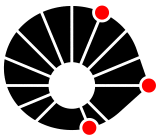
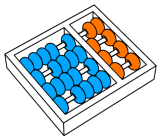




Avaliação e melhoria de testes baseados em modelos de estados para APIs REST

J. V. Silva *E. Martins*

Relatório Técnico - IC-PFG-24-35
Projeto Final de Graduação
2024 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Avaliação e melhoria de testes baseados em modelos de estados para APIs REST

J. V. Silva E. Martins

Resumo

Uma forma de determinar a qualidade de conjuntos de teste é pela avaliação da cobertura. Na geração automática de casos de testes baseados em modelos de estado, vários critérios de cobertura do modelo são implementados pelas ferramentas, tais como cobertura de estados ou cobertura de transições. Será que conjuntos de testes que satisfazem aos critérios de cobertura do modelo de estados garantem uma boa cobertura da API REST em teste? Uma especificação de uma API contém requisições, respostas, recursos (ou endpoints), entre outras, que precisam ser exercitados durante os testes. Este trabalho procura avaliar quão bem os conjuntos de testes gerados de acordo com os diferentes critérios baseados em modelos de estado são capazes de cobrir adequadamente os elementos de uma API. Com a utilização de um conjunto de métricas de cobertura para APIs, o objetivo é ter uma forma de auxiliar as equipes de testes a determinar aonde introduzir melhorias nos conjuntos de testes gerados.

Palavras-chave: API REST; testes baseados em modelos de estado; testes de dados; critérios de cobertura de API

1 Introdução

Uma API (Application Programming Interface), em português, Interface de Programação de Aplicação, permite que dois componentes se comuniquem na forma de requisições (requests) e respostas (responses). Para acesso a aplicações na Web, é comum o uso de APIs que usam o estilo arquitetural REST (REpresentational State Transfer), em português, Transferência Representacional de Estado, em que clientes e servidores interagem usando protocolo HTTP [5]. Os elementos básicos da arquitetura REST são os recursos, que representam as informações de interesse as quais se tem acesso pela interface. Recursos têm endereço, e a API permite realizar operações para criar, ler, atualizar ou apagar recursos, entre outras.

Em uma arquitetura orientada a serviços, mais conhecidas pela sigla em inglês, SOA, uma aplicação usa componentes de software heterogêneos, chamados de serviços. Cada serviço é responsável por uma função de negócio, exposta por sua interface, sem que se conheça sua estrutura interna. Uma arquitetura de microsserviços, por outro lado, divide as aplicações em partes menores e independentes, que podem ser alocados e desalocados dinamicamente. As APIs REST são usadas neste tipo de arquitetura para permitir a comunicação entre serviços (ou microsserviços) em aplicações Web.

Testar APIs REST é importante para garantir que, os serviços ou microsserviços são capazes de realizar as funções desejadas e de interagir adequadamente. Dentre os desafios dos testes de aplicações orientadas a serviços podemos citar: (i) geralmente o código fonte não está disponível, e (ii) os serviço, e principalmente os microsserviços, podem ser conhecidos e incorporados dinamicamente à aplicação.

Neste trabalho propomos o uso de uma técnica de testes de caixa preta, i.e., que não necessita do código fonte do serviço, e se baseia unicamente na descrição de sua interface. Mais especificamente, utiliza-se neste trabalho a técnica de testes baseados em modelos de estado para representar o comportamento de um serviço REST. Trabalhos anteriores propuseram o uso da UML, mais especificamente, diagrama de classes e modelo de estados de protocolo, para representar o comportamento de serviços [Porres e Rauf 2011; Rauf e Porres, 2011; Rauf et al., 2010]. Ulteriormente esta modelagem foi utilizada para fins de teste REST [Pinheiro et al 2013; relatório técnico do semestre passado]. A técnica de testes baseados em modelos tem a vantagem de permitir a geração automática de casos de teste de acordo com diferentes critérios. No caso de modelos de estado, as ferramentas geralmente garantem critérios como cobertura de estados ou cobertura de transições, como é o caso do trabalho de Pinheiro et al. citado anteriormente.

