Geração de Testes automáticos para sistemas web a partir de casos de uso

Title: Generation of automated tests for web systems from Use Cases

Lucas C. F. Batista[1], **Sidney C. Nogueira**[1]

[1]Departamento de Computação – Universidade Federal Rural de Pernambuco (UFRPE)
Recife, Pernambuco – Brasil

{lucas.cfbatista, sidney.nogueira}@ufrpe.br

Abstract. *Use cases are widely used to describe interactions between users and systems in natural language, serving as a valuable source for deriving test cases. This work explores the generation of automated test cases for web systems through the extension of TaRGeT, a tool that generates manual test cases from a system model derived from use cases, to enable the generation of scripts for automated test execution implemented in Cypress. The extension introduces mappings from use case elements to the corresponding Cypress code, allowing TaRGeT to generate automated tests in addition to manual tests. To evaluate the proposed extension, it was applied to a real system, demonstrating its applicability.*

Keywords. *Use Cases; Model-Based Testing; Automated Test; Test Generation.*

Resumo. *Casos de uso são amplamente utilizados para descrever interações entre usuários e sistemas em linguagem natural, uma fonte valiosa para derivar casos de teste. Este trabalho explora a geração de testes automáticos para sistemas web através da extensão de TaRGeT, uma ferramenta de geração de casos de teste manuais a partir de um modelo do sistema derivado de casos de uso, para permitir gerar scripts para execução automática de testes implementados em Cypress. A extensão consiste na introdução de mapeamentos dos itens do caso de uso para o respectivo código, permitindo que TaRGeT gere testes automáticos em adição aos testes manuais. Para avaliar a extensão proposta, a mesma foi aplicada em um sistema real, demonstrando sua aplicabilidade.*

Palavras-Chave. *Casos de Uso; Testes Baseados em Modelos; Testes automáticos; Geração de Testes.*

1. Introdução

Desde 1979, é de conhecimento prático que aproximadamente 50% do custo total de desenvolvimento de um software é gasto com testes [Myers et al. 2011]. Testes de software

são feitos com o intuito de garantir a qualidade do sistema sobre teste (SUT - System Under Test) através da identificação de divergências das funcionalidades do SUT com as suas especificações. O grande interesse em teste de software se dá principalmente em razão de seu impacto econômico, quanto mais tarde o erro é encontrado no ciclo de desenvolvimento do software, mais caro é o seu reparo. Um erro encontrado após a entrega do programa, pelo consumidor, pode custar de 440 a 2900 vezes mais do que um encontrado na elaboração dos requisitos [Baziuk 1995].

Casos de teste são muito afetados por alterações nos requisitos de um sistema, e consequentemente, quanto mais alterações nos requisitos, mais manutenção se faz necessária nesses artefatos. Dentro do contexto ágil de desenvolvimento de software, onde a fase de testes do software ocorre de forma simultânea com o desenvolvimento, existe uma dependência dos testes automatizados, por serem menos laboriosos para executar em comparação com a execução manual dos testes, além de economizar tempo e evitar erros humanos [Myers et al. 2011].

A abordagem de Testes Baseados em Modelos (MBT - Model-based Testing) permite um alto grau de automação para a criação e execução dos testes. A partir de uma representação formal das especificações do SUT, é possível gerar casos de teste automáticos de forma automática, além de facilitar a validação de todos os fluxos desse modelo [Utting and Legeard 2010].

Modelos de Casos de uso são bastante utilizados na elaboração de requisitos, dada sua facilidade de identificar as interações entre os atores e o sistema, os casos de uso geralmente são documentados em diagramas [Sommerville 2011]; casos de teste derivam de casos de uso em vários projetos, dado que casos de teste também servem para validar requisitos e atender as expectativas dos usuários [Sommerville 2011]. Entretanto, a criação de casos de teste a partir de casos de uso de forma manual é bastante custosa.

Por outro lado, a geração automática de casos de teste a partir de casos de uso promove redução de esforço, uma cobertura abrangente, diminuição de risco de erros humanos, e facilidade de manutenção dos artefatos gerados.

A ferramenta TaRGeT (Test and Requirements Generation Tool) traz uma abordagem onde casos de uso descritos em linguagem natural são a entrada para a geração automática de casos de teste para a execução manual, que são acompanhados de informações de rastreabilidade dos requisitos associados aos casos de teste [Ferreira et al. 2010]. Além disso, TaRGeT também possibilita o uso de dados de entrada, como variáveis e constantes de forma a enriquecer a descrição dos fluxos dos casos de uso e permitir a geração de testes que possuem dados associados[Nogueira et al. 2019]. Apesar de ter ser mostrado eficaz para a geração e seleção de testes manuais, uma importante limitação da versão original de TaRGeT é não ser capaz de gerar testes para execução automática

Diversos *frameworks* [Framework 2025, Selenium 2025] tem sido propostos para suportar a automação de testes para sistemas *web*. Dentre os existentes, destacamos Cypress [Cypress 2024b], que é um *framework* de código aberto feito para desenvolvedores e analistas de controle de qualidade para escrita de testes automáticos em JavaScript para sistemas Web, com principal propósito a automatização de testes end-to-end

(E2E), ou teste de sistema, onde os componentes integrados são testados como um todo [Sommerville 2011]. Cypress fornece diversas facilidades para seus usuários, entre elas a interface própria com histórico de ações para depurações mais eficientes; o framework aguarda automaticamente os componentes do SUT carregarem, antes de executar ações e afirmações, o que aumenta a robustez dos testes; também é possível testar o front-end isoladamente com a criação de esboços de métodos, conhecidos como stubs, permitindo controlar o comportamento de funções do SUT [Cypress 2024b].

Este trabalho tem como principal objetivo utilizar casos de uso como fonte de informação para gerar casos de teste automáticos codificados em Cypress. Para isto, a ferramenta TaRGeT foi estendida com o objetivo de gerar scripts automáticos em Cypress, pela introdução de um novo artefato que define o mapeamento das instruções textuais dos casos de teste gerados para os respectivos trechos de código Cypress. A descrição do mapeamento permite a utilização de dados que são especificados nos passos do caso de uso, o que dá maior flexibilidade e facilidade para geração dos testes. O mapeamento é composto por colunas que relacionam entradas fornecidas ao sistema e saídas esperadas, que estão presentes nos fluxos dos casos de uso com o código automatizado correspondente. Durante a geração dos testes com TaRGeT, o mapeamento é utilizado para transformar as entradas e saídas especificados no caso de uso em comandos de ação e asserção do *framework* Cypress, permitindo que vários casos de testes automáticos completos sejam gerados de forma automática. O código fonte da solução está disponível [1].

