Apoiando o Entendimento e a Evolução de Software Legado: Um Estudo de Caso com o iVProg

Guilherme Vieira dos Santos
DCC, IME/USP
São Paulo, Brasil
guilhermevieiras@gmail.com

Anarosa Alves Franco Brandão
PCS, Poli/USP
São Paulo, Brasil
anarosa.brandao@usp.br

Resumo

A compreensão de sistemas Web legados orientados a eventos é dificultada pela fragmentação dos fluxos de controle assíncronos entre manipuladores de eventos, *callbacks*, chamadas JavaScript nativas, mutações de estado e atualizações dinâmicas de interface. Embora modelos de linguagem de grande escala (LLMs) possam apoiar tarefas de compreensão de programas, *prompts* baseados em código bruto tendem a incluir informação irrelevante, exceder limites práticos de contexto e produzir explicações pouco rastreáveis. Este artigo apresenta uma abordagem para extrair *fluxos de interação* a partir de sistemas Web, combinando Grafo de Propriedades de Código (CPG), delimitação por grafo de chamadas e fatiamento reverso restrito por dependências. A ideia central é usar análise simbólica para selecionar apenas os elementos estruturalmente relacionados a uma interação de usuário e aos seus efeitos observáveis, produzindo uma representação compacta e rastreável tanto para explicação assistida por LLMs quanto para futuras visualizações de comportamento. A abordagem identifica pontos de entrada associados a eventos, delimita uma zona de contenção interprocedural, filtra *sinks* alcançáveis e reconstrói dependências relevantes para cada fluxo. Avaliamos a abordagem em três cenários de compreensão de um sistema Web educacional, comparando explicações geradas a partir de código bruto e de fluxos extraídos. Os resultados indicam redução de aproximadamente 69% no tamanho do contexto textual e sugerem potenciais ganhos em foco e rastreabilidade das explicações geradas. Esse avanço apoia a manutenção de software ao reduzir o esforço de compreensão em sistemas legados extensos, que frequentemente dependem de bibliotecas descontinuadas e carecem de documentação.

Palavras-chave

compreensão de programas, manutenção de software, sistemas Web, fluxos de interação, grafo de propriedades de código, fatiamento de programa bidirecional, modelos de linguagem, abordagem neuro-simbólica

1 Introdução

A compreensão de programas é uma atividade central em manutenção, evolução e reengenharia de software, pois envolve reconstruir relações entre estruturas de implementação, comportamento observado, intenção computacional e conceitos do domínio da aplicação [3, 5, 20]. Essa tarefa é particularmente difícil em sistemas Web legados orientados a eventos, nos quais uma funcionalidade percebida pelo usuário pode estar distribuída entre arquivos JavaScript, manipuladores de interface, *callbacks*, seletores DOM, estruturas de estado e atualizações visuais.

Em tais sistemas, desenvolvedores frequentemente precisam responder a perguntas como: "o que acontece quando este botão é acionado?", "quais estruturas de estado são modificadas por esta interação?" ou "quais funções participam deste comportamento?". Essas perguntas são recorrentes em depuração, documentação, refatoração e evolução de software. Respondê-las manualmente exige navegar por cadeias de chamadas, dependências de dados, estruturas de controle e efeitos colaterais espalhados pelo código.

Modelos de linguagem de grande escala (LLMs) têm sido empregados para apoiar tarefas de compreensão, resumo e explicação de código. Entretanto, seu uso direto sobre código bruto apresenta limitações: bases de código reais podem exceder janelas de contexto, conter grande quantidade de informação irrelevante e induzir explicações que não são diretamente rastreáveis ao comportamento implementado. Além disso, estudos sobre contexto longo mostram que modelos de linguagem podem apresentar degradação quando a informação relevante está dispersa em entradas extensas [16].

A análise estática de sistemas Web modernos e legados, por sua vez, enfrenta obstáculos próprios. Linguagens dinâmicas como JavaScript, com forte uso de bibliotecas abstrativas como jQuery, quebram o determinismo de análises tradicionais baseadas em fluxo de controle [1, 2, 9, 10]. Registros de eventos, seletores de DOM e delegação assíncrona introduzem descontinuidades que dificultam a construção de grafos de chamada precisos.

Este trabalho parte da hipótese de que a compreensão assistida por LLMs em sistemas Web orientados a eventos pode ser aprimorada quando o contexto fornecido ao modelo é previamente restringido por uma análise simbólica do programa. Em vez de enviar grandes volumes de código bruto à LLM, propomos extrair *fluxos de interação*: subconjuntos do programa que conectam uma ação de usuário a estados, chamadas ou atualizações observáveis relevantes para uma funcionalidade.

Para nortear esta investigação focada na evolução de sistemas legados e na mitigação das limitações das LLMs, formulamos as seguintes Questões de Pesquisa (QPs):

- QP1: A restrição simbólica do contexto por meio de fluxos de interação reduz de forma expressiva o volume de *tokens* submetidos à LLM em comparação ao uso de código bruto?
- QP2: A representação em fluxos de interação apoia a superação de limitações das LLMs, melhorando o foco e a rastreabilidade das explicações geradas para a compreensão do sistema?