Neste trabalho utilizamos Testes Baseados em Modelos de Estado como forma de automatizar a geração dos casos de testes. O objetivo é não somente a cobertura do modelo de estados, mas também determinar o quanto a API foi coberta. Uma especificação de uma API contém requisições, respostas, recursos (ou endpoints), entre outras, que precisam ser exercitados durante os testes. Para isso, nos baseamos no trabalho de Martin-Lopez et al. (2019) que define diferentes critérios de cobertura para os testes de uma API REST. Com isso, pode-se ajudar aos testadores onde concentrar esforços para a melhoria dos testes da API.

Este texto está organizado da seguinte forma: a Seção 2 apresenta alguns fundamentos

sobre APIs e testes baseados em modelos, que são necessários para o entendimento deste trabalho. A Seção 3 apresenta os passos do método de testes utilizados. A Seção 4 apresenta as ferramentas utilizadas para a realização do método. A Seção 5 apresenta o estudo de caso, e os passos para a realização dos testes. A Seção 6 contém uma discussão sobre os resultados obtidos. Por fim, a Seção 7 apresenta as conclusões, seguida de uma lista das referências bibliográficas utilizadas para a escrita deste trabalho.

2 Fundamentação

Esta seção aborda os principais temas utilizados neste trabalho, assim como os trabalhos que deram origem ao método empregado no projeto.

2.1 Testes baseados em modelos de estado

Modelos são cada vez mais utilizados para representar requisitos. Com a UML (Unified Modelling Language) se tornando um padrão de fato para a modelagem de sistemas, o uso de modelos de especificação vem se tornando mais comum em testes.

O teste baseado em modelos (TBM) consiste em usar ou derivar modelos do comportamento esperado do software para produzir casos de teste que podem revelar discrepâncias entre o comportamento real do programa e o especificado pelo modelo [12][cap 14]. O principal objetivo de TBM é automatizar a geração de testes. Uma grande parte das ferramentas de apoio a TBM utilizam como base um modelo de estados.

Modelos de estados servem para representar sequências de interações entre um sistema e seu ambiente. São baseados em um modelo matemático denominado autômato finito [18], aplicados desde os anos 1950 para modelar circuitos eletrônicos e a partir dos anos 1960 para modelar o comportamento de sistemas reativos.

Para a geração de casos de teste, um modelo de estados pode ser visto como um **grafo direcionado**, em que os nós representam os estados e as arestas, as transições. **Estados** podem representar estágios de um processo, condições sobre os dados, consequências da ocorrência de eventos, configuração de hardware ou status de dispositivos [6][cap 4.2]. Um **evento** de entrada causa transição de um estado origem a um estado destino, e representam sinais de hardware, mudança nas condições dos dados, passagem de tempo, entre outras. Além dos eventos, as transições também podem conter **ações**, que são respostas produzidas pelo sistema em resposta a um evento.

Os casos de teste gerados a partir de modelos são sequências de passos que caracterizam um cenário de uso do sistema em teste. As sequências correspondem a caminhos no modelo. Antes, é importante introduzir alguns termos utilizados nesta seção; a terminologia é baseada em [18]. Um **caminho** é uma sequência de k estados adjacentes, i.e., estados interconectados por transições. Um caminho com k estados contém, portanto, (k-1) transições, e este número dá o **comprimento** do caminho. Quando os k estados do caminho são distintos, tem-se um **caminho simples**. Um caminho que começa e termina no mesmo estado é chamado de **circuito**. Os caminhos que interessam para os testes sempre começam no estado inicial.

Dado um modelo, como fazer para a obtenção dos casos de teste? Para isso, são definidos os critérios de teste (ou de adequabilidade dos testes). Um critério de adequabilidade **C** determina um conjunto finito de elementos do modelo que devem ser exercitados durante os testes [20]. A esse conjunto finito de elementos se dá o nome de requisitos de teste (**RTc**). Por exemplo, em um modelo de estado, esse conjunto pode ser formado por estados, indicando que o conjunto de testes adequado contém caminhos que visitem cada estado pelo menos uma vez.

