
**UNIVERSIDADE FEDERAL DE CAMPINA GRANDE**
**CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA**
**CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO**

**FRANCISCO IGOR DE LIMA MENDES**

# UMA ABORDAGEM PARA A DESCOBERTA AUTOMÁTICA DE ELEMENTOS WEB ACIONÁVEIS PARA TESTE SISTEMÁTICO DE GUI

**CAMPINA GRANDE - PB**

**2024**

**FRANCISCO IGOR DE LIMA MENDES**

# UMA ABORDAGEM PARA A DESCOBERTA AUTOMÁTICA DE ELEMENTOS WEB ACIONÁVEIS PARA TESTE SISTEMÁTICO DE GUI

**Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.**

**Orientador : Professor Dr. Everton Leandro Galdino Alves**

**CAMPINA GRANDE - PB**

**2024**

**FRANCISCO IGOR DE LIMA MENDES**

# UMA ABORDAGEM PARA A DESCOBERTA AUTOMÁTICA DE ELEMENTOS WEB ACIONÁVEIS PARA TESTE SISTEMÁTICO DE GUI

**Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.**

## BANCA EXAMINADORA:

**Everton Leandro Galdino Alves**

**Orientador – UASC/CEEI/UFCG**

**Patrícia Duarte de Lima Machado**

**Examinador – UASC/CEEI/UFCG**

**Francisco Vilar Brasileiro**

**Professor da Disciplina TCC – UASC/CEEI/UFCG**

**Trabalho aprovado em: 15 de MAIO de 2024.**

**CAMPINA GRANDE - PB**

# RESUMO

A geração automática de testes da interface gráfica do usuário (GUI) desempenha um papel crucial na detecção de faltas em aplicações web. Nesse contexto, *scriptless testing* automatiza a geração e execução de casos de teste com base nos elementos localizados da GUI. No entanto, a localização automática eficaz e não repetitiva dos elementos acionáveis em páginas complexas permanece um desafio. Técnicas existentes dependem de adaptações extensivas à aplicação sob teste, podendo ser onerosas e complexas. Como solução para essa problemática, propomos a utilização de uma técnica de localização automática, *Unique Actionable Element Search* (UAES), com o intuito de distinguir elementos acionáveis por meio do seu papel funcional, identificando-os unicamente sem intervenção no código-fonte. Nosso estudo empírico comparou e avaliou a eficácia da técnica de localização UAES com uma técnica de marcação explícita. Essa comparação foi conduzida em quatro aplicações *open-source* das quais testadores experientes destacaram os elementos acionáveis como parte da técnica de marcação explícita. Os resultados indicaram que nossa abordagem descobriu 79.81% dos elementos destacados, enquanto identificou novos elementos (8.1% de todos os elementos descobertos) que não foram evidenciados pelos testadores.

# AN APPROACH TO AUTOMATICALLY DISCOVER ACTIONABLE WEB ELEMENTS FOR SYSTEMATIC GUI TESTING

## ABSTRACT

The automatic generation of graphical user interface (GUI) tests plays a crucial role for detecting faults at web applications. In that regard, scriptless testing streamlines the process of generation and executing test cases through the identified GUI elements. However, identifying such actionable elements in an automatic and unique manner is still a challenge when dealing with complex web pages. Current approaches are tied with intricate and extensive adaptations to fit the Application Under Test (AUT). As an alternative and solution to this problem, we propose the adoption of a technique called Unique Actionable Element Search (UAES) that aims to uniquely discover actionable web elements by its functional role in an automatic fashion. Our empirical study assessed the effectiveness of UAES localization against a manual explicit markup approach. The study was conducted using four open-source projects where experienced testers identified the actionable elements as part of the explicit markup approach. The results show that our approach managed to discover 79.81% of the marked elements, while identifying new ones (8.1% of all discovered elements) that weren't highlighted by the testers.

# Uma Abordagem para a Descoberta Automática de Elementos Web Acionáveis para Teste Sistemático de GUI

Francisco Igor de Lima Mendes
Universidade Federal de Campina Grande
Campina Grande Brasil
francisco.mendes@ccc.ufcg.edu.br

Everton L. G. Alves
Universidade Federal de Campina Grande
Campina Grande Brasil
everton@computacao.ufcg.edu.br

Thiago Santos de Moura
Universidade Federal de Campina Grande
Campina Grande Brasil
thiago.moura@copin.ufcg.edu.br

## Resumo

A geração automática de testes da interface gráfica do usuário (GUI) desempenha um papel crucial na detecção de faltas em aplicações web. Nesse contexto, *scriptless testing* automatiza a geração e execução de casos de teste com base nos elementos localizados da GUI. No entanto, a localização automática eficaz e não repetitiva dos elementos acionáveis em páginas complexas permanece um desafio. Técnicas existentes dependem de adaptações extensivas à aplicação sob teste, podendo ser onerosas e complexas. Como solução para essa problemática, propomos a utilização de uma técnica de localização automática, *Unique Actionable Element Search* (UAES), com o intuito de distinguir elementos acionáveis por meio do seu papel funcional, identificando-os unicamente sem intervenção no código-fonte. Nosso estudo empírico comparou e avaliou a eficácia da técnica de localização UAES com uma técnica de marcação explícita. Essa comparação foi conduzida em quatro aplicações open-source das quais testadores experientes destacaram os elementos acionáveis como parte da técnica de marcação explícita. Os resultados indicaram que nossa abordagem descobriu 79.81% dos elementos destacados, enquanto identificou novos elementos (8.1% de todos os elementos descobertos) que não foram evidenciados pelos testadores.

## 1 Introdução

Testes de GUI vêm ganhando cada vez mais destaque devido a tendência da maioria das aplicações possuírem interface gráfica [23]. Esses testes visam criar casos que simulam a experiência do usuário com o intuito de verificar se os componentes se comportam da maneira esperada e detectando faltas a partir de falhas visíveis [7, 16]. Uma GUI recebe eventos (cliques, seleções e digitação em campos de texto) como entrada do usuário e pode mudar o estado dos elementos exibidos a partir dessas ações [4].

A natureza dinâmica das aplicações modernas na web, bem como a demanda crescente por sistemas complexos, seja no ramo da comunicação, comércio, educação e entretenimento [24, 36] tendem a requerer meios cada vez mais rebuscados para validar a qualidade e robustez dessas aplicações, tornando crucial a adoção de técnicas adequadas de teste para assegurar os padrões de qualidade esperados do sistema. Dito isso, testes em GUI se mostram uma opção eficaz uma vez que os casos de teste são executados por meio de interações com a interface gráfica da aplicação realizáveis por um usuário final. Apesar de efetivos, testes de GUI enfrentam desafios relacionados a mudanças significativas na interface, bem como à alta frequência dos ciclos de entrega do sistema [17].

Os frameworks para teste em GUI podem ser categorizados em três abordagens [34]: baseadas em coordenadas, baseadas no Modelo de Objeto de Documento (DOM), e por reconhecimento de imagem. As técnicas estudadas neste trabalho concentram-se somente no DOM da página. Essa categoria de frameworks, amplamente utilizada na indústria devido a sua simplicidade e usabilidade, faz uso do DOM para interagir com elementos da web por meio de suas propriedades. Selenium[1] e Cypress[2] são exemplos de frameworks bem conhecidos desta categoria.

Suítes de testes em GUI podem ser implementadas manualmente por meio de *scripts* (*scripted testing*) ou automaticamente (*scritpless testing*) [7]. Para implementar os *scripts* de teste, o testador deve interpretar visualmente a página web, encontrar os elementos relevantes e escrever os testes apropriados [18]. A compreensão do contexto em que a aplicação está inserida permite que o testador entenda o papel de cada elemento e construa testes que correspondam a demandas específicas. Ferramentas de *capture-and-replay* podem ajudar a automatizar esse processo [7]. Ainda assim, a implementação e manutenção frequente dos *scripts* de teste em sistemas robustos é custosa, sendo visto como um desafio complexo para testes em GUI, uma vez que a *Application Under Test* (AUT) passa por mudanças frequentes. Tornando a alteração na suíte de testes, a cada mudança significativa no design da GUI, impraticável [2, 29, 31].