Como validação deste trabalho, a versão extendida de TaRGeT foi utilizada para testar a funcionalidade de um sistema real. Casos de uso foram especificados em TaRGeT que gerou testes automáticos em Cypress, que foram executados no sistema. Na validação a ferramenta mostrou-se capaz de gerar testes funcionais e automatizados com sucesso, exigindo apenas configurações adicionais mínimas, como ajustes em variáveis de ambiente e definição de *fixtures*, que são arquivos que armazenam dados utilizados para alimentar os testes. Com as modificações, a ferramenta cresceu na sua utilidade e possibilidades de utilização. Como contribuição mais geral, enfatizamos que tanto o mapeamento, como os componentes do código que foram criados para permitir a geração de scripts, possuem a flexibilidade para serem facilmente adaptados para geração de testes automáticos em outras tecnologias além do Cypress.

No restante deste trabalho, temos na Seção 2 os principais conceitos sobre a geração de testes a partir de casos de uso. Na Seção 3, discutimos os trabalhos relacionados. A Seção 4 detalha como a ferramenta de MBT foi estendida para suportar a geração de scripts automáticos, enquanto na Seção 5, relatamos a aplicação da ferramenta estendida em um sistema real. Por fim, na Seção 6, trazemos as conclusões do estudo.

2. Geração de testes a partir de casos de uso

Usando casos de uso como entrada para MBT, podemos garantir que os testes gerados possuem uma cobertura mais completa dos cenários descritos nos casos de uso. O uso de MBT traz consigo a necessidade de uma ferramenta capaz de ler casos de uso. Neste trabalho, utilizamos como base a ferramenta TaRGeT.

[1] URL do repositório omitida em razão do blind review

2.1. Ferramenta de geração de testes (TaRGeT)

TaRGeT é uma ferramenta de geração de casos de teste manuais a partir de um modelo de casos de uso descrito em uma Linguagem Natural Controlada (*CNL - Controlled Natural Language*). O documento de casos de uso é utilizado como entrada para o processo de geração de testes da ferramenta, que dispõe de uma interface gráfica para criação desse documento [Ferreira et al. 2010]. O fluxo da ferramenta é mostrado na Figura 1, um documento de casos de uso é utilizado como entrada, e internamente a ferramenta gera um modelo dos testes em CSP (Communicating Sequential Processes) [Hoare et al. 1978], este modelo é entrada para a geração automática dos testes que resulta em uma Suíte de Casos de Teste em CSP, que é traduzida em uma Suíte de Casos de Teste descrita em CNL.

Figura 1. Fluxo da ferramenta TaRGeT [Nogueira et al. 2019]

Um documento de caso de uso em TaRGeT possui uma estrutura própria de elementos são eles Funcionalidade (*Feature*), Caso de Uso (*Use Case*), Fluxo (*Flow*) e Passo (*Step*). Cada documento deve ter pelo menos uma *feature*, cada *feature* deve ter pelo menos um *use case*, cada *use case* deve ter pelo menos um *flow*, e cada *flow* deve ter pelo menos um *step* [Ferreira et al. 2010]. Segue o detalhamento de cada elemento.

- ***Feature***: Uma funcionalidade é composta por um identificador, um nome, um conjunto de casos de uso, e opcionalmente uma seção para Definição de Dados (*Data Definition*).
- ***Use Case***: Um caso de uso é composto por um identificador, um nome, descrição, informações de configuração e fluxos.
- ***Flow***: Um fluxo é composto por uma descrição, referências para os passos de origem e de destino do fluxo, e uma sequência de passos.
- ***Step***: Um passo é composto por um identificador, uma ação de entrada para o sistema, e uma resposta produzida pelo sistema. Opcionalmente, o passo pode conter uma condição necessária para que o passo seja executado (expressão booleana).

A seção *Data Definition* permite definir variáveis, constantes e tipos dentro do escopo de um caso de uso que são utilizados como parte da especificação do caso de uso [Nogueira et al. 2019]. Essas definições são organizadas em tabelas como as da Figura 2. Neste exemplo, temos a definição do tipo `User`, que pode assumir os valores {`E_0`, `E_1`, `E_2`}, e a definição da variável `Users_Test`, um conjunto de usuários representados pelos valores {`E_0`, `E_1`}.

O caso de uso exemplificado na Figura 3, utiliza os dados definidos na Figura 2. O caso de uso deste exemplo descreve o preenchimento dos campos de e-mail e senha, e a realização do login. No passo `1M` existe um carácter `$`, que é utilizado para indicar uma entrada no caso de uso, que é definida pela expressão matemática definida dentro dos caracteres `%`. Neste exemplo, o dado é um usuário que pertence ao conjunto de usuário

Data Definition

New Type

Id	Description	Value
User	User types	E_0, E_1, E_2

Variables

Id	Description	VarType	Value
Users_Test		set of User	{E_0, E_1}

Figura 2. Definição de Dados em uma Feature na ferramenta TaRGeT.

definido na variável `Users_Test` demonstrada na Figura 2. O passo `1M` descreve a ação de escrever o e-mail do usuário no campo correspondente, e o resultado esperado para essa ação é o e-mail estar escrito no campo correto. Este mesmo passo demonstra a sintaxe para utilização dos dados, neste exemplo, a entrada `x` é definida matemáticamente como um subconjunto não-vazio de Usuários selecionados a partir da variável `Users_Test`. O passo `2M` descreve a ação de escrever a senha do usuário no campo correspondente, e o resultado esperado para essa ação é que o conteúdo da senha não deve ser mostrado. O sistema deve exibir caracteres `"*"` com a mesma quantidade de caracteres na senha do usuário. O passo `3M` descreve a ação de clicar no botão ”Entrar”, e o resultado esperado para essa ação é que o usuário deve ser logado e a tela de visualização correspondente é mostrada.

UC_02 - Login

Description: This use case fills the login fields with values and performs the login

Setup:

FLOWS

Description: Filling the email and password fields and performing the login
From Step: START
To Step: END

Step Id	User Action	System Condition	System Response
1M	Type $ email on the email field % Input x : set of User from powerset of Users_Test such that x is non-empty %		Email is writen in field
2M	Type the user password on the password field		Input box shows "*" for each character input on the password field
3M	Click on the Entrar button		Logged in page with user's view is shown

Figura 3. Exemplo de Caso de Uso na ferramenta TaRGeT.

Ainda neste exemplo é utilizado o campo `From Step`, que representa o ponto inicial do fluxo do Caso de Uso. Quando este campo é definido como `START`, significa que o fluxo não possui dependência com passos anteriores. Já o campo `To Step` indica que o fluxo atual continua nos passos de outros casos de uso. Quando o valor deste campo é definido como `END` significa que o fluxo termina após seu ultimo passo.

Por concisão, o exemplo apresentado possui apenas um fluxo, entretanto, na prática os casos de uso possuem vários fluxos (alternativos, de exceção e de erro) que são combinados e interconectados através dos campos `From Step` e `To Step`. Desta forma, o modelo de casos de uso pode conter diversas ramificações no comportamento que dão origem a diferentes casos de teste. A medida que crescem as combinações surgem novos casos de teste a serem gerados.

