Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Análise de Evolução de Linhas de Produtos de Software

Herval Alexandre Dias Hubner

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Orientador
Prof. Dr. Vander Ramos Alves

Brasília
2023

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Análise de Evolução de Linhas de Produtos de Software

Herval Alexandre Dias Hubner

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Prof. Dr. Vander Ramos Alves (Orientador)
CIC/UnB

Prof. Dr. Leopoldo Motta Teixeira
CLN/UFPE

Prof.a Dr.a Genaína Nunes Rodrigues
CIC/UnB

Prof. Dr. Ricardo Pezzuol Jacobi
Coordenador do Programa de Pós-graduação em Informática

Brasília, 26 de outubro de 2023

Dedicatória

Dedico esse trabalho a:

- meus pais Wilmar e Rosaly, a quem devo a minha vida;

- minha esposa Joedna e a meus filhos, Ester e João Alexandre, que sempre acreditaram em mim e me deram forças;

- minha avó Glaucia que orou por mim todos os dias e

- meus irmãos Wilmar Ernesto, Augusto e Amanda que sempre se alegram com minhas conquistas.

Agradecimentos

Agradeço a Deus, autor e consumador da minha fé, que cuidou de mim e me guiou em todos os momentos.

Ao ilustre professor Dr. Vander Alves, por dar sentido à palavra orientador e, além de se preocupar comigo, viu o meu potencial e me levou a superar minhas limitações.

À Secretaria de Educação do Estado do Piauí, por me apoiar durante o curso.

Ao CETI Dr. Dionísio Rodrigues Nogueira e toda a sua equipe, por me apoiarem e compreenderem minha ausência.

Ao Prof. Dr. Leopoldo Motta Teixeira, por todas as suas colaborações para a execução do trabalho.

Acknowledgements

I thank God, the author and finisher of my faith, who took care of me and guided me at all times.

To the illustrious professor Dr. Vander Alves, for giving meaning to the word advisor and, in addition to worrying about me, he saw my potential and led me to overcome my limitations.

To the Department of Education of the State of Piauí, for supporting me during the course.

To CETI Dr. Dionísio Rodrigues Nogueira and his entire team, for supporting me and understanding my absence.

To Prof. Dr. Leopoldo Motta Teixeira, for all his collaboration in carrying out the work.

Resumo

No cenário atual da engenharia de software, as Linhas de Produtos de Software (LPS) destacam-se como uma abordagem fundamental para enfrentar os desafios da customização em massa. LPS permitem a construção de soluções individuais com base em componentes reutilizáveis, proporcionando eficiência e flexibilidade no desenvolvimento de software. As LPS são fundamentais para melhorar a produtividade e a qualidade no desenvolvimento de software, graças à reutilização de componentes e rápida adaptação a novos requisitos. A variabilidade é central em LPS, facilitando a adaptação a diversas situações de produtos através de recursos que podem ser ativados. A análise de LPS é crucial para identificar requisitos comuns e variantes, mas enfrenta desafios de falta de dados concretos e limitações de tempo. No entanto, a análise de LPS enfrenta desafios, como a escassez de estudos empíricos caracterizando e detalhando a evolução de LPS. Neste trabalho, desenvolvemos a ferramenta ASTool (software de análise de alterações na árvore de sintaxe abstrata) para examinar diversas Linhas de Produto de Software e assim, caracterizar a evolução das mesmas. Os resultados obtidos nesta análise revelam que, no que diz respeito à média de profundidade das alterações na Abstract Syntax Tree (AST), as modificações ocorrem em níveis superficiais, ou seja, próximas à raiz da árvore de sintaxe. Em relação à média de arquivos alterados por commit, observou-se uma baixa quantidade de arquivos modificados por commit. Quanto à média de lacunas (gaps) entre as linhas alteradas no código, os valores obtidos indicam uma baixa frequência de alterações. Os resultados deste estudo podem influenciar a decisão de utilizar ou não a técnica de memoização para melhorar a eficácia das análises.

Palavras-chave: Linhas de Produto de Software, variabilidade, análise de evolução, ASTool.

Abstract

Command and Control (C2), in its historical origin, is related to the application of classic military strategies where there was a single centralized command and an inflexible chain of command between the elements that composed the acting forces. C2 is not an end in itself, but a process whose goal is to optimize the application of resources in order to accomplish a mission. However, in a modern C2 context, the dynamism of the mission, the team and the environment is a necessary assumption and, thus, the organization of the team to accomplish a mission becomes a challenge requiring constant adaptations. This ability to adapt to new circumstances characterizes C2 Agility. However, the state-of-the-art does not assess how this ability is affected by the choices of C2 approach, represented by the level of information spread, by the organization of the team and by the capacity of decision making. In addition, recent works do not consider the measurement of Quality Attributes (QA), which makes the models and simulations poorly adherent to the reality of missions, where at least the cost can be an obstacle to their achievement. To address these issues, we apply concepts of Self-Adaptive Systems (SAS) with an approach using Dynamic Software Product Lines (DSPL) to represent the elements that make up the C2 System and that are organized into teams. Relying on configuration and coordination, we propose two models that seek to ensure C2 agility. These models provide for the choice of the C2 approach, combined with the ability to reconfigure the team members in order to ensure agility to face the changes in circumstances that may occur. To evaluate the proposed models, we perform a set of simulations to indicate the agility level obtained by the approach and we apply questionnaires to C2 domain experts to validate models' usability and compatibility with realistic scenarios faced by domain experts.

Keywords: Software Product Lines, variability, evolution analysis, ASTool

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Linhas de Produto de Software (LPS) são conjuntos de soluções de software que compartilham uma base comum de código e são adaptadas para atender a diferentes necessidades dos usuários. Pohl et al. [23] definem LPS como um paradigma de desenvolvimento de software que envolve a criação de uma família de produtos relacionados, aproveitando a reutilização de componentes de software comuns, a fim de atender a diferentes necessidades dos usuários ou requisitos de mercado. As LPS são importantes porque permitem a criação de soluções de software personalizadas e eficientes, reduzindo o tempo e o custo de desenvolvimento de novos produtos, o que é afirmado por Krueger [16] ao dizer que LPS são fundamentais para melhorar a produtividade e a qualidade no desenvolvimento de software, permitindo a reutilização de componentes e a rápida adaptação a novos requisitos.

A variabilidade é central em LPS, pois permite a adaptação a diferentes contextos de produtos. Para Apel et al. [5], variabilidade é a capacidade de um sistema de software de oferecer diferentes características, comportamentos ou configurações para atender a requisitos específicos de diferentes produtos ou variantes da linha. A variabilidade é implementada por meio de *features*, que são unidades de funcionalidade que podem ser ativadas ou desativadas de acordo com as necessidades do usuário.

Segundo Thüm et al. [26], a LPS é uma família de produtos de software que compartilham um conjunto comum de recursos. A análise de LPS é importante para identificar oportunidades de melhoria e resolver problemas que podem surgir durante a evolução do sistema. Por exemplo, uma empresa de software que desenvolve uma linha de produtos de gerenciamento de projetos, ao analisar seus processos, percebe que todos os aplicativos da linha compartilham uma necessidade comum: a geração de relatórios. A empresa decide que, em vez de desenvolver um módulo de geração de relatórios separado para cada aplicativo, seria mais eficiente e econômico criar um módulo centralizado que pudesse ser compartilhado por todos os aplicativos. Essa mudança permitiria que a equipe de desen-

volvimento economizasse tempo e recursos, além de melhorar a experiência do usuário, pois os clientes teriam acesso a um único padrão de relatórios.

No entanto, a análise de LPS enfrenta desafios, como a falta de dados concretos de evolução e a limitação de tempo. Essa ausência de dados torna difícil compreender como esses sistemas mudam, crescem e se adaptam, prejudicando a capacidade de realizar análises profundas e informadas.

Apesar da evolução de LPS ser um tópico de pesquisa ativo, Marques et al. [18], observa-se uma escassez de estudos empíricos caracterizando e detalhando a evolução de LPS. Essa carência de evidências concretas dificulta a compreensão das tendências e padrões de evolução das LPS, o que é fundamental para a melhoria contínua e otimização das técnicas de análise.

Outro obstáculo é que a evolução da LPS é complexa devido à natureza variável desses sistemas. À medida que novas *features* são adicionadas, requisitos modificados e bugs corrigidos, a estrutura e o comportamento da LPS podem se tornar cada vez mais intrincados. Isso pode dificultar a compreensão das mudanças ao longo do tempo e a avaliação de seu impacto nas análises subsequentes. A complexidade da LPS em evolução exige uma abordagem cuidadosa e sistemática.

Outro ponto é que a falta de ferramentas específicas e eficazes para a análise de evolução em LPS é uma preocupação recorrente. Embora existam diversas ferramentas de análise de software disponíveis, poucas são direcionadas especificamente para a análise de LPS, levando a lacunas na capacidade de identificar e avaliar mudanças significativas ao longo do tempo. De acordo com Gamma et al. [9], a disponibilidade de ferramentas adequadas é fundamental para a eficácia da análise de software em contextos complexos como Linhas de Produto de Software.

É necessário entender como a evolução de LPS impacta suas características e desempenho para melhorar as soluções de software.

Para tratar esse problema, este trabalho contribui com um estudo de caso, novas métricas e uma ferramenta visando caracterizar a evolução de LPS. O estudo de caso analisa a evolução de algumas LPS, utilizando métricas para avaliar a profundidade das alterações na Abstract Syntax Tree (AST), a média de arquivos alterados por commit e a média de gaps (linhas não alteradas) por arquivo. A ferramenta desenvolvida ASTool (ferramenta de análise de AST), disponível em https://sites.google.com/view/astools/in%C3%ADcio, permite a visualização e análise dos dados coletados, facilitando a identificação de padrões e oportunidades de otimização.

Como resultado, observamos alguns padrões comuns na evolução das LPS estudadas, como a profundidade das alterações na AST e a média de arquivos alterados por commit. No entanto, também foram identificadas abordagens distintas de evolução nas LPS estu-

dadas, destacando a importância de compreender como a evolução das mesmas impacta suas características e desempenho.

O restante do trabalho é organizado da seguinte forma. No Capítulo 2, apresentamos a fundamentação teórica sobre os conceitos de variabilidade e evolução de LPS. No Capítulo 3, descrevemos em detalhes os métodos e técnicas utilizados, além de apresentarmos as estratégias e instrumentação utilizados para a análise de evolução em LPS, bem como os resultados obtidos. No Capítulo 4, destacamos alguns trabalhos relacionados à evolução das LPS. E finalmente, na quinta e última seção, apresentamos as conclusões do trabalho e suas contribuições para a área de análise de evolução de linhas de produto de software.

Capítulo 2

Fundamentação Teórica

Para um conhecimento mais aprofundado do problema e da solução que discutimos neste trabalho, serão apresentados conceitos relacionados à linha de produtos de software, variabilidade, evolução de linhas de produto de software, bem como a maneira como a ferramenta ComAn conduz análises em LPS e por fim o conceito de Árvores Sintáticas Abstratas.

2.1 Linhas de Produto de Software e Variabilidade

As Linhas de Produto de Software têm emergido como uma abordagem estratégica e eficaz para o desenvolvimento de software. O conceito de LPS remonta à década de 90 e tem sido amplamente adotado em diversas indústrias. De acordo com Clements and Northrop [7], uma LPS é uma abordagem que permite a criação eficiente de uma família de produtos de software relacionados, que compartilham um conjunto comum de características essenciais, enquanto também possuem variações específicas para atender às necessidades individuais dos clientes.

Uma característica essencial das LPS é a sua capacidade de reutilização. Conforme definido por Czarnecki and Eisenecker [8], a reutilização é fundamental nas LPS, pois permite que artefatos de software sejam compartilhados entre os produtos da LPS, reduzindo o esforço de desenvolvimento e melhorando a consistência e a qualidade dos produtos. Além disso, as LPS são caracterizadas por um modelo de domínio bem definido, que descreve as características comuns e as variações esperadas entre os produtos da linha. Essa modelagem de domínio é essencial para guiar o desenvolvimento de produtos específicos da linha.

As LPS oferecem uma série de benefícios significativos. Elas podem melhorar a produtividade do desenvolvimento de software, uma vez que a reutilização de artefatos reduz a duplicação de esforços. Além disso, as LPS possibilitam uma resposta ágil às mudan-

ças nas demandas do mercado, pois as variações podem ser facilmente incorporadas aos produtos existentes da linha.

A relação entre a eficiência na gestão da variabilidade de software e os benefícios das Linhas de Produto de Software (LPS) é fundamental para compreender o sucesso dessa abordagem, conforme enfatizado por Metzger and Pohl [20] e Jaring and Bosch [11]. Assim, a capacidade de adaptação eficiente dos produtos de software, destacada nas LPS, está intrinsecamente ligada ao conceito de variabilidade de software.

Segundo Metzger and Pohl [20], a variabilidade de software refere-se à capacidade dos sistemas ou artefatos de software de serem estendidos, alterados, personalizados ou configurados com eficiência. Gurp et al. [10], por sua vez, definem que a variabilidade é a capacidade de mudar ou customizar um sistema de software. Eles também enfatizam a importância de compreender onde a mudança deve ser planejada e as opções possíveis em situações particulares, especialmente quando a arquitetura é usada para diferentes versões de produto.

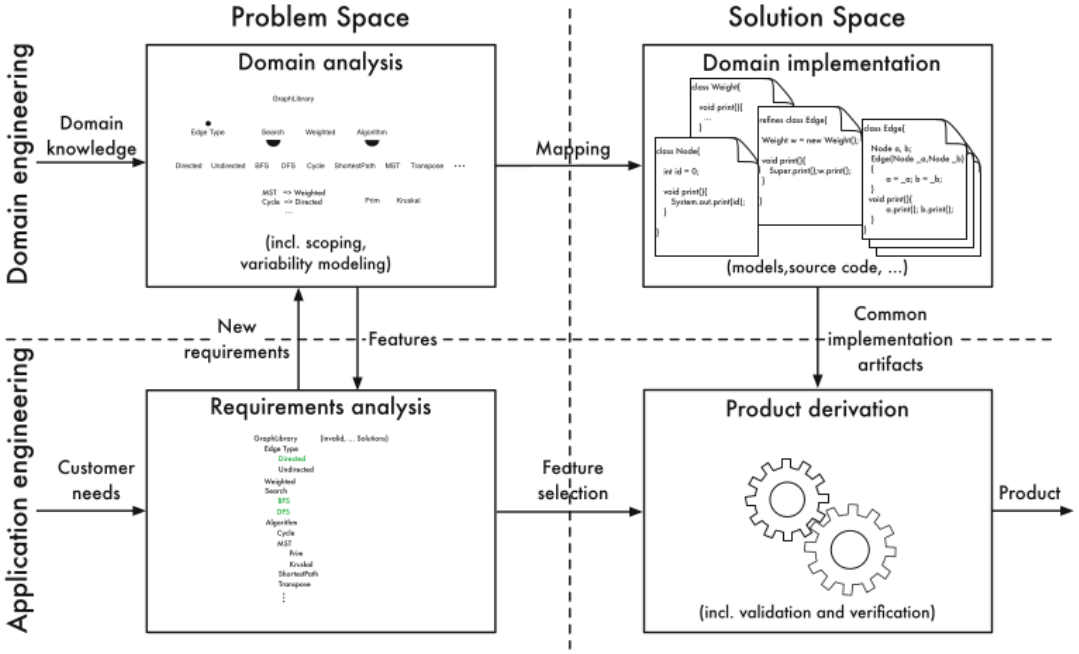

Figura 2.1: Processo de análise de profundidade de alterações na AST.

A Figura 2.1 demonstra uma linha de produtos de software na qual os requisitos são um conjunto de entradas de software que são recebidos. A variabilidade ocorre na linha de produção, sendo que cada produto é definido por escolhas relacionadas a requisitos opcionais e variáveis no modelo de decisão. A variabilidade representa um dos principais fatores que determinam a LPS e sua evolução.

Para preservar o comportamento consistente dos produtos de uma LPS, frequentemente é necessário realizar análises abrangentes que considerem diversos artefatos, como:

- *Feature Model*: segundo Apel et al. [5], feature model documenta as features de uma linha de produtos e seus relacionamentos. Lembrando que feature é uma característica ou comportamento visível ao usuário final de um sistema de software Czarnecki and Eisenecker [8]. Como destacado por Apel et al. [5], os *Feature Models* constituem a espinha dorsal das Linhas de Produtos de Software, possibilitando uma gestão eficiente da variabilidade e a criação de produtos diversificados.