Em *scriptless testing*, os casos de teste são gerados em cada execução a medida que as páginas da AUT são exploradas e as ações do usuário são reproduzidas [27, 37]. Com isso, a efetividade das ferramentas de *scriptless testing* está correlacionada com a capacidade dela de descobrir os elementos acionáveis da página para que testes expressivos sejam gerados com o intuito de alavancar a exploração do sistema ao navegar a GUI [13]. Por outro lado, é válido ressaltar que *scriptless testing* não consegue prover o mesmo grau de granularidade e fidelidade com as demandas específicas da aplicação encontrada nas implementações de *scripts* de teste [37]. Assim, na maioria das vezes esse tipo de teste visa detectar faltas a partir de falhas visíveis.

Na exploração sistemática da GUI, a suíte de testes objetiva verificar todos os elementos acionáveis da interface derivados do modelo da aplicação [3, 26, 35, 40]. Esse modelo é comumente baseado nos elementos encontrados em cada estado alcançado da AUT. Entretanto, identificar automaticamente um conjunto de elementos acionáveis sem repetições é um desafio que exige avanços nas técnicas de exploração [32]. Por exemplo, a ferramenta Cytestion [27], que faz uso de *scriptless testing*, emprega a exploração sistemática da GUI por meio da implantação de propriedades personalizadas

[1] https://selenium.dev
[2] https://www.cypress.io

(denominado marcações) para descobrir os elemento no DOM, possibilitando a geração dinâmica de *scripts* de teste para realizar a exploração sistemática. A necessidade dessas marcações impõem limitações, uma vez que o esforço manual se faz necessário para destacar os componentes relevantes, resultando na violação do código fonte, sendo uma técnica contraintuitiva para testes caixa preta.

Outras ferramentas de scriptless testing, como o TESTAR [37] e o WEBMATE [10], empregam o uso da API do Selenium para auxiliar na localização dos elementos sem invasão no código fonte. Entretanto, o TESTAR pode fazer uso de Monkey Testing [11, 39], em que sua abordagem aleatória torna improvável a exploração de funcionalidades profundamente aninhadas em tempo hábil, reduzindo o escopo apenas às funcionalidades facilmente encontradas [38]; ou pode utilizar modelos descritivos que guiem a exploração da AUT, exigindo uma fase de implementação e validação manual do modelo específico para a AUT e considerando o conhecimento contextual da aplicação. Enquanto isso, o WEBMATE utiliza seu usage model para mapear a exploração baseando-se em abstrações que causam perda de informações, podendo necessitar da implementação de *scripts* adaptadores para mitigar essa perda.

Neste trabalho, apresentamos a técnica *Unique Actionable Element Search* (UAES), uma abordagem automática para descobrir elementos acionáveis para testes de GUI via exploração sistemática em aplicações web. Para isso, exploramos o DOM, que reflete o GUI da aplicação, e fizemos uso de *strings* de busca para identificar os elementos interativos na página web. Além disso, para evitar repetições, usamos um conjunto de localizadores expressivos e únicos que remetem a propriedades funcionais dos elementos. Por fim, adotamos a ferramenta Cytestion [27] para incluir o UAES como parte do processo de geração de casos de teste.

Este trabalho traz as seguintes contribuições:

- uma técnica (UAES) para descobrir automaticamente elementos acionáveis sem repetições em aplicações web;
- uma implementação do UAES em uma ferramenta que realiza exploração sistemática em teste de GUI;
- um estudo empírico utilizando aplicações open-source com o intuito de comparar a efetividade do UAES em relação a uma técnica manual de marcação explícita.

Ao longo deste trabalho, as seções foram divididas no seguinte formato: na seção 2 apresentamos os conceitos mais pertinentes ao longo do texto, na seção 3 tratamos a técnica UAES em detalhes, na seção 4 expomos o estudo empírico nas aplicações open-source e os resultados alcançados, na seção 5 discutimos as ameaças à validade do trabalho, na seção 6 revisamos trabalhos relacionados e na seção 7 compartilhamos as conclusões derivadas do estudo.

## 2 Fundamentação Teórica

Nesta seção, abordamos os principais conceitos pertinentes ao trabalho. Além disso, definimos a distinção entre elementos estáticos e dinâmicos, em que tal classificação se mostrou crucial para refinar a efetividade da técnica UAES e nos ajudou a observar as limitações nos trabalhos relacionados presentes na seção 6. Ademais, apresentamos a técnica de marcação explícita e demonstramos como ela lida com a diferenciação de elementos estáticos e dinâmicos. Por fim, apresentamos a ferramenta Cytestion, veículo pelo qual executaremos a exploração sistemática para testes em GUI, com o intuito de realizar o experimento comparativo das duas técnicas de localização na seção 4.

### 2.1 Testes de GUI e Frameworks

Scripts de teste para testes em GUI utilizam propriedades dos elementos no DOM para destacar os agentes dos casos de teste. Tais propriedades servem como identificadores de localização e interação com elementos da página web, podendo preencher caixas de texto (por exemplo, campos de formulário), executar cálculos ao localizar e clicar em botões, e verificar a corretude do resultado da ação localizando elementos de exibição de resultados [19]. Esses localizadores estáticos precisam ser averiguados regularmente para manter sua validade e podem passar por ajustes a cada nova versão da aplicação. Desta forma, pequenas mudanças na interface podem causar grande impacto nos localizadores adotados. Soluções como Robula+, Sidereal e Similo objetivam manter a natureza descritiva desses scripts ao mesmo tempo que melhora a robustez dos localizadores [21, 22, 30]. O framework Cypress, por outro lado, encoraja a adoção de uma propriedade específica para testes com o intuito de mitigar os problemas oriundos de manutenção, marcando os elementos a serem localizados por essa propriedade em tempo de execução[3].

Selenium é um framework open-source para realizar testes funcionais e de regressão em aplicações web [12]. Ele permite a interação com páginas da web por meio do navegador, realizando ações como clicar em botões e navegar entre páginas. Apesar do suporte extensivo a várias linguagens e aos *browsers* mais famosos, Selenium enfrenta problemas relacionados a relatórios de resultados e limitações quando usado em sistemas robustos e dinâmicos [20]

Por outro lado, Cypress é uma alternativa moderna que tem o propósito de facilitar o processo de automação de testes de aplicações web. Ela oferece uma API simples, que interage diretamente com o DOM e, ao contrário do Selenium, possui técnicas modernas de adaptação à natureza dinâmica das aplicações atuais. Além disso, em Janeiro de 2024, Cypress tinha mais de 5.5 milhões de novos downloads, enquanto Selenium estava significativamente atrás com 1.8 milhões de novos downloads[4].

### 2.2 Elementos Estáticos x Dinâmicos

Em sistemas web, a distinção entre componentes interativos estáticos e dinâmicos desempenha um papel crucial na compreensão do comportamento das técnicas em relação a abordagem de descobrir e localizar unicamente os elementos acionáveis. Elementos estáticos se mantêm fixos e não são alterados baseados na interação do usuário na aplicação, ainda assim, são cruciais para navegar na GUI aplicação. Na figura 1, ilustramos esses elementos (destacados em verde). Exemplos clássicos de elementos estáticos estão contidos em footers, headers, barra lateral e menus de navegação.

Em contrapartida, elementos dinâmicos (botões, links e formulários) são encarregados de providenciar uma resposta baseada nos dados do usuário e do sistema. Eles são responsáveis por facilitar a navegação do conteúdo da página, sendo associados com ações que levam a efeitos colaterais. Como mostrado na figura 1,

[3]https://docs.cypress.io/guides/references/best-practices
[4]https://npmtrends.com/cypress-vs-selenium-webdriver

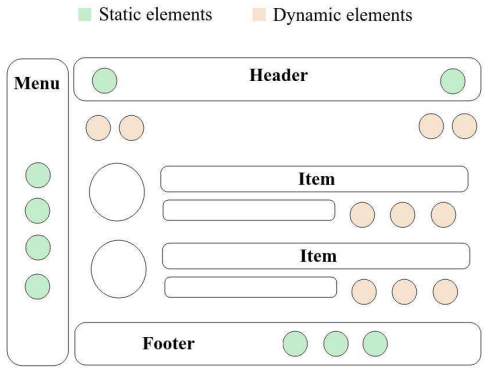


**Figure 1: Exemplo de elementos estáticos e dinâmicos.**

elementos dinâmicos (destacados em vermelho) estão ligados a itens de lista ou funcionalidades dependentes do estado atual da AUT, como criação, atualização e remoção de dados exibidos.