A contribuição central deste artigo é mostrar que a combinação de *chopping* sobre CPGs com delimitação por grafo de chamadas seleciona contexto estruturalmente relevante para LLMs explicarem fluxos de interação em sistemas Web. A abordagem combina: (i) identificação de pontos de entrada baseados em eventos; (ii) delimitação interprocedural por grafo de chamadas; (iii) filtragem de *sinks*

alcançáveis; e (iv) rastreamento reverso de dependências restrito à zona de contenção do fluxo.

As principais contribuições são:

(1) uma estratégia de extração de fluxos de interação baseada em CPGs, grafo de chamadas e dependências de dados, adaptada às particularidades de sistemas Web orientados a eventos;
(2) um pipeline neuro-simbólico que transforma artefatos JavaScript/HTML em contextos compactos e rastreáveis, adequados tanto para consumo por LLMs quanto para futuras visualizações de comportamento;
(3) uma avaliação em três cenários de compreensão de um sistema Web real, comparando explicações produzidas a partir de código bruto e a partir de fluxos extraídos;
(4) uma discussão dos desafios de aplicação de chopping e análise estática em sistemas Web orientados a eventos.

Como resultado principal, a avaliação demonstrou que a utilização dos fluxos extraídos reduziu o tamanho do contexto enviado à LLM em aproximadamente 69% quando comparado ao *baseline* de código bruto. Além dessa expressiva redução quantitativa, os dados qualitativos sugerem potenciais ganhos no foco e na rastreabilidade das explicações geradas pelo modelo.

O restante do artigo está organizado da seguinte forma. A Seção 2 apresenta a fundamentação e o posicionamento em relação a trabalhos relacionados. A Seção 3 descreve a abordagem proposta. A Seção 4 detalha a implementação do pipeline. A Seção 5 apresenta o estudo. A Seção 6 discute resultados, limitações e próximos passos. Por fim, a Seção 7 conclui o artigo.

2 Trabalhos Relacionados

2.1 Compreensão de Programas

A compreensão de programas envolve mais do que reconhecer chamadas de função ou estruturas sintáticas. Brooks descreve a compreensão como um processo de reconstrução de hipóteses sobre o programa, no qual o desenvolvedor relaciona o texto do código a conhecimentos sobre problema, domínio e implementação [5]. Biggerstaff et al. formulam o problema de atribuição de conceitos, no qual compreender um programa significa associar estruturas formais do código a conceitos humanos de mais alto nível, como intenções, comportamentos e efeitos no domínio operacional [3]. Storey discute como teorias, métodos e ferramentas de compreensão de programas evoluíram para apoiar atividades de manutenção e evolução [20].

A abordagem proposta se alinha a essa tradição ao buscar recuperar, a partir do código, fluxos que possam ser explicados em termos comportamentais. O objetivo é produzir uma representação intermediária que ajude desenvolvedores e LLMs a relacionarem ações de interface, chamadas internas, dependências e efeitos observáveis.

2.2 CPGs e Fatiamento de Programa Bidirecional

O Grafo de Propriedades de Código (CPG) foi proposto por Yamaguchi et al. como uma representação unificada que combina árvore sintática abstrata, grafo de fluxo de controle e grafo de dependência de programas em uma única estrutura baseada em grafo [24]. Embora originalmente introduzido no contexto de descoberta de vulnerabilidades, o CPG é adequado para compreensão de programas porque permite navegar conexões entre chamadas, dependências, estruturas de controle e elementos sintáticos. O Grafo de Dependência do Programa (PDG), base para as arestas de fluxo de dados e controle utilizadas neste trabalho, foi formalizado por Ferrante et al. [8].

O Fatiamento de Programa (*slicing*) foi introduzido por Weiser como técnica para decompor programas com base em dependências relevantes a determinado critério de análise [23]. O Fatiamento de Programa Bidirecional (*chopping*) generaliza essa ideia ao considerar explicitamente uma origem e um destino. Jackson e Rollins definem *chopping* como a extração dos elementos que transmitem efeitos de um conjunto de origem para um conjunto de destino [12]. Reps e Rosay descrevem o *chop* como uma fatia filtrada capaz de responder quais elementos do programa conectam uma origem s a um alvo t [17].

Essa formulação é particularmente adequada para sistemas orientados a eventos. Em vez de perguntar apenas "o que afeta esta variável?", a análise pode ser formulada como: "quais elementos conectam esta interação de interface ao comportamento observado?". Neste artigo, chamamos esse subconjunto de *fluxo de interação*.

2.3 Análise Estática em Sistemas Web Dinâmicos

Análises estáticas tradicionais foram concebidas sobre linguagens de tipagem estática e fluxos síncronos. Em JavaScript, no entanto, o dinamismo de tipos, o uso intensivo de funções de ordem superior e a manipulação assíncrona do DOM introduzem explosão de estados que compromete o determinismo dos analisadores [2, 9, 18]. Frameworks legados como jQuery agravam esse cenário ao intermediar registros de eventos e seletores de interface por meio de padrões que fogem a análises ingênuas [1]. Esses trabalhos motivam a necessidade de tratamentos específicos para resolução de vínculos de eventos, ponto central do pipeline aqui apresentado.