- *Core Assets*: compreendem componentes, código-fonte, bibliotecas e outros recursos compartilhados que são utilizados de forma consistente em toda a linha de produtos. Facilitam a reutilização de funcionalidades e características comuns, economizando tempo e esforço no desenvolvimento de produtos específicos da linha. Ao mesmo tempo, garantem a manutenção da integridade e consistência do software. Em consonância com a observação de Apel et al. [5], os core assets constituem o alicerce sólido sobre o qual as diferentes variantes da linha de produtos são construídas, facilitando, assim, tanto a reutilização quanto a gestão da variabilidade de forma eficaz.

- *Configuration Knowledge*: desempenha um papel importante na escolha e combinação dos elementos do sistema para a concepção de produtos específicos, com o intuito de garantir a eficiência e consistência. Conforme destacado por Apel et al. [5], o *Configuration Knowledge* assume uma posição essencial dentro das Linhas de Produtos de Software, fornecendo informações críticas acerca das alternativas de configuração disponíveis e assegurando a obtenção de uma configuração precisa e eficaz dos produtos pertencentes à referida linha.

2.2 Evolução de Linhas de Produto de Software

De acordo com Bastarrica et al. [6], a evolução da LPS é como um processo de crescimento e mudança. Ela pode ser motivada por várias coisas, como as necessidades dos clientes, as novas tecnologias ou a concorrência. Os processos para a evolução da LPS são bem definidos, mas ainda precisam de muito esforço humano. Isso significa que, embora existam ferramentas e procedimentos para ajudar, ainda é preciso que pessoas analisem dados, tomem decisões e implementem mudanças. Os autores também explicam que existem alguns desafios para a evolução da LPS. Um deles é a falta de consenso sobre como ela deve acontecer. Diferentes pessoas podem ter prioridades e objetivos diferentes, o que pode levar a conflitos e atrasos. Outro desafio é a necessidade de mais pesquisas para desenvolver métodos e ferramentas que suportem a evolução da LPS de maneira

mais comparável. Isso ajudará a garantir que diferentes LPSs possam ser comparadas e avaliadas com base em critérios comuns.

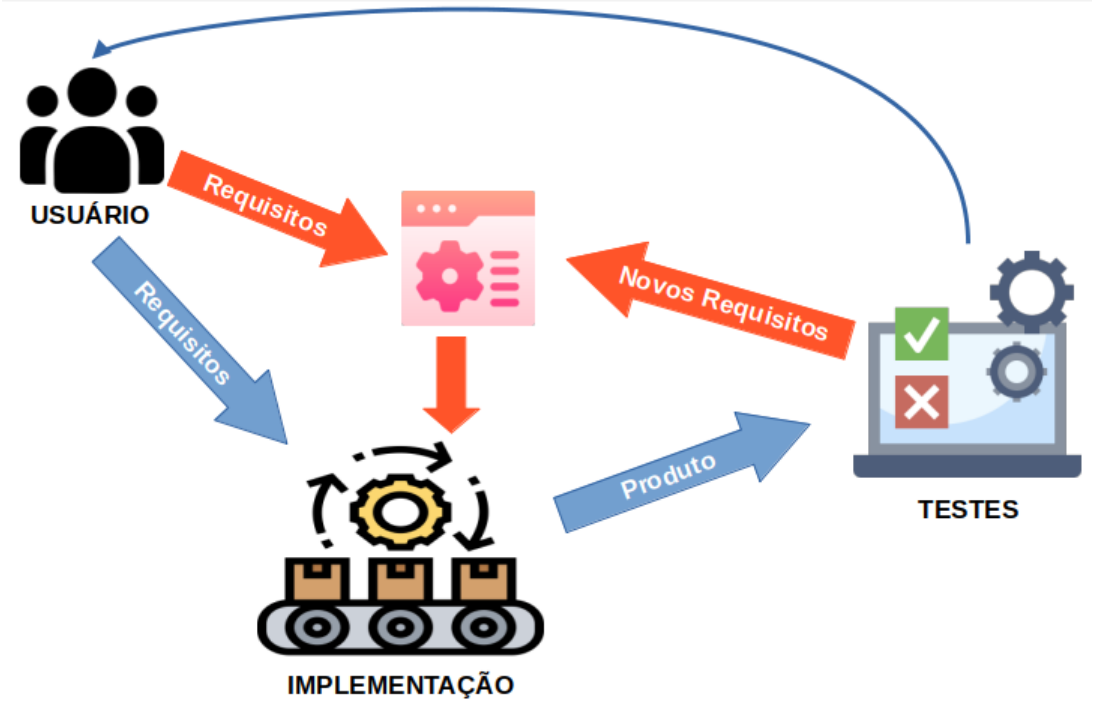

Figura 2.2: Processo de desenvolvimento contínuo na LPS.

Como ilustrado na 2.2, a evolução das linhas de produtos de software representa um processo contínuo de desenvolvimento de produtos de software, com o propósito de atender a diversos requisitos e aprimorar as funcionalidades existentes.

É importante assegurar que as modificações não afetem adversamente o comportamento dos produtos existentes. Nesse contexto, a análise de linhas de produtos de software assume um papel importante, uma vez que fornece informações valiosas sobre como as funcionalidades e recursos existentes podem ser aprimorados e otimizados.

Conforme observado por Thüm et al. [26] as Linhas de Produtos de Software representam um desafio substancial para as técnicas de análise tradicionais, tais como a verificação de tipos, a verificação de modelos e a prova de teoremas, à medida que buscam garantir a correção e confiabilidade do software. Para Meinicke et al. [19], as análises relacionadas às Linhas de Produtos de Software enfrentam o desafio de lidar com um vasto conjunto de produtos que pode crescer de maneira exponencial em função do número de funcionalidades, levando, assim, à necessidade de diversas abordagens terem sido propostas para a análise eficiente das Linhas de Produtos de Software.

De acordo com Thüm et al. [26] uma abordagem sugerida para minimizar o esforço de análise é a utilização de análises baseadas em famílias (Family-based analyses). Esta estratégia considera todas as variantes configuráveis de um programa como uma única

unidade de análise, em vez de examinar separadamente cada variante individualmente. Esta abordagem visa evitar a duplicação de esforços, uma vez que as partes comuns são analisadas apenas uma vez, e a análise se concentra exclusivamente nas discrepâncias existentes entre as diferentes variantes.

Nesse contexto, Kroher et al. [14] desenvolveram uma ferramenta que extrai e analisa dados de evolução dos repositórios de LPS, permitindo uma investigação sobre a frequência das alterações nas informações de variabilidade em diferentes tipos de artefatos.

2.3 Ferramenta ComAn

As Árvores Sintáticas Abstratas, frequentemente abreviadas como ASTs, são uma estrutura de dados hierárquica que representa a estrutura gramatical de um programa de computador, linguagem de programação ou qualquer outro texto fonte. As ASTs são utilizadas para representar a estrutura semântica do código-fonte, fornecendo uma visão simplificada e abstrata do programa.

Com o propósito de elucidar a frequência com que os desenvolvedores efetuam modificações nas informações de variabilidade presentes em diversos tipos de artefatos, Kroher et al. [14] conceberam a ferramenta denominada ComAn. Esta ferramenta se destina à extração e análise de dados relativos à evolução de repositórios de LPS, conforme ilustrado na Figura 2.3.

Figura 2.3: Abordagem da análise da ComAn.

A primeira fase operacionalizada pela referida ferramenta consiste na extração de commits. Commits representam modificações efetuadas no código-fonte com o propósito de aprimorar ou corrigir um produto de software. Essas alterações são arquivadas em repositórios e podem abranger correções, implementações de novas funcionalidades ou melhorias. Além disso, desempenham um papel essencial na rastreabilidade das mudanças ao longo do tempo, facultando aos desenvolvedores a capacidade de reverter alterações quando julgarem necessário.

Após a extração dos commits, a ferramenta procede à análise das modificações de linhas em relação ao tipo de informação afetada em cada categoria de artefato. Os tipos de artefatos examinados incluem:

- Mudanças de Código (*Code*): englobam as alterações realizadas no código-fonte do software;
- Mudanças no Construtor (*Build*): abarcam as modificações nas diretrizes de compilação (*Configuration Knowledge*);
- Mudanças no Modelo de Variabilidade (*Variability Model*): referem-se à informação de variabilidade em si.

A ferramenta ComAn utiliza expressões regulares para efetuar a busca, em cada linha, de modificações referentes às marcações de adição (+) e remoção (-), como exemplificado na Figura 2.4. Ao concluir o processo de análise, a ferramenta gera relatórios e gráficos contendo os resultados, os quais evidenciam a quantidade de modificações ocorridas.

No âmbito deste trabalho, ampliamos o escopo da análise, inspirando-nos na abordagem da ComAn, para desenvolver uma ferramenta que, por sua vez, concentra-se na análise das Árvores Sintáticas Abstratas (ASTs) dos artefatos, buscando conhecer a profundidade das alterações introduzidas ao longo do ciclo de vida do software.

2.4 Árvores Sintáticas Abstratas (ASTs

As Árvores Sintáticas Abstratas, frequentemente abreviadas como ASTs, são uma estrutura de dados hierárquica que representa a estrutura gramatical de um programa de computador, linguagem de programação ou qualquer outro texto fonte. As ASTs são utilizadas para representar a estrutura semântica do código-fonte, fornecendo uma visão simplificada e abstrata do programa.

Ranta [24], afirma que as árvores de sintaxe abstratas são como o mapa de um compilador. Elas são criadas pelo analisador, que "lê"o código-fonte e identifica sua estrutura. As árvores de sintaxe abstratas são então usadas para realizar as principais etapas da compilação, como:

- *Type Checking*: envolve a análise da sintaxe e da estrutura do código para detectar quaisquer erros ou inconsistências de tipo.
- *Code Generator*: faz a conversão de código de alto nível em instruções legíveis por máquina. Este processo, apesar de pouco comum atualmente, envolve a tradução da árvore de sintaxe abstrata em código executável, que pode ser compreendido pelo computador.

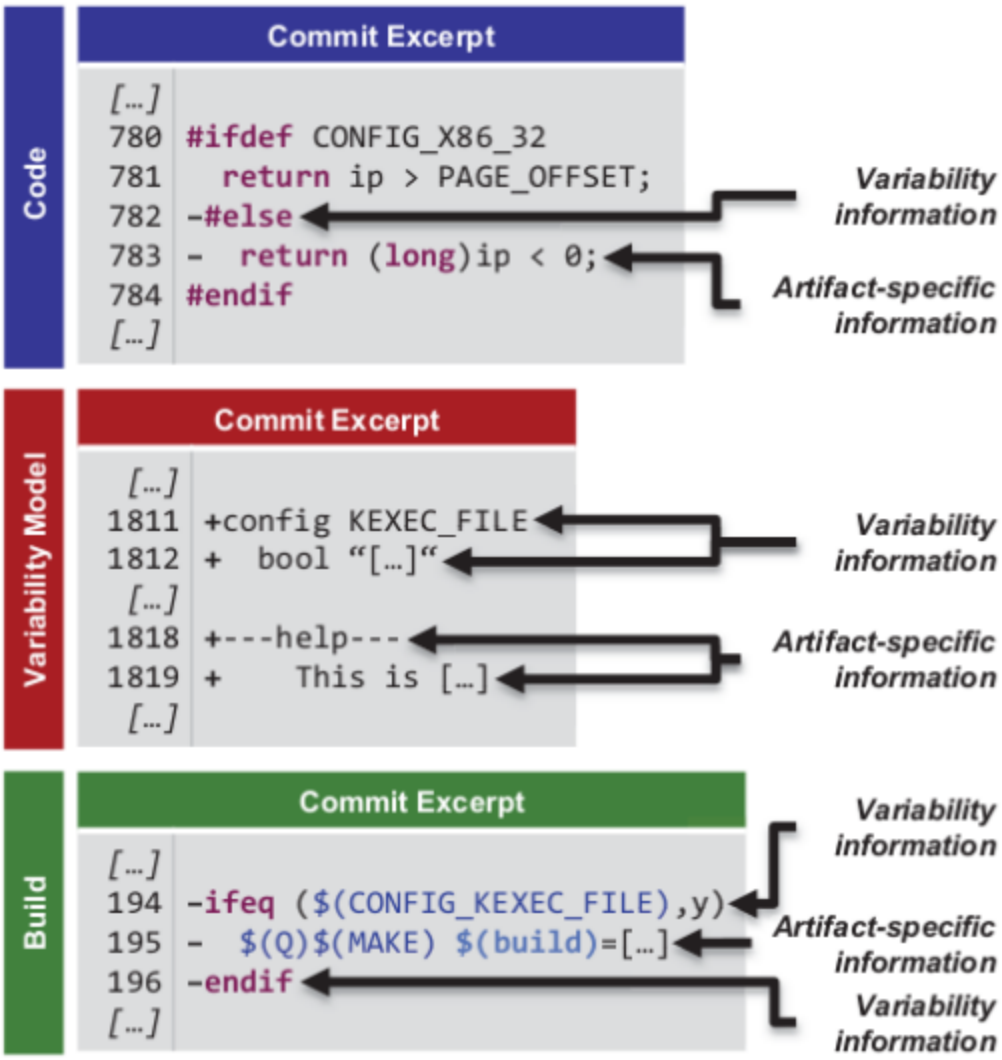

Figura 2.4: Classificação de alterações da ComAn.

A Figura 2.5 traz um trecho de código em linguagem Python junto a uma representação simplificada da AST correspondente. Esta AST captura a estrutura semântica essencial do código Python, mostrando a declaração de função, os parâmetros, a atribuição e a declaração de retorno.

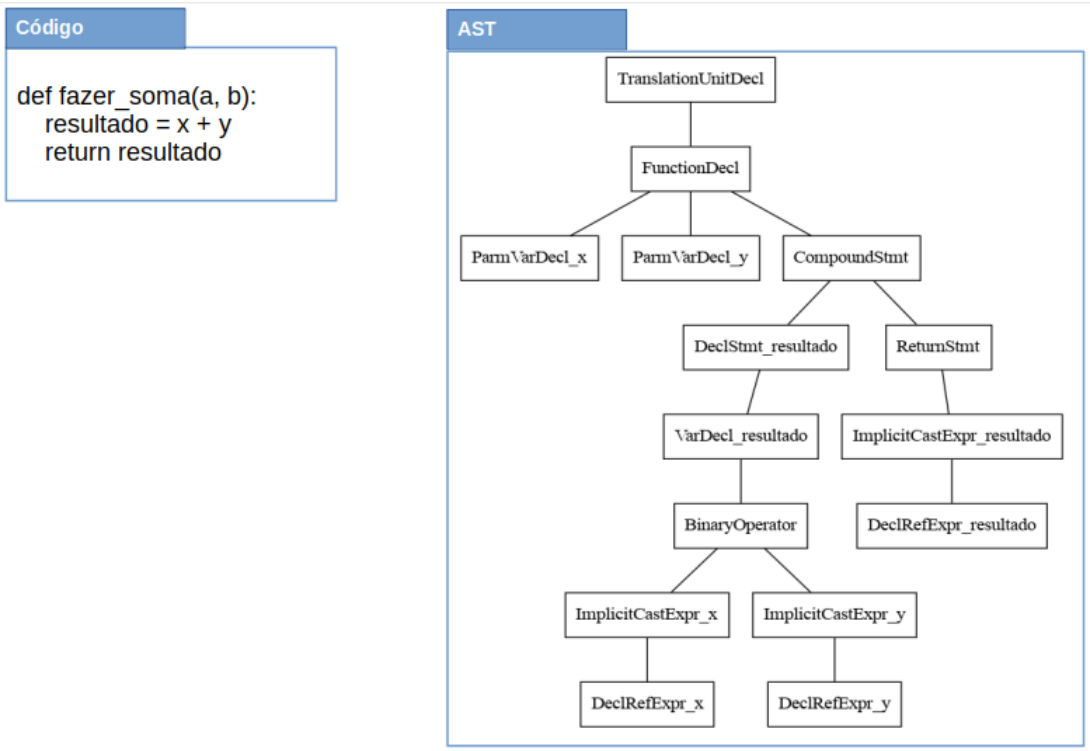

Figura 2.5: Exemplo de Árvore Sintática Abstrata.

Capítulo 3

Estudo de Caso

Conforme destacado por Wohlin et al. [30], estudos de caso não têm a mesma capacidade de estabelecer relações causais que os experimentos controlados, mas oferecem uma valiosa compreensão dos fenômenos estudados em seu contexto real. Nesse contexto, essa pesquisa contribui com (1) um estudo de caso que visa examinar diferentes LPS e (2) a ferramenta de análise, a ASTool.

Para realizar as análises, definimos os repositórios de LPS apropriadas para o estudo. Em seguida, procedemos ao download desses repositórios e conduzimos a análise por meio da ferramenta ASTool, que foi empregada de maneira similar ao que foi realizado com a ferramenta ComAn na abordagem de Kroher et al. [14]. Através dessa análise, foram gerados diversos arquivos contendo informações sobre as modificações identificadas nos arquivos e commits realizados nos repositórios selecionados.