É essencial para as ferramentas de automação de testes em GUI distinguir efetivamente os elementos estáticos e dinâmicos. Os elementos estáticos serão acionados uma única vez, enquanto os elementos dinâmicos são reavaliados a cada novo estado da AUT.

## 2.3 Abordagem de marcação explícita

Uma vez que não há uma definição formal generalizada da GUI de uma aplicação web, e analisar os resultados presentes no HTML exposto pode produzir resultados imprecisos e incompletos, tem-se que o código-fonte da aplicação é a única maneira de identificar de forma autêntica todas as interfaces da aplicação [13]. Dito isso, essa abordagem requer que o testador manualmente inspecione o código fonte das páginas web da aplicação e destaque (pela marcação) os elementos acionáveis que devem ser avaliados na GUI durante a geração dos testes. A marcação explícita então é denominada pela inserção manual de propriedades inequívocas nos elementos a serem testados. Tais propriedades são então usadas pelas ferramentas de geração automática de testes.

A figura 2 mostra um componente React com a marcação. Seguindo o conceito de componentes reutilizáveis, a propriedade *data-cy* é incluída em todas as instâncias do componente. Para garantir unicidade dos elementos dinâmicos, um valor hash é associado ao elemento baseado na URL da página em que está inserido em conjunto com um valor definido pelo testador (o retorno da função *hashCode* é concatenado com o *buttonId*). A reusabilidade dos componentes acelera o processo de marcação, todavia, pode ser impraticável aplicar essa técnica em código de terceiros. Por exemplo, é impossível aplicar a marcação em um componente modal que não ofereça opções extensivas de customização como o componente de Modal do Ant Design[5].

A Figura 3 mostra como a marcação de elementos estáticos e dinâmicos deve ser distinta. O item de menu *Home Page* está marcado apenas como `data-cy="home"` (a), enquanto o botão *Download* tem um hash único associado ao estado atual (b). Essa distinção

[5]https://ant.design/components/modal

```jsx
import React from 'react';
import { hashCode } from './utils';
const CustomButton = ({ buttonId, children }) => (
  <button data-cy={`${hashCode()}-${buttonId}`}>
    {children}
  </button>
);
export default CustomButton;
```

**Figure 2: Exemplo de um componente React com a marcação.**

permite que múltiplos componentes de download existam na aplicação, cada um deles com seu identificador, enquanto *Home Page* permanece o mesmo independente do estado alcançado.

A adoção de uma propriedade personalizada para indicar quais elementos em um dado estado da GUI devem ser acionados é visto como um guia para ferramentas de scriptless testing. Durante a fase de geração, a ferramenta pode manter localizadores previamente utilizados para conseguir gerenciar quais valores de *data-cy* ainda não foram averiguados. Isso garante que elementos dinâmicos sejam apropriadamente explorados dependendo do estado nos quais estão inseridos, enquanto elementos estáticos são acionados uma única vez.


**Figure 3: Exemplo de elementos HTML com marcação.**

A técnica de marcação explícita consegue lidar com o desafio de encontrar elementos acionáveis ao mesmo tempo que lida com o problema dos localizadores ao assumirmos que os testadores associaram valores únicos à propriedade personalizada. Entretanto, isso introduz limitações cruciais: i) certas partes da AUT podem se tornar inacessíveis pela impossibilidade de marcar tais elementos (código de terceiros); ii) o tempo necessário para implantar esses marcadores em sistemas robustos e complexos pode ser inviável; iii) os testadores precisam conhecer o código fonte para conseguirem definir os elementos apropriados; e iv) a técnica requer que o testador garanta a unicidade do valor ligado ao elemento dinâmico, que pode ser falível (colisão de hash).

## 2.4 Cytestion

Cytestion [27] é uma ferramenta baseada em Cypress que gera e executa testes automáticos na GUI de sistemas web por meio da exploração sistemática da aplicação recorrendo à estratégia de scriptless testing em que, a partir de um único caso de teste, ele descobre elementos acionáveis presentes no estado inicial da aplicação e progressivamente gera novos casos de teste ao explorar cada elemento acionável que pode levar a novos estados da GUI. O foco das suítes de teste geradas pelo Cytestion está em detectar faltas a partir de falhas visíveis, ou seja, falhas que se apresentam na GUI por meio de mensagens de erro, em status de requisições e no console do browser. Cytestion requer que o testador tenha acesso ao código fonte que gera a GUI da AUT para que os elementos

sejam manualmente destacados para serem acessados durante o processo de geração de testes (técnica de marcação explícita).

## 3 Unique Actionable Elements Search (UAES)

Nesta seção, apresentamos a técnica Unique Actionable Elements Search (UAES), uma nova abordagem para descobrir unicamente elementos acionáveis na GUI da aplicação web. Seu objetivo é fornecer uma técnica de baixa manutenção para aplicar exploração sistemática em testes de GUI, priorizando a não redundância de testes. Nossa abordagem opera independente de APIs e oferece uma maneira de distinguir os elementos que serão utilizados pelas ferramentas de teste em GUI. Isso é feito por meio da abstração do DOM da página como uma abstração textual (string), permitindo o uso de trechos (substrings) para aplicar algoritmos de busca textuais na página da aplicação.

UAES consegue descobrir os elementos acionáveis a partir de um conjunto de substrings que estão associadas a trechos de código dos elementos acionáveis. A tabela 1 apresenta o conjunto de trechos utilizados, incluindo partes do código HTML dos elementos bem como chaves que servem como localizadores. O conjunto foi definido após consultar os testadores que adicionaram as marcações sobre os elementos marcados com maior frequência, bem como quais propriedades eram mais proeminentes para denunciar acionabilidade.

| HTML Element Snippets | | Possible Locator keys |
|---|---|---|
| **Tag** | **Attributes** | `inner-text` |
| `<a` | `onclick=` | `id` |
| `<button` | `href=` | `name` |
| `<input` | `type="button"` | `placeholder` |
| `<select` | `type="submit"` | `tooltip` |
| `<textarea` | `type="reset"` | `aria-label` |
| `<datalist` | `class="*value*"` | `value` |
| `<details` | | `class` |

**Table 1: Trechos de elementos HTML divididos entre tags e atributos, bem como seus possíveis termos chave usados como localizadores.**

Os trechos de elementos HTML incluem tags, atributos, além de valores de classes que podem ser fornecidos pelos testadores para destacar elementos acionáveis nas páginas web. Esses trechos servem como padrões para algoritmos de busca em strings, como Aho-Corasick [1], para descobrir os elementos acionáveis durante o processo de busca. Com as informações dos elementos encontrados pelo algoritmo, podemos extrair efetivamente a tag na string do DOM e validar a unicidade do elemento encontrado.

O *inner text*, também visto como o texto do conteúdo encapsulado em tags HTML, tem a capacidade de representar a semântica do elemento mais diretamente do que seus atributos para derivar seu propósito na GUI [15]. Na tabela 1, definimos chaves localizadoras para associar um localizador único aos elementos encontrados. Esse localizador valida e distingue os elementos descobertos elegíveis. Tal identificador é escolhido com base em uma ordem na lista, sendo *inner text* a primeira chave. Elementos que não incluam um *inner text*, como botões com ícones, são associados aos localizadores subsequentes, podendo ser *id*, *name*, *tooltip*, *aria-label*, *description*, and *placeholder*, que oferecem evidência de funcionalidade na GUI. Atributos como *value* e *class* podem ser usados, mas oferecem pouco valor semântico.

A concatenação do localizador (chave + valor do atributo) é crucial para rotular unicamente os elementos descobertos pelo UAES. Para evitar redundâncias, os localizadores de estados previamente explorados são considerados ao definir o localizador para um novo elemento. Essa checagem é executada comparando os elementos da URL atual com os elementos da URL anterior. Tal estratégia consegue evitar efetivamente a descoberta de elementos estáticos que não mudaram depois da execução de uma ação que causou a mudança da URL. Aplicações web podem mudar completamente suas páginas sem alterar a URL. Dito isso, a comparação em uma mesma URL nos permite encontrar os elementos dinâmicos que apareceram após a execução da ação mais recente.