Critérios de adequabilidade também são úteis para avaliar a qualidade dos testes. Esta qualidade pode ser medida através da análise de cobertura dos testes com relação ao critério **C**, que é normalmente dada em forma de porcentagem:

$$\frac{\text{(número de elementos de } \mathbf{RTc} \text{ exercitados)}}{\text{(número total de elementos de } \mathbf{RTc})}$$

A cobertura pode ser definida tanto antes, para indicar quantos casos de teste devem ser gerados, quanto depois, para obter a cobertura atingida pelos testes. Desta forma, uma resposta à pergunta "quando terminam os testes?" pode ser: quando o conjunto de testes atingir a cobertura desejada.

Nos testes baseados em modelos de estados, os critérios comumente usados são [1][cap 7.2]:

- **Cobertura de estados:** cada estado do modelo deve ser visitado pelo menos uma vez, i.e., RT contém todos os estados do modelo.
- **Cobertura de transições:** cada transição do modelo deve ser visitada pelo menos uma vez, ou seja, RT contém todas as transições do modelo.
- **Cobertura de pares de transições:** cada par de transições alcançáveis deve ser visitado pelo menos uma vez, ou seja, RT contém pares de transições alcançáveis do modelo.
- **Cobertura de caminhos primos (prime paths):** um caminho primo é um caminho simples que não está contido em nenhum outro caminho simples. O critério então requer que RT contenha todos os caminhos primos do modelo.
- **Cobertura de elementos específicos:** uma transição ou um estado específica/o deve ser alcançada/o, ou seja, RT contém as transições ou os estados de interesse.

O objetivo de TBM é permitir que a geração dos casos de teste seja automatizada. Diversos algoritmos foram propostos para a geração de casos de teste a partir de modelos de estado para atender a um ou mais dos critérios citados. Apesar de não ser esta a preocupação de quem usa uma ferramenta de geração automática de testes, é interessante conhecê-los, para determinar as vantagens e limitações de cada um; a seguir, alguns dos principais algoritmos [19][cap 11]; [4]:

- Passeio aleatório (Random Walk), em que, a partir de um estado corrente, uma próxima transição é escolhida aleatoriamente. Apesar de ser uma técnica simples de implementar e que permite exercitar cenários inesperados ou pouco comuns, é difícil garantir um nível adequado de cobertura de um critério.

- Técnicas baseadas em análises exaustivas do modelo, tais como técnicas de busca em grafos ou verificação de modelos (model checking), garantem cobertura de critérios, mas têm custos elevados em termos de tempo e memória no caso de modelos muito grandes.

- Técnicas baseadas em busca metaheurística, usadas quando as técnicas baseadas em busca exaustiva são limitadas devido ao tamanho ou complexidade do modelo.

Uma outra dimensão de TBM diz respeito à relação entre as fases de geração e execução dos testes. Nos testes fora de linha (offline), os casos de teste são gerados antes da execução. Já nos testes em linha (online) a geração e a execução dos testes ocorrem concomitantemente: os casos de teste são gerados e executados, e o resultado da execução orienta a geração dos próximos casos de teste.

2.2 Sobre APIs REST

Uma API (sigla em inglês para Interface de Programação de Aplicação) permite que dois componentes se comuniquem na forma de requisições (requests) e respostas (responses). Para acesso a aplicações na Web, é comum o uso de APIs que usam o estilo arquitetural REST, sigla em inglês para Transferência Representacional de Estado, em que clientes e servidores interagem usando protocolo HTTP (Protocolo de Transferência de Hipertexto) [5], que representam as informações de interesse as quais se tem acesso pela interface. Recursos têm endereço, e a API permite realizar operações para criar, ler, atualizar ou apagar recursos, entre outras. O endereçamento de recursos se dá via um URI (Identificador de Recurso Uniforme), que especifica o caminho para o recurso. Por exemplo, o acesso ao recurso produtos de uma loja virtual pode se dar pela URI https://umaloja.com.br/api/produtos. Já as operações para manipulação de um recurso são mapeadas para métodos HTTP, dentre os quais, os mais comumente utilizados são: POST (criação do recurso), GET (consulta a informações do recurso), PUT (atualização de um recurso) e DELETE (exclusão do recurso).