Apresentamos os campos dos casos de teste gerados por TaRGeT usando como entrada casos de uso, são eles:

- ***Use Case***: Uma lista de todos os casos de uso que são relacionados com o caso de teste.
- **Requisitos (*Requirements*)**: O conteúdo deste campo é extraído do identificador dos requisitos, que são anotados no caso de uso.
- **Configuração (*Setup*)**: Informações de configuração necessárias para execução do caso de teste derivadas do caso de uso. Essa informação é extraída do campo *setup* da *feature*.
- **Condições iniciais (*Initial Conditions*)**: Condições necessárias que precisam ser atendidas antes do início da execução do caso de teste.
- **Passos (*Steps*)**: Sequência de ações do usuário derivadas do caso de uso.
- **Resultados Esperados (*Expected Results*)** : Sequência de respostas do sistema para cada ação correspondente, também derivadas do caso de uso.

Esses elementos, e mais alguns, podem ser observados no exemplo de caso de teste presente na Figura 4. A instância dos dados de entrada especificadas nos passos do casos aparecem nos testes gerados. Por exemplo, no passo 3, o valor {`E_1`} representa um dos valores possíveis para a entrada `x` do passo `1M` da Figura 3.

Test Case ID: WIA.Input.FUNC:001-001

Regression Level: na
Execution Type: Man
Description: This use case launches the browser and opens the login page. This use case fills the login fields with values and performs the login.
Objective: Launching the browser and navigating to Acolhe login page. Filling the email and password fields and performing the login.

Use Case References: 11170#UC_01, 11170#UC_02
Requirements: None.
Setups: None.

Initial Conditions: None.

Steps	Expected Results
1) Launch Browser and go to Acolhe website.	Browser is launched and Acolhe website is opened.
2) Click the Entrar button.	The login page is displayed.
3) Type {E_1} email on the email field.	Email is written in the field.
4) Type the user password on the password field.	Input box shows "*" for each character input on the password field.
5) Click on the Entrar button.	Logged-in page with user's view is shown.

Final Conditions: None.
Cleanup: None.
Notes: Test case auto-generated by TaRGeT system.

Figura 4. Exemplo de Caso de Teste gerado pela ferramenta TaRGeT

2.2. Framework de automação Cypress

Cypress é um framework de automação de testes voltado para aplicações web, amplamente utilizado para realizar testes end-to-end (E2E). Um exemplo básico de um script de teste Cypress pode ser visto na Figura 5.

```
const INPUT_USUARIO = '#usuario';

describe('Teste de login com fixture', () => {
    it('Deve realizar o login com sucesso', () => {
        cy.fixture('usuario.json').then((usuarios) => {
            cy.visit(`${Cypress.config('baseUrl')}`);
            cy.get(INPUT_USUARIO).type(this.usuarios.usuario);
            cy.get('#senha').type(this.usuarios.senha);
            cy.get('button[type="submit"]').click();
            cy.url().should('include', '/dashboard');
        });
    });
});
```

Figura 5. Exemplo de script Cypress para teste de login.

A estrutura do código Cypress baseia-se em alguns elementos principais, são eles:

- ***baseUrl***: definido no arquivo de configuração do Cypress, é uma propriedade utilizada para definir uma URL base que será automaticamente prefixada em todos os comandos que utilizam o método `cy.visit()`.
- **Comandos**: Na figura acima, `cy.visit()` navega até a baseURL definida anteriormente, `cy.get()` busca elementos na página, e `cy.type()` insere valores nos campos de entrada.
- **Afirmações (*Assertions*)**: são usadas para verificar se o sistema se comporta conforme esperado. No exemplo, `cy.url().should('include', '/dashboard')` valida que a URL mudou e contém o endpoint `'/dashboard'` após o login [Cypress 2024a].
- **Blocos para estruturação**: Cypress utiliza blocos para estruturar e organizar os testes, o bloco `describe` agrupa um conjunto de casos de teste, enquanto o bloco `it` descreve um caso de teste específico [Cypress 2022].

Além dos comandos e *assertions*, Cypress também oferece funcionalidades avançadas que facilitam a automação de testes complexos. Entre elas, destacam-se o uso de *fixtures* e constantes que auxiliam na organização e reutilização de dados dentro dos testes.

Fixtures são arquivos que armazenam dados utilizados para alimentar os testes. Eles permitem a separação entre o fluxo do teste e dados utilizados, facilitando a reutilização de dados em diferentes cenários de teste e tornando os scripts mais limpos e fáceis de manter. Por exemplo, no script de exemplo da Figura 5, em vez de definir diretamente no script o nome de usuário e a senha, esses dados podem ser armazenados em um arquivo de *fixture* como um JSON (notação de objeto de JavaScript), um formato de serialização de dados baseado em literais de JavaScript [Flanagan 2012] como o da Figura 6.

Outro conceito importante no Cypress é o uso de constantes que são valores que não mudam durante a execução dos testes e podem ser definidas diretamente no código ,

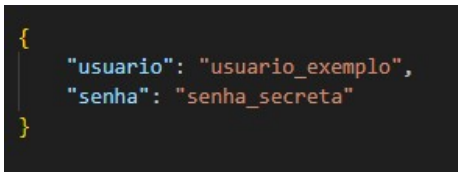

Figura 6. Exemplo de arquivo `usuario.json` usado como fixture no script da Figura 5.

são usados para garantir a consistência e evitar repetição de valores em diferentes partes dos testes. No exemplo de script da Figura 5 uma constante chamada `INPUT_USUARIO` é definida no início do script e referenciada ao longo do teste. Caso esse valor seja alterado no futuro, a modificação é feita apenas em um local, sem a necessidade de alterar múltiplos pontos no código.

3. Trabalhos relacionados

A geração automática de casos de teste a partir de modelos tem sido amplamente estudada na literatura [Liu and Nakajima 2022, Zafar et al. 2021, Wang et al. 2015, Sarmiento et al. 2014, Hasling et al. 2008, Nebut et al. 2006]. No entanto, um dos desafios recorrentes é a necessidade de criar e estruturar manualmente os modelos de entrada, o que exige habilidades especializadas e representa um esforço significativo para os testadores.

Uma abordagem recorrentemente adotada é o uso de modelos UML para derivar casos de teste utilizando MBT. [Zafar et al. 2021] e [Hasling et al. 2008] propõem estratégias para gerar automaticamente casos de teste a partir de diagramas UML. No entanto, esses diagramas precisam ser criados manualmente, o que exige alto nível de experiência dos usuários e um conjunto específico de habilidades. De maneira similar, [Wang et al. 2015] utilizam diagramas de sequência e de casos de uso como entrada para a geração de testes, mas ressaltam que a modelagem prévia do SUT continua sendo um dos principais desafios para a automação efetiva.

