metodo_Chamado: _ZN1XC2Ev
                Metodo_Chamado_Classe: X    Objeto_nome: this_value
                Objeto_classe: Y
12      Método: _ZN1Y1sEv
13          Chamadas:
14              Tipo: Chamada -     Metodo_Chamado: _ZN1X1oEv
                Metodo_Chamado_Classe: X    Objeto_nome: this_value
                Objeto_classe: Y
```

Fonte: O Autor

Esta anomalia ocorre quando uma classe descendente de outra classe não sobrescreve os métodos da classe herdada de forma consistente, sendo assim, a classe herdeira faz uso de métodos da classe herdada, a partir de chamadas, e estes métodos possuem uma semântica diferente quanto à alteração do objeto. A fim de identificar esta situação, para cada classe existente no arquivo de entrada, são exibidas todas as chamadas feitas para métodos de classes ancestrais. Estes métodos devem ser analisados a fim de encontrar eventuais divergências semânticas que causariam inconsistências no estado do objeto.

No exemplo apresentado na Figura 3.41 a classe Y possui apenas dois métodos que possuem uma chamada cada um para sua classe ancestral X. O método _ZN1YC2Ev (construtor de Y) chama o método _ZN1XC2Ev (construtor de X) e o método _ZN1Y1sEv (método “s()” da classe Y) chama o método _ZN1X1oEv (método “o()” da classe X).

Portanto a execução dos métodos o() de X e s() de Y devem ser tentadas de forma alternada, assim como a execução dos construtores de X e Y.

3.3.5.2 Arquivo SDA.txt

O arquivo SDA.txt foi criado a fim de salientar características do código de entrada que podem acarretar em uma anomalia de definição de estado. Visto que esta anomalia ocorre quando as interações de estado de um descendente não são consistentes com as do ancestral, o protótipo compara todos os métodos polimórficos que compartilham a mesma coluna na tabela virtual, ou seja, polimórficos entre si, e, quando há diferenças entre eles quanto às definições ou chamadas feitas, estas divergências serão exibidas no documento SDA.txt.

Figura 3.42 – Exemplo do arquivo de saída SDA.txt

```
 2  SDA - ANOMALIA DE DEFINIÇÃO DE ESTADO
 3  É uma anomalia que ocorre quando as interações de estado de um
    descendente não são consistentes com as do ancestral, ou seja, quando os
    métodos da classe herdeira não fornecem as mesmas definições para as
    variáveis de estado dos métodos da classe herdada que foram sobrescritos.
 4  SAÍDA: Foram comparados todos os métodos polimórficos existentes entre
    si, os que apresentam divergências quanto às definições ou chamadas de
    funções serão exibidos abaixo.
 5  TESTE: Os seguintes métodos podem apresentar esta anomalia entre si e
    devem ser testados a fim de verificar as interações de estado.
 6
 7  -------------------------------------------------------------------------
 8
 9  Possível SDA nos seguintes metodos polimorficos entre si:
10  _ZN1Y1mEv _ZN1W1mEv
11  Os metodos diferem quanto as DEFINIÇÕES, por favor conferir as
    informacoes abaixo:
12      Metodo: _ZN1Y1mEv
13          Tipo: Definição -       Atributo: w     Objeto_nome:
            this_value     Objeto_classe: Y
14      Metodo: _ZN1W1mEv
15          Tipo: Definição -       Atributo: v     Objeto_nome:
            this_value     Objeto_classe: W
16  -------------------------------------------------------------------------
17
18  Possível SDA nos seguintes metodos polimorficos entre si:
19  _ZN1X1nEv _ZN1W1nEv _ZN1X1nEv
20  Os metodos diferem quanto as DEFINIÇÕES, por favor conferir as
    informacoes abaixo:
21      Metodo: _ZN1X1nEv
22          Tipo: Definição -       Atributo: x     Objeto_nome:
            this_value     Objeto_classe: X
23      Metodo: _ZN1W1nEv
```

Fonte: O Autor

No exemplo apresentado na Figura 3.42 existem dois conjuntos polimórficos entre si que não fazem as mesmas alterações sobre as mesmas variáveis de estado. No primeiro conjunto, o método Y::m() define a variável de estado w. Porém, W::m() define v. Portanto, ambos não são consistentes entre si, o que pode acarretar em uma anomalia de definição de estado. No segundo caso W::n() e X::n() não são semelhantes pois o método de X define o atributo x, o que não ocorre no método de W.

3.3.5.3 Arquivo SDIH.txt

O arquivo SDIH.txt foi criado a fim de proporcionar sugestões ao usuário de quais estruturas devem ser testadas a fim de encontrar possíveis anomalias de definição de estado inconsistente. Esta anomalia ocorre quando a inclusão de uma variável local em uma classe herdeira sobrescreve a variável da classe herdada, deixando esta escondida.

A fim de identificar relacionamentos como este, o protótipo percorre todas as classes fornecidas no arquivo de entrada. Para cada classe são investigadas suas classes ancestrais recursivamente e exibidos todos os atributos de mesmo nome existentes nesta estrutura. O usuário deverá testar seus valores a fim de identificar se as variáveis de estados estão sendo alteradas de acordo com a especificação.

No exemplo apresentado na Figura 3.43, é possível observar que para a análise recursiva feita para a classe Y (linha 7), foi identificada, em sua estrutura hierárquica, outra classe que possui um atributo em comum com Y, W. A classe W possui o atributo v assim como a classe Y o possui. Sendo assim, o atributo v de W pode ser sobrescrito por v de Y, escondendo-o e acarretando em uma anomalia de fluxo de dados.

Figura 3.43 – Exemplo do arquivo de saída SDIH.txt

```
2  SDIH - INCONSISTÊNCIA DE DEFINIÇÃO DE ESTADO
3  Ocorre quando a inclusão de uma variável local em uma classe herdeira
   sobrescreve a variável da classe herdada, deixando esta escondida, podendo
   assim causar uma anomalia de fluxo de dados.
4  SAÍDA: Foram comparados todos os atributos entre classes herdeiras e
   herdadas, todos os que possuem mesmo nome serão exibidos abaixo.
5  TESTE: Os atributos exibidos devem ser testados a fim de identificar se as
   variáveis de estado estão sendo alteradas de acordo com a especificação.
6
7  ---------------------------------------------------------------------------
8  Para a classe: Y
9  ----- Atributos de mesmo nome encontrados -----
10 Nome do Atributo: v
11 Este atributo existe nas classes:
12 Y
13 W
```

Fonte: O Autor

3.3.5.4 Arquivo SDI.txt

O arquivo SDI.txt foi criado no intuito de exibir estruturas para orientar o usuário na busca da identificação de possíveis definições incorretas de estado. A anomalia de definição incorreta de estado pode ocorrer quando o método herdeiro define a mesma variável de estado que o método antecessor possui e a computação feita pelo método herdeiro não é semanticamente correspondente à computação do método sobrescrito em relação à mesma variável de estado.

A fim de identificar as estruturas presentes no código de entrada que possam apresentar este comportamento, são exibidos todos os métodos polimórficos que definem variáveis de estado presentes na hierarquia de cada classe pertencente ao código. Para cada classe existente, uma saída é gerada. Supondo que existam as classes X, Y e W, onde X é uma especificação de W e Y é uma especialização de X. Supondo que ambos W e Y possuam os métodos k() e m() e que estes métodos definam a variável de estado v, que pertence a W. O resultado da execução do protótipo para a situação descrita encontra-se representada na Figura 3.44. É possível notar que para cada classe existente (X, Y e W) foi gerada uma análise de todos os métodos polimórficos, presentes dentro de suas estruturas hierárquicas, que possuem definições para a variável de estado v.

A semântica das definições apresentadas para cada classe deve ser testada a fim de identificar divergências no estado do objeto quanto ao conteúdo das variáveis de estado definidas.

Figura 3.44 – Exemplo do arquivo de saída SDI.txt