**Algorithm 1** The UAES Algorithm

```
1:  setAE ← {} \\ set of AE triples < tag, locator, urls >
2:  HTMLsnippets ← list of predefined HTML snippets.
3:  locatorKeys ← list of predefined possible locator keys.
4:
5:  procedure UAES(content, url, previousUrl)
6:      tags ← DiscoverTags(content)
7:      for all tag ∈ tags do
8:          locator ← DefineLocator(tag, url, previousUrl)
9:          if locator is not null then
10:             Add AE triple (tag, locator, url) to setAE
11:         end if
12:     end for
13: end procedure
14:
15: function DiscoverTags(content)
16:     setTags ← {}
17:     discoveredTags ← Aho-Corasick(content, snippetsHTML)
18:     for all tag ∈ discoveredTags do
19:         if tag is visible and available then
20:             Add tag to setTags
21:         end if
22:     end for
23:     return tags
24: end function
25:
26: function DefineLocator(tag, url, previousUrl)
27:     locators ← getPossibleLocators(tag, locatorKeys)
28:     locator ← the first locator in locators, or null if no next exists.
29:     previousAE ← getPreviousAE(setAE, url, previousUrl)
30:     for all AE ∈ previousAE do
31:         locatorsAE ← getPossibleLocators(AE.tag, locatorKeys)
32:         if locators ≡ locatorsAE then
33:             updateAEURLs(setAE, AE, url)
34:             locator ← null
35:             break
36:         else if locator ≡ AE.locator then
37:             locator next locator in locators, or null if no next exists.
38:         end if
39:     end for
40:     return locator
41: end function
```

O pseudocódigo 1 apresenta o algoritmo da técnica UAES, responsável por atualizar dinamicamente o conjunto de elementos acionáveis (`setAE`). Abstrairmos um elemento acionável pela tripla $< tag, locator, urls >$, em que a `tag` corresponde à string da tag encontrada. O `locator` é o identificador único atribuído a essa tag, e `urls` é a lista de URLs em que esse elemento foi encontrado. O algoritmo começa por inicializar um conjunto vazio (linha 1), lendo os trechos de HTML predefinidos assim como os possíveis localizadores (linhas 2-3). A procedure recebe como parâmetros o

conteúdo do estado da GUI (o código HTML da página) como uma string, além da URL atual juntamente com a URL do estado anterior (linha 5).

Inicialmente, a função `DiscoverTags` é chamada para descobrir as possível tags interativas contidas no conteúdo fornecido (linha 6). Para isso, `DiscoverTags` aplica o algoritmo de busca em string Aho-Corasick [1] para localizar todas as instâncias de tags interativas usando os trechos de HTML contidos em `HTMLsnippets`, subsequentemente extraindo as tags apropriadas (linha 17). Então, essas tags são filtradas verificando sua visibilidade e acessibilidade com o objetivo de verificar se aparecem na GUI e se o usuário poderia interagir com essas tags.

Após obter as tags elegíveis, a procedure do UAES começa a definir um localizador para cada tag (linha 8) ao chamar a função `DefineLocator`. Primeiramente, a função `DefineLocator` começa verificando todos os localizadores disponíveis usando a função `getPossibleLocators` e atribui o retorno para a lista `locators` (linha 27). Essa função verifica os localizadores válidos extraídos da tag e fornece uma lista de todos os localizadores (chave + valor) encontrados. Então, é gerada uma lista de todas as triplas anteriormente construídas que contenham a URL atual ou a URL anterior nos elementos correspondentes da tripla.

Subsequentemente, um laço é executado para garantir a unicidade do localizador para cada tag (linhas 30-39). Esse loop percorre todos os elementos de `previousAE`, e durante cada iteração, todos os possíveis localizadores usados na tag do elemento acionável (AE) da URL anterior são incluídos à lista `locatorsAE`. Caso a lista de localizadores atual `locators` seja igual à lista dos localizadores da URL anterior `locatorsAE`, significa que o elemento já foi inserido no conjunto. Sendo assim, atualizamos essa tripla do elemento acionável (AE) para estar associada a URL atual, definindo *null* para o localizador atual (uma vez que não existem novos localizadores) e finalizando a execução do laço (linhas 33-35). Caso contrário, verificamos se o localizador selecionado em `locator` é igual ao localizador do elemento acionável (AE) atual do laço, se sim, `locator` recebe o próximo localizador da lista `locators` ou null caso o próximo não exista (linhas 36-38).

Ao identificar um localizador único para esse elemento elegível, sua tripla correspondente (tag, localizador, e uma lista de `[URL]`), é adicionado ao conjunto `setAE` (linha 10). Dessa forma, UAES mantém um catálogo extenso dos elementos acionáveis (AE) que são únicos na AUT.

## 3.1 Exemplo de execução

Para ilustrar o uso do UAES, apresentamos um fluxo de execução extraído da aplicação open-source Petclinic[6]. Inicialmente, o conjunto `setAE` começa vazio (linha 1) e o conteúdo do estado inicial representado na figura 4 (a) é fornecido à procedure do UAES juntamente com sua URL e a URL anterior, que nesse caso é *null* (linha 5). A função `DiscoverTags` é então chamada usando o conteúdo da página como entrada (linha 6). Após executar o algoritmo de busca em strings, são retornados as três tags acionáveis do menu de navegação (Home, Find Owners e Vecterinarians). Consequentemente, cada tag é associada a um localizador válido como retorno da função `DefineLocator` (linha 8)

[6]https://github.com/spring-projects/spring-petclinic

Para a primeira tag, é feita uma busca para encontrar o localizador apropriado. Isso resulta em uma consulta na lista de localizadores, em que o valor do primeiro localizador, (`inner-text`), é `inner-text="Home"` (linha 27). Como `setAE` está vazio, nenhuma tripla é retornada (linha 29), validando o primeiro localizador encontrado. A tripla gerada do elemento acionável (tag, localizador e lista de URLs em que possuem esse elemento) é então adicionada ao conjunto `setAE`. O processo é similar para as próximas duas tags, a diferença está no processo de validação do localizador, pois como `setAE` não está vazio, `previousAE` retorna a tag recém inserida, uma vez que a URL atual está contida nessa tripla, isso faz com que haja uma verificação da unicidade do localizador, provando-se única e apropriada para ser associada às suas respectivas tags. Com `setAE` atualizado, a ferramenta de teste pode interagir com a aplicação por meio de cliques nos elementos encontrados. Para esse exemplo, suponha que o elemento `inner-text="Find Owners"` foi clicado.

Ao interagir com o menu de navegação, o processo de descobrimento é iniciado novamente ao invocar a procedure do UAES. Dessa vez, seis elementos são encontrados ao chamar a função `DiscoverTags`: três novos (marcados em vermelho na figura 4 (b)) e três itens de menu previamente descobertos. Consequentemente, quando a função `DefineLocator` é aplicada a cada tag, temos que as triplas das três tags do item de menu presentes na URL anterior estão em `setAE`, que são atribuídas a `previousAE` (linha 28). Com isso, como esses elementos são idênticos, todos os localizadores disponíveis também serão idênticos (linha 32). Como consequência, esses três elementos terão suas URLs atualizadas na linha 33, enquanto o localizador, dessa vez *null* leva à saída do laço na linha 30. Assim, como esses três itens de menu não satisfazem o critério estabelecido na linha 9, suas respectivas triplas não são adicionadas a `setAE`.

Um dos novos elementos descobertos têm o mesmo *inner text* do item de menu, "Find Owners" (see Figure 4 (b)). Entretanto, ele também possui um outro localizador válido, em que `id="findOwners"` passa a ser o segundo localizador encontrado na linha 37, que resulta na sua adição válida ao conjunto `setAE`. Além disso, o elemento input não possui *inner text*, mas o localizador `id` foi encontrado, resultando no valor `id="lastName"`. Dessa forma, os três novos elementos foram adicionados com suas respectivas triplas a `setAE`.

Suponha que nesse fluxo de execução, o botão com o localizador `id="findOwners"` foi acionado, que nos leva para uma nova página com novos elementos acionáveis e uma mudança na URL atual (Figura 4 (c)). Um total de oito elementos elegíveis são descobertos, incluindo os mesmos três itens do menu de navegação. Como a lista de URLs foi atualizada na execução anterior, esses itens de menu foram encontrados pela função `getPreviousAE` usando `previousURL` (linha 29). Conseguinte, eles não são considerados elementos novos e não são adicionados ao conjunto `setAE`, porém suas triplas foram atualizadas novamente. Depois de incluir os cinco novos elementos em `setAE`, a procedure do UAES é finalizada e o fluxo de execução prossegue clicando no elemento de localizador `inner-text="George Franklin"`.