Outros trabalhos buscam reduzir a necessidade de modelos gráficos ao empregar especificações formais ou requisitos em linguagem natural como base para a geração de testes. [Liu and Nakajima 2022] utilizam métodos matemáticos para especificar formalmente os sistemas e derivar testes de forma automática. No entanto, essas especificações ainda precisam ser escritas manualmente, tornando a abordagem dependente de conhecimento especializado e limitando sua adoção prática.

Alternativamente, algumas pesquisas exploram técnicas de Processamento de Linguagem Natural (PLN) para extrair informações estruturadas de requisitos textuais. [Sarmiento et al. 2014] propõem a identificação de variáveis e condições de entrada a partir de descrições textuais dos requisitos. Contudo, a extração não é totalmente automatizada, exigindo intervenção humana para validar a precisão dos dados gerados. De maneira semelhante, [Nebut et al. 2006] investigam a conversão de casos de uso escritos em linguagem natural para modelos formais, mas destacam que essa transformação ainda requer um esforço manual considerável.

Uma abordagem distinta é apresentada por [Arruda et al. 2019], que foca na automação de casos de teste escritos em linguagem natural, utilizando as ferramentas AutoMano e Kaki para reutilização e verificação de consistência. Embora essa solução ofereça maior flexibilidade ao trabalhar diretamente com descrições textuais, ela depende da existência de casos de teste previamente descritos, exigindo captura manual de interações para alimentar a base de dados de ações reutilizáveis. Dessa forma, sua aplicabilidade continua limitada pelo esforço necessário para estruturar esses insumos.

A maioria das abordagens existentes depende fortemente de modelos formais, semi-formais ou da intervenção manual para estruturar os insumos necessários à geração de testes. Essa dependência impõe barreiras à adoção prática dessas ferramentas, pois exige conhecimentos específicos e um esforço considerável na modelagem.

Em contrapartida, a abordagem proposta neste trabalho oferece maior flexibilidade ao permitir a geração de casos de teste diretamente a partir de descrições textuais, sem depender de modelos formais ou semi-formais. A extensão da ferramenta TaRGeT possibilita essa automação sem a necessidade de criar manualmente diagramas UML ou especificações matemáticas, tornando o processo mais acessível a testadores sem experiência em modelagem. Além disso, o uso de um mapeamento de passos simplifica a transição entre os requisitos e a execução dos testes, reduzindo significativamente a carga manual associada à criação das suítes de teste.

Outro diferencial da solução proposta é a escolha do Cypress como framework de execução, garantindo compatibilidade com ferramentas modernas amplamente adotadas na indústria. Essa integração permite que os testes gerados possam ser facilmente incorporados aos fluxos de desenvolvimento contínuo, aumentando a aplicabilidade prática da abordagem e reduzindo as barreiras de adoção observadas em soluções anteriores.

4. Estendendo ferramenta de MBT para geração de scripts

Para estender a ferramenta TaRGeT para gerar casos de teste automáticos, foi necessário incluir um novo artefato de entrada, um mapeamento de cada entrada do usuário e resultado esperado dos casos de uso de entrada. O artefato relaciona o texto em linguagem natural a um trecho de código, com a intenção de obter um caso de teste automático. A Figura 7 mostra o fluxo de geração de testes manuais e automáticos em TaRGeT após a extensão. Primeiro são gerados casos de teste manuais a partir dos casos de uso, em seguida, são gerados casos de teste automáticos a partir de cada caso de teste manual.

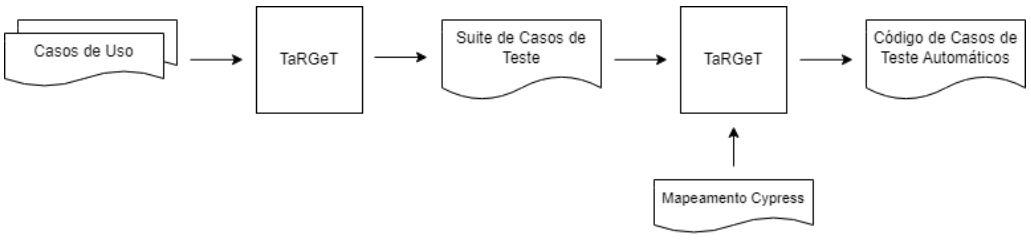

Figura 7. Fluxo da ferramenta TaRGeT para geração de testes automáticos após extensão

Dado que um único caso de uso pode originar múltiplos casos de teste, com diferentes entradas e condições, essa abordagem facilita a cobertura de diversos cenários. Por exemplo, ao invés de escrever manualmente um script para cada situação específica, a ferramenta gera automaticamente todos os scripts necessários, adaptando os passos de acordo com as variações nos dados de entrada ou nas respostas esperadas. Esse processo automatizado elimina o trabalho repetitivo de ajustar cada script manualmente, reduz a possibilidade de erros humanos e permite que o desenvolvedor dedique seu tempo a outras atividades.

4.1. Mapeando passos de caso teste em trechos de código

A utilização de um arquivo de mapeamento, em substituição a inserção de informações diretamente na interface gráfica, apresenta diversas vantagens. Primeiramente, essa abordagem permite desenvolver a automação de forma independente da especificação em linguagem natural controlada (CNL), promovendo a possibilidade de colaboração de pessoas com diferentes níveis de conhecimento no preenchimento do mapeamento. Além disso, o uso de arquivos de mapeamento possibilita a criação de diferentes mapeamentos para um mesmo caso de uso, permitindo a adaptação dos testes a diversas situações.

O mapeamento dos passos dos caso de uso para trechos de código é feito em um novo artefato de entrada usado por TaRGeT, um arquivo `.xlsx` (*Microsoft Excel Open XML Format Spreadsheet*) composto de 6 colunas, onde o valor das cinco primeiras colunas derivam dos passos dos casos de uso usados como entrada para a ferramenta. As colunas do arquivo de mapeamento são explicadas no que segue.

- ***Feature***: o ID da *Feature* no documento de casos de uso.
- ***UC***: o ID do Caso de Uso correspondente.
- ***Step***: o ID de um passo de caso de uso.
- ***Type***: o tipo do passo, `A` para Ação (*Action*) e `R` para Resposta (*Response*).
- ***Value***: o texto do passo do caso de uso.
- ***Code***: o código que será gerado para o passo do caso de teste.

Tabela 1. Tabela com exemplo de um mapeamento.

#	Feature	UC	Step	Type	Value	Code
1	11170	1	1	A	Launch Browser and go to Acolhe website	cy.visit(‘$Cypress. config(’baseUrl’)‘);
2	11170	1	1	R	Browser is launched and Acolhe website is launched	cy.location(’pathname’, timeout:1000).should(’to.eq’, ‘/inicio‘);

A Tabela 1 ilustra um exemplo de mapeamento. Olhando essa tabela, é possível notar que sua estrutura não possui nenhuma característica específica de um *framework* de automação em particular, tornando a solução facilmente adaptável a frameworks diferentes de Cypress.

4.2. Implementando a geração de scripts

Em tempo de geração dos testes manuais a lista de casos de teste manuais gerados por TaRGeT é usada como entrada para a geração dos scripts automáticos escritos em Cypress. Um a um, os testes manuais são traduzidos para a sua versão em Cypress.