Além do método HTTP, as requisições também incluem opcionalmente um cabeçalho e um corpo. O cabeçalho contém informações adicionais sobre a requisição, como por exemplo, credenciais de autenticação ou formato de respostas. O corpo contém informações utilizadas por métodos como POST ou PUT.

As requisições também podem incluir parâmetros, que fornecem ao servidor mais detalhes sobre a requisição, como por exemplo, parâmetros de caminho, que especificam detalhes do caminho. A Figura 1 ilustra uma requisição com parâmetros.

As requisições são processadas pelo servidor que retorna uma resposta que inclui: o código de status HTTP (ex.: 200, sucesso; 404, recurso não encontrado), um cabeçalho e um corpo, contendo as informações solicitadas sobre o recurso.

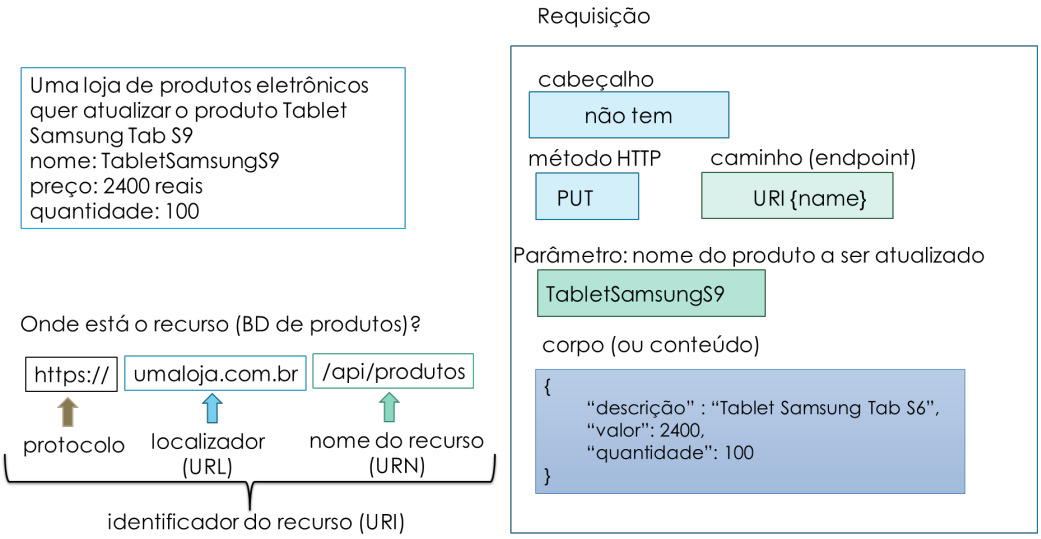
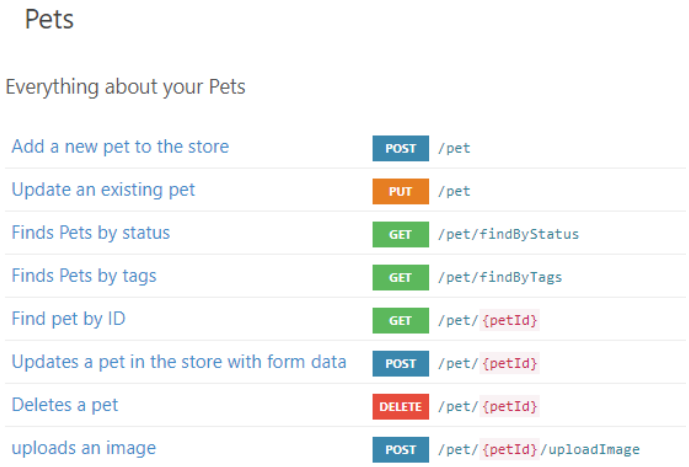


Figura 1: Exemplo de uma requisição com parâmetros.

