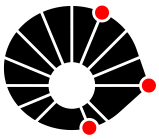

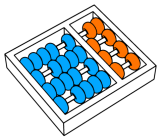

Desenvolvimento de uma API para Análise de Código Fonte com Diretivas de Pré-processamento em Sistemas Configuráveis

Fabio Santos Villar, Bruno Barbieri de Pontes Cafeo

Relatório Técnico - IC-PFG-23-59
Projeto Final de Graduação
2023 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Desenvolvimento de uma API para Análise de Código Fonte com Diretivas de Pré-processamento em Sistemas Configuráveis

Fabio Santos Villar*, Bruno Barbieri de Pontes Cafeo†

12/2023

Resumo

Ao se tratar de sistemas configuráveis utilizando arquivos com código-fonte em C, é extremamente comum o uso de diretivas de pré-processamento para permitir uma maior escalabilidade do sistema. Com isso, torna-se possível a inclusão condicional de trechos de código, de acordo com as condições de presença descritas nas diretivas de pré-processamento.

Este artigo apresenta um projeto de pesquisa que trata da construção de uma API que possui, como propósito, ser uma ferramenta de auxílio para facilitar a extração de informações relevantes das configurações de sistemas configuráveis.

O sistema desenvolvido cria um arquivo XML a partir do código-fonte, com tags que identificam elementos da sintaxe abstrata da linguagem, construindo, assim, uma AST (Abstract Syntax Tree), sem perder nenhuma das informações provenientes das diretivas de pré-processamento, que será analisado e terá informações relevantes devolvidas ao usuário da API. Esta possibilita a obtenção de detalhes sobre as condições de presença e a listagem de segmentos de código variável com base nessa estrutura de dados. O projeto compreendeu uma análise bibliográfica abrangente das diretivas de pré-processamento, a formulação do design da solução e a subsequente implementação.

Prevê-se que a API criada simplifique as tarefas dos programadores que trabalham com sistemas configuráveis em C, proporcionando uma análise de código-fonte mais eficaz e confiável, especialmente no que diz respeito às diretivas de pré-processamento.

1 Introdução

1.1 Motivação

No contexto de sistemas configuráveis, nos quais diferentes configurações de software são necessárias para atender a requisitos específicos, é comum encontrar implementadas diretivas de pré-processamento no código-fonte. Essas diretivas possibilitam a inclusão ou exclusão condicional de trechos de código, conferindo maior flexibilidade e adaptabilidade ao código em diversos cenários.

*Instituto de Computação, Universidade Estadual de Campinas, Campinas, SP, 13083-852
†Instituto de Computação, Universidade Estadual de Campinas, Campinas, SP, 13083-852

No entanto, a análise manual de código contendo essas diretivas pode ser tediosa e suscetível a erros. Diante desse cenário, o objetivo deste projeto é desenvolver uma API que simplifique a extração de informações relevantes dessas configurações. Isso será alcançado ao proporcionar uma Árvore de Sintaxe Abstrata (AST) do código-fonte, juntamente com uma API para acessar informações sobre as condições de presença e a quantidade de trechos de código variável.

Destrinchando um pouco mais a análise manual de código contendo diretivas de pré-processamento, é fácil perceber que, ao se tratar de sistemas maiores e mais complexos, com muitas condições de presença, torna-se maior a chance de ser cometido um erro, visto que programadores podem acabar negligenciando condições específicas, o que acarretaria em bugs e problemas a serem resolvidos. Além disso, deve-se considerar que, ao se desenvolver um código, mudanças feitas no sistema podem comprometer o projeto, causando falhas, e com análises manuais, é mais difícil de se analisar impactos de atualizações feitas. [1]

1.2 Objetivos

O propósito principal deste projeto de pesquisa consistiu em criar um sistema com API que tivesse a capacidade de examinar códigos-fonte que contenham diretrizes de pré-processamento em sistemas configuráveis implementados na linguagem C.

Este projeto envolveu uma investigação aprofundada das diretivas de pré-processamento em linguagem C, com o objetivo de identificar suas diversas formas e comportamentos. Ao se criar um arquivo XML a partir do código-fonte, que possui todas as informações originais, incluindo as condições de presença das diretivas de pré-processamento, é possível analisar o código fonte e extrair informações relevantes dele.

Além disso, o projeto incluiu o design e a implementação de uma API que possibilitasse a extração de informações relevantes do XML. Especificamente, a API seria capaz de fornecer uma lista das condições de presença presentes no código e também dos blocos de código relacionados à cada condição.

Para validar a eficácia da API, foram conduzidos testes utilizando uma variedade de códigos-fonte provenientes de sistemas configuráveis. Esses testes visaram garantir a robustez e a confiabilidade do projeto em diferentes contextos e cenários.

A conclusão do projeto incluiu a documentação abrangente da API, apresentando informações claras e detalhadas. Foram fornecidos exemplos de uso da biblioteca, juntamente com orientações destinadas aos desenvolvedores, a fim de facilitar sua integração e utilização eficiente em projetos futuros.