Na página de detalhes de cadastro, um processo similar acontece para os elementos na barra de navegação. Os dois novos botões são encontrados e apropriadamente adicionados ao conjunto `setAE`. Continuando este fluxo, quando o elemento de localizador

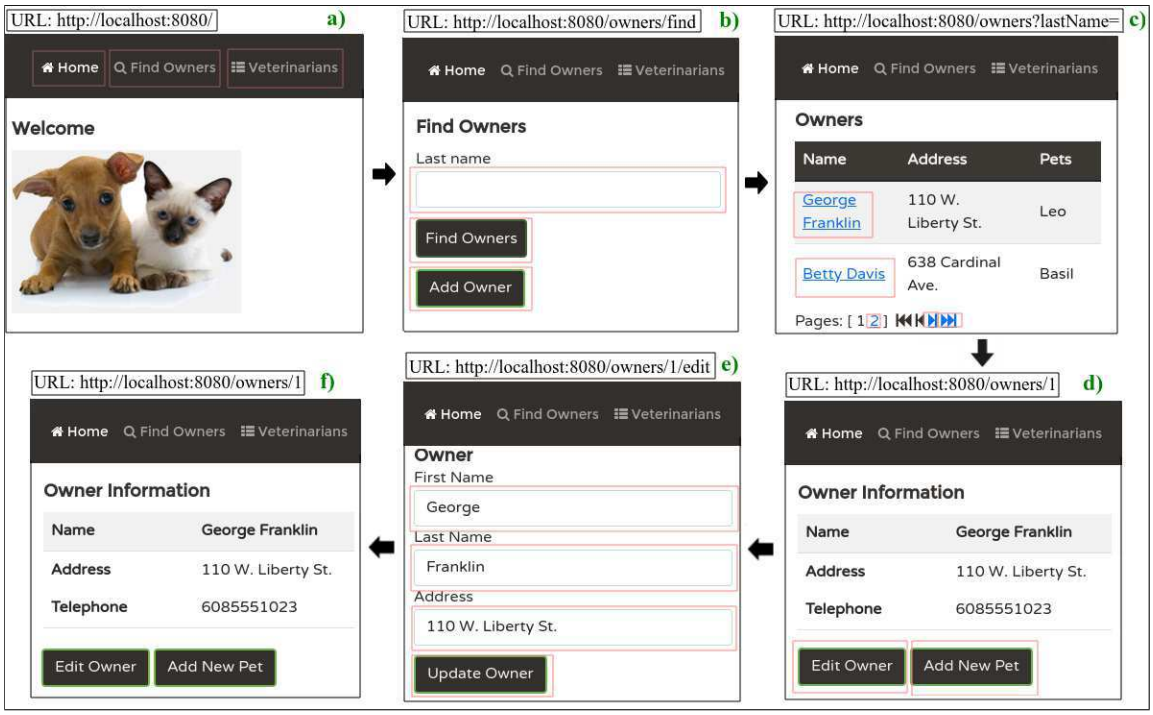


**Figure 4: Exemplo de execução do UAES para ilustrar o processo de geração de testes**

`inner-text="Edit Owner"` é clicado, somos levados à página da figura 4 (e). No formulário dessa página, todos os inputs e o botão de atualização são descobertos, classificados como únicos, e então são adicionados a `setAE`. Por fim, nesse fluxo, apenas o elemento `inner-text="Update Owner"` é clicado, nos levando à página da figura 4 (f). É válido ressaltar que essa nova página possui uma URL já explorada. Ao fornecer essa informação ao UAES, todos os elementos descobertos estão incluídos em `previousAE`, fazendo com que a busca por novos localizadores sempre leve a *null* e nenhuma adição é feita ao conjunto `setAE`.

## 4 Estudo de Avaliação

Nesta seção, apresentamos um estudo empírico com o objetivo de verificar a eficácia da abordagem UAES no processo de descobrir unicamente os elementos acionável em exploração sistemática na GUI. Para isso, estabelecemos a seguinte questão de pesquisa:

- *RQ*: A técnica UAES consegue detectar e selecionar efetivamente os elementos acionáveis em uma aplicação web?

Nós comparamos a abordagem da UAES com a da marcação explícita. Usamos a marcação explícita como base pois sua implantação foi feita por especialistas que identificaram manualmente os elementos acionáveis das páginas. Com isso, quanto mais próximo o resultado do UAES for da marcação, melhor. Além disso, usamos uma versão estendida da ferramenta Cytestion (descrita na seção 2.4) para explorar a geração da suíte de testes em GUI nas aplicações usando tanto a marcação como a UAES.

Para respondermos a questão de pesquisa, realizamos um estudo empírico que foi conduzido usando quatro aplicações web open-source. Esse estudo segue o seguinte fluxo: selecionamos um conjunto de aplicações e para cada uma delas, requisitamos que testadores experientes marcassem as partes do código correspondentes a elementos acionáveis nas páginas usando a técnica de marcação explícita. Então executamos a ferramenta Cytestion para ambas as abordagens. O Cytestion consegue realizar exploração sistemática nas páginas, descobrindo unicamente de forma dinâmica todos os elementos marcados e armazenando-os, bem como os elementos encontrados pela abordagem do UAES. Com isso, nós comparamos os conjuntos de elementos acionáveis resultantes de ambas as técnicas.

É importante destacar que a execução do Cytestion foi feita usando ambas as técnicas simultaneamente (marcação explícita e UAES). Isso nos permite mitigar as distorções de dados causadas, como valores aleatórios de atributos, que possam ocorrer em execuções separadas (e.g., propriedades assumindo um novo valor a cada carregamento da página).

Para clarificar nossa análise, classificamos os elementos acionáveis encontrados tanto na marcação explícita como no UAES em três categorias: *same*, como elementos encontrados por ambas as técnicas; *new* como elementos encontrados apenas pelo UAES; e *missed*, como elementos encontrados apenas pela marcação explícita.

Para avaliar as diferenças estatísticas nos elementos descobertos por ambas as técnicas, usamos *Mann-Whitney U* [28], em conjunto com a medição de *Cliff's delta effect size* [25]. Essas medições foram escolhidas considerando capacidade de lidar com pequenas amostras e a característica não-paramétrica das distribuições dos grupos, uma vez que *Mann-Whitney U* pode ser usado para estipular se dois grupos são homogêneos e possuem a mesma

distribuição. Assumimos que a hipótese nula é a de que não há diferença entre os resultados das duas abordagens ($\alpha$ = 0.05). A medição do *Cliff's delta effect size* é uma medição não-paramétrica que quantifica a magnitude da diferença entre os valores das duas distribuições. Um *delta* maior indica uma maior disparidade entre as distribuições. Ao interpretar o valor de *Cliff's delta*, usamos os limiares citados em [6]: *delta* < 0.147 (negligente), *delta* < 0.33 (pequeno), *delta* < 0.474 (médio), *delta* >= 0.474 (grande). Um *Cliff's delta* positivo significa que o quanto os valores do primeiro grupo são maiores, enquanto um *delta* negativo demonstra o contrário.

Nosso estudo empírico foi executado em uma máquina Desktop com um processador Intel Core i7 10700KF, com 32GB de RAM DDR4 3200MHz, uma placa de vídeo Nvidia GTX 1060 6GB GDDR5 e um SSD SATA 1TB 500Mpbs/s.

## 4.1 Realização do estudo empírico

Para o nosso estudo, quatro aplicações open-source foram selecionadas: i) *petclinic*, uma aplicação SpringBoot que lida com o cadastro de donos de pets e permite gerenciar uma agenda para visitas de pets ao veterinário; ii) *bistro restaurant*, um website desenvolvido com HTML, Javascript e CSS que apresenta portfólios para restaurantes; iii) *learn educational*, um website responsivo que apresenta portfólios para cursos educacionais online; iv) *school educational*, um website em HTML5 que implementa um conjunto de funcionalidades comumente encontradas em aplicações escolares. Essas aplicações são sistemas usados para propósitos acadêmicos e não fazem uso de componentes reutilizáveis[7].