```
 2  SDI - DEFINIÇÃO INCORRETA DE ESTADO
 3  Ocorre quando o método herdeiro define a mesma variável de estado que o
    método antecessor possui e a computação feita pelo método herdeiro não é
    semanticamente correspondente à computação do método sobrescrito em
    relação à mesma variável de estado, sendo assim uma anomalia de
    comportamento em potencial.
 4  SAÍDA: Para cada classe existente e para cada variável de estado dentro
    de sua hierarquia, todos os métodos polimórficos que a definem serão
    exibidos.
 5  TESTE: A semântica de definição da variável de todos os métodos exibidos
    deve ser analisada e testada.
 6
 7  --------------------------------------------------------------------------
 8  Análise a partir da Classe: Y
 9  A variável de estado v da classe W é definida nos métodos:
10      _ZN1Y1mEv:
11          Tipo: Definição -       Atributo: v     Atributo Classe: W
12      _ZN1W1mEv:
13          Tipo: Definição -       Atributo: v     Atributo Classe: W
14      _ZN1Y1kEv:
15          Tipo: Definição -       Atributo: v     Atributo Classe: W
16      _ZN1W1kEv:
17          Tipo: Definição -       Atributo: v     Atributo Classe: W
18  --------------------------------------------------------------------------
19  Análise a partir da Classe: W
20  A variável de estado v da classe W é definida nos métodos:
21      _ZN1W1mEv:
22          Tipo: Definição -       Atributo: v     Atributo Classe: W
23      _ZN1W1kEv:
24          Tipo: Definição -       Atributo: v     Atributo Classe: W
25  --------------------------------------------------------------------------
26  Análise a partir da Classe: X
27  A variável de estado v da classe W é definida nos métodos:
28      _ZN1W1mEv:
29          Tipo: Definição -       Atributo: v     Atributo Classe: W
30      _ZN1W1kEv:
31          Tipo: Definição -       Atributo: v     Atributo Classe: W
```

Fonte: O Autor

3.3.5.5 Arquivo ACB1.txt

Esta anomalia diz respeito à execução de métodos polimórficos a partir de construtores. Quando um construtor chama um método polimórfico, é possível que, dependendo do objeto que acarretou a chamada, um método da classe herdeira seja executado e faça uso de variáveis de estado da classe herdeira que ainda não foram propriamente definidas.

A fim de identificar pontos do código que possam acarretar na anomalia descrita, o protótipo exibe no documento ACB1.txt todos os construtores que possuem chamadas polimórficas e, para cada método polimórfico, todos os seus respectivos usos que referenciam atributos da própria classe. Os testes devem ser gerados com foco na verificação do estado das variáveis afetadas.

Figura 3.45 – Exemplo do arquivo de saída ACB1.txt

```
 2  ACB1 - COMPORTAMENTO ANÔMALO DE CONSTRUÇÃO
 3  Pode ocorrer quando o construtor de uma classe ancestral executa um método
    polimórfico na classe herdeira, se o método chamado utiliza variáveis de
    estado que não foram previamente definidas, pois sua definição encontra-se
    no construtor da classe herdeira.
 4  SAÍDA: Abaixo se encontram todos os construtores que possuem chamadas
    polimórficas, e, para cada método polimórfico, todos os seus respectivos
    usos que referenciam atributos de sua própria classe.
 5  TESTE: Os testes devem ser gerados com foco na verificação do estado das
    variáveis afetadas.
 6
 7  ---------------------------------------------------------------------------
 8  Construtor: _ZN1CC2Ev   Classe: C
 9      Chamada polimórfica para: _ZN1D1fEv _ZN1C1fEv
10
11          Usos do Método: _ZN1D1fEv   Método Classe: D
12              Tipo: Uso -     Atributo: x     Atributo Classe: D
13
14          Usos do Método: _ZN1C1fEv   Método Classe: C
15              Este método não possui usos que façam uso de atributos de sua
                própria classe!
```

Fonte: O Autor

A Figura 3.45 possui um exemplo de arquivo ACB1.txt. Neste exemplo o construtor da classe C possui uma chamada polimórfica para o método f(). A classe D é filha de C e possui um método f() declarado em seu escopo. Como pose ser observado na linha 12, o método f() de D possui um uso para o atributo x, que pertence a sua classe. O método f() de C, por sua vez, não possui usos para nenhuma variável de estado de sua própria classe. Tendo em vista que o método f() de D poderá ser chamado pelo construtor de C, o estado do atributo x de D deverá ser testado a fim de identificar o possível uso deste atributo antes do mesmo ser devidamente definido pelo construtor de D.

3.3.5.6 Arquivo ACB2.txt

Assim como o ACB1, o ACB2 também faz referência ao comportamento anômalo de construção causado por chamadas polimórficas dentro de construtores. O ACB2 pode ocorrer quando um método polimórfico de uma classe herdeira é chamado e este método faz uso de algum atributo da classe ancestral, sem que este atributo tenha sido previamente definido, visto que o método que o faz foi sobrescrito pelo método chamado. A fim de identificar esta anomalia, o arquivo ACB2.txt exibe todos os construtores que possuem chamadas polimórficas, e, para cada método polimórfico, todos os usos de variáveis de estado herdadas presentes neste método, se existirem. Caso não exista nenhum caso, a mensagem “Este método não possui usos que façam uso de atributos de alguma classe ancestral” será exibida.

Figura 3.46 – Exemplo do arquivo de saída ACB2.txt

```
 2  ACB2 - COMPORTAMENTO ANÔMALO DE CONSTRUÇÃO 2
 3  Ocorre quando um método polimórfico é chamado por um construtor em uma
    subclasse e faz uso de variáveis de estado de sua superclasse, sem que os
    mesmos tenham sido previamente definidos, pois o método que o faz foi
    sobrescrito pelo método chamado.
 4  SAÍDA: Abaixo se encontram todos os construtores que possuem chamadas
    polimórficas, e, para cada método polimórfico, todos os seus respectivos
    usos que referenciam atributos de classes herdadas.
 5  TESTE: Os testes devem ser gerados com foco na verificação do estado das
    variáveis afetadas.
 6
 7  ---------------------------------------------------------------------------
 8  Construtor: _ZN1DC2Ev   Classe: D
 9      Chamada polimórfica para: _ZN1D1fEv _ZN1C1fEv
10
11          Usos do método: _ZN1D1fEv   Método Classe: D
12              Tipo: Uso -     Atributo: x     Atributo Classe: C
13
14          Usos do método: _ZN1C1fEv   Método Classe: C
15              Este método não possui usos que façam uso de atributos de
                alguma classe ancestral!
16
17  ---------------------------------------------------------------------------
18  Construtor: _ZN1CC2Ev   Classe: C
19      Chamada polimórfica para: _ZN1D1fEv _ZN1C1fEv
20
21          Usos do método: _ZN1D1fEv   Método Classe: D
22              Tipo: Uso -     Atributo: x     Atributo Classe: C
23
24          Usos do método: _ZN1C1fEv   Método Classe: C
25              Este método não possui usos que façam uso de atributos de
                alguma classe ancestral!
```

Fonte: O Autor

Supondo que existam duas classes, C e D. A classe D é herdeira de C. Ambas as classes possuem um método f() polimórfico que é chamado por ambos construtores. Porém, apenas a classe C possui um o atributo x, que é usado por f() de D. A Figura 3.46 exemplifica o arquivo de saída ACB2.txt para esta situação. Para cada construtor, se um método polimórfico é chamado e este método faz uso de um atributo de alguma classe ancestral, o mesmo é exibido, como é o caso de x para f() de D no exemplo apresentado. A etapa de testes deve ter foco na verificação do valor de x a fim de identificar inconsistências na definição de seu estado.

3.3.5.7 Arquivo IC.txt

Esta anomalia faz referência aos problemas que podem ocorrer quando os atributos das classes que não são devidamente definidos por seus construtores, ou seja, quando o estado inicial do objeto é indefinido. A fim de detectar casos como este, o conteúdo do arquivo IC.txt exibe todos os construtores das classes acompanhados de todos os atributos que não possuem definição neste construtor.

Figura 3.47 – Exemplo do arquivo de saída IC.txt