Encontrar as melhores formas de se configurar um sistema é fundamental para que seu desempenho ótimo seja atingido, visto que as condições de presença se conectam de formas específicas. [2] Espera-se que a ferramenta criada seja capaz de auxiliar programadores em suas análises de desempenho.

1.3 Metodologia

Para o desenvolvimento da API e do restante do sistema, ao início do projeto final de graduação, foi definida uma metodologia para estruturar todo o trabalho que viria a seguir.

O desenvolvimento do sistema passou por diversas etapas cruciais para assegurar sua qualidade e eficácia. Inicialmente, foi conduzida uma revisão bibliográfica detalhada, abrangendo as diretrizes de pré-processamento em C, juntamente com estudos pertinentes relacionados à análise de código-fonte e à construção de Abstract Syntax Trees (ASTs).

Posteriormente, foi realizado o projeto da API, envolvendo a definição da sua estrutura, a identificação das funcionalidades principais e o seu design para interação com os desenvolvedores. Nesse contexto, foi de suma importância considerar o projeto arquitetural de maneira a tornar o sistema facilmente extensível para a incorporação de novas funcionalidades.

A etapa seguinte consistiu na implementação efetiva da API, englobando o desenvolvimento do código, a geração da AST e a implementação da API destinadas à extração de informações relevantes.

Para validar a robustez e eficiência da biblioteca, foram realizados testes utilizando uma variedade de códigos-fonte provenientes de sistemas configuráveis. Esses testes visam garantir a precisão e eficácia da API em diferentes contextos de aplicação.

Finalmente, foi elaborada uma documentação do sistema e API, incluindo um guia de uso, exemplos práticos e informações sobre como contribuir ou expandir suas funcionalidades. Essa documentação visa proporcionar aos usuários uma compreensão abrangente da API, facilitando sua utilização e potencial contribuição para seu aprimoramento contínuo.

1.4 Sistemas configuráveis

Atualmente, várias organizações utilizam programas que compartilham um núcleo comum, mas podem ser adaptados para atender a diferentes propósitos, de acordo com as preferências do usuário. Esses programas oferecem a capacidade de configuração específica para cada cenário de operação, proporcionando assim uma notável flexibilidade e usabilidade. Conjuntos de software ou hardware que atendem a essas especificações são designados como sistemas configuráveis, constituindo a base deste projeto de pesquisa.

Em muitos sistemas contemporâneos, como bancos de dados, servidores web, compiladores e outros, é possível combinar centenas de configurações de opções. A complexidade resultante dessas inúmeras combinações torna desafiador encontrar a configuração ideal para que o sistema funcione conforme as intenções do desenvolvedor. Tem-se que analisar diversas possibilidades de configurações para se encontrar a ideal, e existem muitas formas de se encontrar as melhores possibilidades, como utilizando algoritmos de amostragens [3] ou utilizando técnicas de Machine Learning, por exemplo. [4]

Uma técnica empregada para lidar com o problema de se ter múltiplas funções para um sistema é o pré-processamento. Em códigos C, por exemplo, são incorporadas condições de presença no código para tornar o controle dos blocos de código mais flexível e fácil. As diretrizes de pré-processamento podem abranger desde estruturas completas de código, como funções, até a simples associação de uma variável a uma variabilidade.

Ao analisar o código-fonte de um sistema configurável extenso, repleto de dezenas de variáveis e que realiza chamadas a centenas de funções de APIs, é possível que o número de violações nas configurações resultantes seja consideravelmente alto. Além disso, as implicações decorrentes dessas violações nos padrões permanecem desconhecidas, represen-

tando assim um desafio significativo para a compreensão e correção adequada no contexto mencionado.

2 Trabalhos relacionados

O grande ponto a ser considerado não apenas neste trabalho, como também nos demais, reside na complexidade percebida ao se escalar um sistema configurável, com um aumento no número de condições de presença, e também na utilização de diretivas de pré-processamento aninhadas. Quanto maior esta complexidade, a chance de ocorrer um erro de julgamento ao se analisar a funcionalidade de um sistema aumenta junto, o que dificulta ainda mais o trabalho de programadores que queiram analisar sistemas configuráveis de forma apenas manual, sem utilização de ferramentas de auxílio, como, por exemplo, a API e o sistema desenvolvidos neste trabalho.

No artigo "Performance-Influence Model for Highly Configurable Software with Fourier Learning and Lasso Regression", escrito por Huong Ha e Hongyu Zhang, tem-se um foco muito grande em analisar a performance de um sistema configurável de acordo com cada configuração específica, e entender como cada opção de configuração influencia na performance obtida por meio da análise. [5]

Considerando-se que não seria viável medir a performance do sistema considerando todas as possibilidades, visto que o número poderia aumentar de forma exponencial, os autores Ha e Zhang propuseram PerLasso, um modelo de performance e predição baseado em Fourier e Lasso (Least absolute shrinkage and selection operator - Operador de Redução e Seleção com Menor Valor Absoluto), este último sendo um método estatístico considerado uma melhoria em relação ao método de mínimos quadrados, possuindo o objetivo de escolher variáveis de maneira automática, de forma a regularizar um modelo. Comparados aos modelos vigentes até então, os autores obtiveram uma acurácia maior.