2.4 LLMs e Abordagens Neuro-Simbólicas

Abordagens neuro-simbólicas combinam a capacidade de generalização e geração textual de modelos neurais com a precisão estrutural de métodos simbólicos. No contexto de compreensão de programas, essa estratégia tem sido defendida como alternativa ao uso isolado de LLMs, especialmente devido a desafios de interpretabilidade, custo, confiabilidade e dependência de grandes contextos [21]. Técnicas como *Chain-of-Thought* (CoT) buscam ainda induzir raciocínio passo a passo dentro da própria LLM, melhorando a rastreabilidade das explicações produzidas [22]. A escolha do formato de serialização do contexto também impacta o desempenho: evidências recentes sugerem que representações estruturadas em JSON favorecem o mecanismo de atenção sobre topologias de grafos [25].

Neste trabalho, usamos o termo neuro-simbólico em sentido operacional: a etapa simbólica seleciona e estrutura o contexto por meio de CPGs, grafo de chamadas e dependências; a etapa neural gera explicações em linguagem natural a partir dos fluxos extraídos. Essa divisão não garante que a explicação produzida pela LLM seja correta por construção, mas reduz ruído e aumenta rastreabilidade.

Trabalhos recentes têm explorado a combinação entre CPGs e LLMs. LLMxCPG, por exemplo, utiliza CPGs para construir fatias de código relevantes para detecção de vulnerabilidades [15]. Este

artigo se diferencia em três aspectos: (i) a tarefa é compreensão de fluxos comportamentais, e não detecção de defeitos de segurança; (ii) o domínio é o *Frontend* Web (JavaScript, DOM, jQuery), e não linguagens de *Back-end* compiláveis; (iii) introduz-se um *chopping* bidirecional restrito por grafo de chamadas como mecanismo de contenção interprocedural. O Quadro 1 sintetiza esse posicionamento.

Tabela 1: Posicionamento em relação a trabalhos correlatos.

Abordagem	CPG	Chopping	Web/DOM	LLM p/ compreensão
Análise estática JS clássica [1, 9, 18]	Parcial	Não	Parcial	Não
LLM sobre código bruto [16]	Não	Não	Sim	Sim
LLMxCPG [15]	Sim	Parcial	Não	Parcial
Este trabalho	**Sim**	**Sim**	**Sim**	**Sim**

3 Abordagem Proposta

A nossa inspiração central para definir o fatiamento bidirecional proposto é o conceito de caso de uso. Ele foi classicamente definido como uma "sequência especial de transações, realizada por um usuário e um sistema em um diálogo" [13]. Essas transações podem ser compreendidas como fluxos de eventos (*flow of events*). Esse "diálogo"pode envolver desvios que precisam ser definidos com base no estado do sistema ou nas escolhas do ator, envolvendo regras lógicas e condicionais [14]. O que buscamos explorar aqui é se esses desvios condicionais, que guardam a "essência"da intenção dos desenvolvedores e especificadores de um sistema, podem ser mapeados identificando-se as principais ramificações lógicas e as alterações na informação transmitida pelo sistema após eventos iniciados pelo usuário, bem como os dados iniciais que este transmite à aplicação.

Esta abordagem proposta extrai *fluxos de interação* sobre um CPG para representar comportamentos associados a funcionalidades de sistemas Web orientados a eventos. Um fluxo de interação corresponde a uma sequência recuperada a partir do código que conecta: (i) uma ação de usuário ou evento de interface; (ii) os manipuladores e chamadas relacionados; (iii) as variáveis, condições e dependências relevantes; e (iv) estados, chamadas ou atualizações observáveis produzidos ao longo do processamento.

Seja $G = (V, E)$ o CPG de um sistema Web, em que V representa nós derivados da árvore sintática abstrata e E representa relações sintáticas, de chamada, controle, dependência de dados e contenção. Definimos $S \subseteq V$ como o conjunto de nós de origem de um fluxo (pontos de entrada de interação, como manipuladores de eventos DOM ou *callbacks*) e $T \subseteq V$ como o conjunto de nós-alvo (estados, chamadas ou atualizações de interface relevantes alcançáveis a partir de S).

Formalmente, o *chopping* tradicional entre origem e destino pode ser descrito como a interseção entre alcançabilidade direta e reversa [12, 17]. Neste trabalho, adotamos uma variante operacional: em vez de calcular alcançabilidade global sobre todo o CPG, delimitamos primeiro uma zona de contenção interprocedural $Z(S)$ pelo grafo de chamadas a partir do ponto de entrada, e restringimos a busca reversa de dependências aos *sinks* contidos nessa zona. O fluxo de interação é então aproximado por:

$$F(S,T) = Z(S) \cap Reach^{-}_{ddg/cfg}\big(T \cap Z(S)\big) \tag{1}$$

em que $Reach^{-}_{ddg/cfg}$ representa o rastreamento reverso por dependências de dados e estruturas de controle. Essa formulação preserva a intuição do *chopping*, mas restringe a análise à região interprocedural alcançável a partir da interação investigada, reduzindo ruído, evitando exploração de bibliotecas ou trechos inalcançáveis, tornando a execução do código mais rápida e a análise mais prática para sistemas Web. Embora o uso do grafo de chamadas em $Z(S)$ introduza uma sobreaproximação na alcançabilidade direta (visto que ignora restrições estritas de fluxo de dados na ida), essa escolha de projeto garante que nenhum nó do fatiamento exato seja omitido. A subsequente restrição pela busca reversa atua como filtro, mitigando a perda teórica de precisão da ida e viabilizando a análise em tempo interativo sobre o JavaScript dinâmico. A Figura 1 apresenta a visão geral da abordagem.