A geração de scripts implementada na ferramenta introduziu um novo componente localizado no *plug-in* `On The Fly Generation`. Esse *plug-in* fornece uma pré-visualização dos casos de teste manuais que serão gerados. O novo componente, codificado em Java, utiliza como entrada para o método de geração de scripts uma lista de casos de teste gerados em formato textual, disponíveis durante a pré-visualização dos mesmos, o arquivo de mapeamento no formato .xlsx e o arquivo de saída onde o script será escrito.

TaRGeT organiza os artefatos de entrada e saída em um projeto de forma estruturada, facilitando o gerenciamento e o acesso aos arquivos. Há pastas dedicadas para armazenar os casos de uso, para os requisitos, para arquivos exportados e para as suítes de teste geradas. Na extensão da ferramenta realizada por este trabalho, foi feita a adição do diretório chamado `code_generation` ao projeto, dedicado exclusivamente aos artefatos relacionados à geração de código.

Explicamos os passos do processo de geração dos scripts. Inicialmente, o arquivo de mapeamento é carregado em memória, considerando que o arquivo do mapeamento está no diretório `code_generation`. Em seguida, as colunas UC e Step do arquivo de mapeamento são tratadas, dado que o valor para essas colunas são referências para passos de casos de uso usadas pela ferramenta para gerar, na forma de comentário dos testes automáticos, o conteúdo dos passos. Esse comentário é uma documentação importante para entendimento do script gerado, ao mesmo tempo que ajuda na rastreabilidade entre o código gerado e os passos do casos de uso. Ilustramos em detalhe a tradução de um passo do mapeamento mostrado na Figura 8.

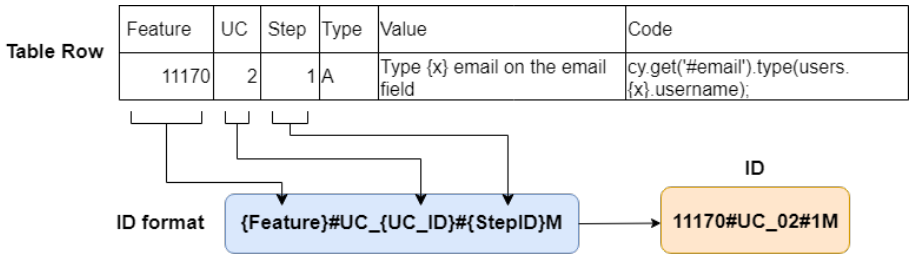

Figura 8. Criação de ID a partir de linha da tabela.

A Figura 9 ilustra como uma frase de um caso de teste é traduzida para código em Cypress. Para isto, usa como exemplo uma linha de mapeamento da Figura 8. Conforme mostra a Figura 9, um passo textual do caso de teste manual é analisado em busca de espaços reservados (*placeholder*), que são caracterizados como valores entre chaves. Na figura, o passo do caso de teste está na cor azul, o mapeamento que é aplicado ao passo esta na cor amarela, e o código gerado está na cor verde. Considerando que o texto dos passos dos casos de teste quanto o código no mapeamento possuem espaços reservados.

No exemplo em questão, o valor {E_1} corresponde a um valor existente no teste. Já a notação {x} corresponde a um espaço reservado no mapeamento. O espaço reservado do caso de teste indica o valor que será extraído. Já o espaço do mapeamento indica onde o valor do teste será incluído. Como resultado final, o espaço reservado no código é substituído pelo valor obtido do texto do caso de teste.

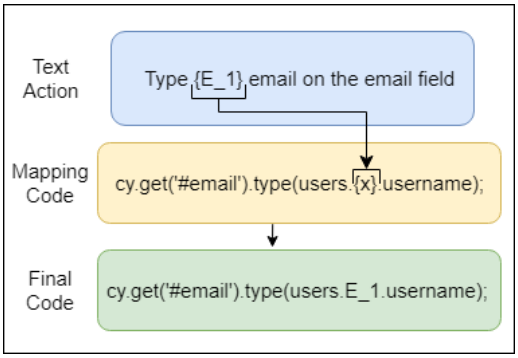

Figura 9. Exemplo de substituição de placeholder feita pelo algoritmo.

Na Figura 10, outro um exemplo mais completo de mapeamento, onde um passo completo de um caso de teste é traduzido para código utilizando o mapeamento de acordo com a abordagem proposta. Nesta figura, podemos observar como o código é estruturado.

Passo e Resultados Esperados de um Caso de Teste Gerado

Steps	Expected Results
1) Launch Browser and go to Acolhe website.	Browser is launched and Acolhe website is opened.

Valores de mapeamento para o Passo e Resultado Esperado acima

Feature	UC	Step	Type	Value	Code
11170	1	1	A	Launch Browser and go to Acolhe website	cy.visit(`${Cypress.config('baseUrl')}`);
11170	1	1	R	Browser is launched and Acolhe website is	cy.location('pathname',{timeout: 1000}).should('to.eq',`/inicio`);

Código gerado com base no Mapeamento e Caso de Teste Gerado

```
3  describe('Feature-11170', () => {
4      //----------------------------------------
5      // Use Case References: 11170#UC_01, 11170#UC_02
6      // Objective:  Launching the browser and navigating to Acolhe login page
7      // then filling the email and password fields and performing the login
8      it(`TC ID - WIA.Input.FUNC:001-001`, () => {
9          cy.fixture('users.json').then((users) => {
10             // Action: Launch Browser and go to Acolhe website
11             cy.visit(`${Cypress.config('baseUrl')}`);
12             // Response: Browser is launched and Acolhe website is launched
13             cy.location('pathname',{timeout: 1000}).should('to.eq',`/inicio`);
14         });
15     });
16 });
```

Figura 10. Exemplo de código gerado a partir de trecho de um caso de teste.

Na escrita do script cada passo do caso de teste é convertido em um trecho de código (linhas 11 e 13) e cada passo e resultado esperado é inserido em forma de co-

mentário (linhas 10 e 12) para que seja claro a qual ação ou asserção aquele código corresponde, como pode ser observado no exemplo de código presente na Figura 10.

Nem sempre o mapeamento está completo para todas as frases, uma vez que é desenvolvido de forma interativa e incremental. Por este motivo, na estratégia proposta, caso um trecho de código não seja encontrado no mapeamento, é inserido um comentário no script indicando a ausência de código para aquela ação ou resposta, assegurando que mesmo na ausência de um mapeamento completo, o teste automatizado seja gerado com comentários que auxiliem na depuração. Após isso, o arquivo com o script é gerado e exportado o diretório `code_generation`.