```
2   IC - CONSTRUÇÃO INCOMPLETA
3   Ocorre quando o estado inicial do objeto é indefinido, ou seja, quando o
    construtor de sua classe não define todos os atributos da classe.
4   SAÍDA: Abaixo encontram-se todos os construtores que não definem os
    atributos de sua classe, seguidos pelos nomes dos atributos não declarados.
5   TESTE: Os valores dos atributos listados abaixo devem ser verificados a fim
    de evitar que os mesmos sejam usados sem a devida inicialização.
6
7   ---------------------------------------------------------------------------
8   O Construtor _ZN1DC1Ev da Classe: D não possui declaração para os atributos
    abaixo:
9   x
10  a
11  c
12
13  ---------------------------------------------------------------------------
14  O Construtor _ZN1CC2Ev da Classe: C não possui declaração para os atributos
    abaixo:
15  y
16  z
```

Fonte: O Autor

A Figura 3.47 exemplifica o conteúdo do arquivo IC.txt. Para cada construtor encontrado, tendo em vista que cada classe pode possuir mais de um construtor, são exibidos

todos os atributos da classe não declarados dentro deste construtor. Os atributos exibidos devem ser testados a fim de verificar se seu estado está de acordo com o esperado, visto que a falta de inicialização dos mesmos pode acarretar em anomalias. No exemplo apresentado, é possível notar que o construtor da classe D não possui declaração para os atributos x, a e c. Também é possível notar que o construtor da classe C não possui definição para os atributos y e z.

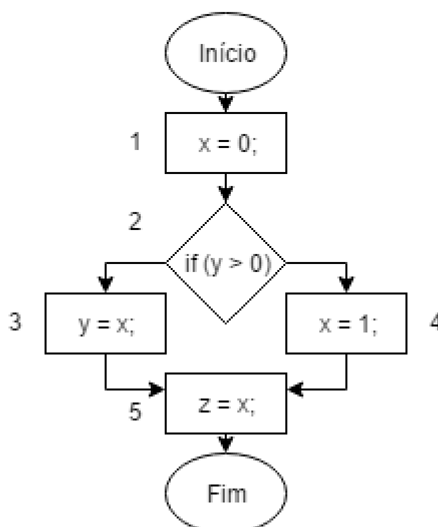
3.4 Expansão do código

A partir do código inicial, outras funcionalidades podem ser implementadas a fim de aprimorá-lo. Nos tópicos seguintes há possíveis sugestões de expansão para o protótipo desenvolvido.

3.4.1 Inclusão de análise de pares DU

Os pares DU são pares de definição e uso que fazem referência ao mesmo atributo ou objeto. Um par DU é formado se existir um caminho no fluxo do programa em que o valor atribuído na definição pode atingir um ponto do uso. Visto que o protótipo mantém, em sua memória, todas as definições e usos de atributos e/ou objetos, a análise de pares DU pode ser feita através da exploração desta estrutura.

Figura 3.48 – Exemplo de fluxo



Fonte: O Autor

O protótipo desenvolvido não realiza segmentação do fluxo interno dos métodos, visto que seu foco inicial é o relacionamento entre classes e métodos. A fim de identificar caminhos no fluxo interno do método, seria necessária a alteração da segunda parte da segunda varredura para a inclusão do código responsável pela identificação de quebras neste fluxo. A identificação do fluxo passa a ser necessária nesta etapa, pois é um ponto importante para a formação de pares DU. Por exemplo, considerando o fluxo da Figura 3.48, se a bifurcação que possui um uso da variável x, representada pelo número 3, for executada, então o último nodo (5), que também faz uso de x, fará parte de um caminho DU livre de definições representado por 1-2-3-5. Por outro lado, se o outro caminho for considerado, então 4-5 é um par DU. No protótipo atual não existe esta distinção causada por bifurcações, portanto estes passos seriam vistos como sequenciais e não seria possível identificar os caminhos de pares DU livres de outras definições.

Outra mudança necessária seria a alteração da estrutura de armazenamento de definições, usos e chamadas. Atualmente estas informações são armazenadas em listas, visto que não há distinção de fluxo interno. A partir do momento em que o fluxo interno é considerado, estas estruturas deverão ser adaptadas a fim de abrigar as novas informações registradas. Esta alteração poderá ser feita através da substituição do uso de listas pelo uso de grafos contendo bifurcações de acordo com o fluxo de execução interno de cada método do código.

A partir dos dados coletados, é possível relacionar as definições e usos que ocorrem dentro de outros métodos em relação ao método que os invoca, inclusive em chamadas polimórficas, visto que esta informação está presente nas estruturas já desenvolvidas. Exemplos de relacionamentos de definição e uso entre métodos podem ser encontrados no Capítulo 7 da obra de Amman;Offut (2008).

3.4.2 Inclusão de todos os métodos presentes no arquivo de entrada

O arquivo LLVM concentra todas as classes e métodos usados no código compilado, isso faz com que este arquivo seja longo e possua classes com nomes que não são necessariamente conhecidos pelo desenvolvedor. Portanto, por motivos de clareza, o protótipo desenvolvido considera apenas os métodos das classes definidas no arquivo de entrada, desconsiderando aqueles importados a partir de outras bibliotecas.

Porém, a expansão do protótipo para a inclusão dos métodos de todas as classes envolvidas, presentes no arquivo LLVM, não seria complexa. Visto que todas as etapas de

validação, que ocorrem toda vez que uma nova linha do arquivo é lida, ocorrem sobre as estruturas obtidas na primeira varredura, a única alteração necessária seria na expressão regular que testa se a linha é uma classe.

As classes importadas aparecem em um formato que difere das pertencentes ao código que originou o arquivo de entrada. A Figura 3.49 exemplifica cinco declarações de classes. As linhas 16, 17 e 18 correspondem a declarações de classes importadas, já as linhas 19 e 20 possuem declarações presentes no código que originou o arquivo LLVM, portanto possui uma sintaxe diferenciada das demais. Tendo essa diferença de padrão em evidência, a expressão regular que identifica a declaração de classes deveria ser ajustada ao padrão desejado a fim de considerar as demais classes como parte do código. A partir do momento em que todas as classes declaradas são consideradas, os métodos pertencentes às mesmas serão também considerados.

Figura 3.49 – Exemplo de declaração de classes

```
16  %"class.std::locale::facet.base" = type <{ i32 (...)**, i32 }>
17  %"class.std::num_put" = type { %"class.std::locale::facet.base", [4 x i8] }
18  %"class.std::num_get" = type { %"class.std::locale::facet.base", [4 x i8] }
19  %class.D = type <{ i32 (...)**, i32, i8, i8, [2 x i8], i32, [4 x i8], %class.C }>
20  %class.C = type { i32 (...)**, i32, i8, %"class.std::__cxx11::basic_string" }
```

Fonte: O Autor

Esta mudança pode causar problemas de processamento, visto que a etapa de validação deverá percorrer estruturas maiores em alguns casos. Por este motivo é recomendado o uso de concorrência, visto que a etapa de validação de estruturas pode ser executada em paralelo.

4 RESULTADOS

Este Capítulo apresenta os resultados obtidos após a execução do protótipo quando aplicado a diferentes códigos.

4.1 Aplicação do protótipo em seu próprio código

O primeiro conjunto de resultados gerados visa à identificação de anomalias no próprio código desenvolvido neste trabalho. Visto que o protótipo desenvolvido não faz uso de polimorfismo, os arquivos de saída que dependem da análise de estruturas polimórficas não retornaram nenhum resultado, como é o caso de: SDA, SDI, ACB1 e ACB2. Os arquivos de saída para estas anomalias possuem apenas o conteúdo: “Nenhum caso encontrado no código analisado!”. As saídas para os demais arquivos são detalhadas nas próximas seções.

4.1.1 Saída para o arquivo Classes.txt

O respectivo arquivo .ll gerado para este código possui 116286 linhas. Isso acontece porque muitas bibliotecas de C++ são importadas ao longo do código desenvolvido. Tendo em vista o grande número de linhas, é possível perceber que os arquivos de saída, como o Classe.txt, serão longos. O arquivo Classes.txt gerado possui 4489 linhas e está disponível no repositório GIT https://bitbucket.org/Jayne_GC/resultados.git, assim como os demais arquivos gerados.

Visto que o arquivo gerado é muito grande para ser exibido e comentado, a Figura 4.1 exemplifica apenas uma parte da saída gerada para a classe “Objeto” no arquivo Classes.txt. Na figura, a classe objeto possui três atributos: “objeto_nome”, “objeto_classe” e “objeto_ordem”. Estes três atributos podem aparecer com variações em seus nomes dentro do arquivo .ll, como é o caso de “objeto_nome”, que em algum momento do arquivo .ll aparece como “objeto_nome2”. Neste fragmento do arquivo Classes.txt são exibidos dois métodos pertencentes à classe Objeto: “_ZN6Objeto15get_objeto_nomeEv” e “_ZN6Objeto17get_objeto_classeEv”. O método “_ZN6Objeto15get_objeto_nomeEv”, por exemplo, é um método que tem por finalidade retornar o nome do objeto, portanto este método possui um uso para o atributo “objeto_nome”.

Figura 4.1 – Resultados: Classes.txt

```
Classe: Objeto
        Atributos:
                objeto_nome
                        objeto_nome2
                objeto_classe
                        objeto_classe3
                objeto_ordem
                        objeto_ordem4
                        objeto_ordem5
        Metodos:
                _ZN6Objeto15get_objeto_nomeEv
                        Objetos:
                                this_value      Objeto_classe: Objeto
                        Definições:
                        Usos:
                                Tipo: Uso - Atributo: objeto_nome  Objeto_nome: this_value ...
                                                                      ...Objeto_classe: Objeto  Ordem: 6
                        Chamadas:
                _ZN6Objeto17get_objeto_classeEv
                        Objetos:
                                this_value      Objeto_classe: Objeto
                        Definições:
                        Usos:
                                Tipo: Uso - Atributo: objeto_classe  Objeto_nome: this_value ...
                                                                      ...Objeto_classe: Objeto  Ordem: 6
                        Chamadas:
```

Fonte: O Autor

4.1.2 Saída para o arquivo Tabela de Polimorfismo.txt

O arquivo Tabela de Polimorfismo.txt não possui uma tabela virtual, pois o código do protótipo não faz uso de polimorfismo. Portanto este arquivo de saída possui apenas o conteúdo "Não existem tabelas de polimorfismo para o código selecionado!" após a execução do protótipo.

4.1.3 Saída para o arquivo Fluxo.txt

O arquivo Fluxo.txt gerado é longo demais para ser totalmente explorado neste trabalho. Portanto, apenas um trecho do arquivo Fluxo.txt é exibido a fim de exemplificar um loop presente no código.

Na Figura 4.2 é possível notar que o método "show_all_methods" chama os métodos: "get_def_ou_uso_tipo", "get_chamada_vfn_table_posição", "get_metodo_nome" e "show_all_methods" recursivamente. O método "show_all_methods" é o método usado para gerar o arquivo Fluxo.txt. Toda vez em que é chamado para um determinado método lido do arquivo .ll, o método "show_all_methods" explora toda a lista de chamadas (normais ou polimórficas) do método recebido.

Figura 4.2 – Resultados: Fluxo.txt