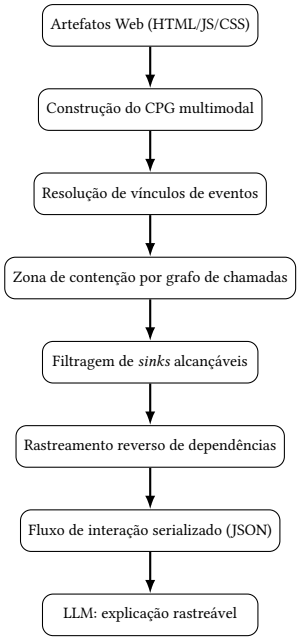

Figura 1: Pipeline CPG–Chop–LLM para extração e explicação de fluxos de interação.

4 Implementação do Pipeline

Esta seção descreve a implementação da abordagem. A construção do CPG foi realizada usando a ferramenta Joern. As travessias foram implementadas em Scala e na linguagem de consulta do Joern [19]. A serialização dos fluxos foi realizada também em Scala. O pipeline possui seis estágios principais.

4.1 Estágio 1: Construção do CPG Multimodal

O primeiro estágio processa os artefatos do sistema Web para produzir o CPG base. A ferramenta Joern é aplicada aos arquivos JavaScript e TypeScript do sistema analisado, gerando uma representação unificada que integra árvore sintática abstrata, grafo de fluxo de controle e grafo de dependências. Arquivos de teste e bibliotecas

externas empacotadas junto ao *Frontend* são excluídos pelo parser jssrc2cpg implementado no Joern, evitando poluir o grafo com código não relevante para a compreensão do sistema alvo.

Como sistemas Web dependem estruturalmente de marcação e seletores presentes em arquivos HTML, o CPG é enriquecido com metadados extraídos dos artefatos de visualização (identificadores, classes CSS e *tags* referenciados por seletores) que serão utilizados posteriormente para reconstruir vínculos entre elementos de interface e código lógico. Esta ingestão multimodal fornece o substrato sobre o qual operam os estágios seguintes.

4.2 Estágio 2: Resolução de Vínculos de Eventos

O segundo estágio identifica instruções de entrada (*entry points*) e instruções de saída (*sinks*). Os *entry points* representam ações iniciadas pelo usuário; os *sinks* representam efeitos observáveis ou resultados esperados. A implementação percorre os nós do CPG buscando assinaturas de registro de eventos e mutações previamente catalogadas. A Tabela 2 resume as categorias de comandos jQuery e nativos considerados.

Tabela 2: Comandos catalogados para identificação de *entry points* e *sinks*.

Categoria	Padrões catalogados
Entry points (jQuery)	`.on(evt,...)`, `.click`, `.change`, `.keydown`, `.submit`
Entry points (nativos)	`addEventListener`, atribuições a `on{click,change,...}`
Sinks de mutação DOM (jQuery)	`.html`, `.append`, `.addClass`, `.removeClass`, `.attr`, `.val`
Sinks de mutação DOM (nativos)	`document.createElement`, `appendChild`, `setAttribute`, `innerHTML`
Sinks de estado/visibilidade	`.show`, `.hide`, `.toggle`, `.css`

Cada ocorrência é classificada de acordo com o padrão encontrado e sua posição no código. O resultado desse estágio é um Dicionário Global de Âncoras, que associa identificadores internos a metadados de interação. Por exemplo:

`Gatilho_Method_42` → `[EVENT: click, SELECTOR: #run_button]`

Esse dicionário permite relacionar elementos do CPG a ações de interface, servindo como ponto de partida para a construção dos fluxos.

4.3 Estágio 3: Delimitação de Contenção por Grafo de Chamadas

O terceiro estágio delimita o escopo máximo de análise para cada fluxo. A partir dos *entry points*, executamos uma travessia transitiva focada exclusivamente no grafo de chamadas, isolando as funções alcançáveis a partir de determinada interação. Funções e chamadas que não podem ser acessadas a partir do ponto de entrada investigado são descartadas da análise daquele fluxo, evitando que a busca atravesse bibliotecas ou utilitários genéricos. Essa zona de contenção define o espaço máximo no qual procuraremos *sinks* e dependências relevantes.