Já no artigo intitulado "Performance-Influence Models for Highly Configurable Systems", redigido por Norbert Siegmund, Alexander Grebhahn, Sven Apel e Christian Kästner, a equipe de pesquisa conduziu uma análise abrangente sobre as complexidades associadas à configuração de sistemas de acordo com requisitos específicos. O objetivo central foi desenvolver modelos de influência no desempenho, bem como modelos para outros atributos mensuráveis de qualidade, como consumo de energia. Esses modelos visam descrever como as opções de configuração e suas interações influenciam o desempenho global de um sistema. Por exemplo, eles exploraram como diferentes configurações impactam variáveis como taxa de transferência ou tempo de execução de benchmarks. A construção desses modelos é fundamental para compreender como as decisões de configuração podem afetar a qualidade e a eficiência de um sistema. [6]

A maioria significativa dos estudos existentes está voltada para a problemática atual relacionada à dificuldade de configurar um sistema conforme desejado. No artigo mencionado anteriormente, a equipe de desenvolvimento empregou técnicas de aprendizado de máquina para identificar disparidades entre diversas abordagens de configuração de um sistema, realizando uma avaliação comparativa abrangente dos diferentes trajetos do sistema por meio de benchmarks. Eles enfrentaram desafios específicos, como lidar com formatos binários e

numéricos, os quais demandam um tratamento mais cuidadoso em comparação com outras opções.

Um outro estudo relacionado a essa área é "Interaction Testing of Highly-Configurable Systems in the Presence of Constraints", redigido por Myra B. Cohen, Matthew B. Dwyer e Jiangfan Shi. Neste trabalho, a equipe concentrou-se em elucidar a diversidade e a categorização de restrições ("constraints") presentes em sistemas altamente configuráveis. O objetivo foi examinar a presença dessas restrições em sistemas de natureza não trivial, com a intenção de compartilhar técnicas e abordagens para lidar com tais situações. [7]

Para isso, utilizaram testes de iterações combinatórias (do inglês CIT - Combinatorial interaction testing), método utilizado para criar amostras de possíveis configurações de sistemas. O artigo dos autores, ao contrário dos demais, tem como foco lidar com as restrições de forma aprimorada, de forma a ter uma análise de performance.

3 Ferramentas

3.1 srcML

O Source ML (srcML) é uma representação em XML de código fonte, no qual as tags identificam elementos da sintaxe abstrata da linguagem. Isso facilita significativamente a análise de arquivos de código. Para o propósito deste projeto, o uso do srcML foi essencial para analisar códigos em linguagem C. As tags do srcML já possuem identificadores apropriados para cada diretiva de pré-processamento, simplificando a construção do sistema e da API.

Pode ser utilizado com linguagens de programação como C, C++, C# e Java. Além disso, não há perda de código durante a conversão, e a ferramenta apresenta uma rápida velocidade de transformação, podendo processar cerca de 3000 arquivos por minuto. É amplamente empregado em sistemas de grande escala e serve como suporte a APIs.

3.2 FastAPI

FastAPI é um framework moderno, de alta performance e notável rapidez, destinado à construção de APIs utilizando Python 3.8+ ou tipos básicos de Python. Destaca-se como um dos frameworks mais velozes do ecossistema Python, rivalizando em velocidade com tecnologias como NodeJS e Go. Sua concepção visa a simplicidade e facilidade de aprendizado e utilização.

O framework automatiza a validação de dados de entrada, gerando mensagens de erro prontas para qualquer anomalia detectada. Adicionalmente, o FastAPI cria automaticamente documentação interativa para a API desenvolvida, acessível através do navegador.

Além de oferecer suporte integrado para autenticação, autorização e websockets (permitindo a criação de APIs em tempo real), o FastAPI também se destaca pela gestão de dependências e compatibilidade com diversas outras ferramentas.

A iniciativa do FastAPI partiu de Sebastian Ramirez, que identificou as limitações de outros frameworks existentes. Muitos deles não acompanhavam as últimas inovações do Python, como os type hints. Assim, o FastAPI surge como uma solução moderna e robusta para atender às demandas mais contemporâneas no desenvolvimento de APIs em Python.

4 Arquitetura

A arquitetura do sistema é baseada em um modelo cliente-servidor, conforme apresentado na imagem abaixo. Essa estrutura utiliza uma API construída com a ferramenta FastAPI, mencionada anteriormente. O cliente pode realizar três tipos de requisições REST à API.