A tabela 2 fornece informações sobre as aplicações, incluindo quantidade de linhas de código (KLOC) e o número de casos de teste gerados e executados pelo Cytestion. Apesar da simplicidade, essas aplicações oferecem funcionalidades de navegação, apresentam funcionalidades relevantes e possuem operações de cadastro. Isso é evidenciado pelo número de casos de teste gerados.

| Projeto | KLOC | # de Testes Gerados |
|---|---|---|
| *petclinic* | 25.7 | 50 |
| *bistro restaurant* | 33.4 | 212 |
| *learn educational* | 19 | 225 |
| *school educational* | 30.2 | 231 |

**Table 2: Projetos, KLOC e número de testes gerados.**

Solicitamos que dois testadores aplicassem a técnica de marcação explícita em todas as quatro aplicações. Tais testadores possuem mais de três anos de experiência em desenvolvimento de aplicações web e teste de GUI com Cypress. Nós os instruímos a navegar todo o código, apontando todos os elementos que poderiam ser interagidos em um cenário de testes em GUI e validar os efeitos dessas ações. O tempo estimado para completar essa atividade foi de quatro horas.

Com isso, comparamos o número de elementos descobertos pela marcação explícita e o UAES.

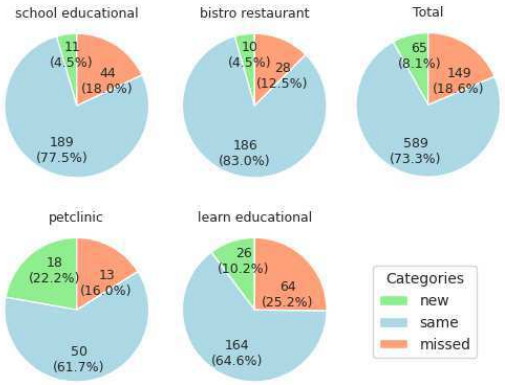


**Figure 5: Resultados das acplicações open-source: elementos acionáveis equal, new e missed.**

## 4.2 Resultados e Discussão

A figura 5 exibe todos os elementos acionáveis encontrados, tanto no total como por aplicação. Categorizamos eles como *same*, *new* e *missed*. 73.3% dos elementos foram descobertos por ambas as técnicas (*same*). Entretanto, quando consideramos apenas os elementos encontrados pela técnica de marcação explícita (*same* + *missed* = 738 elementos), que pode ser vista como nosso baseline, UAES conseguiu descobrir automaticamente 79.81% deles.

8.1% dos elementos foram identificados apenas pelo UAES. Esses elementos não foram notados pelos testadores e por isso não foram incluídos nos testes gerados da marcação explícita. Nós manualmente verificamos esses novos elementos e todos eles foram considerados elementos acionáveis válidos. Eles incluíam funcionalidades essenciais como paginação de tabelas. Essas descobertas contribuíram significativamente para demonstrar a efetividade da UAES em evitar erros oriundos do processo manual da marcação explícita e os riscos associados a tal estratégia.

Nós investigamos os elementos classificados como *missed* (20.19%) e concluímos que a maior parte deles não estão em conformidade com padrões básicos de boas práticas de implementação. Por exemplo, encontramos elementos na mesmo página compartilhando todos os localizadores possíveis quando na verdade possuem propósitos diferentes, impossibilitando a descoberta única desses elementos. A figura 6 ilustra um exemplo desse caso, em que os elementos estáticos (Home, About, Courses, Fees, Portfolio, and Contact) aparecem tanto no menu de navegação como no rodapé. Apesar da duplicação desses elementos para navegação do usuário ser comum, eles não possuem outros atributos como IDs únicos, impossibilitando a diferenciação desses elementos somente pelo texto.

UAES perdeu alguns elementos ao confundir dinâmicos como estáticos, que deveriam ser visitados apenas uma vez. Isso ocorreu por conta da grande semelhança a outros elementos de estados anteriores. Por exemplo, na figura 7, o botão *Add Owner* aparece em ambas as páginas *Find Owners* e *Owner* com atributos e texto idênticos. Apesar do estado e das URLs serem diferentes, considerar apenas isso faria com que elementos verdadeiramente

[7] https://gitlab.com/lsi-ufcg/cytestion/gui-testing-study/applications

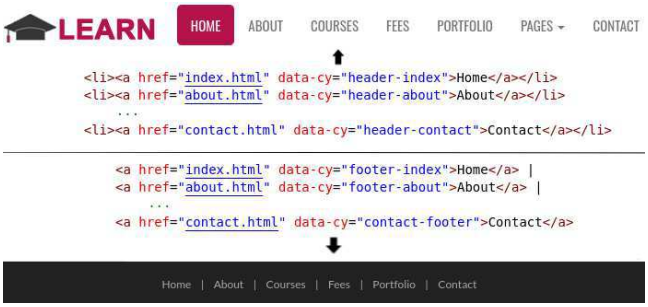


**Figure 6: Exemplo de elementos estáticos perdidos (missed) devido à igualdade do itens no menu de navegação e no rodapé (footer).**

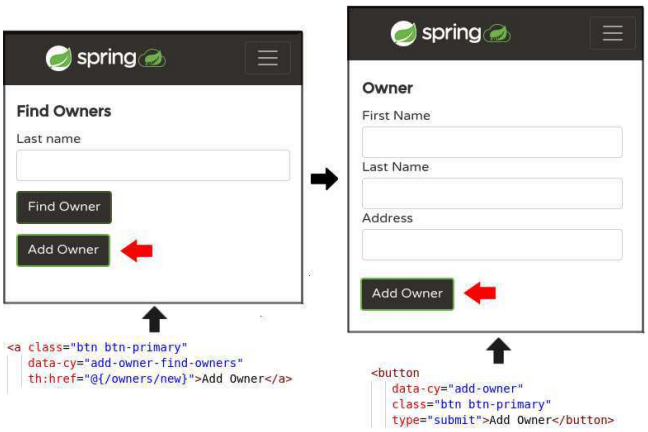


**Figure 7: Exemplo de elementos perdidos (missed) oriunda da falta de localizadores únicos.**

estáticos fossem acionados repetidamente (itens de menu) após qualquer mudança na URL, como demonstrado na seção 3.1. UAES sempre compara os elementos encontrados na URL anterior com os da URL atual com o intuito de descobrir elementos verdadeiramente únicos. Ou seja, fica a cargo do desenvolvedor atuar na solução dessa questão ao implementar diferenciadores para elementos semelhantes. Retomando ao exemplo (Figura 7), se os botões possuíssem IDs únicos ou textos diferentes (*Save*, por exemplo), o UAES conseguiria diferenciá-los com sucesso e seriam validados como elementos acionáveis.

Ademais, descobrimos que alguns dos elementos *missed* eram na realidade tags desprovidas de quaisquer localizadores válidos, evidenciando problemas de implementação e, consequentemente, ausência de informações relevantes para a localização.

O teste de hipótese realizado entre os grupos de elementos encontrados pela marcação explícita e aqueles marcados identificados pelo UAES revelou um *p-value* de 0.2, indicando que a diferença entre os dois grupos não é estatisticamente significativa, falhando em rejeitar a hipótese nula de que não há diferença na distribuição dos resultados das técnicas. Além disso, o *effect size* foi de 0.0625, indicando que há uma diferença negligente entre os dois grupos, ou seja, eles seguem uma distribuição parecida. Esses resultados sugerem, considerando esta pequena amostra, que a diferença de ambas as técnicas em relação a efetividade ao identificar elementos acionáveis não é expressiva.

Esse estudo sugere que o UAES mostrou uma eficácia comparável à técnica de marcação explícita para as aplicações estudadas. Ademais, alguns dos elementos que não foram identificados pelos testadores foram descobertos pelo UAES, o que pode melhorar a qualidade da suíte de testes gerada. Finalmente, a maioria dos elementos *missed* poderiam ser detectados pelo UAES se boas práticas de programação fossem seguidas durante o desenvolvimento. Faz-se necessário incluir atributos que auxiliem a distinção dos elementos no código fonte da GUI para cada elemento acionável para que seja alcançada uma automação de testes efetiva. Esses atributos servem como localizadores para quaisquer ferramentas de testes automáticos.

## 5 Ameaças à Validade

Nossos resultados não generalizam além das aplicações usadas em nossos estudos. Ainda, não avaliamos nossa abordagem em aplicações industriais.

A avaliação da efetividade do UAES está ligada principalmente à descoberta de elementos acionáveis, outros aspectos de testes de GUI não foram considerados, como cobertura de teste, capacidade para detectar faltas e escalabilidade. Todavia, a descoberta eficaz é fundamental para a efetividade dos testes de GUI, uma vez que fornece a fundação para uma cobertura de testes expressiva. Ao automatizar a descoberta de elementos acionáveis, UAES melhora a eficiência e diminui as particularidades de erro humano, contribuindo para um acréscimo da qualidade geral da suíte de testes.