4.4 Estágio 4: Interseção Lógica e Redução de Sinks

O quarto estágio identifica quais *sinks* possuem efeito observável dentro da zona de contenção. Cruzamos as funções e nós contidos na zona delimitada com os *sinks* listados durante a resolução de eventos. Se uma mutação de interface existe no código mas está inalcançável a partir das ramificações lógicas do gatilho atual, ela é podada da análise. Com isso, reduzimos ruído e mantemos apenas efeitos plausivelmente conectados ao fluxo investigado, obtendo um conjunto reduzido de destinos $T \cap Z(S)$.

4.5 Estágio 5: Rastreamento de Dependências Restritas

O quinto estágio reconstrói recursivamente a árvore de dependências que controla cada *sink* alcançável. A busca retrocede pelo Grafo de Dependência de Dados (DDG), mapeando onde variáveis receberam atribuição e adicionando novas variáveis relevantes a uma fila de análise. Quando uma estrutura de controle, como `if` ou `switch`, é encontrada, as variáveis associadas a suas expressões booleanas de teste são adicionadas à fila, e a estrutura é marcada como relevante para o fluxo. Quando uma atribuição é encontrada, seu valor é registrado como possível elemento de saída ou transformação intermediária.

A recursão continua até que a fila se esgote ou até que a análise atinja os limites da zona de contenção. Para garantir estabilidade da travessia sobre nós heterogêneos do Joern, a implementação aplica *pattern matching* de tipo (por exemplo, `case cfgNode: CfgNode`) para restringir o acesso a arestas de dependência apenas a nós de fluxo de controle válidos, ignorando nós puramente sintáticos que não possuem propriedades de dependência de dados. O resultado é um fatiamento reverso delimitado pelo grafo de chamadas, distinto do fatiamento reverso global clássico.

Em suma, a construção do fluxo de interação que propomos envolve a identificação recursiva de estruturas de desvio condicional e de variáveis que interfiram diretamente na alcançabilidade de um *sink* e nos valores que este recebe como parâmetros. Portanto, o processo abrange identificar:

- estruturas condicionais que definem se um *sink* será ou não chamado;
- estruturas de desvio condicional que definem se um dado valor será passado para um *sink*; e
- atribuições de variáveis dentro do fatiamento bidirecional, seja uma variável passada como parâmetro para o *sink* ou utilizada como condição em um desvio.

Variáveis que não sofrem mutação dentro do fatiamento bidirecional são tratadas na nossa análise como variáveis globais.

4.6 Estágio 6: Reconstrução Topológica e Serialização

Cada fluxo de interação é construído a partir de cada *entry point* identificado no Estágio 2, percorrendo nós de controle e atribuições marcadas no Estágio 5 até chegar aos *sinks*. Para preservar o significado de estruturas de controle cujos ramos não levam a nenhum *sink*, um nó especial `SILENT_ABORT` é inserido nesses ramos, evitando que a estrutura seja artificialmente linearizada. *Entry points* que não geram fluxos até pelo menos um *sink* são descartados.

Cada fluxo é serializado em JSON, formato empiricamente favorável ao mecanismo de atenção de LLMs sobre representações de grafos [25], com os seguintes tipos de nós:

- `ROOT_TRIGGER`: evento e seletor associados ao ponto de entrada, com filhos `EVENT` (função de evento acionada) e `SELECTOR` (seletor HTML utilizado);
- `DATA_MUTATION`: alterações em variáveis relevantes para estruturas de controle e/ou passadas como parâmetros para *sinks*;
- `DECISION`: estruturas de controle que governam cada *sink* alcançável ou variável relevante;
- `children`: relação entre um nó pai e um nó filho alcançável;
- `SILENT_ABORT`: ponto terminal de ramo condicional (como um bloco `try-catch`) que não gera efeito observável direto, mas cujo trecho de código preservado internamente no JSON permite à LLM inferir chamadas de funções relevantes contidas nele (como, por exemplo, o disparo do `SemanticAnalyser`);
- `SINK_UI_MUTATION`: *sinks* alcançáveis.

O objetivo da serialização não é reproduzir todo o código original, mas fornecer um contexto estruturado e rastreável: cada afirmação esperada em uma explicação deve poder ser associada a elementos presentes no fluxo serializado. Além do consumo por LLMs, esta representação também pode servir como substrato para visualizações interativas dos comportamentos, alinhando-se aos objetivos de visualização de software para apoio à manutenção e evolução.

5 Avaliação

A abordagem foi avaliada em um estudo de caso envolvendo o iVProg [4, 7], um sistema Web de ensino de programação por meio de programação visual, implementado com JavaScript, TypeScript e manipulação dinâmica de interface, fortemente dependente da biblioteca jQuery. O sistema apresenta características típicas de aplicações Web orientadas a eventos: múltiplos manipuladores de interface, *callbacks*, estruturas de estado, atualizações visuais e fluxos comportamentais distribuídos em diferentes trechos do código. Neste estudo de caso, o CPG gerado pelo Joern resultou em 183.147 nós. Após a execução do pipeline de extração, foi obtido um total de 174 fluxos de interação a partir do código fonte. O tempo de execução deste demonstrou-se viável, levando cerca de 1,1 segundo para ser finalizado (excluindo o tempo de construção inicial do CPG).