3.1 Definição

A metodologia adotada para conduzir o estudo de caso, cujo objetivo é caracterizar a evolução da LPS usando a abordagem Goal-Question-Metric (GQM) Van Solingen et al. [28], que é uma abordagem eficaz e amplamente reconhecida para a avaliação da qualidade de software.

Para atingir o objetivo, implementamos a ASTool para realizar análises em LPS explorando o modo como as LPS evoluem. As perguntas e métricas correspondentes são as seguintes:

Questão de Pesquisa 1 (QP1): Qual é a intensidade da evolução da linha de produtos de software em relação ao número de alterações e à frequência de atualizações?

Métrica 1: Média de Arquivos Alterados por Commits é um indicador da quantidade de mudanças que ocorrem em uma LPS. Uma média mais alta sugere uma evolução mais intensa, indicando que as modificações são frequentes e impactam um grande número

de arquivos. Isso é um indicativo de uma LPS que está em constante desenvolvimento, com requisitos em constante evolução e adaptação a novas demandas. Por outro lado, uma média mais baixa sugere uma evolução menos intensa, indicando que as alterações ocorrem de forma mais esporádica e em um conjunto menor de arquivos. Isso sugere que a LPS está em um estado de maior estabilidade, onde as mudanças acontecem de forma menos frequente e com menor impacto nos componentes existentes.

Métrica 2: Na Média de Gaps (linhas não alteradas) entre modificações no código os gaps representam as linhas de código que permanecem inalteradas entre as modificações. Uma média de gaps maior pode indicar áreas estáveis do código que raramente sofrem alterações. Isso pode ser útil para identificar componentes ou funcionalidades que são menos propensos a mudanças, o que, por sua vez, pode orientar a otimização das análises, direcionando mais recursos para áreas críticas.

Questão de Pesquisa 2 (QP2): Qual é a estrutura da evolução da linha de produtos de software em termos de complexidade das modificações e impacto nas funcionalidades ao longo do tempo?

Métrica 3: A média de profundidade de nós alterados na AST fornece informações sobre a complexidade das alterações realizadas ao longo da evolução da LPS. Ela ajuda a identificar se as mudanças são pequenas modificações, como ajustes menores, correções de bugs ou alterações simples que não têm um impacto profundo na estrutura do código-fonte, ou envolvem modificações significativas na estrutura do código-fonte, ou seja, que têm um impacto substancial na organização e lógica do código-fonte. Isso pode incluir a adição de novas funcionalidades, grandes reorganizações, reescrita de partes essenciais do código ou qualquer outra modificação que afete consideravelmente a estrutura geral do programa. Isso é importante para perceber o impacto das alterações e como elas afetam a evolução da LPS.

Uma compreensão precisa da magnitude da evolução (QP1) auxilia na determinação se uma LPS está em um estado de mudança constante, com requisitos em constante evolução, ou se está em uma fase de estabilidade, onde as alterações ocorrem de forma mais esporádica. Se uma LPS está progredindo rapidamente é fundamental focar na flexibilidade e na capacidade de adaptação. Por outro lado, se a LPS está em um estado estável, os recursos podem ser direcionados para melhorias em termos de desempenho, qualidade e eficiência. Esse foi um dos objetivos de Kroher et al. [14] ao desenvolverem a ComAn. Eles exploraram a intensidade das mudanças de variabilidade na evolução de linhas de produtos de software e propor uma abordagem mais refinada para análise e verificação durante a evolução de LPS. Os autores destacam a importância de gerenciar a variabilidade em LPS para melhorar a qualidade do software, reduzir custos e aumentar a eficiência do desenvolvimento de software.

ComAn e ASTool são duas ferramentas que podem ajudar os desenvolvedores a entender a evolução de linhas de produtos de software. ComAn é voltada para questões estratégicas, como a gestão da variabilidade e a tomada de decisões para melhorar a qualidade e a eficiência do desenvolvimento. ASTool é voltada para questões técnicas, como a análise do código-fonte dos aplicativos para fornecer informações que podem ajudar os desenvolvedores a otimizar suas análises de evolução.

É essencial ter uma compreensão da estrutura de evolução de uma LPS (QP2) porque isso possibilita a gestão eficiente da evolução contínua da linha ao longo do tempo. Compreender a estrutura de evolução envolve conhecer como as mudanças ocorrem, quais componentes são mais propensos a modificações e como essas mudanças impactam os produtos derivados. Isso é crucial para garantir que atualizações e melhorias sejam feitas de forma consistente e para sugerir estratégias de otimização em técnicas de análise de LPS.

3.2 Planejamento

Apresentamos a abordagem metodológica adotada no presente estudo. O intuito deste trabalho consiste em examinar diversas LPS por meio da ferramenta ASTool, caracterizando sua evolução. Para tanto, aplicamos uma série de procedimentos e técnicas. Inicialmente, descrevemos as LPS selecionadas para a realização do estudo. Depois, mostramos como é feita a extração e análise dos arquivos quanto à informação de profundidade de alteração dos nós na AST. Na sequência vemos como são feitas as análises de quantidade de arquivos alterados por commits e de média de gaps. Por último, trazemos todas as informações de configuração para o uso da ferramenta de análise.

3.2.1. Linhas de Produtos de Software selecionadas

Para a execução das análises deste trabalho foram selecionados 4 repositórios:

AXTLS [1]: abreviação de "Another Xtreme TLS,"é uma biblioteca de código aberto amplamente utilizada para implementar o protocolo de segurança TLS (Transport Layer Security) em sistemas embarcados e aplicativos de tamanho reduzido. Sua principal finalidade é fornecer funcionalidades criptográficas essenciais para a comunicação segura pela Internet, como criptografia de dados, autenticação de servidores e clientes, bem como integridade dos dados transmitidos.

BUSYBOX [2]: O BusyBox é uma ferramenta de software de código aberto que combina várias utilidades essenciais do sistema operacional Linux em um único executável compacto. Ele foi projetado para sistemas embarcados e ambientes com recursos limitados, nos quais a economia de espaço em disco e recursos de memória é fundamental. O

	Commits	Arquivos	Linhas de Código
axTLS	179	239	28573
BusyBox	17604	2823	209508
Ccoreboot	54557	18225	1451710
Soletta	3086	1537	191751

Tabela 3.1: Características das LPS analisadas.

BusyBox oferece uma série de comandos comuns do Linux, como ls, cp, mv, grep, entre outros, mas todos esses comandos estão consolidados em um único binário.

COREBOOT [3]: O Coreboot é um firmware de código aberto projetado para substituir o BIOS tradicional ou outros firmwares proprietários em sistemas de computador. Tem como objetivo oferecer uma alternativa de firmware mais flexível, rápida e segura, permitindo que os desenvolvedores personalizem e otimizem o processo de inicialização do sistema.

SOLETTA [4]: é um framework de código aberto projetado para simplificar o desenvolvimento de sistemas embarcados e IoT (Internet das Coisas). Ele fornece uma plataforma flexível e modular para criar aplicativos e dispositivos conectados de forma eficiente. O Soletta é conhecido por sua versatilidade, suportando uma ampla variedade de dispositivos e sistemas operacionais. A Tabela 3.1 traz informações técnicas sobre cada uma das linhas de produtos de software.

3.2.2. A ASTool

A ASTool (Abstract Syntax Tree Tool), é uma ferramenta para a análise de Linhas de Produto de Software (LPS) escritas em linguagem C. A principal função da ASTool é realizar uma análise sintática detalhada do código-fonte em linguagem C. Ao identificar tokens e estruturas gramaticais específicas, a ferramenta constrói uma representação hierárquica do código, gerando a AST e essa árvore oferece uma visão estruturada e organizada do programa.

No âmbito da análise estática de código em linguagem C, a ASTool destaca-se por identificar padrões de codificação e métricas específicas para essa linguagem. Sua aplicação vai além da mera análise sintática, permitindo a extração de informações relevantes para Linhas de Produto de Software.

Além disso, a ASTool incorpora funcionalidades adicionais que enriquecem sua capacidade analítica. A análise de arquivos alterados por commits fornece informações valiosas sobre as mudanças ao longo do tempo, enquanto a análise da média de gaps identifica a estabilidade de áreas específicas do código.

Ao abordar as alterações nos arquivos por meio de commits, a ASTool contribui para uma compreensão mais profunda da evolução do código. A distribuição das alterações e a média de gaps entre modificações complementam a análise estática, oferecendo uma perspectiva abrangente sobre a estabilidade e dinâmica do código-fonte ao longo do tempo.

3.2.3. Extração de Arquivos

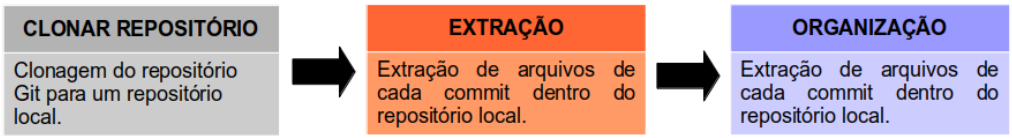

Figura 3.1: Processo de extração e organização de arquivos.

Como mostrado na Figura 3.1, o procedimento de extração dos arquivos compreende três etapas distintas. Inicialmente, é necessário clonar o repositório para uma pasta local. Uma vez clonado o repositório, a ASTool extrai os arquivos de acordo com as informações contidas nos commits.

Por último, procedemos à organização dos arquivos extraídos, categorizando-os com base na data e no hash de commit correspondente. Isso simplifica a identificação e o acesso a versões específicas do código-fonte, o que torna mais fácil o acompanhamento e a caracterização da evolução dos produtos de software ao longo do tempo.

3.2.4. Análise de Arquivos

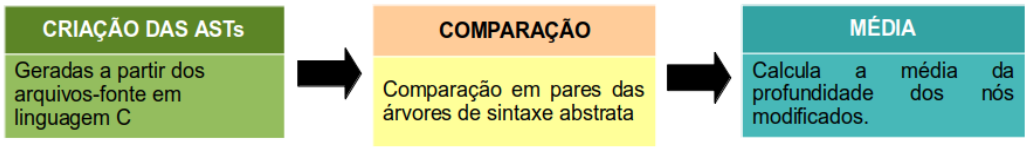

Figura 3.2: Análise das ASTs.

Na etapa inicial da análise, as árvores de sintaxe abstrata (AST) são geradas a partir dos arquivos-fonte em linguagem C. Essa operação é realizada de forma recursiva, percorrendo uma pasta que contém os arquivos em C. A AST é gerada para cada arquivo identificado e armazenada em arquivos de texto (extensão txt). Todas as ASTs geradas são armazenadas em uma estrutura de diretórios no local especificado pelo usuário.

Por último, ocorre a comparação em pares das árvores de sintaxe abstrata (ASTs) que estão nos arquivos de texto, e as diferenças identificadas são registradas em arquivos

CSV. Adicionalmente, o script calcula a média da profundidade dos nós modificados nas ASTs e a insere nos mesmos arquivos CSV, como mostra a Figura 3.2. Na Figura 3.3, é apresentado o fluxo completo, desde a aquisição dos arquivos até a criação das ASTs, a comparação entre elas e a exibição dos resultados.

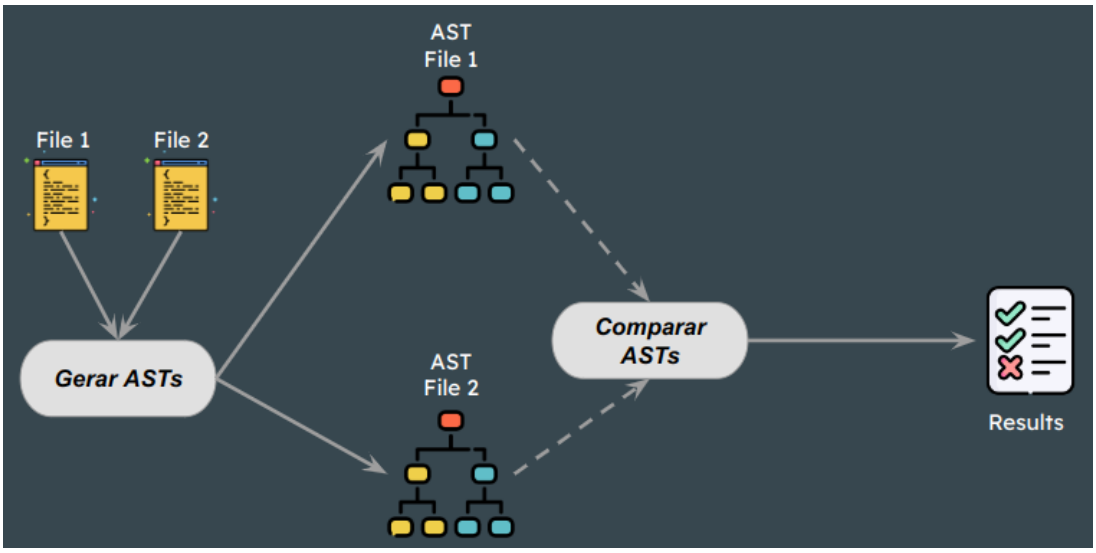

Figura 3.3: Fluxo da análise das ASTs.

Realizamos ainda duas análises distintas nos commits das LPS. Na primeira análise, coletamos informações quantitativas sobre o histórico de commits (Média de Arquivos Alterados por Commits).

A segunda análise concentra-se na detecção de linhas gap (ou seja, linhas não alteradas) e na geração de estatísticas relacionadas. Percorremos todos os commits no repositório, identificando os arquivos de código-fonte (.c e .h) e calculando a média das linhas gap em cada arquivo . Além disso, é calculada a média geral das linhas gap em todos os arquivos (Média de Gaps - Linhas não Alteradas entre Modificações no Código). Na Figura 3.4, é apresentado o fluxo de análises dos commits.

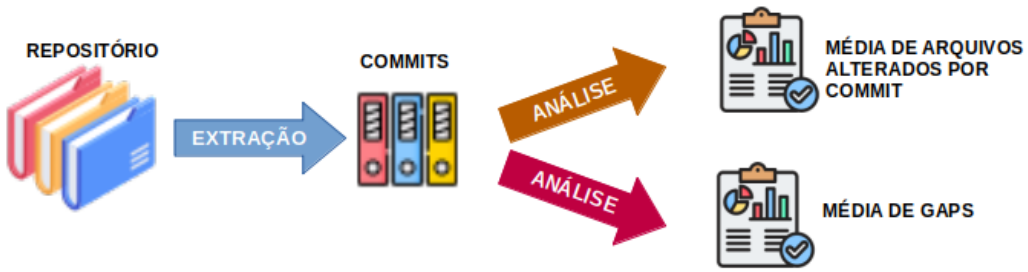

Figura 3.4: Fluxo da análise dos Commits.

3.2.5. Instrumentação e Operação

Abordaremos aqui o hardware e o software que compõem a configuração essencial utilizada neste trabalho.

O hardware da máquina em que executamos as análises consiste em uma CPU Intel Core i7-2630QM com 2.00GHz e 8 núcleos. Além disso, possui 8 GB de RAM e 500 GB de armazenamento no HD. Com essa configuração, todas as análises foram realizadas; no entanto, para um melhor desempenho, é aconselhável aumentar a memória RAM, pois quanto maior for o commit a ser analisado, maior será o consumo de memória.

No que diz respeito ao software, o sistema operacional utilizado para as análises foi o Ubuntu 22.04.3 LTS (Jammy Jellyfish) de 64 bits. Para executar nossa implementação, instalamos o Python 3.9, o Git 2.34.1, o Clang 14.0.0 e como ambiente de desenvolvimento utilizamos o Visual Studio Code 1.82.1.

A linguagem de programação Python é de uso geral e oferece bibliotecas e ferramentas disponíveis para a manipulação de árvores de sintaxe abstrata (ASTs) e a análise de código-fonte, o que é fundamental para o nosso propósito de caracterizar a evolução das LPS.

A expressividade da linguagem Python também nos permitiu implementar os procedimentos descritos nessa seção, desde a extração e organização dos arquivos dos repositórios até a análise das mudanças nas ASTs. A natureza modular da linguagem Python nos permitiu criar plug-ins e módulos especializados para cada etapa do processo, tornando a ferramenta ASTool adaptável a diferentes LPS e cenários de análise.

No contexto da ferramenta ASTool, que é uma aplicação desenvolvida para análise de evolução de Linhas de Produto de Software (LPS) em linguagem Python, um conjunto de bibliotecas essenciais foi empregado com o objetivo de possibilitar a realização de diversas tarefas críticas para o sucesso do projeto. As bibliotecas utilizadas foram:

- Os: é utilizada para realizar operações relacionadas ao sistema operacional. No âmbito da ASTool, a biblioteca desempenha um papel crucial na interação com o sistema de arquivos. Ela possibilita a criação de diretórios, verificação de existência de arquivos, navegação por pastas, e a manipulação de caminhos de arquivo. Essas funcionalidades são vitais para a organização e estruturação dos dados de repositórios de código-fonte, bem como na gestão de arquivos temporários e resultados da análise.