```
-> _ZN6Metodo16show_all_methodsESt6vectorIS_SaIS_EEiS0_IS0_INSt7__... (Metodo)
    -> _ZN10Def_ou_Uso19get_def_ou_uso_tipoEv (Def_ou_Uso)
    -> _ZN7Chamada29get_chamada_vfn_table_posicaoEv (Chamada)
    -> _ZN6Metodo15get_metodo_nomeEv (Metodo)
    -> _ZN6Metodo16show_all_methodsESt6vectorIS_SaIS_EEiS0_IS0_INSt7__... (Metodo)
```

Fonte: O Autor

Para cada chamada do método corrente, o método "show_all_methods" primeiramente chama "get_def_ou_uso_tipo" a fim de verificar se a chamada é normal ou polimórfica, se a chamada for polimórfica, então o método "get_chamada_vfn_table_posição" é chamado a fim de identificar a coluna da tabela polimórfica que será chamada. Em seguida, o método "get_metodo_nome" é chamado a fim de imprimir o nome do método corrente no arquivo de saída. Por fim, uma nova execução de "show_all_methods" é disparada recursivamente para cada um dos métodos que são chamados pelo método corrente.

4.1.4 Saída para o arquivo YoYo.txt

Assim como os arquivos Classe.txt e Fluxo.txt, a saída exibida no arquivo YoYo.txt não pode ser apresentada por inteiro neste trabalho devido ao tamanho do arquivo gerado. O arquivo YoYo.txt pode ser encontrado em sua totalidade no mesmo repositório GIT mencionado anteriormente.

Visto que o código desenvolvido neste trabalho não faz uso de polimorfismo, o arquivo YoYo.txt não parece fazer tanto sentido neste contexto. Porém, se o usuário deseja analisar apenas o fluxo de execução de um método especifico cuja execução não é desencadeada pelo método main, este arquivo pode auxiliá-lo nesta tarefa.

A fim de exemplificar a saída gerada, um trecho do arquivo de saída YoYo.txt é exibido na Figura 4.3. Este trecho exibe a saída gerada para o método "output_print_OBJ". Este método recebe, por parâmetro, um objeto do tipo "Objeto" e um ponteiro para um arquivo de saída e tem por objetivo imprimir, no arquivo de saída, o nome e a classe do objeto recebido. Para tanto, o método "output_print_OBJ" aciona os métodos: "get_objeto_nome" e "get_objeto_classe". O método "get_objeto_nome" retorna o nome do objeto e o método "get_objeto_classe" retorna sua respectiva classe.

Figura 4.3 – Resultados: YoYo.txt

```
------------------------------------------------------------------------
Método: _ZN6Output16output_print_OBJESt6vectorI6ObjetoSaIS1_...(Output)

    -> _ZN6Output16output_print_OBJESt6vectorI6ObjetoSaIS1_... (Output)
        -> _ZN6Objeto15get_objeto_nomeEv (Objeto)
        -> _ZN6Objeto17get_objeto_classeEv (Objeto)

------------------------------------------------------------------------
```

Fonte: O Autor

4.1.5 Saída para o arquivo ITU.txt

O código desenvolvido possui um relacionamento de herança que não é de subtipo. A classe “Chamada” herda da classe “Def_ou_Uso”, porém uma chamada não é uma definição ou um uso. Sendo assim, se um objeto da classe “Chamada” for usado no contexto da classe “Def_ou_Uso”, uma anomalia de uso de tipo inconsistente pode ocorrer.

Figura 4.4 – Resultados: ITU.txt

```
ITU - USO DE TIPO INCONSISTENTE
É uma anomalia que pode ocorrer quando o relacionamento de herança é usado e contextos nos quais
não existe uma relação de subtipo entre as classes pai e filha.
SAÍDA: Para cada classe existente, são exibidos todos os seus métodos que efetuam chamadas para
métodos de classes herdadas.
TESTE: Para cada chamada entre métodos descrita abaixo, verificar se o objeto da classe herdeira é
usado no contexto da classe herdada.

---------------------------------------------------------------------------
Chamadas a partir dos métodos da Classe: Chamada
    Método: _ZN7ChamadaC2Ev
        Chamadas:
            Tipo: Chamada -     Metodo_Chamado: _ZN10Def_ou_UsoC2Ev   Metodo_Chamado_Classe: ...
                                ...Def_ou_Uso   Objeto_nome: this_value   Objeto_classe: Chamada
            Tipo: Chamada -     Metodo_Chamado: _ZN10Def_ou_UsoD2Ev   Metodo_Chamado_Classe: ...
                                ...Def_ou_Uso   Objeto_nome: this_value   Objeto_classe: Chamada
    Método: _ZN7ChamadaD2Ev
        Chamadas:
            Tipo: Chamada -     Metodo_Chamado: _ZN10Def_ou_UsoD2Ev   Metodo_Chamado_Classe: ...
                                ...Def_ou_Uso   Objeto_nome: this_value   Objeto_classe: Chamada
    Método: _ZN7ChamadaC2EOS_
        Chamadas:
            Tipo: Chamada -     Metodo_Chamado: _ZN10Def_ou_UsoC2EOS_  Metodo_Chamado_Classe: ...
                                ...Def_ou_Uso   Objeto_nome: this_value   Objeto_classe: Chamada
```

Fonte: O Autor

Portanto o arquivo de saída ITU.txt exibe todos os pontos em que os objetos da classe “Chamada” são usados no contexto da classe “Def_ou_Uso”, como exibido na Figura 4.4. A Figura 4.4 possui um trecho do arquivo ITU.txt gerado. Neste trecho há dois construtores e um destrutor da classe “Chamada”. Geralmente todos os construtores e destrutores serão

exibidos, visto que os atributos privados do tipo herdado também devem ser inicializados pela classe herdada.

No exemplo apresentado, o construtor "_ZN7ChamadaC2EOS_" chama o construtor "_ZN10Def_ou_UsoC2EOS_" utilizando um objeto do tipo "Chamada". Isso significa que um objeto do tipo da classe herdeira está sendo usado no contexto da classe herdada. Esta situação pode acarretar em uma anomalia de uso de tipo inconsistente.

4.1.6 Saída para o arquivo SDIH.txt

O código do protótipo foi desenvolvido utilizando um padrão para a nomenclatura de atributos. Neste padrão, o nome da classe correspondente foi incluído no nome do atributo a fim de trazer clareza na leitura do código. Esta padronização foi feita apenas por preferência pessoal e não é necessária para o funcionamento do protótipo.

O arquivo SDIH.txt exibe todos os atributos entre classes herdeiras e herdadas que possuem o mesmo nome. Visto que o padrão de nomenclatura descrito anteriormente foi utilizado, o código não possui atributos com mesmo nome dentro da mesma hierarquia de classes. Portanto o arquivo de saída SDIH.txt possui a apenas a indicação de que nenhum caso foi encontrado.

4.1.7 Saída para o arquivo IC.txt

O arquivo IC.txt exibe, para cada construtor encontrado, todos os atributos da respectiva classe que não foram definidos dentro do escopo do construtor. A Figura 4.5 ilustra alguns trechos do arquivo IC.txt.

Nos trechos exibidos é possível notar que o construtor "_ZN8AtributoC2Ev" da classe "Atributo" não possui declaração para dois de seus atributos: "atributo_outros_nomes" e "atributo_id". Sendo assim, o construtor "_ZN8AtributoC2Ev" executa uma construção incompleta para um objeto do tipo "Atributo".

Também é possível perceber que o construtor "_ZN6ObjetoC2Ev", da classe "Objeto", possui declarações para todos os atributos da classe "Objeto". Porém o construtor "_ZN6ObjetoC2ERKS_", também da classe "Objeto", não possui declarações para dois de seus atributos, sendo assim uma construção incompleta. Esta construção incompleta pode acarretar em anomalias ao longo do código, assim como as construções feitas pelo construtor "_ZN8AtributoC2Ev".

Figura 4.5 – Resultados: IC.txt

```
IC - CONSTRUÇÃO INCOMPLETA
Ocorre quando o estado inicial do objeto é indefinido, ou seja, quando o construtor de sua
classe não define todos os atributos da classe.
SAÍDA: Abaixo encontram-se todos os construtores que não definem os atributos de sua classe,
seguidos pelos nomes dos atributos não declarados.
TESTE: Os valores dos atributos listados abaixo devem ser verificados a fim de evitar que os
mesmos sejam usados sem a devida inicialização.