5.1 Definição dos Cenários e Padrão-Ouro (*Ground Truth*)

Para garantir a representatividade arquitetural, foram selecionados três cenários de uso do ecossistema iVProg, com níveis crescentes de complexidade topológica interprocedural. Para cada cenário, definiu-se um padrão-ouro (*ground truth*):

(1) **Cenário 1: Adição de Função (Baixa Complexidade):** Disparado pelo clique na âncora `.add_function_button`. O padrão-ouro exige que o modelo identifique o gatilho, descreva a adição de um novo bloco visual e a mutação de estado no DOM (anexação ao contêiner `.all_functions`).
(2) **Cenário 2: Inserção de Caso Condicional (Média Complexidade):** Disparado pelo botão `.button_add_case`. O padrão-ouro estipula a identificação do gatilho, a avaliação da pré-condição lógica (`test_case`) e a injeção do elemento `new_row` na tabela `.content_cases`.
(3) **Cenário 3: Execução do Motor Lógico (Alta Complexidade):** Disparado pelo `#run_button`. Por tratar-se da chamada ao interpretador do sistema, o padrão-ouro exige o mapeamento da alternância de visibilidade do terminal do console (`ivprog-term`) e o disparo da análise semântica (`SemanticAnalyser`), omitindo detalhes estritamente infraestruturais assíncronos que não representam regras de negócio.

5.2 *LLM-as-a-Judge*

Para cada cenário foram produzidas duas condições. Na condição **baseline**, a LLM recebeu um arquivo de texto bruto contendo o conteúdo de arquivos JavaScript e TypeScript encadeados em sequência, incluindo *scripts* incorporados em HTML e excluindo arquivos de teste, bibliotecas externas e especificamente o `ivprogParser.js`, que executa o *parser* de código visual via ANTLR e foi tratado como biblioteca externa. Na condição **fluxos**, a LLM recebeu uma representação compacta contendo todos os fluxos extraídos pelo pipeline, serializados em um único JSON.

As respostas foram geradas usando Gemini 2.5 Flash [11] com temperatura $T = 0$ para reduzir aleatoriedade. A escolha do modelo deveu-se à sua janela de contexto ampla, necessária para acomodar a condição *baseline*. A engenharia de *prompt* adotou *Chain-of-Thought* [22], instruindo o modelo a descrever primeiro o gatilho, depois as transformações intermediárias e, por fim, o efeito observável.

A avaliação foi realizada por meio de *LLM-as-a-Judge*, também com Gemini 2.5 Flash [11] e $T = 0$. O juiz recebeu respostas anonimizadas junto ao padrão-ouro (*Ground Truth*) definido por um desenvolvedor com conhecimento sobre o iVProg, e avaliou cada resposta segundo dois critérios:

- **Precisão:** a explicação está alinhada ao comportamento implementado?
- **Completude:** a resposta cobre os passos relevantes do fluxo?

A Tabela 3 apresenta as notas atribuídas pelo juiz.

Tabela 3: Desempenho inferencial por cenário: *Baseline* vs. Fluxos ($T = 0$).

Caso de Uso	Abordagem	Precisão	Completude
Cenário 1:	*Baseline* (Texto Bruto)	2	3
Adição de Função	**Fluxos (JSON)**	**5**	**5**
Cenário 2:	*Baseline* (Texto Bruto)	3	4
Caso Condicional	**Fluxos (JSON)**	**5**	**4**
Cenário 3:	*Baseline* (Texto Bruto)	2	3
Execução de Código	**Fluxos (JSON)**	**5**	**5**

Em termos de tamanho de contexto, a condição *baseline* totalizou 289.130 *tokens*, enquanto a representação serializada totalizou 89.685 *tokens*, uma redução de aproximadamente 69%. Em ambos os casos, o número de *tokens* foi calculado empiricamente por meio do *upload* direto dos arquivos no chat do Google AI Studio, utilizando o tokenizador do modelo Gemini 2.5 Flash carregado.

5.3 Exemplo de Fluxo de Interação

Para ilustrar concretamente a saída do pipeline, escolhemos um cenário com poucos nós, este é mostrado abaixo. O gatilho é o clique em `.editing_name_var`. Então uma sucessão de mudanças na interface visual acontece: a variável que contém o nome é zerada `input_name.val('')`; o valor do nome é substituído pelo novo valor salvo na variável `tmpStr`, através da instrução `input_name.val(tmpStr);` e, por fim o tamanho da caixa de texto é alterado no CSS para o novo tamanho na variável `inputWidth`, através do comando `input_name.css({width: inputWidth})`.


```
{
  "type": "ROOT_TRIGGER", "line": 120,
 "code": "Trigger:<lambda>0 | Anchors:[EVENT:click, SELECTOR:.editing_name_var]",
  "children": [
    { "type": "SINK_UI_MUTATION", "line": 928,
      "code": "input_name.val('')", "children": [] },
    { "type": "SINK_UI_MUTATION", "line": 929,
      "code": "input_name.val(tmpStr)", "children": [] },
    { "type": "SINK_UI_MUTATION", "line": 931,
      "code": "input_name.css({width: inputWidth})",
      "children": [] }
  ]
}
```


6 Discussão e Ameaças à Validade

Os resultados sugerem que os fluxos de interação podem melhorar o foco e a rastreabilidade das explicações geradas por LLMs. Ao receber um contexto previamente estruturado, notou-se que o modelo tende a discutir menos elementos irrelevantes e produz explicações que se aproximam mais da sequência evento–chamada–dependência–efeito. No Cenário 3, por exemplo, o *baseline* dedicou parte substancial da resposta a rotinas de sincronização assíncrona (*polling*), enquanto a condição baseada em fluxos concentrou a explicação nas mutações de interface relevantes.