- Csv: é empregada para a manipulação de arquivos CSV (Comma-Separated Values), que são frequentemente utilizados para armazenar dados tabulares. Na ASTool, a biblioteca csv permite a leitura e escrita de dados estruturados em formato CSV.

Essa funcionalidade é especialmente relevante quando se deseja armazenar os resultados das análises de evolução das Linhas de Produto de Software em um formato tabular para futuras referências ou análises estatísticas.

- Diffilib: oferece funcionalidades para comparar e encontrar diferenças entre sequências de texto. No contexto da ASTool, ela é empregada para comparar diferentes versões de código-fonte, destacando as mudanças realizadas entre commits em um repositório. Essa capacidade é essencial para a identificação e análise das alterações nas Árvores de Sintaxe Abstrata ao longo do tempo.

- Subprocess: permite a interação com processos externos ao programa Python. Na ASTool, ela é usada para executar comandos do sistema operacional, como a invocação do Clang para analisar o código-fonte e gerar as ASTs. Através do subprocess, é possível automatizar a execução desses processos externos diretamente a partir da ferramenta.

- Git: é uma interface Python para interagir com repositórios Git. No contexto da ASTool, ela é fundamental para realizar operações relacionadas ao controle de versão, como clonar repositórios, navegar por históricos de commits e recuperar informações sobre as alterações nos arquivos. Isso permite a análise de evolução das LPS diretamente a partir dos repositórios Git.

- Shutil: é utilizada para operações de alto nível relacionadas à manipulação de arquivos e diretórios. No contexto da ASTool, a shutil é empregada para cópia, movimentação e exclusão de arquivos e pastas, tornando possível a organização e gerenciamento eficiente dos dados e resultados da análise.

- Pandas: a biblioteca pandas é uma ferramenta essencial para a análise de dados em Python, tornando mais fácil o trabalho com dados estruturados, a realização de análises estatísticas e a preparação de dados para modelagem e visualização.

A escolha de utilizar o Clang para a geração das árvores de sintaxe abstrata de códigos escritos na linguagem C foi motivada por diversas razões técnicas e práticas. O Clang é um frontend de compilador de código aberto amplamente reconhecido e utilizado na comunidade de desenvolvedores C e C++. Aqui estão algumas das justificativas para a escolha do Clang:

1. Precisão e Conformidade: O Clang é conhecido por sua precisão e conformidade com os padrões de linguagem C, como o padrão ANSI C e suas iterações mais recentes, e também trabalha com a representação de diretrizes de pré-processamento (ex.

Or.	Script	Função	Análise
1	ExtComm.py	Permite a extração do conteúdo de commits em um repositório Git local.	Alterações na AST
2	ReComm.py	Reorganizar arquivos provenientes de commits em um repositório Git.	Alterações na AST
3	MakeTree.sh	Gera árvores de sintaxe abstrata (AST) a partir de arquivos fonte em linguagem C.	Alterações na AST
4	Analise.py	Compara pares de árvores de sintaxe abstrata (ASTs) presentes em arquivos de texto e armazenar as diferenças encontradas em arquivos CSV.	Alterações na AST
5	Medias.py	Calcula a média de alterações de profundiade para todos os commits no repositório e as adiciona ao arquivo CSV.	Alterações na AST
6	Commit_Files.py	Fornece informações sobre a quantidade de arquivos modificados em cada commit e a média dessa quantidade em todo o repositório.	Modificações por commits
7	GapsComm.py	Calcula a média dos gaps entre cada alteração do código-fonte demonstrando a dispersão dessas alterações	Modificações por commits

Tabela 3.2: Scripts da ASTool em ordem de execução

#ifdefs). Isso garante que as ASTs geradas sejam representações fiéis do código-fonte original, tornando-as ideais para análises detalhadas e confiáveis.

2. Acesso à Estrutura Interna: O Clang oferece APIs de fácil acesso para a estrutura interna do código-fonte durante o processo de compilação. Isso permite uma análise profunda das estruturas de controle, tipos de dados, funções e outras características do código C, o que é essencial para a análise de evolução de código e otimizações.

3. Portabilidade: O Clang é altamente portátil e está disponível em várias plataformas, tornando-o uma escolha viável para desenvolvedores que desejam realizar análises em diferentes ambientes de desenvolvimento.

4. Desempenho: O Clang é conhecido por sua eficiência em termos de consumo de recursos e velocidade de compilação. Isso é especialmente importante ao lidar com grandes conjuntos de código, como os repositórios de LPS.

A seguir, descreveremos brevemente o funcionamento dos scripts que compõem a ASTool, mais detalhes podem ser consultados em:

- https://sites.google.com/view/astools/in%C3%ADcio.

A Tabela 3.2 cita cada script em sua ordem de execução.

Os scripts de 1 a 5 desempenham papéis interdependentes na preparação e geração das ASTs. Por outro lado, os scripts 6 e 7, que se concentram na análise de commits, não possuem uma dependência estrita em relação aos scripts anteriores. Isso significa que eles podem ser executados em qualquer ordem, dependendo dos requisitos específicos do usuário ou das análises a serem realizadas. A 3.4 mostra o fluxo de execução dos scripts sobre o repositório git.

Por exemplo, para realizar a análise de média de gaps em linhas de produto de software, basta executar o comando python GapsComm.py. O resultado da análise será gerado automaticamente.

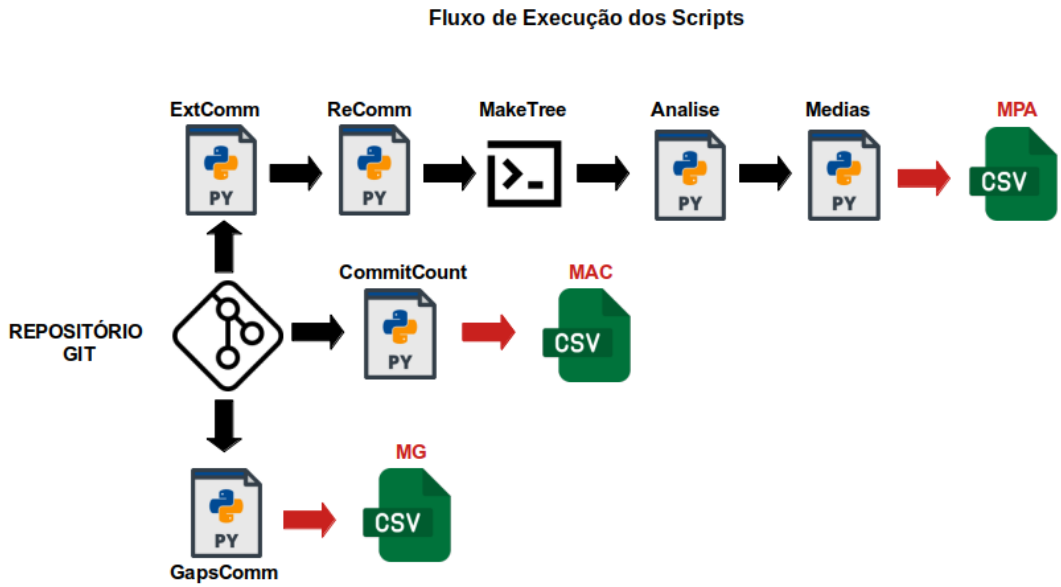

Figura 3.5: Fluxo de execução de scripts.

3.3 Resultados e Análises

3.3.1. Média de Profundidade de Nós Alterados na AST (MPA)

Para realizar esta análise, foram usados repositórios de arquivos-fonte escritos em C. A ASTool foi então aplicada a esses arquivos para identificar a profundidade das modificações na AST. No entanto, é importante salientar que foram desconsiderados arquivos nos quais não foram encontradas modificações ou nos quais as modificações consistiam apenas em comentários.

Para fins de testes, disponibilizamos a ASTool para outros usuários. Nos testes iniciais, selecionamos uma quantidade de arquivos na linguagem C (de 10 a 20), submetidos à ferramenta por nós e pelos usuários. Os resultados foram comparados. Uma vez que os resultados correspondiam, selecionamos uma LPS que não fosse muito grande, enviamos aos usuários para rodarem a ferramenta e comparamos os resultados, que também se mostraram correspondentes. Ressaltamos que não aplicamos a ferramenta a outras LPS que não estão relacionadas neste trabalho.

A Figura 3.6 ilustra o processo de análise executado pela ferramenta. Essa ferramenta é responsável por gerar a AST a partir do arquivo original e dos arquivos alterados. Posteriormente, ela compara essas árvores, nó a nó, registrando a posição de qualquer alteração identificada. Uma vez que todas as alterações na AST são identificadas, é calculada a média para o arquivo em questão. Por fim, após analisar todos os arquivos, a média geral é calculada.

A Figura 3.7 mostra as médias de cada uma das Linhas de Produtos de Software

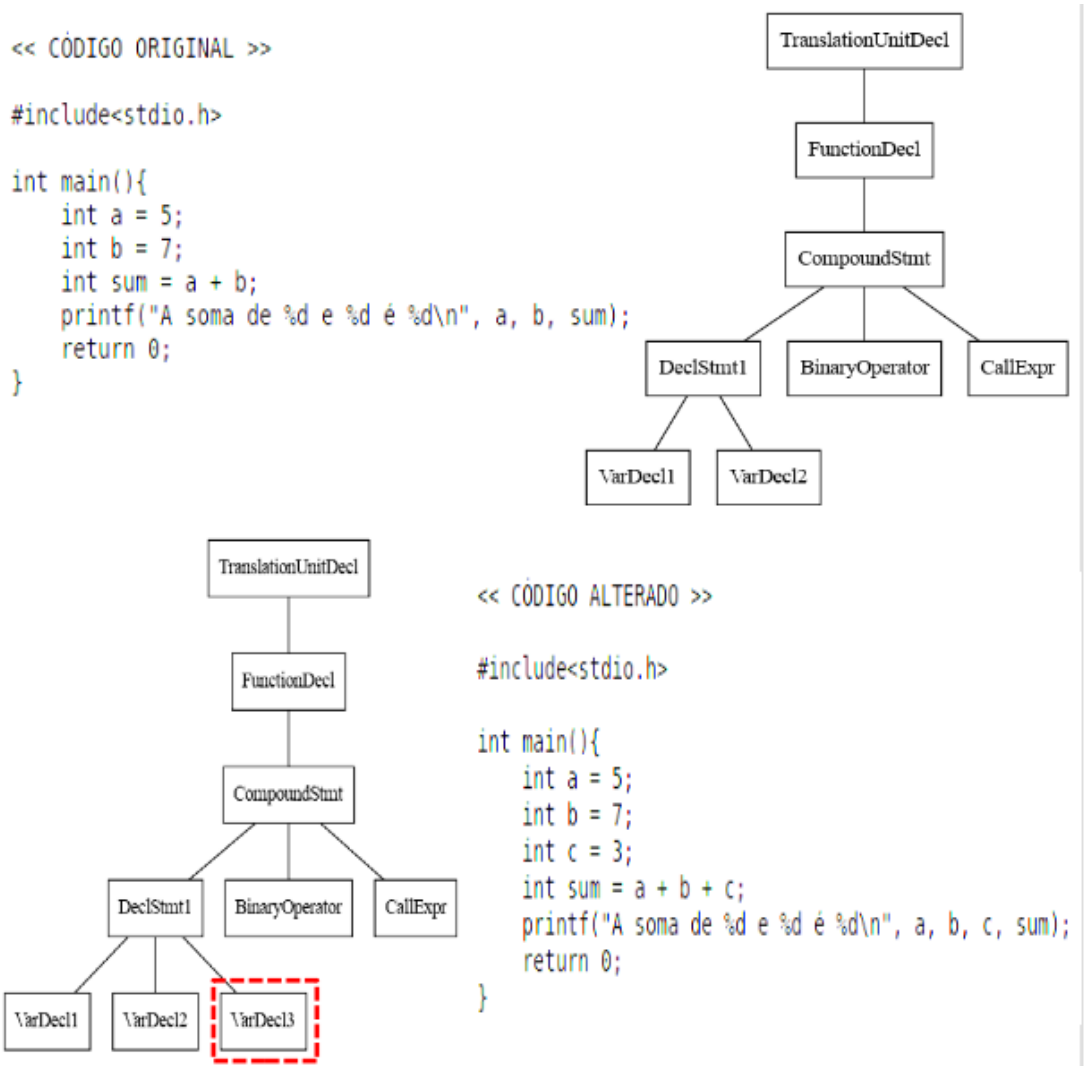

Figura 3.6: Processo de análise de profundidade de alterações na AST.

selecionadas. Já a Tabela 3.3 informa a LPS, a quantidade de arquivos analisados e a média de profundidade de alterações (MPA), encontrada de cada uma.

A LPS axTLS revela a mais alta média de profundidade de nós modificados, atingindo 2,93, enquanto o coreboot obteve uma média de profundidade de nós alterados de 2,03 mesmo com um número consideravelmente maior de arquivos examinados.

O BusyBox alcançou uma média de 2,79, sugerindo que as modificações em seu código-fonte não são excessivamente profundas, assim como o Soletta, que registrou uma média de 2,70.

As Figuras 3.8 e 3.9 trazem um histograma e os resultados referentes à distribuição da quantidade de arquivos por média para cada LPS. Neste contexto, observamos que, em três das LPS analisadas, as alterações ocorrem principalmente entre as profundidades 3 e 4 da AST. Uma exceção é o coreboot, no qual a maioria das modificações ocorre ainda mais próximas à raiz da AST, entre as profundidades 1 e 2. No entanto, uma informação

LPS	Quant. Arquivos	MPA na AST
axTLS	31	2,93
BusyBox	1417	2,79
coreboot	15832	2,03
Soletta	449	2,70

Tabela 3.3: Média de Profundidade de Alterações na AST

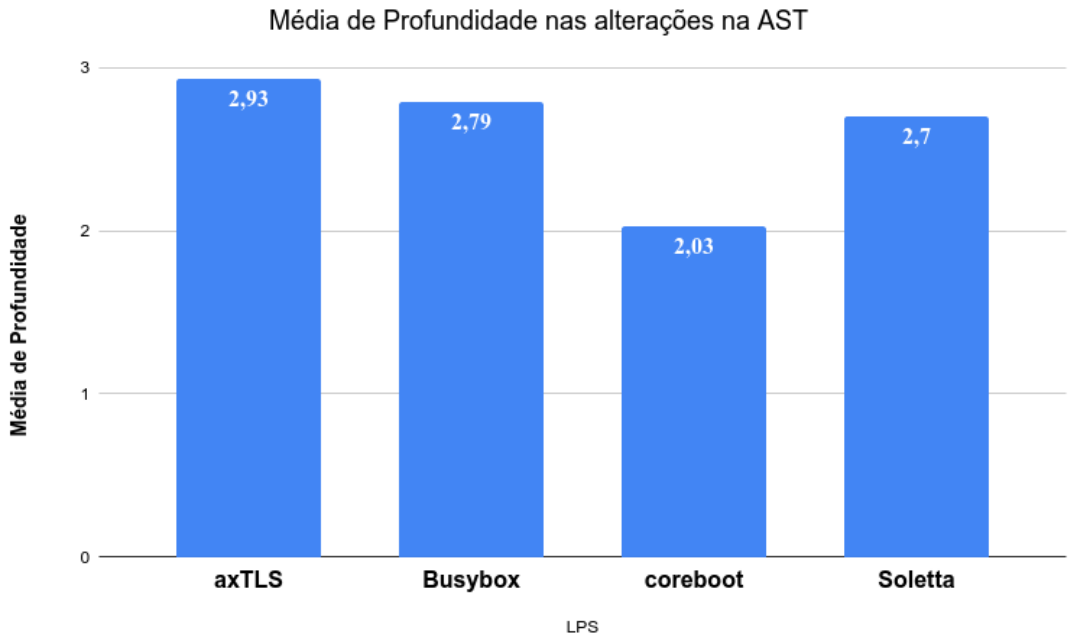

Figura 3.7: Média de Profundidade de Alterações na AST por LPS.

importante que os gráficos nos fornecem é que há uma maior concentração de evolução em baixa profundidade de AST em todas as LPS.

3.3.2. Média de Arquivos Alterados por Commits (MAC)

A figura 3.10 apresenta um exemplo de um commit extraído da LPS Soletta. O commit em questão é identificado por um código hash, que é identificado e verificado pela ASTool para determinar o número de arquivos alterados. No exemplo apresentado na figura, podemos observar que 12 arquivos foram alterados.