Além das informações obtidas diretamente do mapeamento, no código, existem algumas estruturas específicas do framework Cypress foram adicionadas de forma embutida ao código gerado para adequação aos exemplos utilizados. Por exemplo, conforme pode ser visto na Figura 10, cada *Feature* é representada por um bloco `Describe`, e cada teste automático gerado é encapsulado em um bloco `it`. Além disso, é realizada a inclusão de `fixtures`, para pleno funcionamento das entradas incluídas nos testes. Essas estruturas são facilmente adaptáveis a outra linguagem.

4.2.1. Geração e atualização do arquivo de mapeamento

Uma das vantagens da abordagem proposta é que o usuário não precisa criar o arquivo de mapeamento manualmente: a ferramenta foi adaptada para gerar o mapeamento parcialmente preenchido de forma automática, com as colunas *Feature*, *UC*, *Step*, *Type* e Value preenchidas com base nos casos de uso, conforme a Tabela 2. Com isto, o usuário só precisa completar a tabela preenchendo apenas a coluna `Code`. Em adição, de forma automática a ferramenta checa a existência do arquivo de mapeamento, e caso não exista, o diretório `code_generation` é criado e o arquivo é gerado.

Tabela 2. Modelo de mapeamento parcialmente gerado.

#	Feature	UC	Step	Type	Value	Code
1	11170	1	1	A	Action text	
2	11170	1	1	R	Response text	
3	11170	1	2	A	Action text	
4	11170	1	2	R	Response text	

Caso o arquivo já exista, quando um novo passo no caso de uso é adicionado, não é preciso refazer o mapeamento. A ferramenta automaticamente insere novas linhas em suas posições correspondentes do mapeamento já existente.

4.3. Modificando a interface

Descrevemos as mudanças na interface gráfica da ferramenta TaRGeT para possibilitar a geração automática de scripts de teste e do mapeamento correspondente. A interface da tela `Test Case Viewer` foi modificada com a adição de dois botões: um destinado à geração do script e outro para a criação ou atualização do arquivo de mapeamento, conforme ilustrado na Figura 11.

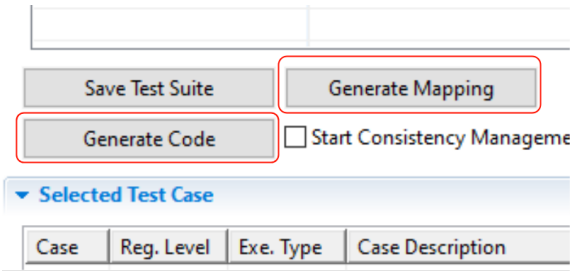

Figura 11. Modificações na interface do usuário da ferramenta TaRGeT

Após a geração dos testes pela ferramenta, ao selecionar o botão *"Generate Code"*, o processo descrito na Seção 4.2 é executado. De forma semelhante, ao selecionar o botão *"Generate Mapping"*, o processo detalhado na Seção 4.2.1 é iniciado. Em ambos os casos, o usuário é notificado sobre o sucesso ou falha da operação, conforme ilustrado na Figura 12.

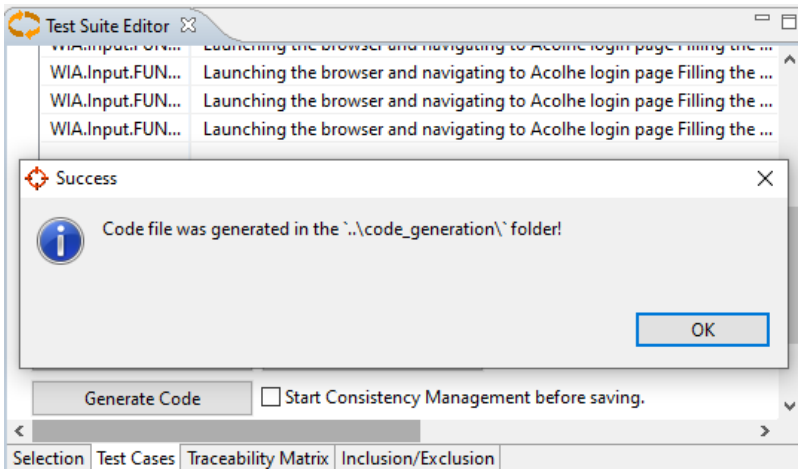

Figura 12. Retorno de uma geração de código bem-sucedida na ferramenta.

5. Avaliação da ferramenta em um sistema real

Avaliamos a utilização da ferramenta TaRGeT estendida para gerar testes automáticos em um sistema web real. A avaliação considera a adesão às práticas recomendadas do framework, incluindo o uso de Fixtures e Constantes, para garantir que os resultados da avaliação sejam representativos do desempenho da ferramenta em um ambiente de produção.

Para a avaliação foi escolhido o sistema Acolhe[Alves et al. 2024], que é um sistema web projetado para atender as necessidades dos cidadãos em momentos de crise provocados por eventos naturais. O Sistema foi feito para ser utilizado por órgãos públicos, principalmente órgãos ligados à Assistência Social e Defesa Civil. O Sistema dispõe de diferentes perfis de usuário Gestor, Cadastrador, Coordenador e Visualizador. Além dos usuários que tem acesso ao sistema, o sistema persiste Famílias, Abrigos, Pessoas e Voluntários. Na avaliação da ferramenta foram utilizadas credenciais de teste para Gestor e Visualizador.

5.1. Modelando login e cadastro utilizando casos de uso

Para avaliação, 3 casos de uso distintos foram criados para as funcionalidades de login e cadastro do sistema. Na Figura 13 temos o `UC_01`, um caso de uso auxiliar- casos de uso auxiliares são utilizados como parte de outros casos de uso. ara indicar em TaRGeT que um caso de uso é auxiliar é preciso colocar o valor `true` no campo `Auxiliary`. O `UC_01` foi criado com o objetivo de acessar o site do acolhe, e posteriormente a página de login do sistema.

UC_01 - Launching login page

Description:

This use case launches the browser and opens the login page

Auxiliary:

true

Main Flow

Description: Launching the browser and navigating to Acolhe login page
From Steps: START
To Steps: END

Step Id	User Action	System Response
1M	Launch Browser and go to Acolhe website	Browser is launched and Acolhe website is launched
2M	Click the Entrar button	The login page is displayed

Figura 13. Caso de Uso UC_01.

O `UC_02`, ilustrado na Figura 14 inclui os passos do `UC_01`, desde o início do mesmo, em seu fluxo principal, isso é especificado pelo atributo `Relations` que mostra que `UC_01` é incluído a partir da posição `START`. O `UC_02` espera que o usuário preencha os valores de e-mail e senha, e então faz login no sistema, considerando os usuários `E_0` ou `E_1`, conforme a sintaxe de entrada presente no passo `1M` descrita na Seção 2.1.

Finalmente, o `UC_03`, apresentado na Figura 15 inclui os passos do `UC_02` - de forma transitiva, inclui os passos do caso de uso `UC_01` em seu fluxo principal. O `UC_03` descreve em seus passos o fluxo de cadastro de um novo usuário do tipo `Gestor`, navega pela interface até a aba de cadastro e faz o preenchimento das informações de um novo usuário, após isso checa que o sistema retorna que o usuário foi cadastrado com sucesso.