A correção das explicações depende, no entanto, da completude do CPG, da precisão das relações de dependência, da qualidade da serialização e da capacidade da LLM de interpretar o contexto fornecido. Assim, a principal indicação preliminar oferecida por este estudo, sem ser conclusiva, aponta para a potencial redução de ruído e o apoio indireto à rastreabilidade.

A validade dos resultados está sujeita a algumas limitações. Primeiro, o estudo foi conduzido em um único sistema Web, o que limita a generalização. Sistemas baseados em React, Vue ou Angular, e arquiteturas fortemente assíncronas, podem exigir adaptações na identificação de *entry points* e *sinks*. Segundo, a análise estática não captura todos os comportamentos dinâmicos de aplicações JavaScript, pois linguagens dinâmicas frequentemente não observam a propriedade de *frame* (*frame property*) [18]. Eventos registrados dinamicamente, *closures*, delegação de eventos e geração dinâmica de código podem produzir fluxos incompletos, levando a explicações focadas porém omissas. Terceiro, o *baseline* utilizado (concatenação completa do código bruto) altera simultaneamente a seleção estrutural do contexto, a redução do tamanho e a serialização em JSON, o que impede atribuir os ganhos exclusivamente ao fatiamento (*chopping*) e à contenção por grafo de chamadas; logo, estudos futuros devem incluir um *baseline* mais forte e representativo para isolar esses efeitos. Quarto, as métricas de precisão e completude não operacionalizam diretamente os ganhos de foco e rastreabilidade reportados. Para se definir métricas claras de comparação para rastreabilidade e foco, seria preciso criar um *baseline* que restrinja somente informações do fluxo (como um *slicing* ingênuo ou o uso de Geração Aumentada por Recuperação - RAG). Por isso, neste estudo, a rastreabilidade e o foco só puderam ser medidos indiretamente pelas notas de precisão e completude atribuídas pela *LLM-as-a-Judge*. Quinto, a avaliação baseada em *LLM-as-a-Judge* foi adotada por demonstrar resultados consistentes na literatura recente [6]. Esse método ajuda a mitigar o viés de confirmação que um avaliador humano poderia introduzir, especialmente quando os avaliadores estão diretamente envolvidos com a pesquisa. Contudo, reconhecemos que essa abordagem também possui limitações, como possíveis vieses de posição e preferência por respostas mais longas [6]; além disso, não se pode descartar que o código do iV-Prog, por ser público, já tenha sido analisado pelo modelo (Gemini 2.5 Flash) durante o seu treinamento. Para enriquecer a análise, estudos futuros devem incluir a avaliação das respostas utilizando múltiplos modelos LLM, além de avaliadores humanos.

Apesar dessas limitações, a abordagem apresenta potencial para apoiar entendimento, manutenção e evolução de sistemas Web. Além de servir como contexto para LLMs, o fluxo de interação extraído pode ser usado como artefato visual ou navegável para desenvolvedores. Em trabalhos futuros pretendemos explorar visualizações que destaquem eventos, chamadas, condições, dependências e efeitos observáveis, permitindo que o desenvolvedor navegue do ponto de entrada ao resultado produzido, bem como investigar a integração dos fluxos e a re-injeção dos conceitos atribuídos pela LLM no CPG, construindo um grafo de conhecimento de código. Com esse mapeamento semântico pode-se contornar as dificuldades de navegação no CPG, dado o seu tamanho extenso de nós 183.147.

7 Conclusão

Este artigo apresentou uma abordagem para extrair fluxos de interação em sistemas Web, combinando CPG, delimitação por grafo de chamadas, fatiamento reverso restrito e LLMs. A proposta mitiga a dificuldade de compreender como ações de usuário se propagam por manipuladores, estados e atualizações observáveis.

A contribuição central está em usar análise simbólica para selecionar e estruturar o contexto antes da geração textual por LLMs. Em vez de delegar ao modelo a tarefa de localizar todos os elementos relevantes em uma base de código extensa, a abordagem extrai uma representação compacta e rastreável do fluxo de interação, que pode apoiar tanto explicações em linguagem natural quanto futuras visualizações de comportamento. No estudo com três cenários de um sistema Web educacional, os contextos baseados em fluxos de interação produziram explicações mais focadas e rastreáveis do que contextos baseados em código bruto, com redução de aproximadamente 69% no tamanho do contexto textual.

Como trabalhos futuros, pretendemos ampliar a avaliação para múltiplos sistemas Web, incluir avaliadores humanos, comparar diferentes modelos de linguagem, medir a redução real usando outras métricas além do número de *tokens*, refinar *baselines* e investigar mecanismos de visualização e navegação interativa dos fluxos extraídos.