A figura 3.11 mostra a média de arquivos alterados por commits em cada uma das LPSs analisadas. A Tabela 3.4 informa a LPS, a quantidade de commits analisadas e a média encontrada de cada uma.

Os resultados da análise indicam que a média geral de arquivos alterados por commit em cada LPS está situada em um intervalo que vai de 3 a 7 arquivos. Embora esse valor seja considerável, não é tão elevado. No caso do BusyBox, que teve a menor média entre as

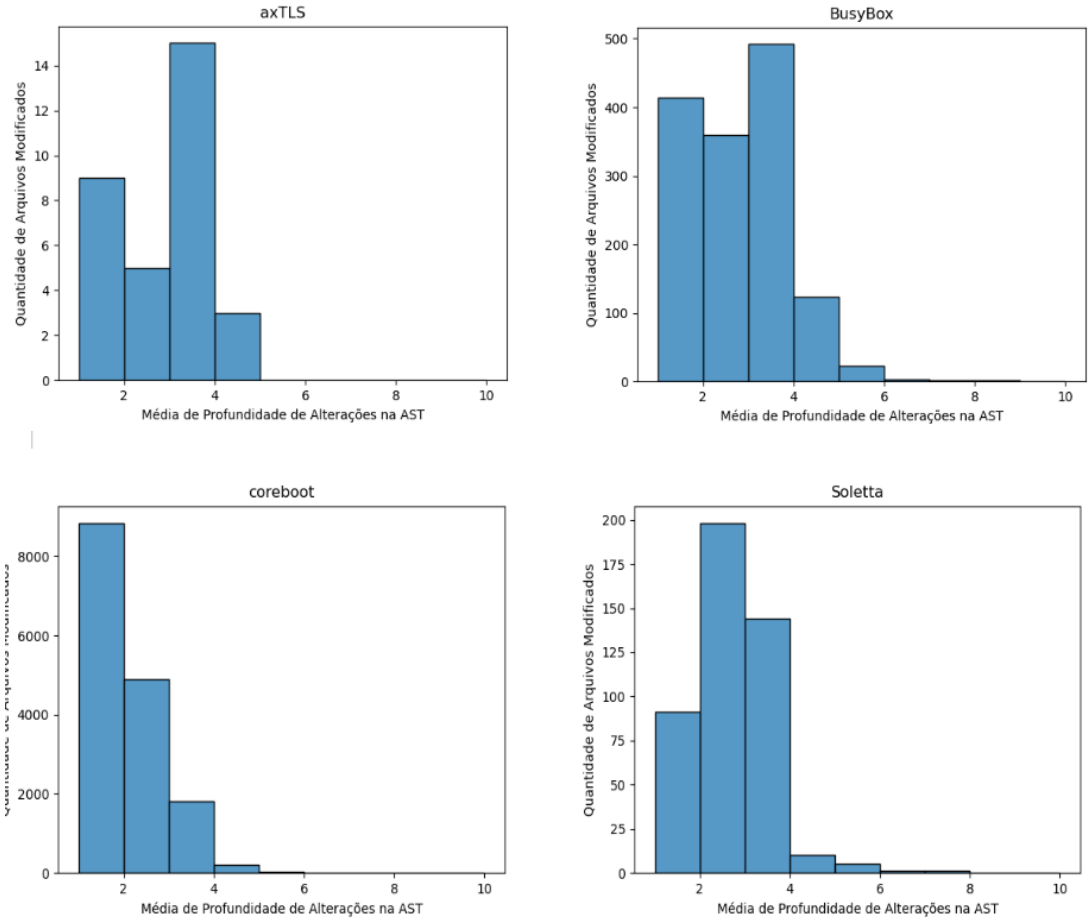

Figura 3.8: Distribuição das Médias de Profundidade de Alterações na AST por LPS.

LPS analisadas, as alterações têm um impacto menos abrangente em relação à quantidade de arquivos envolvidos.

LPS	Quant. Commits	Md. Arq. Commit
axTLS	138	6,85
BusyBox	17604	3,17
coreboot	54023	5,77
Soletta	3086	4,67

Tabela 3.4: Média de Arquivos Alterados por Commits

A figura 3.12 mostra que em todas as LPS a maioria dos commits alteram três ou menos arquivos. A figura 3.13 confirma essa tendência mostrando que em três das LPS analisadas a quantidade de commits que alteram de 1 a 3 arquivos passa de 70%

No contexto da axTLS, que passou por análise de 138 commits, observamos uma média de 6,85 arquivos alterados por commit. Essa média mostra que cada commit altera um número relativamente considerável de arquivos.

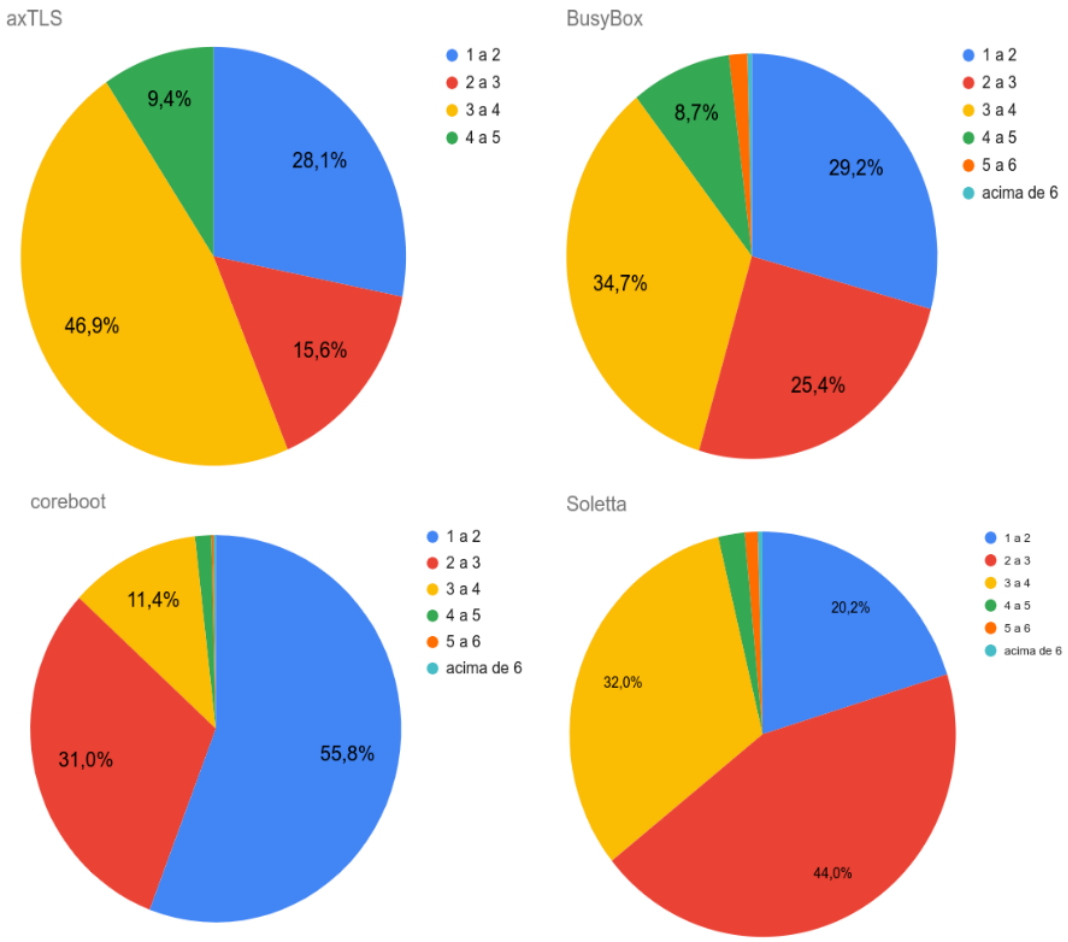

Figura 3.9: Resumo das Médias de Profundidade de Alterações na AST por LPS.

3.3.3. Média de Gaps (Linhas não Alteradas entre Modificações no Código) (MG)

A figura 3.14 apresenta um trecho de código do arquivo sol_flow_builder_add_node.c da LPS Soletta. O cálculo das lacunas (gaps) é realizado através de uma análise minuciosa das diferenças entre o conteúdo de um arquivo em diferentes commits. Ao percorrer essas alterações, contabilizam-se as linhas removidas consecutivas identificadas pelo símbolo '-' e, ao encontrar adições marcadas com '+', as lacunas são normalizadas em relação ao número total de linhas não modificadas. A normalização no contexto deste código refere-se à técnica utilizada para ajustar e padronizar as lacunas (gaps) identificadas durante as alterações nos arquivos de um repositório Git. Esse processo tem como finalidade obter uma medida proporcional que representa a extensão das alterações em relação ao conteúdo não modificado, permitindo, assim, uma análise mais equitativa das mudanças ao longo do tempo. O resultado desse processo é uma média ponderada das lacunas, proporcionando uma métrica significativa para avaliar a evolução dos arquivos no repositório. Esse método permite uma compreensão mais aprofundada das mudanças, considerando não apenas a

Figura 3.10: Exemplo de arquivos alterado em um commit.

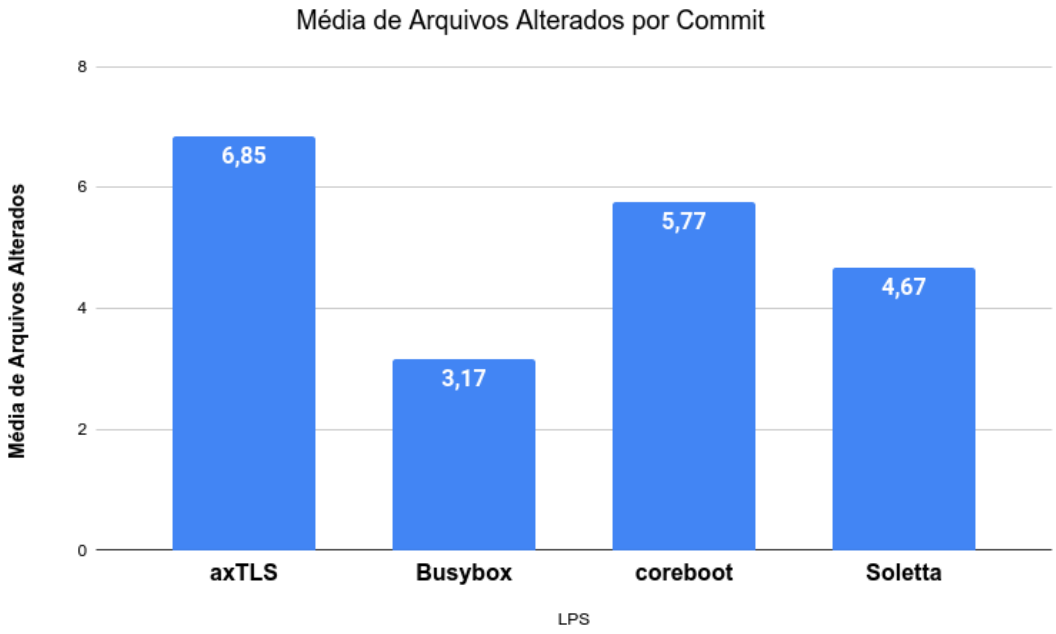

Figura 3.11: Exemplo de arquivos alterado em um commit.

quantidade bruta de adições e remoções, mas também a proporção em relação ao conteúdo original não alterado.

A Tabela 3.5 informa a LPS, a quantidade de arquivos analisadas e a média encontrada de cada uma. E a figura 3.10 mostra as médias de gaps para cada LPS.

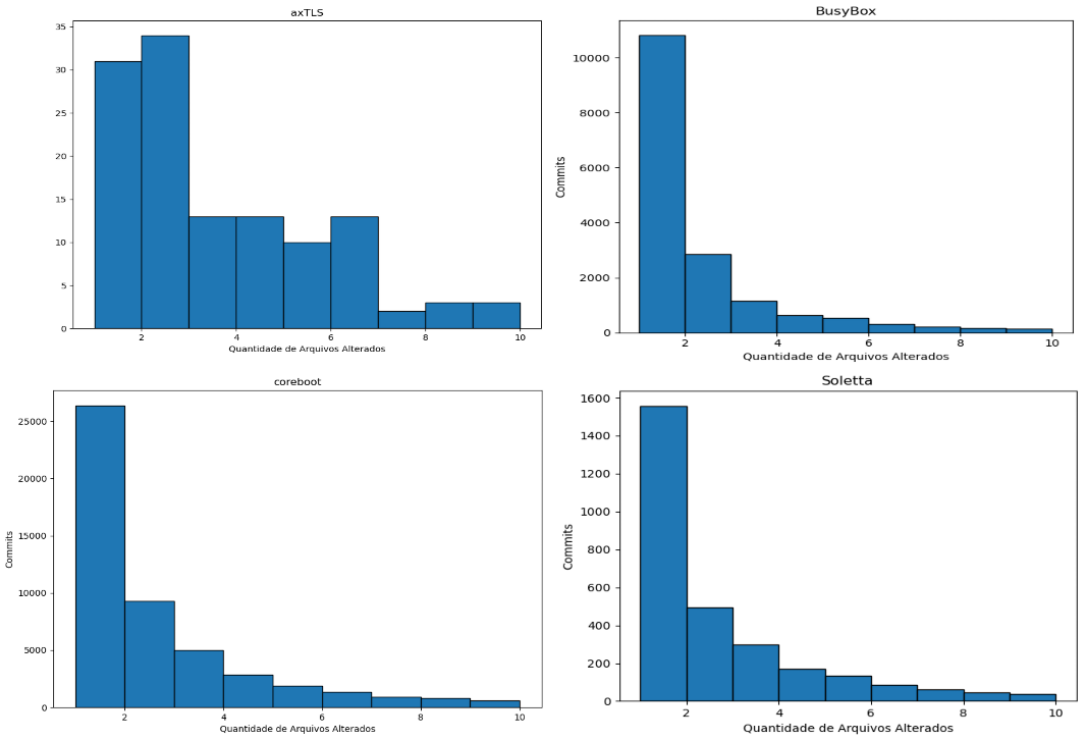

Figura 3.12: Quantidade de arquivos alterados por commit em cada LPS.

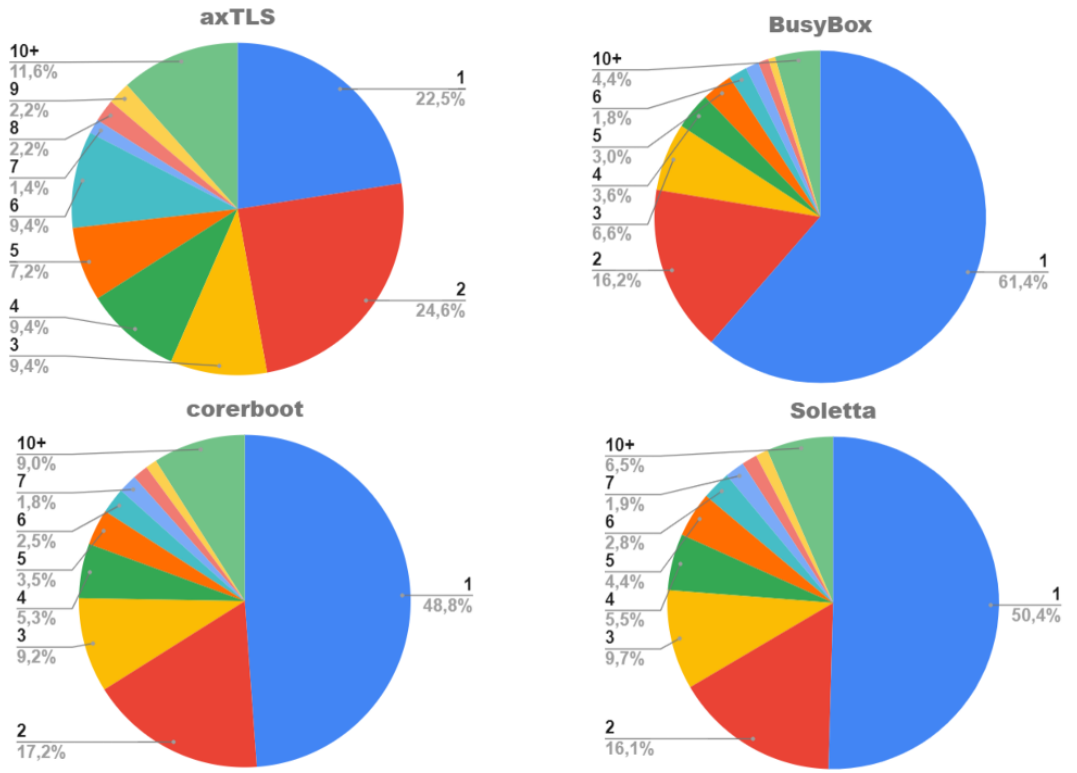

Figura 3.13: Resumo das Médias de Arquivos Alterados por commits em cada LPS.