No primeiro cenário, um POST Request possibilita o envio de um arquivo em linguagem C para análise pelo sistema. O arquivo é armazenado no banco de dados da aplicação para uso posterior, caso o cliente decida obter informações relacionadas a ele.

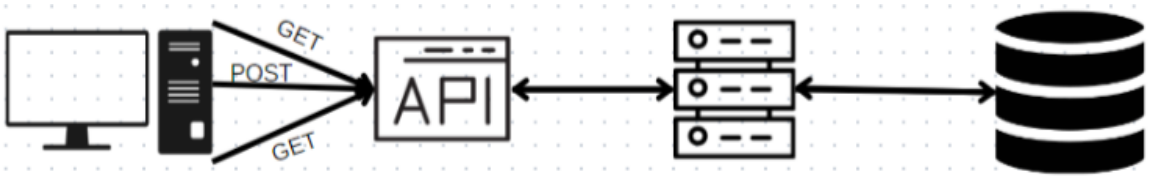

Figura 1: Diagrama do sistema

Em um segundo cenário, um GET Request é utilizado para obter uma lista com todas as condições de presença existentes em um arquivo específico de código em linguagem C. O cliente deve fornecer o nome do arquivo na requisição, e a resposta é emitida em formato JSON. Se o nome do arquivo não for encontrado no banco de dados, um erro é gerado.

No terceiro cenário, outro GET Request é empregado para obter um objeto em formato JSON contendo todos os blocos de código relativos a cada condicional de presença em um arquivo de código em linguagem C. Assim como no cenário anterior, o cliente deve fornecer o nome do arquivo na requisição, e a resposta é emitida em formato JSON. Se o nome do arquivo não for encontrado, um erro é gerado.

Essas funcionalidades proporcionam uma interação eficiente entre o cliente e a API, permitindo a realização de análises e a obtenção de informações específicas relacionadas a arquivos em linguagem C.

5 Desenvolvimento

5.1 Criação da AST

O processo de desenvolvimento do sistema e da API teve início com uma minuciosa análise do funcionamento do comando de linha de comando do srcML. Para realizar essa investigação, foram empregados diversos códigos de exemplo, contendo diretivas de pré-processamento. A execução do comando específico para esse propósito foi o seguinte: "srcml test.c -o test.c.xml".

Esse comando pressupõe a existência de um arquivo em linguagem C, nomeado "test.c", no mesmo diretório do terminal. Em situações em que o arquivo não estivesse localizado nesse diretório, seria necessário incluir o caminho adequado para o comando de terminal e, posteriormente, executá-lo. Após conduzir análises em diversas instâncias com códigos distintos, tornou-se evidente que a geração do XML a partir do código fonte estava operando de maneira impecável.

```c
#include stdio.h

#define FEATURE_ENABLED

int main() {
    #ifdef FEATURE_ENABLED
        printf(Feature is enabled!n);
    #else
        printf(Feature is disabled!n);
    #endif

    return 0;
}
```

Figura 2: Código fonte de exemplo simples

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<unit xmlns="http://www.srcML.org/srcML/src" xmlns:cpp="http://www.srcML.org/srcML/cpp" revision="1.0.0" language="C" filename="u

<cpp:define>#<cpp:directive>define</cpp:directive> <cpp:macro><name>FEATURE_ENABLED</name></cpp:macro></cpp:define>

<function><type><name>int</name></type> <name>main</name><parameter_list>()</parameter_list> <block>{<block_content>
    <cpp:ifdef>#<cpp:directive>ifdef</cpp:directive> <name>FEATURE_ENABLED</name></cpp:ifdef>
        <expr_stmt><expr><call><name>printf</name><argument_list>(<argument><expr><literal type="string">"Feature is enabled!\n"<
    <cpp:else>#<cpp:directive>else</cpp:directive></cpp:else>
        <expr_stmt><expr><call><name>printf</name><argument_list>(<argument><expr><literal type="string">"Feature is disabled!\n"
    <cpp:endif>#<cpp:directive>endif</cpp:directive></cpp:endif>