------------------------------------------------------------------------------------------

...

------------------------------------------------------------------------------------------
O Construtor _ZN8AtributoC2Ev da Classe: Atributo não possui declaração para os atributos abaixo:
atributo_outros_nomes
atributo_id

------------------------------------------------------------------------------------------

...

------------------------------------------------------------------------------------------
O Construtor _ZN6ObjetoC2Ev da Classe: Objeto possui declaração para todos os atributos da classe!

------------------------------------------------------------------------------------------
O Construtor _ZN6ObjetoC2ERKS_ da Classe: Objeto não possui declaração para os atributos abaixo:
objeto_nome
objeto_classe

------------------------------------------------------------------------------------------
...
```

Fonte: O Autor

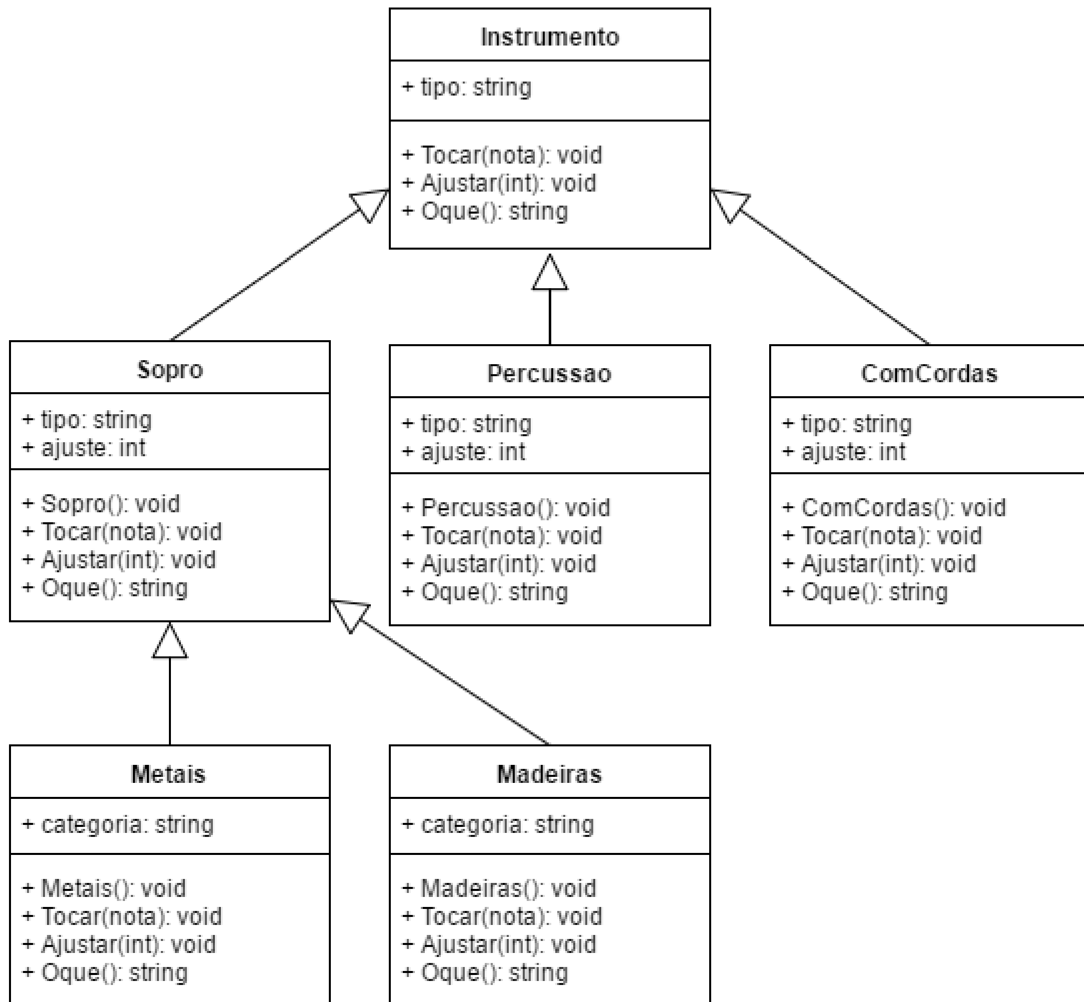
4.2 Aplicação do protótipo no código Instrumentos.cpp

O código “Instrumentos.cpp” define uma hierarquia de classes baseada em categorias de instrumentos musicais. Este código foi adaptado de um código obtido na página http://www.zemris.fer.hr/predmeti/os1/book/tic0153.html. Os nomes das classes, métodos e atributos foram traduzidos e novos métodos, atributos e chamadas foram incluídos.

A partir da análise do diagrama de classes referente ao código “Instrumentos.cpp” presente na Figura 4.6, é possível constatar a existência de herança e polimorfismo. A classe “Instrumento” possui métodos virtuais que são sobrescritos por seus descendentes. No exemplo apresentado, um instrumento musical pode ser: de sopro, de percussão ou com cordas. É possível notar que cada tipo de instrumento musical apresenta uma computação distinta para a função ”tocar”, evidenciando a existência de polimorfismo. No terceiro nível hierárquico, os instrumentos de sopro ainda são divididos em duas categorias distintas: metais e madeiras. Portanto, o código de entrada possui métodos polimórficos e três níveis de

herança, sendo assim propenso a apresentar anomalias provenientes de relacionamentos interclasse.

Figura 4.6 – Instrumentos.cpp – Diagrama de Classes



Fonte: O Autor

O arquivo de entrada possui: comportamento polimórfico, relacionamentos de herança e declaração de construtores. Estas características permitem que todas as análises de estruturas anômalas sejam efetuadas pelo protótipo.

Após a compilação do arquivo “Instrumentos.cpp”, utilizando Clang, o arquivo “Instrumentos.ll” foi gerado. O protótipo desenvolvido neste trabalho foi aplicado no arquivo “Instrumentos.ll”. Os arquivos de saída gerados encontram-se detalhados nas próximas seções. Todos os arquivos gerados nesta execução podem ser encontrados no repositório GIT https://bitbucket.org/Jayne_GC/resultados.git.

4.2.1 Saída para o arquivo Classes.txt (Instrumentos.cpp)

O arquivo Classes.txt é longo devido à existência de um grande número de classes, métodos e chamadas. Na Figura 4.7 é exibida a saída obtida no arquivo Classes.txt apenas para a classe "Percussao". É possível notar que seu construtor ("_ZN9PercussaoC2Ev") possui duas chamadas polimórficas, duas chamadas usuais, quatro usos e duas definições. As estruturas apresentadas neste arquivo ajudam a entender o que acontece no construtor da classe e em seus demais métodos.

Figura 4.7 – Resultados: Classes.txt (Instrumentos.cpp)