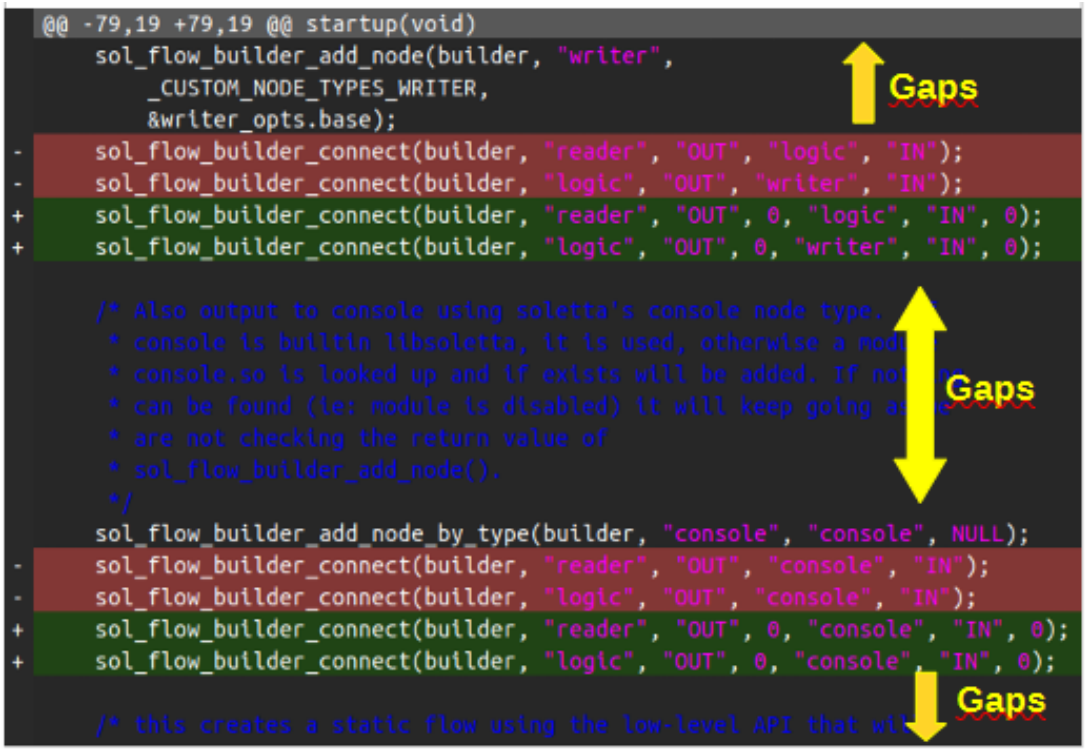

Figura 3.14: Gaps no código.

LPS	Quant. Arquivos	Md. Gaps por LPS
axTLS	58	0,37
BusyBox	747	0,39
coreboot	-	-
Soletta	545	0,31

Tabela 3.5: Média de Gaps po LPS

Como pode ser observado, não apresentamos resultados para a LPS coreboot nessa análise. Isso se deve à necessidade de examinar um grande número de arquivos do coreboot, o que fez com que a capacidade da máquina utilizada para a análise fosse insuficiente. Consequentemente, mesmo com modificações no código e otimizações na análise, o tempo estimado para a conclusão da mesma superou significativamente o ponto de viabilidade. De fato, o tempo estimado para concluir a análise da métrica de média de gaps no Coreboot excedeu 110 horas, o que ultrapassa em muito o limite razoável de tempo para uma análise desse tipo.

Os resultados da métrica "Média de Gaps (Linhas não Alteradas entre Modificações no Código)"para as LPS analisadas indicam que essas linhas de produtos de software mantêm uma relativa estabilidade em várias áreas do código-fonte. Os valores obtidos são bastante semelhantes entre as LPS, variando de 0,31 a 0,39.

Esse resultado sugere que, em média, uma proporção considerável do código-fonte

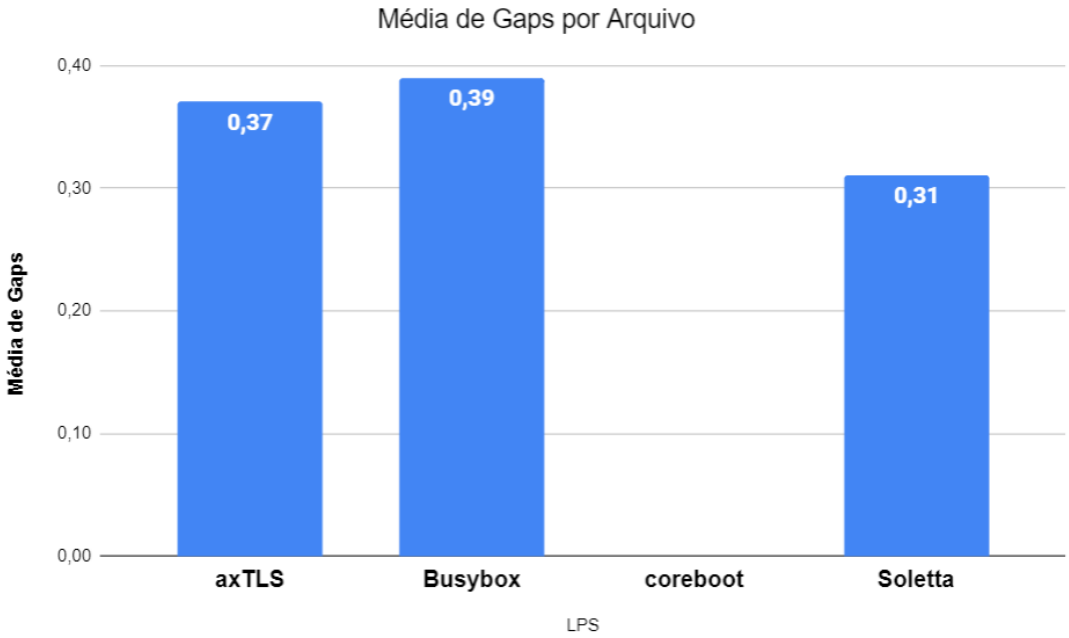

Figura 3.15: Média de Gaps por LPS.

permanece inalterada entre as modificações. Em outras palavras, mesmo com a evolução contínua das LPS, há áreas específicas do código que permanecem estáveis ao longo do tempo.

Os resultados mostrados nas figuras 3.16 e 3.17 indicam que, embora haja uma concentração de médias de gaps entre 0 e 0.2, a maioria das médias está acima de 0.2.

Esses resultados sugerem que, em cada um desses projetos, áreas substanciais do código permanecem relativamente inalteradas ao longo do tempo.

3.4 Discussão

Nesta seção, abordaremos os principais resultados da pesquisa, interpretando-os à luz dos objetivos estabelecidos e destacando os padrões observados. A pesquisa tinha como objetivo caracterizar a evolução de Linhas de Produtos de Software, com foco em duas questões de pesquisa: a intensidade da evolução (QP1) e a estrutura da evolução (QP2).

A métrica "Média de Arquivos Alterados por Commits"revelou que as LPS analisadas tendem a realizar modificações em um número reduzido de arquivos por commit. Em três das LPS análisadas a quantidade de commits que alteram até 3 arquivos ficou acima dos 70%, o que indica que essas LPS mantêm uma taxa de evolução equilibrada.

A métrica "Média de Gaps"(linhas não alteradas entre modificações no código) revelou que, em três dos projetos analisados - axTLS, BusyBox e Soletta - uma proporção significativa dos arquivos possui médias de gaps superiores a 0,2. No axTLS, mais de

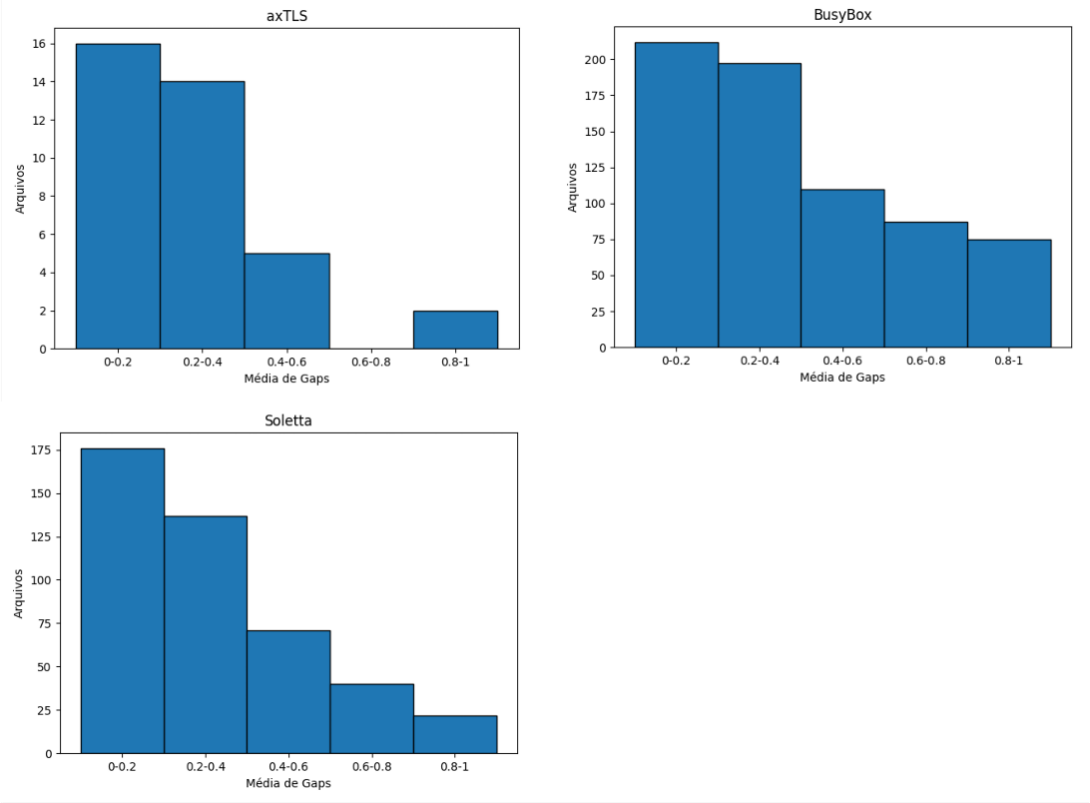

Figura 3.16: Quantidade de gaps por arquivo.

51% dos arquivos apresentam essa característica; no BusyBox, mais de 71% dos arquivos demonstram médias acima de 0,2; e no Soletta, mais de 60% dos arquivos compartilham esse padrão. Os resultados indicam que, em todos esses projetos, grandes partes do código permanecem essencialmente as mesmas ao longo do tempo.

A métrica "Média de Profundidade de Nós Alterados na AST"revelou um padrão nas LPS analisadas. Todas elas apresentaram uma média de profundidade abaixo de 3, indicando que as modificações tendem a ser de natureza mais superficial. Essas mudanças envolvem alterações que não afetam profundamente a organização do código-fonte.

Os resultados destas análises ainda destacam que as LPS analisadas tendem a manter um equilíbrio entre mudanças frequentes e estabilidade. As modificações realizadas são, em sua maioria, não são profundas, o que sugere uma abordagem de desenvolvimento que valoriza a estabilidade da estrutura existente.

Os resultados obtidos sugerem que a memoização, uma técnica que envolve o armazenamento em cache de resultados de cálculos para evitar o processamento redundante, pode ser explorada na evolução da LPS. Dado o padrão de modificações mais superficiais, a memoização pode ser empregada para otimizar o desempenho das análises de LPS - Michie [21], reduzindo a sobrecarga de processamento em áreas críticas que sofrem modificações frequentes - Wimmer et al. [29]. Isso contribuiria para a eficiência da evolução

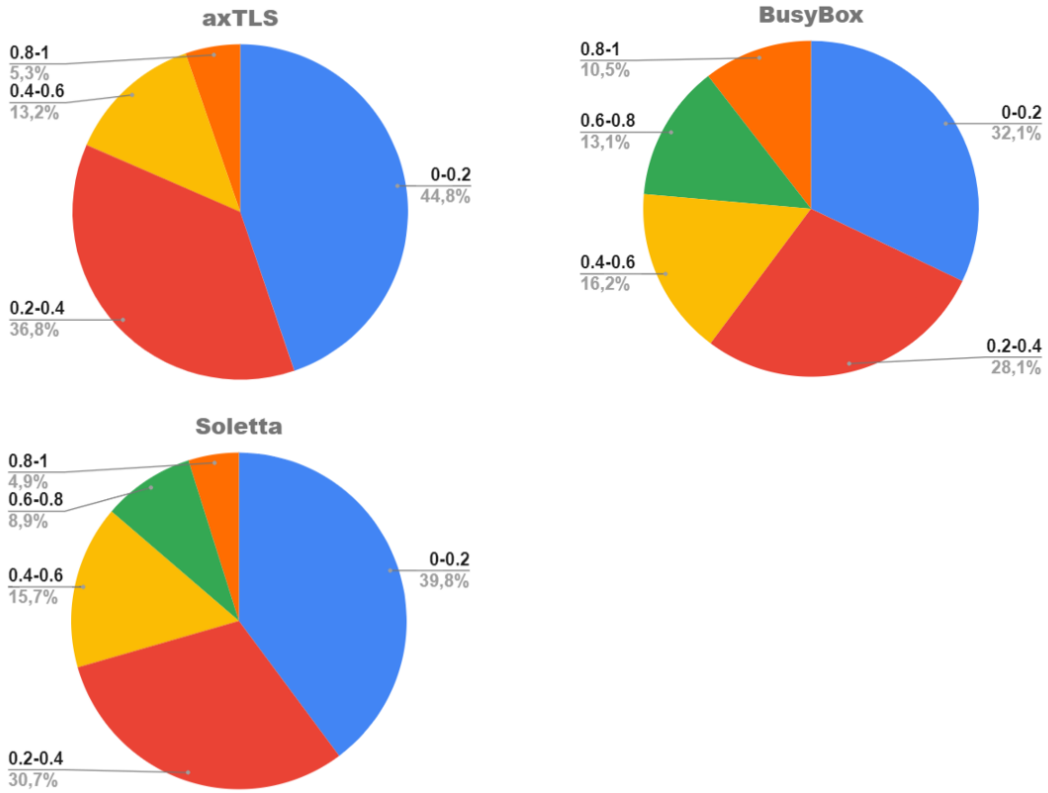

Figura 3.17: Resumo das Médias de Gaps por LPS.

de LPS, preservando a estabilidade em áreas estáveis e aprimorando a flexibilidade nas partes mais dinâmicas.

Por exemplo, imaginemos um programa que calcula os números de Fibonacci. Esses números são uma sequência infinita, onde cada número é a soma dos dois números anteriores. Sem usar memoização, toda vez que for preciso calcular um número de Fibonacci, será necessário calcular todos os números anteriores na sequência, mesmo que já tenha sido calculado antes. Isso pode ser muito ineficiente, especialmente quando a sequência é grande.

A memoização é uma técnica que pode ajudar a melhorar a eficiência desse cálculo. Com a memoização, podemos armazenar os resultados dos cálculos anteriores em um cache. Dessa forma, se for necessário calcular um número de Fibonacci que já foi calculado anteriormente, podemos simplesmente consultar o cache para obter o valor, em vez de calcular a sequência novamente.

A figura 3.18 mostra uma árvore de chamadas para a função Fibonacci, sem e com memoização. Sem memoização, a árvore é muito densa e conteria muitos cálculos redundantes. Por exemplo, fib(3) é calculado duas vezes, fib(2) três vezes, e assim por diante. Com memoização, a árvore é otimizada. Valores já calculados são armazenados em cache, o que evita que eles precisem ser calculados novamente.