    <return>return <expr><literal type="number">0</literal></expr>;</return>
</block_content>}</block></function>
</unit>
```

Figura 3: AST do código fonte em XML

Essa etapa crítica de avaliação assegurou que o srcML estava apto a interpretar e converter eficazmente códigos-fonte em linguagem C para sua representação em XML. A observação detalhada do comando, aliada à execução em cenários variados, contribuiu para validar a integridade e a precisão do processo de geração do XML, constituindo um marco fundamental no desenvolvimento do sistema e da API.

Considerando o código fonte da Figura 2 como exemplo, pode-se verificar a existência de cinco diretivas de pré-processamento: “include”, que traz ao código uma biblioteca da linguagem C; “define”, que define uma variável, neste caso, FEATURE_ENABLED; “ifdef”, que é uma diretiva condicional de pré-processamento, ligada ao valor da variável definida anteriormente; “else” corresponde ao caso em que a “ifdef” possui valor falso, e “endif” indica o fim do bloco de condicional iniciado por “ifdef” anteriormente.

Ao ser executado o comando de linha de terminal, tem-se como resultado o arquivo da Figura 3.

O arquivo XML é gerado de maneira abrangente, capturando todos os dados do arquivo original com código fonte, sem qualquer perda de informação. Uma observação detalhada revela que o XML é estruturado com tags de identificação para cada elemento

presente no código. Por exemplo, encontramos a tag "cpp:define", que corresponde a uma diretiva de pré-processamento para definir uma variável, como no caso da variável FEATURE_ENABLED. Além disso, identificam-se as tags "cpp:ifdef", "cpp:else"e "cpp:endif", representando diretivas condicionais de pré-processamento. Vale mencionar que as diretivas "include"também são representadas no XML, mesmo que não estejam visíveis na imagem.

A presença dessas tags específicas no XML facilita significativamente a identificação e extração de informações cruciais do código fonte original. Essa representação estruturada simplifica o processo de continuação do desenvolvimento, proporcionando uma base sólida para a obtenção de dados essenciais a partir do arquivo XML gerado. Essa abordagem aprimorada promove a eficiência na extração de informações pertinentes para o prosseguimento do trabalho de desenvolvimento.

5.2 Utilização da API

O XML gerado a partir do código-fonte em C é armazenado no banco de dados, acompanhado pelo próprio arquivo original em C, para permitir acesso em caso de requisições à API desenvolvida. A construção da API foi realizada utilizando o FastAPI, empregando a linguagem de programação Python. A API oferece três opções de requisições REST:

1. POST para upload do código-fonte em C: Este método permite o envio e armazenamento do código-fonte em C, o qual é gravado juntamente com o arquivo original.

2. GET para obter um JSON com uma lista de todas as condições de presença: Essa requisição retorna um JSON contendo uma lista completa de todas as condições de presença originadas das diretivas de pré-processamento presentes no código-fonte.

3. GET para obter um JSON com uma lista das linhas de comando relacionadas a cada condição de presença: Ao efetuar essa requisição, é possível obter um JSON que contém uma lista das linhas de comando associadas a cada condição de presença identificada no código-fonte.

Na Figura 4, apresenta-se uma captura de tela da documentação da API criada, exibindo os três tipos de solicitações REST disponíveis. Iniciando pelo método POST e considerando o arquivo em linguagem C previamente mencionado (Figura 2), é notável que o upload para o banco de dados é bem-sucedido. A resposta é acompanhada de um status 200, indicando sucesso, e uma mensagem textual confirmando o salvamento do arquivo.

Considerando que seja executado o primeiro GET Request, da lista das condições, tem-se que um arquivo JSON será retornado ao cliente, contendo uma listagem de todas as condições de presença existentes no código fonte em C.

No caso sendo considerado, existem duas possibilidades de condições: a representada pelo "ifdef", na qual a variável criada possui valor verdadeiro; e a representada pelo "else", com valor contrário.

Por conseguinte, ao se exibir uma lista das condições de presença deste caso, tem-se apenas duas, corretamente retornadas pela API, e exibidas na Figura 6, no arquivo JSON de resposta.

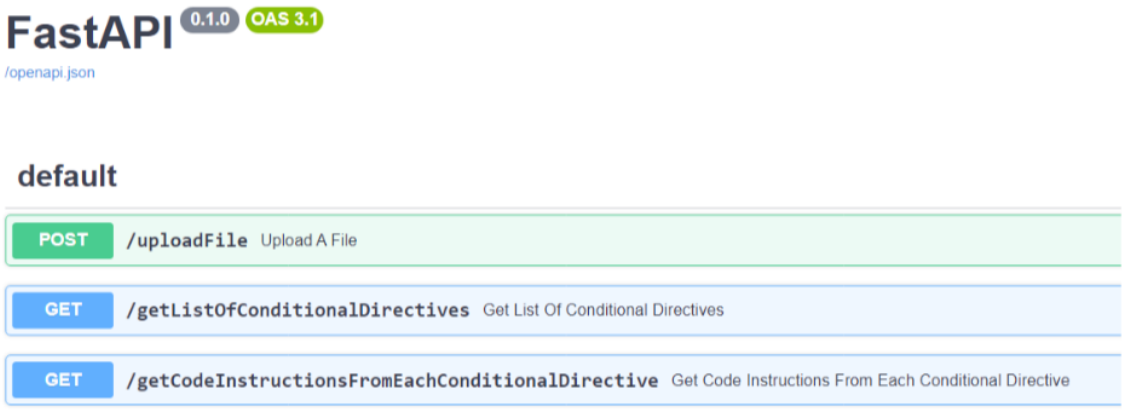

Figura 4: Docs da API construída

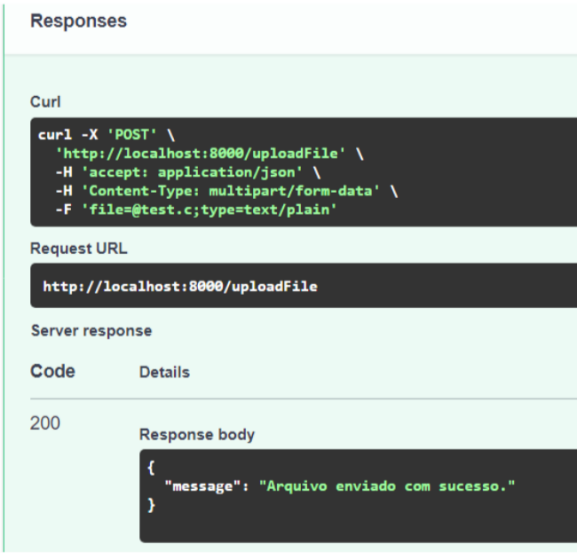

Figura 5: Resposta do POST Request

Figura 6: Resposta do GET Request da listagem das condições

E, por fim, considerando-se o segundo GET Request possível, que tem como finalidade exibir os blocos de código relativos a cada condição de presença das diretivas de pré-processamento, tem-se como resposta a Figura 7.

Figura 7: Resposta do GET Request dos blocos de código

Como pode-se verificar, caso a variável "FEATURE_ENABLED" seja definida como verdadeira na configuração do sistema, a linha de comando "printf("Feature is enabled!");" será executada. Caso contrário, "printf("Feature is disabled!");" é que será usada no lugar. A API funcionou corretamente para este caso, que corresponde a um código fonte simples.

O código fonte considerado para este teste foi simples, sendo aquele mostrado na Figura 2. O sistema e a API também funcionam para casos mais complexos, com múltiplas