As APIs são expostas pelos desenvolvedores para que outras aplicações (clientes) possam se comunicar com suas aplicações programaticamente. Para facilitar essa comunicação, as APIs REST podem ser documentadas segundo um padrão, como o OAS (Open API Specification). Esta especificação é contida em um arquivo com extensão ".yaml", também conhecido como arquivo Swagger. A Figura 2 mostra trechos de um arquivo Swagger para uma pet Store:



((a)) Arquivo Swagger.

```yaml
openapi: 3.0.2
servers:
  - url: /v3
info:
  description: |-
    This is a sample Pet Store Server based on the OpenAPI 3.0 specification.  You can find out more about
    Swagger at [http://swagger.io](http://swagger.io). In the third iteration of the pet store, we've switched to the design first appr
    You can now help us improve the API whether it's by making changes to the definition itself or to the code.
    That way, with time, we can improve the API in general, and expose some of the new features in OAS3.

    Some useful links:
    - [The Pet Store repository](https://github.com/swagger-api/swagger-petstore)
    - [The source API definition for the Pet Store](https://github.com/swagger-api/swagger-petstore/blob/master/src/main/resources/opena
  version: 1.0.20-SNAPSHOT
  title: Swagger Petstore - OpenAPI 3.0
  termsOfService: 'http://swagger.io/terms/'
  contact:
    email: apiteam@swagger.io
  license:
    name: Apache 2.0
    url: 'http://www.apache.org/licenses/LICENSE-2.0.html'
tags:
  - name: pet
    description: Everything about your Pets
    externalDocs:
      description: Find out more
      url: 'http://swagger.io'
  - name: store
    description: Access to Petstore orders
    externalDocs:
      description: Find out more about our store
      url: 'http://swagger.io'
  - name: user
    description: Operations about user
paths:
  /pet:
    post:
      tags:
        - pet
      summary: Add a new pet to the store
      description: Add a new pet to the store
      operationId: addPet
      responses:
        '200':
```

((b)) Arquivo ".yaml". Extraído do site[1]

Figura 2: Trechos da especificação OAS para a petStore.

2.3 Modelagem de APIs REST usando UML

Dado que os elementos básicos da arquitetura REST são os recursos, que são fornecidos pelos serviços, pode-se considerar este tipo de arquitetura como orientada a recursos, ou ROA [15], da sua sigla em inglês. Recursos têm atributos ou campos, representados em formato XML ou JSON. Os atributos de um recurso podem ser acedidos por métodos HTTP e são endereçáveis por uma URI (ver seção 2.2).

Nesse contexto, Porres e Rauf (2011) fizeram uma analogia entre recursos e objetos do paradigma de orientação a objetos (OO). Como os autores ressaltam, há diferenças importantes entre os dois paradigmas; por exemplo, na ROA, o objetivo é expor informações relevantes para os clientes, e não as ocultar, como no paradigma OO. Apesar das diferenças, os autores propõem o uso do diagrama de classes da UML para representar os recursos, sua estrutura e as conexões entre eles. Assim, esse diagrama constitui o modelo conceitual dos recursos.

Para a representação do comportamento foi proposto o uso de modelos de estado, que descrevem a sequência na qual os métodos podem ser invocados, as condições nas quais eles podem ser invocados, bem como os resultados esperados. Mais especificamente, é utilizado o modelo de protocolo da UML, pois este é apropriado para a representação de interfaces [10]. O modelo especifica quais operações da interface podem ser chamadas, sob quais condições, indicando assim quais as sequências de chamadas permitidas. Os estados representam condições que são válidas para a interface em um dado momento, e são representados por invariantes de estados. As transições especificam as operações que podem ser chamadas, as condições necessárias para que a operação possa ser chamada (pré-condições) e as condições esperadas após a realização da operação (pós-condições).