```
Classe: Percussao    Filho de: Instrumento
    Atributos:
        tipo
            tipo2
        ajuste
    Metodos:
        _ZN9PercussaoC2Ev
            Objetos:
                this_value      Objeto_classe: Percussao
            Definições:
                Tipo: Definição - Atributo: tipo   Objeto_nome: this_value   Objeto_classe: Percussao ...
                Tipo: Definição - Atributo: tipo   Objeto_nome: this_value   Objeto_classe: Percussao ...
            Usos:
                Tipo: Uso - Atributo: tipo   Objeto_nome: this_value   Objeto_classe: Percussao   Ordem: 13
                Tipo: Uso - Objeto_nome: this_value   Objeto_classe: Percussao   Ordem: 18
                Tipo: Uso - Objeto_nome: this_value   Objeto_classe: Percussao   Ordem: 30
                Tipo: Uso - Atributo: tipo   Objeto_nome: this_value   Objeto_classe: Percussao   Ordem: 56
            Chamadas:
                Tipo: Chamada - Metodo_Chamado: _ZN11InstrumentoC2Ev   Metodo_Chamado_Classe: Instrumento ...
                Tipo: Chamada Polimórfica - Posicao_na_tabela_de_polimorfismo: 1   Objeto_nome: this_value ...
                Tipo: Chamada Polimórfica - Posicao_na_tabela_de_polimorfismo: 2   Objeto_nome: this_value ...
                Tipo: Chamada - Metodo_Chamado: _ZN11InstrumentoD2Ev   Metodo_Chamado_Classe: Instrumento ...
        _ZNK9Percussao5tocarE4nota
            Objetos:
                this_value      Objeto_classe: Percussao
            Definições:
                Tipo: Definição - Atributo: tipo   Objeto_nome: this_value   Objeto_classe: Percussao ...
            Usos:
                Tipo: Uso - Atributo: ajuste   Objeto_nome: this_value   Objeto_classe: Percussao   Ordem: 11
            Chamadas:
        _ZNK9Percussao4OqueEv
            Objetos:
                this_value      Objeto_classe: Percussao
            Definições:
            Usos:
            Chamadas:
        _ZN9Percussao7ajustarEi
            Objetos:
                this_value      Objeto_classe: Percussao
            Definições:
                Tipo: Definição - Atributo: ajuste   Objeto_nome: this_value   Objeto_classe: Percussao ...
            Usos:
            Chamadas:
```

Fonte: O Autor

4.2.2 Saída para o arquivo Tabela de Polimorfismo.txt (Instrumentos.cpp)

A saída gerada para o arquivo Tabela de Polimorfismo.txt é apresentada na Figura 4.8. É possível notar a existência de três colunas: uma para o método "tocar", uma para o método

"Oque" e uma para o método "ajustar". Todos estes métodos são polimórficos entre si, assim como exibido no diagrama de classes da Figura 4.6.

Figura 4.8 – Resultados: Tabela de Polimorfismo.txt (Instrumentos.cpp)

	0	1	2
Sopro	_ZNK5Sopro5tocarE4nota	_ZNK5Sopro4OqueEv	_ZN5Sopro7ajustarEi
Instrumento			
Percussao	_ZNK9Percussao5tocarE4nota	_ZNK9Percussao4OqueEv	_ZN9Percussao7ajustarEi
ComCordas	_ZNK9ComCordas5tocarE4nota	_ZNK9ComCordas4OqueEv	_ZN9ComCordas7ajustarEi
Metais	_ZNK6Metais5tocarE4nota	_ZNK6Metais4OqueEv	_ZN6Metais7ajustarEi
Madeiras	_ZNK8Madeiras5tocarE4nota	_ZNK8Madeiras4OqueEv	_ZN8Madeiras7ajustarEi

Fonte: O Autor

4.2.3 Saída para o arquivo Fluxo.txt (Instrumentos.cpp)

A Figura 4.9 exibe um trecho do arquivo Fluxo.txt gerado. Neste trecho, no código original, ocorre a declaração de cinco objetos, um de cada tipo. Portanto o trecho do arquivo exibido possui as chamadas para os construtores das respectivas classes e as chamadas desencadeadas a partir das mesmas.

Figura 4.9 – Resultados: Fluxo.txt (Instrumentos.cpp)

```
-> main ()
        -> _ZN5SoproC2Ev (Sopro)
                -> _ZN11InstrumentoC2Ev (Instrumento)
                -> _ZNK5Sopro4OqueEv (Sopro)
                -> _ZN5Sopro7ajustarEi (Sopro)
                -> _ZN11InstrumentoD2Ev (Instrumento)
        -> _ZN9PercussaoC2Ev (Percussao)
                -> _ZN11InstrumentoC2Ev (Instrumento)
                -> _ZNK9Percussao4OqueEv (Percussao)
                -> _ZN9Percussao7ajustarEi (Percussao)
                -> _ZN11InstrumentoD2Ev (Instrumento)
        -> _ZN9ComCordasC2Ev (ComCordas)
                -> _ZN11InstrumentoC2Ev (Instrumento)
                -> _ZNK9ComCordas4OqueEv (ComCordas)
                -> _ZN9ComCordas7ajustarEi (ComCordas)
                -> _ZN11InstrumentoD2Ev (Instrumento)
        -> _ZN6MetaisC2Ev (Metais)
                -> _ZN5SoproC2Ev (Sopro)
                        -> _ZN11InstrumentoC2Ev (Instrumento)
                        -> _ZNK6Metais4OqueEv (Metais)
                        -> _ZN6Metais7ajustarEi (Metais)
                        -> _ZN11InstrumentoD2Ev (Instrumento)
                -> _ZNK6Metais4OqueEv (Metais)
                -> _ZN6Metais7ajustarEi (Metais)
                -> _ZN5SoproD2Ev (Sopro)
                        -> _ZN11InstrumentoD2Ev (Instrumento)
        -> _ZN8MadeirasC2Ev (Madeiras)
                -> _ZN5SoproC2Ev (Sopro)
                        -> _ZN11InstrumentoC2Ev (Instrumento)
                        -> _ZNK8Madeiras4OqueEv (Madeiras)
                        -> _ZN8Madeiras7ajustarEi (Madeiras)
                        -> _ZN11InstrumentoD2Ev (Instrumento)
                -> _ZNK8Madeiras4OqueEv (Madeiras)
                -> _ZN8Madeiras7ajustarEi (Madeiras)
                -> _ZN5SoproD2Ev (Sopro)
                        -> _ZN11InstrumentoD2Ev (Instrumento)
```

Fonte: O Autor

4.2.4 Saída para o arquivo YoYo.txt (Instrumentos.cpp)

A Figura 4.10 apresenta uma parte do arquivo YoYo.txt onde existem chamadas polimórficas. Este trecho refere-se ao método construtor da classe “ComCordas”. Com base na Figura 4.10 é possível notar que há duas chamadas polimórficas dentro deste método. A partir do arquivo YoYo.txt é possível perceber as opções de chamada polimórfica e o comportamento caso o fluxo de uma das mesmas seja seguido. Neste caso, nenhum dos métodos polimórficos chama outras funções, mas se caso o fizesse, estes seriam explorados e exibidos neste arquivo.

Figura 4.10 – Resultados: YoYo.txt (Instrumentos.cpp)

```
-> _ZN9ComCordasC2Ev (ComCordas)
        -> _ZN11InstrumentoC2Ev (Instrumento)
        -> P (_ZNK5Sopro4OqueEv _ZNK9Percussao4OqueEv _ZNK9ComCordas4OqueEv _ZNK6Metais4OqueEv _ZNK8Madeiras4OqueEv)
                -> _ZNK5Sopro4OqueEv (Sopro)
                -> _ZNK9Percussao4OqueEv (Percussao)
                -> _ZNK9ComCordas4OqueEv (ComCordas)
                -> _ZNK6Metais4OqueEv (Metais)
                -> _ZNK8Madeiras4OqueEv (Madeiras)
        -> P (_ZN5Sopro7ajustarEi _ZN9Percussao7ajustarEi _ZN9ComCordas7ajustarEi _ZN6Metais7ajustarEi _ZN8Madeiras7ajustarEi)
                -> _ZN5Sopro7ajustarEi (Sopro)
                -> _ZN9Percussao7ajustarEi (Percussao)
                -> _ZN9ComCordas7ajustarEi (ComCordas)
                -> _ZN6Metais7ajustarEi (Metais)
                -> _ZN8Madeiras7ajustarEi (Madeiras)
        -> _ZN11InstrumentoD2Ev (Instrumento)
```

Fonte: O Autor

4.2.5 Saída para o arquivo ITU.txt (Instrumentos.cpp)

Neste caso, o arquivo ITU.txt gerado possui apenas os construtores das classes, visto que todos os relacionamentos hierárquicos presentes no código são relacionamentos de subtipo. Os construtores das classes geralmente estarão presentes no arquivo ITU.txt, visto que os construtores das classes herdadas geralmente são acionados pelas classes herdeiras a fim de definir o estado do objeto. O mesmo se aplica aos destrutores.

4.2.6 Saída para o arquivo SDA.txt (Instrumentos.cpp)

A saída gerada no arquivo SDA.txt visa identificar inconsistências entre métodos polimórficos. O arquivo de saída gerado possui todos os métodos polimórficos que diferem quando às chamadas ou definições. A Figura 4.11 apresenta um trecho do arquivo de saída gerado para o código “Instrumentos.ll”. Este arquivo salienta que os métodos “tocar” e “ajustar” podem apresentar anomalias causadas por divergências nas definições de variáveis de estado. A Figura 4.11 possui a saída fornecida apenas para o método “tocar”. Neste caso, quando o método chamado é do tipo “Metais” ou “Madeiras”, o método “tocar” define dois

atributos, “tipo” e “categoria”. Porém, quando o método chamado é do tipo “Sopro”, “Percussao” ou “ComCordas”, o método “tocar” define apenas o atributo “tipo”.

Figura 4.11 – Resultados: SDA.txt (Instrumentos.cpp)