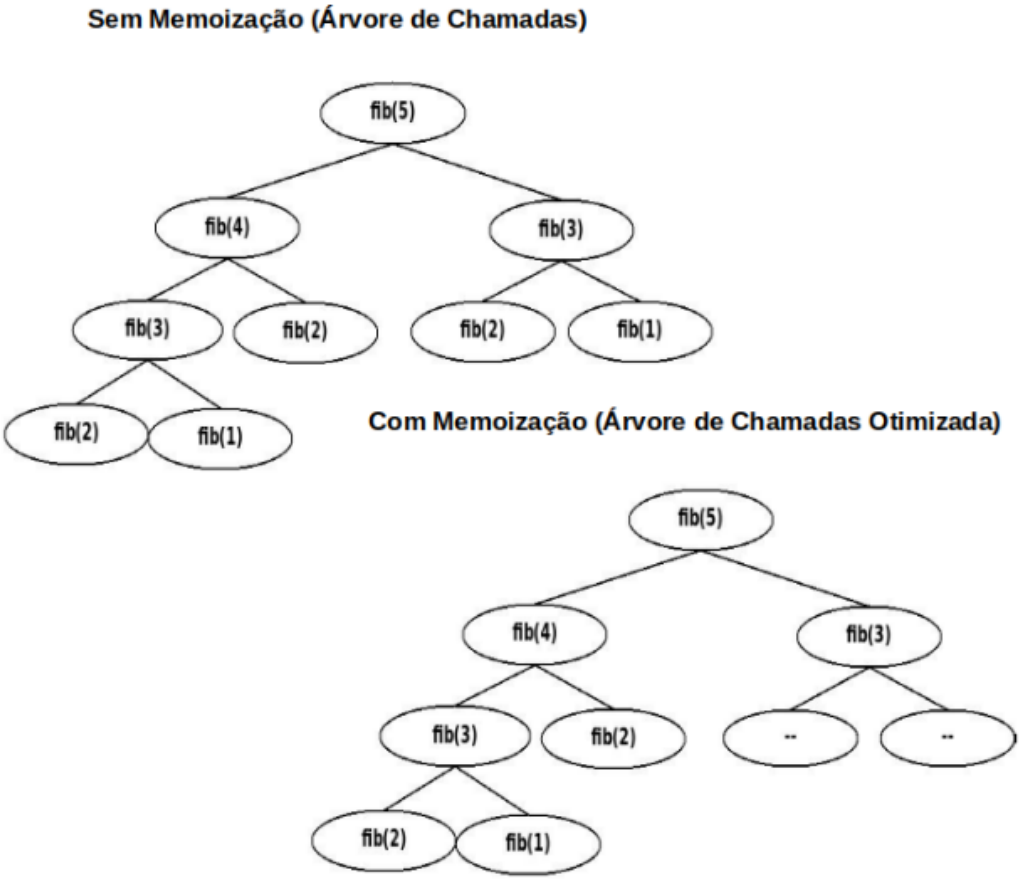

Figura 3.18: Exemplo de memoização.

Em resumo, os resultados desta pesquisa fornecem informações para a gestão e o desenvolvimento de Linhas de Produtos de Software, ao mesmo tempo em que destacam o potencial uso da memoização como uma técnica de otimização de análises de evolução desses sistemas altamente configuráveis, considerando a estrutura e a intensidade da evolução das LPS.

3.5 Ameaças a Validade

Uma das ameaças à validade diz respeito à representatividade da amostra de LPS analisadas. Os resultados apresentados neste estudo são baseados em um conjunto específico de LPS, incluindo axTLS, coreboot, BusyBox e Soletta. Essas LPS foram escolhidas com base em critérios específicos, como tamanho e relevância. No entanto, a generalização dos resultados para todas as LPS pode ser comprometida, uma vez que outras LPS podem apresentar características e dinâmicas de código diferentes. Como destacado por Kontio et al. [13], a seleção de amostras não representativas pode levar a conclusões inadequadas sobre a evolução do software.

Além disso, a análise das métricas de código-fonte depende da eficácia das ferramentas de análise estática utilizadas, como a ASTool mencionada neste estudo. A qualidade das métricas coletadas e a precisão na identificação de elementos da AST (Abstract Syntax Tree) podem ser afetadas por limitações nas ferramentas de análise. Como apontado por Spinellis et al. [25], a escolha inadequada ou a configuração incorreta das ferramentas de análise estática pode levar a resultados imprecisos. Portanto, é importante garantir que as ferramentas utilizadas sejam confiáveis e estejam configuradas adequadamente. A ASTool é confiável porque foi projetada especificamente para análise estática de LPSs em C. Ela foi extensivamente testada em várias LPSs sem a necessidade de alterações na ferramenta ou tratamento específico para cada LPS. Isso demonstra sua capacidade de lidar com diferentes contextos de código sem perder sua eficácia ou precisão.

Outra ameaça à validade diz respeito à interpretação das métricas de código-fonte. As métricas apresentadas neste estudo, como a média de profundidade de nós alterados na AST, a média de arquivos alterados por commits e a média de gaps entre modificações no código, fornecem informações quantitativas sobre a complexidade e a dinâmica do software. No entanto, a interpretação dessas métricas pode ser subjetiva e depender do contexto. Como sugerido por Kitchenham et al. [12], a falta de uma interpretação sólida das métricas pode levar a conclusões equivocadas.

Por fim, é importante considerar que a evolução do software é influenciada por diversos fatores, como mudanças nos requisitos do sistema, pressões de mercado e decisões de desenvolvimento. Portanto, a análise das métricas de código-fonte por si só pode não capturar completamente todos os aspectos da evolução do software. Como apontado por Osterweil [22], a validade externa de estudos de evolução de software pode ser ameaçada pela falta de consideração de fatores contextuais.

Capítulo 4

Trabalhos Relacionados

A análise da evolução em LPS é fundamental para reconhecer como esses sistemas se desenvolvem ao longo do tempo.

Kroher et al. [14] desenvolveram um trabalho que apresenta uma abordagem detalhada para analisar mudanças em artefatos de código, *build* e *variability model* em LPS. Os autores propõem uma técnica para identificar a intensidade das mudanças de variabilidade, que pode ser usada para melhorar a análise e verificação de tarefas durante a evolução da LPS. O estudo de caso realizado no kernel do Linux mostrou que as mudanças na informação de variabilidade ocorrem com pouca frequência e afetam apenas pequenas partes dos artefatos analisados. Apesar de não se concentrar somente em informações de variabilidade, o nosso trabalho chegou à conclusões similares ao desse trabalho com relação à frequência e intensidade de mudanças.

Posteriormente, Kröher et al. [15] ampliaram sua pesquisa utilizando uma abordagem baseada em análise de commit para coletar dados de evolução e que os resultados são baseados em uma análise de vários projetos de LPS de código aberto. Os principais resultados do estudo incluem uma análise detalhada da evolução de LPS, incluindo mudanças de variabilidade, e uma comparação dos resultados com outros estudos existentes na literatura. Os autores concluíram que as mudanças nas informações de variabilidade ocorrem com pouca frequência e afetam apenas pequenas partes dos artefatos analisados em LPS. Além disso, eles descobriram que existem diferenças nos detalhes dessas mudanças entre as LPS analisadas que não podem ser explicadas apenas pelas características das LPS ou seus processos de desenvolvimento. Essas descobertas sugerem que as abordagens de gestão de variabilidade devem ser adaptadas às características específicas de cada LPS.

Thüm et al. [27] apresentam uma análise eficiente de variação no tempo e no espaço em desenvolvimento de software. Nesse trabalho, os autores discutem como as técnicas de análise de regressão e de linha de produtos podem ser aplicadas para analisar variações em *featurer/models* e estratégias de desempenho para variantes e versões de software. É

destaca a necessidade de estudos empíricos para avaliar a eficácia das técnicas de análise de variação no tempo e no espaço e a necessidade de desenvolver ferramentas de análise de variação mais eficientes e precisas.

Lanna et al. [17] em seu trabalho, discutem desafios na verificação de qualidade e requisitos de sistemas de software em LPS, enfocando técnicas de verificação aplicadas. Destaca-se a dificuldade de lidar com a explosão exponencial de produtos e a limitação das técnicas atuais, especialmente na análise de confiabilidade. O artigo propõe um método eficiente para calcular a confiabilidade de todas as configurações em LPS, baseando-se em modelos comportamentais UML. Esse método se baseia em estratégias de *feature* e *family-based*, dividindo em unidades menores para análise mais eficiente e calculando a confiabilidade para todas as configurações simultaneamente. Resultados empíricos demonstram que essa abordagem supera outras estratégias de última geração em tempo e espaço, sendo escalável mesmo com um aumento significativo no espaço de configuração. Conclui-se que essa estratégia combina *feature* e *family-based*, evitando a enumeração de produtos e controlando o tamanho dos modelos analisados, representando uma alternativa promissora para a análise de confiabilidade em linhas de produtos de software. Os fundamentos da pesquisa em nosso projeto ASTool, embora se concentrem no código fonte, têm a possibilidade de serem utilizados em modelos, o que poderia maximizar a melhoria de análises em cache para esses modelos. Isso resultaria na criação de versões em cache da ReAna.

Capítulo 5

Conclusões

Nesta pesquisa, buscamos uma compreensão da evolução das Linhas de Produtos de Software, explorando duas questões de pesquisa fundamentais: a intensidade da evolução (QP1) e a estrutura da evolução (QP2).

Em relação à QP1 (Intensidade da Evolução), nossas análises revelaram que as LPS estudadas mantêm um equilíbrio notável entre mudanças frequentes e estabilidade. A métrica "Média de Arquivos Alterados por Commits"indicou que as modificações ocorrem em um número moderado de arquivos por commit, demonstrando que essas LPS estão em um estado de equilíbrio, onde as alterações ocorrem de forma controlada. Além disso, a métrica "Média de Gaps (Linhas não Alteradas entre Modificações no Código)"destacou a presença de áreas substanciais do código que permanecem inalteradas ao longo das modificações, sugerindo a existência de componentes ou funcionalidades estáveis.

Em relação à QP2 (Estrutura da Evolução), observamos um padrão consistente em todas as LPS analisadas. A métrica "Média de Profundidade de Nós Alterados na AST"indicou que as modificações tendem a ser de natureza mais superficial, envolvendo ajustes menores, correções de bugs e outras alterações que não afetam profundamente a organização do código-fonte.

Nossas descobertas contribuem para a área de Linhas de Produtos de Software, dando orientações valiosas para a gestão eficiente da evolução de LPS. Além disso, destacamos a relevância da memoização como uma técnica potencialmente benéfica para otimizar análises de LPS que evoluem, dada a natureza predominante de modificações superficiais.

Esta pesquisa contribui para o conhecimento existente sobre a dinâmica da evolução em LPS, fornecendo uma visão abrangente de sua intensidade e estrutura. E este estudo se conecta com a literatura relevante, destacando a importância da gestão da variabilidade em LPS e examinando a evolução de LPS com foco nas mudanças de variabilidade. Além disso, ele contribui para a compreensão de como a memoização pode ser aplicada

na evolução de sistemas altamente configuráveis, ampliando o corpo de conhecimento existente.

Nosso trabalho apresenta algumas limitações, como a análise de um conjunto específico de LPS e a necessidade de considerar outros fatores, como o contexto do projeto, ao interpretar os resultados. A ferramenta ASTool precisa ser continuamente aprimorada para realizar análises mais precisas e confiáveis. Essas limitações podem afetar a validade e a generalização de nossas descobertas.

Sugerimos que pesquisas futuras explorem a aplicação prática de técnicas como memoização em LPS. Por exemplo, a média de profundidade de nós alterados na AST mostra que a memoização se torna valiosa para evitar recalculos de ao armazenar resultados previamente computados, a memoização pode otimizar o desempenho em áreas de código sujeitas a mudanças complexas.

A análise da média de arquivos alterados por commits indica que a maioria das modificações impacta um número limitado de arquivos. O uso de timestamp, que registra a data e hora de cada alteração, permite rastrear quando as modificações foram feitas. A memoização pode ser aplicada para armazenar resultados calculados associados a cada versão do código. Isso evita recalculos não apenas com base nos arquivos alterados, mas também nas versões do código, melhorando a eficiência.

E a análise de média de gaps mostrou que quando partes significativas do código permanecem relativamente inalteradas a memoização se torna eficaz para áreas estáveis. Ela pode ser aplicada nas seções de código com médias de gaps mais elevadas, permitindo que resultados previamente calculados estejam prontamente disponíveis, economizando tempo de cálculo.

Além disso, a análise de LPS em diferentes contextos e o estudo de outras métricas de evolução podem ampliar ainda mais nossa compreensão desse fenômeno.

Referências

[1] axTLS github repository. https://github.com/dsheets/axtls. Acessado em 19 de agosto de 2023. 14

[2] BusyBox. https://busybox.net/. Accessed: 2023-9-22. 14

[3] Coreboot. https://www.coreboot.org/. Accessed: 2023-9-22. 15

[4] Soletta. https://https://github.com/solettaproject/soletta. Accessed: 2023-9-22. 15

[5] S Apel, D Batory, C K¨astner, and G Saake. Feature-oriented software product lines: Concepts and implementation, berlin/heidelberg, 2013, 308 pages. *URL http://www. springer. com/computer/swe/book/978-3-642-37520-0*. 1, 6

[6] María Cecilia Bastarrica, Pedro Rossel Cid, Jocelyn Simmonds, and Maíra Marques. Software product line evolution: A systematic literature review. 2019. 6

[7] Paul Clements and Linda Northrop. *Software product lines*. Addison-Wesley Boston, 2002. 4

[8] K. Czarnecki and Ulrich W. Eisenecker. Generative programming - methods, tools and applications. 2000. URL https://api.semanticscholar.org/CorpusID:31097208. 4, 6

[9] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Design Patterns. Elements of reusable object-oriented software. *Design Patterns*, 1995. 2

[10] Jilles Van Gurp, Jan Bosch, and Mikael Svahnberg. On the notion of variability in software product lines. In *2001 Working IEEE / IFIP Conference on Software Architecture (WICSA 2001), 28-31 August 2001, Amsterdam, The Netherlands*, pages 45–54. IEEE Computer Society, 2001. doi: 10.1109/WICSA.2001.948406. URL https://doi.org/10.1109/WICSA.2001.948406. 5

[11] Michel Jaring and Jan Bosch. Representing variability in software product lines: A case study. In *Software Product Lines: Second International Conference, SPLC 2 San Diego, CA, USA, August 19–22, 2002 Proceedings 2*, pages 15–36. Springer, 2002. 5

[12] Barbara A Kitchenham, Shari Lawrence Pfleeger, Lesley M Pickard, Peter W Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on software engineering*, 28(8):721–734, 2002. 33

[13] Jyrki Kontio, Gianluigi Caldiera, and Victor R Basili. Defining factors, goals and criteria for reusable component evaluation. In *CASCON*, volume 96, pages 12–14. Citeseer, 1996. 32

[14] Christian Kroher, Lea Gerling, and Klaus Schmid. Identifying the intensity of variability changes in software product line evolution. In *Proceedings of the 22nd International Systems and Software Product Line Conference-Volume 1*, pages 54–64, 2018. 8, 12, 13, 34

[15] Christian Kröher, Lea Gerling, and Klaus Schmid. Comparing the intensity of variability changes in software product line evolution. *Journal of Systems and Software*, 203:111737, 2023. 34

[16] Charles W Krueger. Software reuse. *ACM Computing Surveys (CSUR)*, 24(2):131–183, 1992. 1

[17] André Lanna, Thiago M. Castro, Vander Alves, Genaína Nunes Rodrigues, Pierre-Yves Schobbens, and Sven Apel. Feature-family-based reliability analysis of software product lines. *Inf. Softw. Technol.*, 94:59–81, 2018. doi: 10.1016/J.INFSOF.2017.10.001. URL `https://doi.org/10.1016/j.infsof.2017.10.001`. 35

[18] Maíra Marques, Jocelyn Simmonds, Pedro O. Rossel, and María Cecilia Bastarrica. Software product line evolution: A systematic literature review. *Information and Software Technology*, 105:190–208, 2019. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2018.08.014. URL `https://www.sciencedirect.com/science/article/pii/S0950584918301848`. 2

[19] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, and Gunter Saake. An overview on analysis tools for software product lines. In *Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools-Volume 2*, pages 94–101, 2014. 7

[20] Andreas Metzger and Klaus Pohl. Software product line engineering and variability management: achievements and challenges. *Future of software engineering proceedings*, pages 70–84, 2014. 5

[21] Donald Michie. "memo" functions and machine learning. *Nature*, 218:19–22, 1968. 30

[22] Leon J Osterweil. Software processes are software too, revisited: an invited talk on the most influential paper of icse 9. In *Proceedings of the 19th international conference on Software engineering*, pages 540–548, 1997. 33

[23] Klaus Pohl, Günter Böckle, and Frank Van Der Linden. *Software product line engineering: foundations, principles, and techniques*, volume 1. Springer, 2005. 1

[24] Aarne Ranta. Implementing programming languages, 2016. URL `https://www.cse.chalmers.se/edu/year/2012/course/DAT150/lectures/plt-book.pdf`. 9

[25] Diomidis Spinellis, Panos Louridas, and Maria Kechagia. The evolution of c programming practices: a study of the unix operating system 1973–2015. In *Proceedings of the 38th International Conference on Software Engineering*, pages 748–759, 2016. 33

[26] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys (CSUR)*, 47(1):1–45, 2014. 1, 7

[27] Thomas Thüm, Leopoldo Teixeira, Klaus Schmid, Eric Walkingshaw, Mukelabai Mukelabai, Mahsa Varshosaz, Goetz Botterweck, Ina Schaefer, and Timo Kehrer. Towards efficient analysis of variation in time and space. In *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume B*, pages 57–64, 2019. 34

[28] Rini Van Solingen, Vic Basili, Gianluigi Caldiera, and H Dieter Rombach. Goal question metric (gqm) approach. *Encyclopedia of software engineering*, 2002. 12

[29] Simon Wimmer, Shuwei Hu, and Tobias Nipkow. Verified memoization and dynamic programming. In J. Avigad and A. Mahboubi, editors, *Interactive Theorem Proving (ITP 2018)*, volume 10895, pages 579–596, 2018. 30

[30] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012. 12

Apêndice A

Instalação e Execução da ASTool

Este anexo fornece uma visão geral das etapas necessárias para instalar e executar a ASTool. Siga essas instruções para garantir uma integração bem-sucedida da ferramenta em seu ambiente de pesquisa.