Disponibilidade de Artefatos

Não foram disponibilizados artefatos adicionais.

Referências

[1] Esben Andreasen and Anders Møller. 2014. Determinacy in Static Analysis of jQuery. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*. ACM, 17–31. doi:10.1145/2660193.2660214

[2] Gábor Antal, Péter Hegedűs, Zoltán Tóth, Rudolf Ferenc, and Tibor Gyimóthy. 2018. Static JavaScript Call Graphs: A Comparative Study. In *Proceedings of the 18th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 177–186. doi:10.1109/SCAM.2018.00028

[3] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas E. Webster. 1994. Program Understanding and the Concept Assignment Problem. *Commun. ACM* 37, 5 (1994), 72–82.

[4] Leônidas de Oliveira Brandão and Anarosa Alves Franco Brandão. 2012. iVProg – Uma Ferramenta de Programação Visual para o Ensino de Algoritmos. (2012).

[5] Ruven Brooks. 1983. Towards a Theory of the Comprehension of Computer Programs. *International Journal of Man-Machine Studies* 18, 6 (1983), 543–554.

[6] Guiming Hardy Chen, Shunian Chen, Ziche Liu, Feng Jiang, and Benyou Wang. 2024. Humans or LLMs as the Judge? A Study on Judgement Bias. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. 8301–8327.

[7] Igor Felix, Lucas de Souza, Leônidas Brandão, Bernardo Ferreira, and Anarosa Brandão. 2019. iVProg: Programação Interativa Visual e Textual na Internet. In *Anais dos Workshops do VIII Congresso Brasileiro de Informática na Educação (WCBIE 2019)*. Sociedade Brasileira de Computação (SBC), Brasília, DF, Brasil, 1164. doi:10.5753/cbie.wcbie.2019.1164

[8] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems* 9, 3 (1987), 319–349. doi:10.1145/24039.24041

[9] José Fragoso Santos, Petar Maksimović, Théotime Grohens, Julian Dolby, and Philippa Gardner. 2018. Symbolic Execution for JavaScript. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming (PPDP '18)*. ACM, New York, NY, USA, 11:1–11:14. doi:10.1145/3236950.3236956

[10] Philippa Gardner, Sergio Maffeis, and Gareth Smith. 2012. Towards a program logic for JavaScript. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. ACM, New York, NY, USA, 31–44. doi:10.1145/2103621.2103663

[11] Google DeepMind. 2025. Gemini 2.5 Flash. https://deepmind.google/technologies/gemini/. Acessado em: 10 ago. 2026.

[12] Daniel Jackson and Eugene J. Rollins. 1994. A New Model of Program Dependences for Reverse Engineering. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*. 2–10.

[13] Ivar Jacobson. 1987. Object-oriented development in an industrial environment. *SIGPLAN Not.* 22, 12 (Dec. 1987), 183–191. doi:10.1145/38807.38824

[14] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. 1992. *Object-Oriented Software Engineering: A Use Case Driven Approach* (revised ed.). Addison-Wesley.

[15] Ahmed Lekssays, Hamza Mouhcine, Khang Tran, Ting Yu, and Issa Khalil. 2025. LLMxCPG: Context-Aware Vulnerability Detection Through Code Property Graph-Guided Large Language Models. In *Proceedings of the 34th USENIX Security Symposium*. 489–507.

[16] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics* 12 (2024), 157–173.

[17] Thomas Reps and Genevieve Rosay. 1995. Precise Interprocedural Chopping. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*. 41–52.

[18] José Fragoso Santos, Petar Maksimović, Gabriela Sampaio, and Philippa Gardner. 2019. JaVerT 2.0: compositional symbolic execution for JavaScript. *Proc. ACM Program. Lang.* 3, POPL (2019), 66:1–66:31. doi:10.1145/3290379

[19] ShiftLeft Inc. and Joern Contributors. 2024. Joern: The Bug Hunter's Workbench. https://joern.io/. Acessado em: 10 jul. 2026.

[20] Margaret-Anne Storey. 2005. Theories, Methods and Tools in Program Comprehension: Past, Present and Future. In *Proceedings of the 13th International Workshop on Program Comprehension*. 181–191.

[21] Alejandro Velasco, Aya Garryyeva, David N. Palacio, Antonio Mastropaolo, and Denys Poshyvanyk. 2025. Toward Neurosymbolic Program Comprehension. In *Proceedings of the IEEE/ACM International Conference on Program Comprehension*.

[22] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[23] Mark Weiser. 1984. Program Slicing. *IEEE Transactions on Software Engineering* SE-10, 4 (1984), 352–357.

[24] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 590–604.

[25] Kerui Zhu, Bo-Wei Huang, Bowen Jin, Yizhu Jiao, Ming Zhong, Kevin Chang, Shou-De Lin, and Jiawei Han. 2024. Investigating Instruction Tuning Large Language Models on Graphs. arXiv:2408.05457 [cs.CL] https://arxiv.org/abs/2408.05457