A dependência da identificação manual dos elementos adequados por testadores, apesar de necessária como parâmetro de comparação, introduz a possibilidade de erro humano e subjetividade na técnica de marcação explícita, pontos que podem impactar negativamente na precisão da análise comparativa. Entretanto, os testadores que adicionaram as marcações têm mais de três anos de experiência em desenvolvimento de aplicações web. As marcações foram realizadas com o objetivo de usar a ferramenta Cytestion com a maior efetividade possível, e os resultados obtidos em detecção de faltas até agora demonstram a confiabilidade das marcações.

A evolução acelerada das tecnologias e frameworks web caracteriza uma ameaça para a efetividade do UAES em termos de longevidade. Como as práticas e padrões de desenvolvimento web continuam a evoluir, a compatibilidade do UAES com as tecnologias de ponta juntamente com diferentes padrões de design podem se tornar um problema. Porém, a adaptabilidade da abordagem do UAES é evidente ao tratar o DOM da página web como strings, pois nos permite extrair eficientemente os elementos relevantes por meio de trechos pré definidos e algoritmos de busca em strings, tornando essa técnica extremamente configurável.

## 6 Trabalhos Relacionados

TESTAR é uma ferramenta que faz uso de scriptless testing utilizada em diversos estudos [5, 7, 8, 14, 37, 38]. Ela explora diferentes caminhos da AUT por meio da GUI e depende do Selenium WebDriver como API de acessibilidade para interagir com os elementos

do DOM. Não é necessário ter conhecimento do código fonte da GUI para utilizar o TESTAR e descobrir os elementos acionáveis. Porém, sua implementação padrão (Monkey testing) não avalia se os elementos foram unicamente descobertos ou não e opera sob um número definido de rodadas, o que pode levar a várias funcionalidades não interagidas e repetições desnecessárias. Implementações baseada em modelo ou Q-learning mostraram resultados satisfatórios em seus respectivos experimentos, mas a complexidade em relação ao ganho efetivo se mostra inferior ao uso do UAES, uma vez que o processo de implantação dessas técnicas exige conhecimento especializado.

WEBMATE é um gerador de testes em GUI [9, 10] que faz uso da API do Selenium para controlar remotamente o navegador e gerar seu usage model, que captura todos os caminhos possíveis de interações dentro da aplicação. Esse usage model é implementado como uma máquina de estados finito (FSM), em que cada estado representa um nó do autómato e, devido a essa estrutura, estados previamente visitados não são explorados. Cada estado é visto como um conjunto dos elementos que foram usados para interagir com a aplicação. Apesar de utilizar exploração sistemática, WEBMATE representa cada elemento como uma combinação de suas possíveis funcionalidades, levando a perda de informação uma vez que não existe uma distinção de elementos estáticos e dinâmicos. Ademais, ela depende da API do Selenium e é uma ferramenta de uso comercial que não possui o mesmo suporte da comunidade visto para ferramentas open-source como TESTAR.

A3E [3] é uma ferramenta open-source que permite exploração sistemática de aplicativos Android sem intervenção no código fonte. Ela visa complementar a suíte de testes por meio da geração de casos oriundos da exploração sistemática da GUI do aplicativo. Porém, ela permite interações repetitivas com os elementos Android durante a exploração automática. UAES por outro lado, oferece exploração sistemática para aplicações web e garante a unicidade dos elementos encontrados.

AMOGA [33] é uma ferramenta feita para exploração sistemática em aplicativos Android. Ela visa extrair um grafo (Windows Transition Graph) da aplicação sob teste para derivar as ações disponíveis no aplicativo e então estressar os elementos acionáveis extraídos. Seu foco está em descobrir unicamente os elementos acionáveis no aplicativo; entretanto, ela requer análise do código fonte do aplicativo ao invés de empregar teste caixa-preta

Zimmermann et al. [41] integra GPT-4 com Selenium WebDriver para melhorar a precisão e a cobertura de testes por meio do modelo integrado de IA. O modelo do GPT-4 interpreta o estado da GUI bem como seus elementos por meio do DOM atualizado provido pelo Selenium WebDriver, permitindo testes automáticos sem a necessidade de interação humana. Todavia, esse método não se configura como exploração sistemática e não apresenta foco em descobrir unicamente os elementos acionáveis. Além disso, UAES fornece resultados semanticamente significativos para as aplicações testadas sem fazer uso da licença industrial do GPT-4.

Kirinuki et al. [15] usa técnicas de NLP para identificar e selecionar elementos acionáveis em uma página web por meio da interpretação de casos de teste escritos em uma linguagem de domínio específico. Essa abordagem objetiva localizar elementos acionáveis sem necessitar de scripts de teste tradicionais, enfatizando a importância de selecionar informações legíveis que expressem a identidade do elemento acionável e oferece pistas robustas sobre sua semântica. Textos visíveis são considerados representações imediatas da semântica do que valores de atributos. Nós adotamos uma ideia similar de representação semântica em nosso trabalho, usado para definir unicidade dos elementos encontrados. Adicionalmente, a técnica mostrada por Kirinuki et al. requer que testadores especifiquem a existência dos elementos; concluindo que identificação automática dos elementos acionáveis não faz parte desse trabalho.

## 7 Conclusões

Neste trabalho, nós apresentamos a técnica *Unique Actionable Element Search* (UAES), uma abordagem automática para descobrir unicamente elementos acionáveis em aplicações web para testes de GUI por exploração sistemática. Essa técnica opera independentemente de qualquer API, utilizando o DOM da página web como *strings* para extrair eficientemente os elementos relevantes com a ajuda de trechos (*substrings*) predefinidos para o algoritmo de busca em *strings*. O UAES consegue diferenciar elementos que podem ser interagidos por ferramentas de teste de GUI. Nossa abordagem foca em elementos HTML comumente associados a elementos acionáveis, bem como chaves que funcionam como localizadores para facilitar a descoberta de novos elementos.

Nós avaliamos o UAES por meio de um estudo empírico em aplicações open-source. Comparamos o UAES com uma abordagem manual de marcação explícita, em que testadores manualmente identificaram os elementos acionáveis. Os resultados mostraram que o UAES tem potencial para ser tão efetivo quanto a técnica de marcação explícita quando se trata de descobrir elementos acionáveis. Entretanto, pelo estudo realizado, não há evidências suficientes para afirmar que um método seja superior ao outro em termos de eficácia.

Mesmo assim, o UAES é relevante pelo seu valor prático, que está em sua habilidade de automatizar a descoberta desses elementos sem intervenção manual, permitindo teste sistemático da GUI de forma simples. Adicionalmente, o UAES consistentemente consegue identificar elementos estruturalmente acionáveis e oferece uma forma de indicar valores de classes que forçam interatividade.

Como parte dos trabalhos futuros, planejamos refinar nossa técnica em um ambiente industrial, realizando o mesmo estudo para aplicações que sigam padrões de implementação definidos e possuam controle de qualidade robusto.

## Agradecimentos

Para todos aqueles que viram em mim algo que jamais encontrei.

## References

[1] Alfred V Aho and Margaret J Corasick. 1975. Efficient string matching: an aid to bibliographic search. *Commun. ACM* 18, 6 (1975), 333–340.

[2] Emil Alégroth, Robert Feldt, and Lisa Ryrholm. 2015. Visual gui testing in practice: challenges, problemsand limitations. *Empirical Software Engineering* 20 (2015), 694–744.

[3] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. Association for Computing Machinery, New York, NY, USA, 641–660. https://doi.org/10.1145/2509136.2509549

[4] Ishan Banerjee, Bao Nguyen, Vahid Garousi, and Atif Memon. 2013. Graphical user interface (GUI) testing: Systematic mapping and repository. *Information and Software Technology* 55, 10 (2013), 1679–1694.

[5] Sebastian Bauersfeld and Tanja E. J. Vos. 2014. User interface level testing with TESTAR: what about more sophisticated action specification and selection?. In *Seminar on Advanced Techniques and Tools for Software Evolution (SATToSE 2014) (CEUR Workshop Proceedings)*, Davide Di Ruscio and Vadim Zaytsev (Eds.). CEUR-WS.org, Germany, 60–78. http://sattose.org/2014 7th Edition Advanced Techniques and Tools for Software Evolution, SATToSE 2014 ; Conference date: 09-07-2014 Through 11-07-2014.