Esses são os programas e bibliotecas necessários para o funcionamento da ASTool:

PROGRAMAS

VISUAL STUDIO CODE

1. Abra o terminal pressionando 'Ctrl + Alt + T'.
2. Atualize o cache do repositório do sistema com o seguinte comando: 'sudo apt update'
3. Instale o Visual Studio Code usando o comando apt: 'sudo apt install code'
4. Após a conclusão da instalação, você pode iniciar o Visual Studio Code a partir do menu de aplicativos ou executando 'code' no terminal.

GIT

1. Abra o terminal.
2. Atualize o cache do repositório do sistema novamente: 'sudo apt update'
3. Instale o Git com o seguinte comando: 'sudo apt install git'
4. Verifique se a instalação foi bem-sucedida digitando: 'git –version'

PYTHON

1. O Ubuntu 22.04 já deve incluir o Python 3 por padrão. Você pode verificar a versão do Python 3 instalada com o comando: 'python3 –version'

PIP

1. Abra o terminal
2. Certifique-se de que o Python 3 já esteja instalado, como mencionado acima.
3. Instale o pip usando o comando apt: 'sudo apt install python3-pip'
4. Verifique se a instalação foi bem-sucedida digitando: 'pip3 –version'

Agora que você instalou o Python e o Pip, você pode usar o Pip para instalar as bibliotecas Python.

CLANG

1. Abra o terminal pressionando Ctrl + Alt + T
2. Atualize o cache do repositório do sistema com o seguinte comando: 'sudo apt update'
3. Instale o Clang e o LLVM com o comando a seguir: 'sudo apt install clang llvm'
4. Após a conclusão da instalação, você pode verificar a versão do Clang com o comando: 'clang –version'

BIBLIOTECAS

Abra o terminal.

Para instalar cada biblioteca individualmente, use o comando pip3 seguido pelo nome da biblioteca. Por exemplo, para instalar a biblioteca os, use o seguinte comando: 'pip3 install os' Repita o processo para as outras bibliotecas mencionadas:

- Csv
- Git
- Shutil
- Difflib
- Pandas

Certifique-se de que os nomes das bibliotecas estejam escritos corretamente. Note que algumas dessas bibliotecas, como os e csv, são módulos padrão do Python e não precisam ser instaladas separadamente.

Depois de seguir essas etapas, você terá o Visual Studio Code, Git, Python e as bibliotecas Python instaladas no seu sistema Ubuntu 22 e poderá executar a ASTool.

EXECUÇÃO

1. Descompacte o arquivo em uma pasta local
2. Clone o repositório Git com o comando: git clone <url do repositório>
3. Abra o terminal e execute o script Executar.py com o comando: python Executar.py
4. Siga as instruções da tela

EXECUÇÃO PARTICIONADA

Depois de descompactar o arquivo da ASTool e clonar o repositório git você pode executar o programa script por script dependendo da sua necessidade. Porém, para fazer a análise de profundidade na AST os scripts devem ser executados na seguinte ordem:

- ExtComm.py
- ReComm.py
- MakeTree.sh
- Analise.py Medias.py

Apêndice B

Código Fonte ASTool

Este anexo traz o código fonte de todos os scripts que fazem parte da ASTool.
EXTCOMM.PY

```
import subprocess
import os

print("Extraindo commits. AGUARDE...\n\n")

def extract_commits():
    repo_path = input("Digite o caminho para o repositório local: ")

    commit_hash_command = 'git -C {} log --pretty=format:%H'.format(repo_path)  # Obtém os hashes de commit
    commit_hashes = subprocess.check_output(commit_hash_command.split()).decode().split('\n')

    for commit_hash in commit_hashes:
        if not commit_hash:
            continue
        # Obtém os arquivos alterados no commit
        commit_files_command = 'git -C {} diff-tree --no-commit-id --name-only -r {}'.format(repo_path, commit_hash)
        commit_files = subprocess.check_output(commit_files_command.split()).decode().split('\n')

        for file_path in commit_files:
            if not file_path:
                continue

            # Obtém o conteúdo do arquivo no commit
            commit_content_command = 'git -C {} show --raw --no-renames --no-textconv --no-color --binary {}:{}'
            .format(repo_path, commit_hash, file_path)
            try:
                commit_content = subprocess.check_output(commit_content_command.split())
            except subprocess.CalledProcessError:
                print('Arquivo não encontrado no commit {}: {}'.format(commit_hash, file_path))
                continue

            # Cria o diretório de saída (se não existir)
            output_dir = os.path.join('commits', commit_hash[:7])
            os.makedirs(output_dir, exist_ok=True)

            # Escreve o conteúdo binário do commit em um arquivo separado
            output_file = os.path.join(output_dir, file_path.replace('/', '-'))
            with open(output_file, 'wb') as f:
                f.write(commit_content)

            print('Arquivo criado: {}'.format(output_file))

if __name__ == '__main__':
    extract_commits()
```

RECOMM.PY

```
import os
import shutil

print("Organizando os commits. AGUARDE...\n\n")

def reorganize_files():
    commits_path = 'commits'
    output_path = 'commits_org'

    for root, _, files in os.walk(commits_path):
        for file_name in files:
            if file_name.endswith('.c'):
                commit_date = os.path.basename(root)
                commit_hash = os.path.basename(os.path.dirname(root))
                file_path = os.path.join(root, file_name)

                output_dir = os.path.join(output_path, file_name)
                os.makedirs(output_dir, exist_ok=True)

                output_file = os.path.join(output_dir, commit_hash[:7] + '_' + commit_date + '.c')
                shutil.copy2(file_path, output_file)

                print('Arquivo movido: {}'.format(output_file))

if __name__ == '__main__':
    reorganize_files()
```

MAKETREE.SH

```
#!/bin/bash

# Pasta de entrada
pasta_origem="commits_org"

# Pasta de saída
pasta_destino="asts"

# Função recursiva para percorrer as subpastas
function percorrer_subpastas() {
    local pasta_atual=$1
    local pasta_relativa=${pasta_atual#$pasta_origem} # Obter a parte relativa do caminho

    # Loop pelos arquivos C na pasta atual
    for arquivo in "$pasta_atual"/*.c; do
        nome_arquivo=$(basename "$arquivo")
        nome_sem_extensao="${nome_arquivo%.*}"

        # Caminho de saída com a estrutura de subpastas preservada
        arquivo_saida="${pasta_destino}${pasta_relativa}/${nome_sem_extensao}.txt"
        # Criar a pasta de destino, se não existir
        mkdir -p "$(dirname "$arquivo_saida")"

        # Gerar a AST do arquivo e gravar em um arquivo de texto (suprimindo os erros)
        clang -Xclang -detailed-preprocessing-record -Xclang -ast-dump "$arquivo" > "$arquivo_saida" 2>/dev/null
    done

    # Loop pelas subpastas na pasta atual
    for subpasta in "$pasta_atual"/*; do
        if [ -d "$subpasta" ]; then
            percorrer_subpastas "$subpasta"
        fi
    done
}

# Verificar se a pasta de entrada existe
if [ ! -d "$pasta_origem" ]; then
```

```
    echo "A pasta de entrada 'commits_org' não foi encontrada!"
    exit 1
fi

# Verificar se a pasta de saída existe ou criar se não existir
if [ ! -d "$pasta_destino" ]; then
    mkdir -p "$pasta_destino"
fi

# Chamada inicial para percorrer as subpastas
percorrer_subpastas "$pasta_origem"

echo "ARVORES CRIADAS!"
```

ANALISE.PY

```
import os
import csv
import difflib

print("Fazendo as análises. AGUARDE... \n\n")

# Função para calcular a média de uma lista de valores numéricos
def calcular_media(lista):
    return sum(lista) / len(lista)

# Função para comparar as ASTs em pares e armazenar as diferenças em um arquivo CSV
def comparar_ast(arquivos, pasta_resultados):
    for i in range(len(arquivos) - 1):
        arquivo_antigo = arquivos[i]
        arquivo_novo = arquivos[i + 1]

        with open(arquivo_antigo, 'r') as file_antigo, open(arquivo_novo, 'r') as file_novo:
            linhas_antigo = file_antigo.readlines()
            linhas_novo = file_novo.readlines()

            diff = difflib.unified_diff(linhas_antigo, linhas_novo, lineterm='')
            mudancas = list(diff)

            if mudancas:
                nome_pasta_antigo = os.path.basename(os.path.dirname(arquivo_antigo))
                nome_pasta_novo = os.path.basename(os.path.dirname(arquivo_novo))

                # Criar o nome do arquivo de resultado
                nome_arquivo_resultado = f'resultadoMedia_{nome_pasta_novo}.csv'

                # Caminho completo para o arquivo de resultado
                caminho_arquivo_resultado = os.path.join(pasta_resultados, nome_arquivo_resultado)

                profundidades = []  # Lista para armazenar as profundidades para o cálculo da média

                with open(caminho_arquivo_resultado, 'w', newline='') as csvfile:
                    fieldnames = ['Arquivo', 'Arquivo Comparado', 'Nó Modificado', 'Profundidade']
                    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
                    writer.writeheader()

                    for linha in mudancas:
                        if linha.startswith('+'):
                            # Identificar o nó alterado
```

```
                        no_alterado = linha[1:].strip()

                        # Calcular a profundidade do nó alterado
                        profundidade = no_alterado.count('  ')
                        profundidades.append(profundidade)

                        if profundidade > 0:
                            writer.writerow({'Arquivo': nome_pasta_novo,
                                             'Arquivo Comparado': nome_pasta_antigo,
                                             'Nó Modificado': no_alterado,
                                             'Profundidade': profundidade})

                # Calcular a média das profundidades e escrever a linha "média" no arquivo
                if profundidades:
                    media_profundidades = calcular_media(profundidades)
                    writer.writerow({'Arquivo': '', 'Arquivo Comparado': '', 'Nó Modificado': 'Média',
                    'Profundidade': media_profundidades})

# Solicitar o caminho da pasta
pasta = 'asts'

# Verificar se o caminho da pasta é válido
if not os.path.isdir(pasta):
    print("Caminho inválido!")
    exit(1)

# Solicitar o caminho da pasta para os resultados
pasta_resultados = os.path.join('analises')
os.makedirs(pasta_resultados, exist_ok=True)

# Verificar se o caminho da pasta para os resultados é válido
if not os.path.isdir(pasta_resultados):
    print("Caminho inválido!")
    exit(1)

# Obter a lista de arquivos de texto nas subpastas da pasta principal
arquivos_ast = []
for diretorio_raiz, subpastas, arquivos in os.walk(pasta):
    for arquivo in arquivos:
        if arquivo.endswith('.txt'):
            caminho_arquivo = os.path.join(diretorio_raiz, arquivo)
            arquivos_ast.append(caminho_arquivo)

# Comparar as ASTs nos arquivos e armazenar os resultados em arquivos CSV separados
comparar_ast(arquivos_ast, pasta_resultados)

print("Os resultados foram armazenados em arquivos CSV na pasta indicada.")
```

MEDIAS.PY

```
import os
import csv

print("\nCalculando as médias de profundidade. AGUARDE... \n\n")

def calcular_media_arquivo(nome_arquivo):
```

```
    soma = 0
    contador = 0
    with open(nome_arquivo, 'r') as arquivo:
        leitor = csv.DictReader(arquivo)
        for linha in leitor:
            profundidade = float(linha['Profundidade'])
            soma += profundidade
            contador += 1
    if contador > 0:
        media = soma / contador
        return media
    return 0

def calcular_media_geral(pasta):
    lista_arquivos = os.listdir(pasta)
    valores_media = []
    for arquivo in lista_arquivos:
        if arquivo.endswith('.csv'):
            caminho_arquivo = os.path.join(pasta, arquivo)
            media = calcular_media_arquivo(caminho_arquivo)
            valores_media.append(media)

    media_geral = sum(valores_media) / len(valores_media) if len(valores_media) > 0 else 0

    with open('media_geral.csv', 'w', newline='') as arquivo_saida:
        escritor = csv.writer(arquivo_saida)
        escritor.writerow(['Arquivo', 'Média'])
        for arquivo, media in zip(lista_arquivos, valores_media):
            # Formatar a média com até 4 casas decimais
            media_formatada = "{:.4f}".format(media)
            escritor.writerow([arquivo, media_formatada])
        media_geral_formatada = "{:.4f}".format(media_geral)
        escritor.writerow(['Média Geral', media_geral_formatada])

    print("Cálculo das médias concluído. O arquivo 'media_geral.csv' foi gerado.")

# Solicitar ao usuário a pasta de origem dos arquivos CSV
pasta_origem = "analises"

calcular_media_geral(pasta_origem)
```

COMMIT_FILES.PY

```
import csv
from git import Repo

# Caminho para o repositório Git clonado
repo_path = input("\nDigite o caminho para o repositório: ")

print("Calculando a média de arquivos por commit. AGUARDE... \n\n")

# Caminho para o arquivo CSV de saída
output_csv_path = 'ArquivosPorCommits.csv'

# Inicializa o repositório
repo = Repo(repo_path)
```

```
# Cria o arquivo CSV e escreve o cabeçalho
with open(output_csv_path, 'w', newline='') as csvfile:
    fieldnames = ['Commit', 'Quant_Arquivos']  # Cabeçalho base
    commits = list(repo.iter_commits())
    max_files = 0

    # Calcula o número máximo de arquivos alterados por commit
    for commit in commits:
        max_files = max(max_files, len(commit.stats.files))

    # Adiciona os nomes dos arquivos como colunas
    for i in range(max_files):
        fieldnames.append(f'Arquivo{i + 1}')

    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
    writer.writeheader()

    # Preenche as informações dos commits no arquivo CSV
    total_quant_arquivos = 0
    for commit in commits:
        commit_info = {
            'Commit': commit.hexsha,
            'Quant_Arquivos': len(commit.stats.files)
        }
        total_quant_arquivos += len(commit.stats.files)

        for i, file in enumerate(commit.stats.files.keys()):
            commit_info[f'Arquivo{i + 1}'] = file

        writer.writerow(commit_info)

    # Calcula e escreve a média da coluna Quant_Arquivos
    average_quant_arquivos = total_quant_arquivos / len(commits)
    writer.writerow({'Commit': 'Média', 'Quant_Arquivos': average_quant_arquivos})

print(f"As informações dos commits e a média foram escritas em '{output_csv_path}'.")
```

GAPSCOMM.PY

```
import csv
from git import Repo

# Caminho para o repositório Git clonado
repo_path = input("\nDigite o caminho para o repositório: ")

print("Calculando a média de arquivos por commit. AGUARDE... \n\n")

# Caminho para o arquivo CSV de saída
output_csv_path = 'ArquivosPorCommits.csv'

# Inicializa o repositório
repo = Repo(repo_path)

# Cria o arquivo CSV e escreve o cabeçalho
with open(output_csv_path, 'w', newline='') as csvfile:
```

```
    fieldnames = ['Commit', 'Quant_Arquivos']  # Cabeçalho base
    commits = list(repo.iter_commits())
    max_files = 0

    # Calcula o número máximo de arquivos alterados por commit
    for commit in commits:
        max_files = max(max_files, len(commit.stats.files))

    # Adiciona os nomes dos arquivos como colunas
    for i in range(max_files):
        fieldnames.append(f'Arquivo{i + 1}')

    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
    writer.writeheader()

    # Preenche as informações dos commits no arquivo CSV
    total_quant_arquivos = 0
    for commit in commits:
        commit_info = {
            'Commit': commit.hexsha,
            'Quant_Arquivos': len(commit.stats.files)
        }
        total_quant_arquivos += len(commit.stats.files)

        for i, file in enumerate(commit.stats.files.keys()):
            commit_info[f'Arquivo{i + 1}'] = file

        writer.writerow(commit_info)

    # Calcula e escreve a média da coluna Quant_Arquivos
    average_quant_arquivos = total_quant_arquivos / len(commits)
    writer.writerow({'Commit': 'Média', 'Quant_Arquivos': average_quant_arquivos})

print(f"As informações dos commits e a média foram escritas em '{output_csv_path}'.")
```