```
Possível SDA nos seguintes metodos polimorficos entre si:
_ZNK5Sopro5tocarE4nota _ZNK9Percussao5tocarE4nota _ZNK9ComCordas5tocarE4nota _ZNK6Metais5tocarE4nota
_ZNK8Madeiras5tocarE4nota
Os metodos diferem quanto as DEFINIÇÕES, por favor conferir as informacoes abaixo:
    Metodo: _ZNK5Sopro5tocarE4nota
        Tipo: Definição -       Atributo: tipo      Objeto_nome: this_value     Objeto_classe: Sopro
    Metodo: _ZNK9Percussao5tocarE4nota
        Tipo: Definição -       Atributo: tipo      Objeto_nome: this_value     Objeto_classe:
        Percussao
    Metodo: _ZNK9ComCordas5tocarE4nota
        Tipo: Definição -       Atributo: tipo      Objeto_nome: this_value     Objeto_classe:
        ComCordas
    Metodo: _ZNK6Metais5tocarE4nota
        Tipo: Definição -       Atributo: tipo      Objeto_nome:        Objeto_classe:
        Tipo: Definição -       Atributo: categoria     Objeto_nome: this_value     Objeto_classe:
        Metais
    Metodo: _ZNK8Madeiras5tocarE4nota
        Tipo: Definição -       Atributo: tipo      Objeto_nome:        Objeto_classe:
        Tipo: Definição -       Atributo: categoria     Objeto_nome: this_value     Objeto_classe:
        Madeiras
```

Fonte: O Autor

4.2.7 Saída para o arquivo SDIH.txt (Instrumentos.cpp)

O arquivo SDIH.txt resultante da execução indica que o arquivo de entrada, “Instrumentos.cpp”, possui variáveis de estado de mesmo nome dentro da mesma hierarquia, porém pertencentes a classes distintas. O arquivo se saída gerado é exibido na Figura 4.12. É possível notar que o atributo “tipo” existe na classe pai “Instrumento” e nas classes filhas “Sopro”, “Percussao” e “ComCordas”.

Figura 4.12 – Resultados: SDIH.txt (Instrumentos.cpp)

```
SDIH - INCONSISTENCIA DE DEFINIÇAO DE ESTADO
Ocorre quando a inclusão de uma variável local em uma classe herdeira
sobrescreve a variável da classe herdada, deixando esta escondida, podendo
assim causar uma anomalia de fluxo de dados.
SAÍDA: Foram comparados todos os atributos entre classes herdeiras e
herdadas, todos os que possuem mesmo nome serão exibidos abaixo.
TESTE: Os atributos exibidos devem ser testados a fim de identificar se as
variáveis de estado estão sendo alteradas de acordo com a especificação.

----- Atributos de mesmo nome encontrados -----
Nome do Atributo: tipo
Este atributo existe nas classes:
Sopro
Instrumento

----- Atributos de mesmo nome encontrados -----
Nome do Atributo: tipo
Este atributo existe nas classes:
Percussao
Instrumento

----- Atributos de mesmo nome encontrados -----
Nome do Atributo: tipo
Este atributo existe nas classes:
ComCordas
Instrumento
```

Fonte: O Autor

4.2.8 Saída para o arquivo SDI.txt (Instrumentos.cpp)

A Figura 4.13 representa um trecho do arquivo de saída SDI.txt. Este trecho diz respeito à análise feita para duas classes, “Madeiras” e “Metais”. Neste arquivo, são exibidos todos os métodos polimórficos que definem as variáveis de estado presentes na hierarquia da classe. Neste caso, para as duas classes analisadas, só existem duas variáveis de estado que se enquadram nesta restrição: “ajuste” e “tipo”.

A primeira parte da Figura 4.13 indica que na análise feita a partir da classe “Metais” é possível constatar que os métodos “ajustar” da classe “Sopro” e “ajustar” da classe “Metais” definem a mesma variável de estado da classe “Sopro”,“ajuste”. A semântica de definição do atributo “ajuste”, neste caso, deve ser testada a fim de identificar anomalias de definição incorreta de estado.

Figura 4.13 – Resultados: SDI.txt (Instrumentos.cpp)

```
--------------------------------------------------------------------------
Análise a partir da Classe: Metais
A variável de estado ajuste da classe Sopro é definida nos métodos:
    _ZN5Sopro7ajustarEi:
       Tipo: Definição -      Atributo: ajuste    Atributo Classe: Sopro
    _ZN6Metais7ajustarEi:
       Tipo: Definição -      Atributo: ajuste    Atributo Classe: Sopro
A variável de estado tipo da classe Sopro é definida nos métodos:
    _ZNK5Sopro5tocarE4nota:
       Tipo: Definição -      Atributo: tipo      Atributo Classe: Sopro
    _ZNK6Metais5tocarE4nota:
       Tipo: Definição -      Atributo: tipo      Atributo Classe: Sopro
--------------------------------------------------------------------------
Análise a partir da Classe: Madeiras
A variável de estado ajuste da classe Sopro é definida nos métodos:
    _ZN5Sopro7ajustarEi:
       Tipo: Definição -      Atributo: ajuste    Atributo Classe: Sopro
    _ZN8Madeiras7ajustarEi:
       Tipo: Definição -      Atributo: ajuste    Atributo Classe: Sopro
A variável de estado tipo da classe Sopro é definida nos métodos:
    _ZNK5Sopro5tocarE4nota:
       Tipo: Definição -      Atributo: tipo      Atributo Classe: Sopro
    _ZNK8Madeiras5tocarE4nota:
       Tipo: Definição -      Atributo: tipo      Atributo Classe: Sopro
```

Fonte: O Autor

4.2.9 Saída para o arquivo ACB1.txt (Instrumentos.cpp)

O arquivo ACB1.txt visa tornar chamadas polimórficas feitas por construtores mais visíveis. Na Figura 4.14 há um trecho do arquivo ACB1.txt gerado através do arquivo ”Instrumentos.ll”. O foco do ACB1 é encontrar métodos polimórficos, chamados por construtores, que fazem uso de atributos de sua própria classe, como definido no Capítulo 2. Portanto, entre todos os métodos polimórficos encontrados, são descritos todos os usos relativos a atributos de sua própria classe.

No trecho do arquivo presente na Figura 4.14 é possível constatar que esta situação é encontrada em apenas dois pontos do código. Nas últimas linhas mostradas na Figura 4.14, observa-se que o método “ajustar” da classe “Madeiras” possui um uso para o atributo “categoria”, que pertence à sua própria classe. A mesma situação é encontrada para o método “ajustar” da classe “Metais”.

Figura 4.14 – Resultados: ACB1.txt (Instrumentos.cpp)

```
Construtor: _ZN6MetaisC2Ev  Classe: Metais
    Chamada polimórfica para: _ZNK5Sopro4OqueEv _ZNK9Percussao4OqueEv   _ZNK9ComCordas4OqueEv
    _ZNK6Metais4OqueEv  _ZNK8Madeiras4OqueEv

        Usos do Método: _ZNK5Sopro4OqueEv   Método Classe: Sopro
            Este método não possui usos que façam uso de atributos de sua própria classe!

        Usos do Método: _ZNK9Percussao4OqueEv   Método Classe: Percussao
            Este método não possui usos que façam uso de atributos de sua própria classe!

        Usos do Método: _ZNK9ComCordas4OqueEv   Método Classe: ComCordas
            Este método não possui usos que façam uso de atributos de sua própria classe!

        Usos do Método: _ZNK6Metais4OqueEv  Método Classe: Metais
            Este método não possui usos que façam uso de atributos de sua própria classe!

        Usos do Método: _ZNK8Madeiras4OqueEv    Método Classe: Madeiras
            Este método não possui usos que façam uso de atributos de sua própria classe!

    Chamada polimórfica para: _ZN5Sopro7ajustarEi    _ZN9Percussao7ajustarEi
    _ZN9ComCordas7ajustarEi _ZN6Metais7ajustarEi    _ZN8Madeiras7ajustarEi

        Usos do Método: _ZN5Sopro7ajustarEi Método Classe: Sopro
            Este método não possui usos que façam uso de atributos de sua própria classe!

        Usos do Método: _ZN9Percussao7ajustarEi Método Classe: Percussao
            Este método não possui usos que façam uso de atributos de sua própria classe!

        Usos do Método: _ZN9ComCordas7ajustarEi Método Classe: ComCordas
            Este método não possui usos que façam uso de atributos de sua própria classe!

        Usos do Método: _ZN6Metais7ajustarEi    Método Classe: Metais
            Tipo: Uso -     Atributo: categoria     Atributo Classe: Metais

        Usos do Método: _ZN8Madeiras7ajustarEi  Método Classe: Madeiras
            Tipo: Uso -     Atributo: categoria     Atributo Classe: Madeiras
```

Fonte: O Autor

4.2.10 Saída para o arquivo ACB2.txt (Instrumentos.cpp)

O arquivo ACB2.txt difere do ACB1.txt com relação ao seu foco de busca. Ambos os arquivos focam em encontrar chamadas polimórficas feitas através de construtores. Porém, como exemplificado no Capítulo 2, a fim de identificar a existência de um possível ACB2 deve-se focar em métodos polimórficos que podem ser chamados por construtores e que fazem uso de atributos de classes herdadas.

Um trecho do arquivo ABC2.txt gerado é representado na Figura 4.15. Nesta imagem nota-se que ambos os métodos, "ajustar" da classe "Metais" e "ajustar" da classe "Madeiras", possuem usos para a variável de estado "ajuste". Visto que esta variável de estado pertence à classe "Sopro" e esta classe é herdada por ambas as classes "Metais" e "Madeiras", esta situação pode acarretar em um comportamento anômalo de construção e deve ser testada.

Figura 4.15 – Resultados: ACB2.txt (Instrumentos.cpp)

```
Construtor: _ZN9PercussaoC2Ev   Classe: Percussao
    Chamada polimórfica para: _ZNK5Sopro4OqueEv _ZNK9Percussao4OqueEv   _ZNK9ComCordas4OqueEv
    _ZNK6Metais4OqueEv  _ZNK8Madeiras4OqueEv

        Usos do método: _ZNK5Sopro4OqueEv   Método Classe: Sopro
            Este método não possui usos que façam uso de atributos de alguma classe ancestral!