UC_02 - Login

Description:

This use case fills the login fields with values and performs the login

Relations

Include

UseCaseId	Position
UC_01	START

Main Flow

Description: Filling the email and password fields and performing the login
From Steps: START
To Steps: END

Step Id	User Action	System Response
1M	Type $ email on the email field % Input x : set of User from powerset of Users_Test such that x is non-empty %	Email is writen in field
2M	Type the user password on the password field	Input box shows "*" for each character input on the password field
3M	Click on the Entrar button	Logged in page with user's view is shown

Figura 14. Caso de Uso UC_02.

UC_03 - Register a new user

Description:
Register a new user as a Gestor user
Relations
Include

UseCaseId	Position
UC_02	START

Main Flow
Description: Registering a new user as Gestor
From Steps: START
To Steps: END

Step Id	User Action	System Response
1M	Click on the "Usuarios" tab	The "Usuarios" tab opens
2M	Click on the "Usuario" button	The register form opens
3M	Type the informations of the new user	All informations are properly filled
4M	Click on the "Cadastrar" button	The new user is registered and "Usuario cadastrado com sucesso" message is shown

Figura 15. Caso de Uso UC_03.

5.2. Gerando e executando testes automáticos

O primeiro passo para gerar os casos de teste automáticos foi gerar o arquivo de mapeamento, após isso, um trecho de código Cypress correspondente foi escrito em cada linha de ação e resposta gerada, conforme a Figura 16.

Feature	UC	Step	Type	Value	Code
11170	1	1	A	Launch Browser and go to Acolhe website	cy.visit(`${Cypress.config('baseUrl')}`);
11170	1	1	R	Browser is launched and Acolhe website is	cy.location('pathname',{timeout: 1000}).should('to.eq',`/inicio`);
11170	1	2	A	Click the Entrar button	cy.contains('Entrar').click();
11170	1	2	R	The login page is displayed	cy.location('pathname', {timeout: 1000}).should('to.eq',`/${LOGIN_PATH}`);
11170	2	1	A	Type {x} email on the email field	cy.get('#email').type(users.{x}.username);
11170	2	1	R	Email is written in field	cy.get('#email').should('have.value',users.{x}.username);
11170	2	2	A	Type the user password on the password field	cy.get('#password').type(users.{x}.password, {log:false});
11170	2	2	R	Input box shows for "*" for each character input on the password field	cy.get('#password').should('have.value',users.{x}.password,{log:false});
11170	2	3	A	Click on the Entrar button	cy.contains('Entrar').click();
11170	2	3	R	Logged in page with user's view is shown	cy.url().should('be.equal',`${Cypress.config('baseUrl')}/${DASHBOARD_PATH}
11170	3	1	A	Click on the "Usuários" tab	cy.contains('Usuários').click();
11170	3	1	R	The "Usuários" tab opens	cy.url().should('be.equal', `${Cypress.config('baseUrl')}/${USERS_PATH}`);
11170	3	2	A	Click on the "Usuário" button	cy.contains('button','Usuário').click();
11170	3	2	R	The register form opens	cy.url().should('be.equal', `${Cypress.config('baseUrl')}/${USERS_NEW_PATH
11170	3	3	A	Type the informations of the new user	cy.get('#name').type('NomeTesteGenerated'); cy.get('#cpf').type('04561745084'); cy.get('#email').type('emailtestegenerated@gmail.com'); cy.get('.user-form__permission > .mat-mdc-form-field > .mat-mdc-te cy.get('mat-option').contains(`Gestor do Acolhe`).click(); cy.get('body').click(); cy.get('#phone').type('81999999999');
11170	3	3	R	All informations are properly filled	cy.contains('button','Cadastrar').should('be.visible').should('be.enabled');
11170	3	4	A	Click on the "Cadastrar" button	cy.contains('button','Cadastrar').click();
11170	3	4	R	The new user is registered and "Usuário cadastrado com sucesso" message is shown	cy.url().should('be.equal',`${Cypress.config('baseUrl')}/${USERS_PATH}`); cy.get('.mat-mdc-simple-snack-bar > .mat-mdc-snack-bar-label').should('t

Figura 16. Mapeamento para geração dos scripts do Sistema web ”Acolhe”.

A Figura 17 ilustra os passos na ferramenta TaRGeT para gerar os testes manuais: primeiro o projeto criado é carregado na ferramenta (acessando os menus `File -> Open project`), em seguida os testes manuais são gerados (acessando os menus `Tools -> Generate Test Cases -> All Steps Generation`), e, finalmente os testes gerados são e listados na tela `Test Case Viewer`. Para gerar os testes automáticos, basta clicar no botão *"`Generate Code`"*, então o arquivo com os scripts será gerado. A Figura 18 ilustra um trecho do código gerado a partir do mapeamento de passos para o teste `WIA.Input.FUNC:001-001`.