```c
int main() {
    #ifdef NUMERO
        printf("A macro NUMERO está definida e seu valor é: %d\n", NUMERO);
        open();
    #elif B
        printf("NUMERO nao esta definida e B sim")
    #else
        printf("A macro NUMERO não está definida e B tambem nao.\n");
        close();
    #endif

    #ifndef OUTRA_MACRO
        printf("A macro OUTRA_MACRO não está definida.\n");
    #endif

    #ifdef NUMERO
        int x = 20;
        close();
        #ifdef B
            print("NUMERO esta definida e B tambem");
            #ifdef C
                printf("NUMERO, B e C estao definidos");
            #endif
        #endif
    #endif

    #ifdef A
        printf("A");
        #ifdef C
            printf("A, C");
        #endif
    #elif B || NUMERO
        printf("B");
    #else
        printf("Else");
    #endif
```

Figura 8: Caso de código-fonte com condições aninhadas

diretivas de pré-processamento, incluindo casos com diretivas aninhadas, ou seja, diretivas que possuem outras diretivas em seu bloco de código. Para isso, toma-se como exemplo o código exibido na Figura 8.

Veja que o caso da Figura 8 é muito mais complexo que o anterior, possuindo diretivas de pré-processamento aninhadas umas às outras. Para este exemplo, deve-se tomar vários cuidados diferentes, como de juntar a expressão lógica de várias condições de presença, como "NUMERO && B", para execução da linha de código "print("NUMERO esta definida e B tambem");", dentre outras possibilidades. Para este código, os resultados para cada GET Request são exibidos nas Figuras 9 e 10.

Como mencionado anteriormente, a listagem das condições deve ser feita com bastante cautela, visto que o uso de diretivas aninhadas permite que haja múltiplas possibilidades de expressões lógicas para as condições de presença. A API, para este caso, apresenta a resposta correta.

E para este caso, a API apresenta como resposta corretamente os blocos de código para cada condição de presença. Por meio de testes com esses códigos fonte e muitos outros, foi possível perceber que a API funcionou corretamente em todos os casos. Todavia, considerando como o campo de sistemas configuráveis é extremamente complexo e propenso a diversas possibilidades de condições de presença, como estudado nos trabalhos relacionados a este projeto, é impossível testar a API para todas as possibilidades de condições de presença e sistemas configuráveis em códigos fonte em C, o que compromete a testagem do sistema.

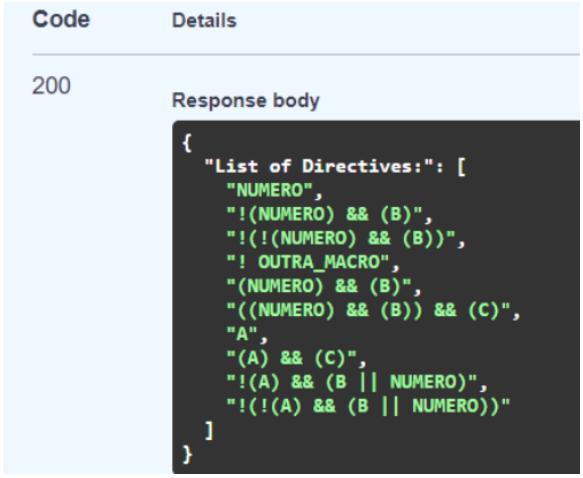

Figura 9: Resposta do GET Request da listagem de condições para o caso com condições aninhadas

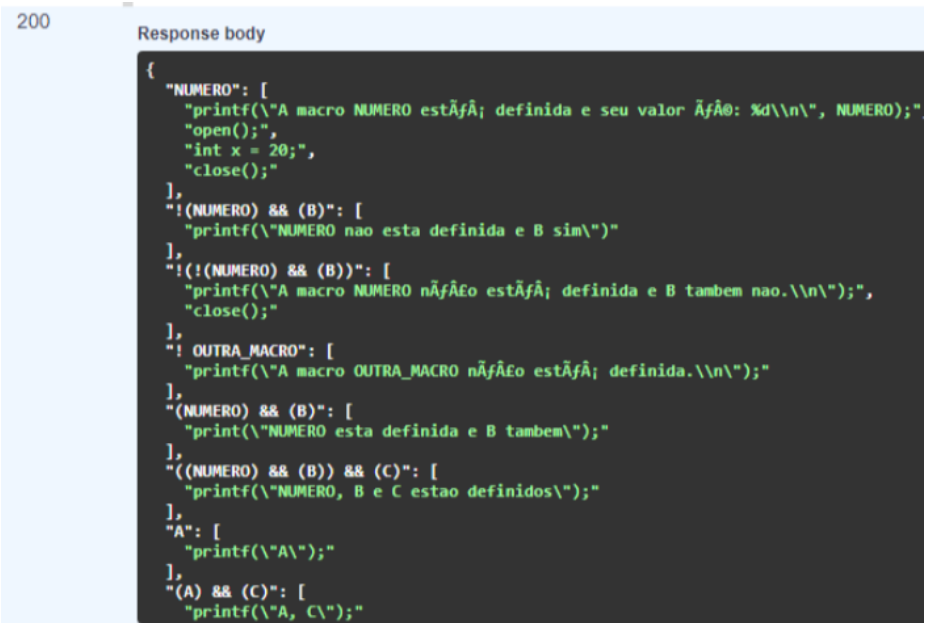

Figura 10: Resposta do GET Request dos blocos de código para o caso com condições aninhadas

5.3 XMLHandler

O arquivo "XMLHandler.py" é o responsável por abrir os arquivos XML salvos no banco de dados por meio do POST Request, e realizar a criação do arquivo JSON que será retornado ao cliente que está utilizando a API.

O GET Request é recebido pelo arquivo "api.py", no repositório do projeto, e este faz uma comunicação com "XMLHandler.py", que começará a análise do arquivo XML salvo. Foi utilizada a biblioteca "BeautifulSoup" para auxiliar no tratamento dos dados provenientes do XML. Tanto para o caso da criação da lista com cada condicional de presença como também para a criação do JSON com os blocos de código de cada condicional, o fluxo de execução de código é similar. Abaixo, um trecho da função principal do dicionário

Primeiramente, todas as condições de presença existentes no código fonte serão salvas em um array auxiliar, que posteriormente será acessado para análise. Essa medida contribui na otimização e otimização do código do projeto.