        Usos do método: _ZNK9Percussao4OqueEv   Método Classe: Percussao
            Este método não possui usos que façam uso de atributos de alguma classe ancestral!

        Usos do método: _ZNK9ComCordas4OqueEv   Método Classe: ComCordas
            Este método não possui usos que façam uso de atributos de alguma classe ancestral!

        Usos do método: _ZNK6Metais4OqueEv  Método Classe: Metais
            Este método não possui usos que façam uso de atributos de alguma classe ancestral!

        Usos do método: _ZNK8Madeiras4OqueEv    Método Classe: Madeiras
            Este método não possui usos que façam uso de atributos de alguma classe ancestral!

    Chamada polimórfica para: _ZN5Sopro7ajustarEi   _ZN9Percussao7ajustarEi
    _ZN9ComCordas7ajustarEi _ZN6Metais7ajustarEi    _ZN8Madeiras7ajustarEi

        Usos do método: _ZN5Sopro7ajustarEi Método Classe: Sopro
            Este método não possui usos que façam uso de atributos de alguma classe ancestral!

        Usos do método: _ZN9Percussao7ajustarEi Método Classe: Percussao
            Este método não possui usos que façam uso de atributos de alguma classe ancestral!

        Usos do método: _ZN9ComCordas7ajustarEi Método Classe: ComCordas
            Este método não possui usos que façam uso de atributos de alguma classe ancestral!

        Usos do método: _ZN6Metais7ajustarEi    Método Classe: Metais
            Tipo: Uso -     Atributo: ajuste    Atributo Classe: Sopro

        Usos do método: _ZN8Madeiras7ajustarEi  Método Classe: Madeiras
            Tipo: Uso -     Atributo: ajuste    Atributo Classe: Sopro
```

Fonte: O Autor

4.2.11 Saída para o arquivo IC.txt (Instrumentos.cpp)

A Figura 4.16 apresenta o arquivo de saída IC.txt gerado. Através deste arquivo é possível notar que alguns dos construtores não definem todos os atributos necessários. Neste caso, os construtores das classes “Percussao”, “Sopro” e “ComCorda” não definem o atributo “ajuste”. Outra divergência existe na classe “Instrumento”, a qual não define o atributo “tipo”.

Figura 4.16 – Resultados: IC.txt (Instrumentos.cpp)

```
--------------------------------------------------------------------------
O Construtor _ZN5SoproC2Ev da Classe: Sopro não possui declaração para os
atributos abaixo:
ajuste

--------------------------------------------------------------------------
O Construtor _ZN11InstrumentoC2Ev da Classe: Instrumento não possui
declaração para os atributos abaixo:
tipo

--------------------------------------------------------------------------
O Construtor _ZN9PercussaoC2Ev da Classe: Percussao não possui declaração
para os atributos abaixo:
ajuste

--------------------------------------------------------------------------
O Construtor _ZN9ComCordasC2Ev da Classe: ComCordas não possui declaração
para os atributos abaixo:
ajuste

--------------------------------------------------------------------------
O Construtor _ZN6MetaisC2Ev da Classe: Metais possui declaração para
todos os atributos da classe!

--------------------------------------------------------------------------
O Construtor _ZN8MadeirasC2Ev da Classe: Madeiras possui declaração para
todos os atributos da classe!
```

Fonte: O Autor

4.3 Análise dos resultados

Quando aplicada ao próprio código, a ferramenta gerou resultados que evidenciam a existência das anomalias ITU e IC. O código original na linguagem C++ da ferramenta possui 2.015 linhas e seu respectivo arquivo .ll possui 116.286 linhas. Este grande número de linhas acarretou em um grande impacto no tempo de execução. O tempo total de execução foi de 11.162,460 segundos (aproximadamente 3 horas).

O tempo de execução foi alto não só devido ao tamanho do arquivo de entrada, como também por outros fatores que influem na complexidade da ferramenta, tais como: quantidade de classes, quantidade de métodos, número de chamadas existentes e número de definições e usos existentes. O código da ferramenta possui 7 classes e 98 métodos e uma grande quantidade de definições, usos e chamadas.

O arquivo "Instrumentos.cpp" utilizado como entrada possui 6 classes e apenas 27 métodos. Os resultados gerados para seu código indicam a existência das anomalias: SDA, SDIH, SDI, ACB1, ACB2 e IC. O código C++ deste arquivo possui 91 linhas e seu respectivo

arquivo .ll possui 1.179 linhas. Neste caso o tempo total de execução foi de 18,762 segundos. Isso ocorreu, pois, além do código possuir um número inferior de linhas, classes e métodos, os relacionamentos entre seus métodos não são tão complexos quanto no caso anterior.

5 CONCLUSÃO

Este trabalho foi desenvolvido no intuito de salientar a importância da automatização de testes no cotidiano, tendo em vista a complexidade dos relacionamentos dos conceitos introduzidos pela orientação a objetos. Com base neste objetivo, foi desenvolvido um protótipo capaz de identificar estruturas relacionadas à orientação a objetos existentes em uma representação intermediária do código, que pode ser compartilhada por outras linguagens de programação.

A elaboração da abordagem proposta foi possível através do uso de LLVM, que gera uma versão intermediária do código que serve como arquivo de entrada para o protótipo desenvolvido que, por sua vez, identifica: classes, métodos, atributos, tabelas virtuais, objetos, chamadas realizadas entre métodos, e definições e usos de atributos e objetos. Além desta etapa de identificação, o protótipo realiza uma etapa de análises estruturais com o propósito de detectar estruturas que possam acarretar nos comportamentos anômalos relacionados à programação orientada a objetos descritos por Amman;Offut (2008).

Visto que a análise fundamenta-se em uma linguagem intermediária, alguns desafios de interpretação fizeram-se presentes ao longo da etapa de desenvolvimento. Alguns destes desafios mostraram-se limitadores, como é o caso da identificação exata do nome original dos métodos, que não foi possível, pois esta é uma combinação temporária baseada na sequência do restante da estrutura do código, no nome da classe do método e no nome atribuído a este método no código original. Embora seja possível distinguir o método desejado a partir do conhecimento do código original, não é uma forma elegante de exibi-lo ao usuário.

Tendo em vista o grande número de validações necessárias ao longo das iterações de leitura do arquivo de entrada, o tempo de processamento mostrou-se um fator de grande impacto no estado final do protótipo. Este impacto pode ser amenizado através do uso de *threads* a fim de paralelizar as validações necessárias, visto que não há dependência entre as mesmas.

Observa-se a possibilidade de trabalhos futuros através do desenvolvimento das funcionalidades descritas nas seções 3.4.1 e 3.4.2, ou seja, através do desenvolvimento da análise de pares DU e da inclusão de todos os métodos importados pelo código original, e presentes no arquivo de entrada, na etapa de análise de estruturas. Outra possibilidade seria a introdução do uso de programação concorrente na etapa de validação de estruturas a fim de reduzir o tempo de execução.

REFERÊNCIAS

AMMAN, Paul; OFFUT, Jeff. **Introduction to Software Testing**. New York: Cambridge University Press, 2008.

DELAMARO, Márcio Eduardo; MALDONADO, José Carlos; JINO, Mario. **Introdução ao teste de software**. Rio de Janeiro: Elsevier Editora Ltda, 2007.

PEZZÈ, Mauro; YOUNG, Michal. **Teste e Análise de Software**: processos, princípios e técnicas. Porto Alegre: Bookman; 2008.

The LLVM Compiler Infrastructure. Disponível em: <llvm.org>. Acesso em: 15 Mai. 2016.

Clang: a C language family frontend for LLVM. Disponível em: <clang.llvm.org>. Acesso em: 15 Mai. 2016.

AMMAN, Paul; OFFUT, Jeff. **Introduction to Software Testing, Chapter 7.1, Engineering Criteria for Technologies**. 2013. Disponível em: <https://cs.gmu.edu/~offutt/softwaretest/powerpoint/>. Acesso em: 15 mai. 2016.

Qian YANG, J. Jenny LI, David M. WEISS. A Survey of Coverage Based Testing Tools. **The Computer Journal**, volume 52 (5), p. 589-59, Agosto 2009.

Sneha SHELKE, Sangeeta NAGPURE. The Study of Various Code Coverage Tools. **International Journal of Computer Trends and Technology (IJCTT)**, volume 13, número 1, Julho 2014.