[6] João Helis Bernardo, Daniel Alencar da Costa, Sérgio Queiroz de Medeiros, and Uirá Kulesza. 2024. How do Machine Learning Projects use Continuous Integration Practices? An Empirical Study on GitHub Actions. arXiv:cs.SE/2403.09547

[7] Axel Bons, Beatriz Marín, Pekka Aho, and Tanja EJ Vos. 2023. Scripted and scriptless GUI testing for web applications: An industrial case. *Information and Software Technology* 158 (2023), 107172.

[8] Hatim Chahim, Mehmet Duran, Tanja EJ Vos, Pekka Aho, and Nelly Condori Fernandez. 2020. Scriptless testing at the GUI level in an industrial setting. In *Research Challenges in Information Science: 14th International Conference, RCIS 2020, Limassol, Cyprus, September 23–25, 2020, Proceedings 14*. Springer, 267–284.

[9] Valentin Dallmeier, Martin Burger, Tobias Orth, and Andreas Zeller. 2013. WebMate: Generating Test Cases for Web 2.0, Vol. 133. 55–69. https://doi.org/10.1007/978-3-642-35702-2_5

[10] Valentin Dallmeier, Bernd Pohl, Martin Burger, Michael Mirold, and Andreas Zeller. 2014. WebMate: Web application test generation in the real world. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 413–418.

[11] Markus Ermuth and Michael Pradel. 2016. Monkey see, monkey do: Effective generation of GUI tests with inferred macro events. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 82–93.

[12] Boni García, Micael Gallego, Francisco Gortázar, and Mario Munoz-Organero. 2020. A survey of the selenium ecosystem. *Electronics* 9, 7 (2020), 1067.

[13] William GJ Halfond and Alessandro Orso. 2007. Improving test case generation for web applications using automated interface discovery. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 145–154.

[14] Thorn Jansen, Fernando Pastor Ricós, Yaping Luo, Kevin van der Vlist, Robbert van Dalen, Pekka Aho, and Tanja EJ Vos. 2022. Scriptless GUI Testing on Mobile Applications. In *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 1103–1112.

[15] Hiroyuki Kirinuki, Shinsuke Matsumoto, Yoshiki Higo, and Shinji Kusumoto. 2021. NLP-assisted web element identification toward script-free testing. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 639–643.

[16] Daniel Kraus, Jeremias Rößler, and Martin Sulzmann. 2020. Visual Testing of GUIs by Abstraction. *arXiv preprint arXiv:2007.10419* (2020).

[17] D Rajya Lakshmi and S Suguna Mallika. 2017. A review on web application testing and its current research directions. *International Journal of Electrical and Computer Engineering* 7, 4 (2017), 2132.

[18] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. 2014. Visual vs. DOM-based web locators: An empirical study. In *Web Engineering: 14th International Conference, ICWE 2014, Toulouse, France, July 1-4, 2014. Proceedings 14*. Springer, 322–340.

[19] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. 2016. Approaches and tools for automated end-to-end web testing. In *Advances in Computers*. Vol. 101. Elsevier, 193–237.

[20] Maurizio Leotta, Boni García, Filippo Ricca, and Jim Whitehead. 2023. Challenges of end-to-end testing with selenium WebDriver and how to face them: A survey. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 339–350.

[21] Maurizio Leotta, Filippo Ricca, and Paolo Tonella. 2021. Sidereal: Statistical adaptive generation of robust locators for web testing. *Software Testing, Verification and Reliability* 31, 3 (2021), e1767.

[22] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2016. ROBULA+: An algorithm for generating robust XPath locators for web testing. *Journal of Software: Evolution and Process* 28, 3 (2016), 177–204.

[23] Grischa Liebel, Emil Alégroth, and Robert Feldt. 2013. State-of-practice in GUI-based system and acceptance testing: An industrial multiple-case study. In *2013 39th Euromicro Conference on Software Engineering and Advanced Applications*. IEEE, 17–24.

[24] Siân E Lindley, Sam Meek, Abigail Sellen, and Richard Harper. 2012. " It's simply integral to what I do" enquiries into how the web is weaved into everyday life. In *Proceedings of the 21st international conference on World Wide Web*. 1067–1076.

[25] Guillermo Macbeth, Eugenia Razumiejczyk, and Rubén Ledesma. 2011. Cliff's Delta Calculator: A non-parametric effect size program for two groups of observations. *Universitas Psychologica* 10 (05 2011), 545–555. https://doi.org/10.11144/Javeriana.upsy10-2.cdcp

[26] Atif Memon, Ishan Banerjee, and Adithya Nagarajan. 2003. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings*. IEEE, 260–269.

[27] Thiago Santos de Moura, Everton LG Alves, Hugo Feitosa de Figueirêdo, and Cláudio de Souza Baptista. 2023. Cytestion: Automated GUI Testing for Web Applications. In *Proceedings of the XXXVII Brazilian Symposium on Software Engineering*. 388–397.

[28] Nadim Nachar. 2008. The Mann-Whitney U: A Test for Assessing Whether Two Independent Samples Come from the Same Distribution. *Tutorials in Quantitative Methods for Psychology* 4 (03 2008). https://doi.org/10.20982/tqmp.04.1.p013

[29] Michel Nass, Emil Alégroth, and Robert Feldt. 2021. Why many challenges with GUI test automation (will) remain. *Information and Software Technology* 138 (2021), 106625.

[30] Michel Nass, Emil Alégroth, Robert Feldt, Maurizio Leotta, and Filippo Ricca. 2022. Similarity-based web element localization for robust test automation. *arXiv preprint arXiv:2208.00677* (2022).

[31] Filippo Ricca, Maurizio Leotta, and Andrea Stocco. 2019. Three open problems in the context of e2e web testing and a vision: Neonate. In *Advances in Computers*. Vol. 113. Elsevier, 89–133.

[32] Olivia Rodríguez-Valdés, Tanja EJ Vos, Pekka Aho, and Beatriz Marín. 2021. 30 years of automated GUI testing: a bibliometric analysis. In *Quality of Information and Communications Technology: 14th International Conference, QUATIC 2021, Algarve, Portugal, September 8–11, 2021, Proceedings 14*. Springer, 473–488.

[33] Ibrahim Anka Salihu and Rosziati Ibrahim. 2016. Systematic exploration of android apps' events for automated testing. In *Proceedings of the 14th International Conference on Advances in Mobile Computing and Multi Media*. 50–54.

[34] Andrea Stocco, Maurizio Leotta, Filippo Ricca, and Paolo Tonella. 2014. PESTO: A tool for migrating DOM-based to visual web tests. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 65–70.

[35] Tommi Takala, Mika Katara, and Julian Harty. 2011. Experiences of system-level model-based GUI testing of an Android application. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 377–386.

[36] Yuan-Hsin Tung, Shian-Shyong Tseng, Tsung-Ju Lee, and Jui-Feng Weng. 2010. A Novel Approach to Automatic Test Case Generation for Web Applications. In *2010 10th International Conference on Quality Software*. 399–404. https://doi.org/10.1109/QSIC.2010.33

[37] Tanja EJ Vos, Pekka Aho, Fernando Pastor Ricos, Olivia Rodriguez-Valdes, and Ad Mulders. 2021. testar–scriptless testing through graphical user interface. *Software Testing, Verification and Reliability* 31, 3 (2021), e1771.

[38] Tanja EJ Vos, Peter M Kruse, Nelly Condori-Fernández, Sebastian Bauersfeld, and Joachim Wegener. 2015. Testar: Tool support for test automation at the user interface level. *International Journal of Information System Modeling and Design (IJISMD)* 6, 3 (2015), 46–83.

[39] Thomas Wetzlmaier, Rudolf Ramler, and Werner Putschögl. 2016. A framework for monkey GUI testing. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 416–423.

[40] Dacong Yan, Shengqian Yang, and Atanas Rountev. 2013. Systematic testing for resource leaks in Android applications. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 411–420. https://doi.org/10.1109/ISSRE.2013.6698894

[41] Daniel Zimmermann and Anne Koziolek. 2023. GUI-Based Software Testing: An Automated Approach Using GPT-4 and Selenium WebDriver. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. IEEE, 171–174.