```
def getConditionalsNamesAndCodeBlocks():
    global allConditionalsDic, allDirectivesArray, bs_data
    currentConditional = ''
    i = 0
    while i < len(allDirectivesArray):
        element = bs_data.find_all()[allDirectivesArray[i]]
        if element.next in ['if', 'ifdef', 'ifndef']:
            # get conditional name:
            conditional = getThisConditionalName(i)
            currentConditional = conditional
            codeBlock = getThisCodeBlock(i)
            updateAllConditionalsDic(conditional, codeBlock)
            nextElement = bs_data.find_all()[allDirectivesArray[i + 1]]
            if nextElement.next in ['if', 'ifdef', 'ifndef']:
                i = getNestedConditionals(i, conditional)
                i -= 1
            i += 1
        elif element.next == 'elif':
            conditional = '!(' + currentConditional + ')'
            thisConditional = getThisConditionalName(i)
            conditional += ' && (' + thisConditional + ')'
            currentConditional = conditional
            codeBlock = getThisCodeBlock(i)
            updateAllConditionalsDic(conditional, codeBlock)
            nextElement = bs_data.find_all()[allDirectivesArray[i + 1]]
            if nextElement.next in ['if', 'ifdef', 'ifndef']:
                i = getNestedConditionals(i, conditional)
                i -= 1
            i += 1
        elif element.next == 'else':
            conditional = '!(' + currentConditional + ')'
            codeBlock = getThisCodeBlock(i)
            updateAllConditionalsDic(conditional, codeBlock)
```

Figura 11: Trecho da função que executa a análise e obtém as expressões lógicas das condições de presença

Depois, utiliza-se um dicionário, inicialmente vazio, que será usado para construção futura do JSON, e nele serão salvos todos os blocos de código relativos às condições de presença existentes no código fonte. Todas as linhas do XML são percorridas, com análise sendo feita por base em suas tags, e usando o array como suporte auxiliar. Selecionada uma condicional de presença, a expressão lógica relativa à ela é salva, e, caso se trate de um caso de condições aninhadas, a expressão é modificada para considerar que tanto a expressão vigente quanto a expressão lógica proveniente da condicional "mãe" devem ser verdadeiras para que a linha de código seguinte seja executada, como no exemplo anterior. Como pode-

se perceber, uma análise minuciosa precisa ser feita para que o dicionário seja preenchido corretamente com as expressões lógicas e seus respectivos blocos de código associados.

Terminada toda a análise anterior, caso o GET Request seja relativo à construção apenas da lista de condições, a lista é criada com base no dicionário, o JSON é construído com base nela e retornado via API para o cliente. Caso seja o terceiro GET Request, dos blocos de código, o dicionário completo com as linhas de código é usado para gerar um JSON e ser retornado ao cliente.

6 Testes

Para a realização dos testes da API, uma abordagem abrangente foi adotada, contemplando uma variedade de cenários. Inicialmente, foram explorados os casos mais simples, caracterizados por um número reduzido de diretivas de pré-processamento e a ausência de quaisquer ocorrências de erro. Este estágio inicial visou assegurar o funcionamento adequado em situações elementares.

Progressivamente, a avaliação foi expandida para englobar cenários mais desafiadores e intricados, incluindo aqueles anteriormente mencionados como exemplos de casos complexos. Nesses contextos mais exigentes, a API continuou a demonstrar robustez, mantendo a capacidade de processamento bem-sucedido, mesmo diante de requisitos mais elaborados. Esse enfoque abrangente na testagem permitiu validar não apenas os aspectos básicos da API, mas também a sua eficácia e estabilidade em situações mais elaboradas e diversificadas.

7 Conclusão

A conclusão deste projeto foi pautada pela estrita adesão ao cronograma predefinido, que estabeleceu uma distribuição eficiente de atividades ao longo das semanas. Ao chegar ao término da fase de desenvolvimento, empreendeu-se a elaboração de um relatório abrangente e documentação minuciosa, abarcando todas as nuances trabalhadas ao longo do período delineado no cronograma.

A análise do campo de pesquisa dedicado ao aprimoramento de análises em sistemas configuráveis revelou sua natureza relativamente nova e pouco explorada. A escassez de artigos acadêmicos dedicados a esse tema específico, evidenciada nos exemplos mencionados na seção de "Trabalhos Relacionados"e discutida anteriormente, ressalta a relevância do presente estudo.

À medida que a complexidade de um sistema configurável é ampliada, sua propensão a erros também aumenta, uma constatação empiricamente comprovada durante a execução de casos de teste mais desafiadores. O grande desafio da área é saber o que acontece a cada diferente configuração de condições de presença. A detecção de novos erros nesse processo exigiu a implementação de medidas corretivas apropriadas. [8]

Diante desse cenário, torna-se imperativo explorar o potencial das ferramentas e softwares concebidos para auxiliar na análise de sistemas configuráveis. Essas ferramentas desempenham um papel crucial ao apoiar programadores envolvidos nessa análise, contribuindo para a redução significativa da probabilidade de ocorrência de erros e falhas no sistema.

Essa abordagem proativa revela-se fundamental para a promoção da integridade e confiabilidade nas análises de sistemas configuráveis, proporcionando uma valiosa salvaguarda contra possíveis falhas.

Referências

[1] S. Smith, O. Lassila, *Configurable Systems for Reactive Production Management.* IFIP Transactions 15-B (1994).

[2] D. Sabin, E. Freuder, *Configuration as Composite Constraint Satisfaction.* Proceedings of the Artificial Intelligence and Manufacturing Research Planning Workshop (1996).

[3] C. Kastner, et al, *A comparison of 10 sampling algorithms for configurable systems* . Proceedings of the 38th International Conference on Software Engineering (2016).

[4] A. Popescu, et al, *An overview of machine learning techniques in constraint solving* . J Intell Inf Syst (2022).

[5] H. Ha, H. Zhang, *Performance-Influence Model for Highly Configurable Software with Fourier Learning and Lasso Regression,* IEEE (2019).

[6] S. Norbert, et al, *Performance-influence models for highly configurable systems.* ESEC/FSE 2015: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. Association for Computing Machinery (2015).

[7] M. Cohen, et al, *Interaction testing of highly-configurable systems in the presence of constraints.* ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis. Association for Computing Machinery (2007).

[8] A. Sarkar, et al, *Cost-Efficient Sampling for Performance Prediction of Configurable Systems.* IEEE (2015).