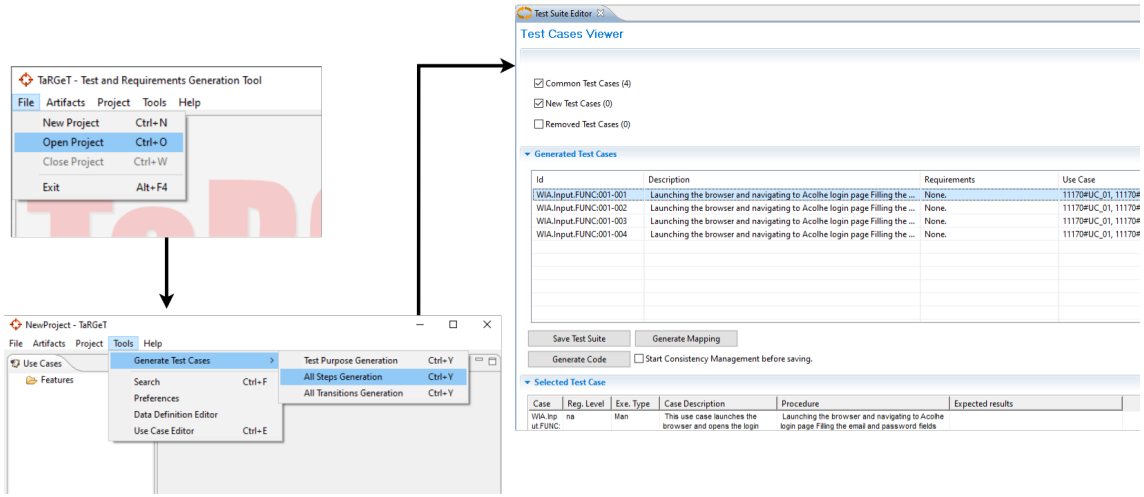

Figura 17. Fluxo de geração de testes manuais.

```
describe('Feature-11170', () => {
    //---------------------------------------
        // Use Case References: 11170#UC_01, 11170#UC_02
        // Objective:  Launching the browser and navigating to Acolhe login page Filling the email and password fields and performing the login
        it(`TC ID - WIA.Input.FUNC:001-001`, () => {
            cy.fixture('users.json').then((users) => {
                // Action: Launch Browser and go to Acolhe website
                cy.visit(`${Cypress.config('baseUrl')}`);
                // Response: Browser is launched and Acolhe website is launched
                cy.location('pathname',{timeout: 1000}).should('to.eq',`/inicio`);

                // Action: Click the Entrar button
                cy.contains('Entrar').click();
                // Response: The login page is displayed
                cy.location('pathname', {timeout: 1000}).should('to.eq',`/${LOGIN_PATH}`);

                // Action: Type {E_1} email on the email field
                cy.get('#email').type(users.E_1.username);
                // Response: Email is writen in field
                cy.get('#email').should('have.value',users.E_1.username);

                // Action: Type the user password on the password field
                cy.get('#password').type(users.E_1.password, {log:false});
                // Response: Input box shows "*" for each character input on the password field
                cy.get('#password').should('have.value',users.E_1.password,{log:false});

                // Action: Click on the Entrar button
                cy.contains('Entrar').click();
                // Response: Logged in page with user's view is shown
                cy.url().should('be.equal',`${Cypress.config('baseUrl')}/${DASHBOARD_PATH}`);

            });
        });
```

Figura 18. Script gerado automaticamente pela ferramenta.

Para que o script gerado possa ser executado corretamente, faz-se necessário configurar as *fixtures* em um arquivo `users.json` para os valores de usuário e também algumas constantes utilizadas no mapeamento, como `LOGIN_PATH` e `DASHBOARD_PATH`. Nas *fixtures* de usuários, `E_0` possui credenciais de um usuário do tipo Visualizador, que não possui permissões para cadastrar um novo usuário, por exemplo, já `E_1` possui credenciais de Gestor.

6. Conclusão

A ferramenta TaRGeT foi estendida desde a estrutura interna até a interface do usuário para permitir a geração de testes automáticos, solucionando a limitação anterior de gerar apenas testes manuais. O processo de geração proporciona flexibilidade no uso de dados através de um mapeamento que considera que passos do caso de uso possuem dados variados, e não apenas fixos. Além disso, o mapeamento pode ser desenvolvido de forma independente dos casos de uso, o que possibilita a colaboração entre pessoas com diferentes níveis de conhecimento técnico. A aplicação prática no sistema real demonstrou que a abordagem proposta viabiliza a automação em um ambiente real

Com base nos resultados obtidos, foi possível gerar casos de teste automáticos a partir de casos de uso. A ferramenta estendida demonstrou capacidade de gerar os scripts de forma adequada. A integração entre os casos de uso descritos em linguagem natural e a geração automática de scripts possibilita uma transição fluida entre a fase de requisitos e a fase de testes, garantindo uma validação mais eficaz das especificações.

Apesar dos avanços, algumas limitações foram identificadas, como a dependência de conhecimento no framework de automação utilizado para preenchimento do mapeamento de passos, no caso deste trabalho, Cypress. Uma investigação futura consiste em

estudar formas de minimizar a necessidade deste conhecimento possivelmente utilizando modelos de linguagem grandes.

Portanto, um ponto de destaque para estudos futuros seria a reutilização de código entre ações e respostas semelhantes, através de uma análise textual e de contexto. Além disso, a integração da ferramenta com outros frameworks de automação de testes, além do Cypress, pode ser uma linha promissora de investigação.

Também são trabalhos futuros explorar os ganhos de eficiência no uso da ferramenta em comparação com a escrita de scripts automáticos sem auxílio da ferramenta estendida.

Referências

Alves, K., Santos, E., L., M., Nogueira, S., Burégio, V., and Brito, K. (2024). Technology at the service of society: A support system for the reception of citizens in natural disaster situations. In *Anais Estendidos do XX Simpósio Brasileiro de Sistemas de Informação*, pages 249–252, Porto Alegre, RS, Brasil. SBC.

Arruda, F., Barros, F., and Sampaio, A. (2019). Automation and consistency analysis of test cases written in natural language: An industrial context. *Science of Computer Programming*, 189:102377.

Baziuk, W. (1995). Bnr/nortel: path to improve product quality, reliability and customer satisfaction. In *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95*, pages 256–262.

Cypress (2022). Bundled libraries.

Cypress (2024a). Introduction to cypress app.

Cypress (2024b). Why cypress?

Ferreira, F., Neves, L., Silva, M., and Borba, P. (2010). Target: a model based product line testing tool. *24th SBES - 17th Tools Session*.

Flanagan, D. (2012). *JavaScript: O Guia Definitivo*. Bookman Editora.

Framework, R. (2025). Robot framework.

Hasling, B., Goetz, H., and Beetz, K. (2008). Model based testing of system requirements using uml use case models. In *1st International Conference on Software Testing*, pages 367–376.

Hoare, C. A. R. et al. (1978). Communicating sequential processes. *Commun. ACM*, 21(8):666–677.

Liu, S. and Nakajima, S. (2022). Automatic test case and test oracle generation based on functional scenarios in formal specifications for conformance testing. *IEEE Transactions on Software Engineering*, 48(2):691–712.

Myers, G. J., Sandler, C., and Badgett, T. (2011). *The Art of Software Testing*. John Wiley & Sons, New Jersey, Hoboken.

Nebut, C., Fleurey, F., Le Traon, Y., and Jézéquel, J. (2006). Automatic test generation: a use case driven approach. *Software Engineering, IEEE Transactions on*, 32:140– 155.

Nogueira, S., Araujo, H., Araujo, R., Iyoda, J., and Sampaio, A. (2019). Test case generation, selection and coverage from natural language. *Science of Computer Programming*, 181:84–110.

Sarmiento, E., Leite, J., and Almentero, E. (2014). C&l: Generating model based test cases from natural language requirements descriptions. In *2014 IEEE 1st International Workshop on Requirements Engineering and Testing (RET)*, pages 32–38.

Selenium (2025). The selenium browser automation project.

Sommerville, I. (2011). *Engenharia de software*. Pearson Prentice Hall.

Utting, M. and Legeard, B. (2010). *Practical Model-Based Testing: A Tools Approach*. Elsevier.

Wang, C., Pastore, F., Goknil, A., Briand, L., and Iqbal, Z. (2015). Automatic generation of system test cases from use case specifications. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, page 385–396, New York, NY, USA. Association for Computing Machinery.

Zafar, M. N., Afzal, W., Enoiu, E., Stratis, A., Arrieta, A., and Sagardui, G. (2021). Model-based testing in practice: An industrial case study using graphwalker. In *Proceedings of the 14th Innovations in Software Engineering Conference (Formerly Known as India Software Engineering Conference)*, ISEC '21, New York, NY, USA. Association for Computing Machinery.