Os autores propuseram modificações ao modelo de forma a atender às restrições da arquitetura REST. Para cumprir a restrição de interface uniforme, as transições se restringem aos métodos HTTP disponíveis para o recurso. No que diz respeito à restrição de ausência de estado (statelessness), são usadas invariantes definidas sobre os recursos; um estado está ativo quando a invariante é satisfeita. Para que uma transição seja disparada em resposta a uma requisição, é necessário determinar se seu estado origem está ativo, ou seja, sua invariante é verdadeira. Caso haja uma pré-condição associada, é necessário que esta também seja satisfeita. As invariantes de estado também servem como pós-condição da transição, ou seja, após a realização da requisição, a invariante do estado destino deve ser satisfeita.

2.4 Critérios de cobertura de testes de APIs REST

Ao testar APIs REST, é importante que se determine quão bem os casos de teste são capazes de exercitar as entradas (requisições, parâmetros) e saídas (respostas, códigos de status). Com este objetivo, Martin-Lopez et al. (2019) propuseram dez critérios de cobertura, baseados na OAS, e atribuem oito níveis para classificar o conjunto de testes com base na cobertura obtida. Os critérios de teste são apresentados na Tabela 1.

[1]Fonte: https://github.com/swagger-api/swagger-petstore/blob/master/src/main/resources/openapi.yaml

Tabela 1: Critérios de cobertura para uma API.

Tipos de critérios	**Critérios**	**Descrição**
Requisições	Cobertura de caminhos	Razão entre o número de caminhos (*endpoints*) testados e o número de caminhos especificados (OAS).
	Cobertura de operações	Razão entre o número de operações testadas pelo número total de operações.
	Cobertura de parâmetros	Razão entre o número de parâmetros usados pelos casos de teste e o número total de parâmetros.
	Cobertura de valores de parâmetros	Razão entre o número de valores de parâmetros exercitados pelo número total de valores de parâmetros. Tal métrica é usada somente para parâmetros com número finito de valores, como por exemplo, *boolean* e *enum*.
	Cobertura de tipo de conteúdo da requisição	Razão entre o número de tipos de conteúdo (JSON, XML, outros) testados pelo número de tipos de conteúdo especificadas. Esta métrica só pode ser calculada quando a requisição tem um corpo (ex.: operações PUT, POST).
Respostas	Cobertura de tipos de código de status	Razão entre o número de classes de código de status (2xx, 4xx, 5xx) obtidos pelo total de classes especificadas.
	Cobertura de códigos de status	Razão entre o número de códigos de status obtidos pelo número total de códigos documentados na especificação.
	Cobertura de tipo de conteúdo da resposta	Razão entre o número de tipos de conteúdo (JSON, XML, outros) testados pelo número de tipos de conteúdo especificadas. Esta métrica só pode ser calculada quando a resposta tem um corpo.
	Cobertura de propriedades do conteúdo da resposta	Razão entre o número de propriedades exercitadas pelos casos de teste e o número total de todas as propriedades de todos os objetos que podem ser obtidas em respostas.

Os autores propõem ainda o critério de cobertura de fluxos de operação, que mede a proporção do número de sequências de operação cobertas. Não inserimos este critério na tabela, pois segundo os autores, não há consenso na literatura quanto ao número total de fluxos de operações. Por isso é assumida uma versão simplificada em que devem ser testados os fluxos de operação básicos, ligando a criação de um recurso com sua leitura, atualização e exclusão.

Com base nos critérios de cobertura estabelecidos, Martin-Lopez et al. propuseram também oito níveis de cobertura de teste ou TCL (Test Coverage Levels). O objetivo é classificar os conjuntos de teste gerados de acordo com o tipo de cobertura atingido. Os níveis são incrementais, i.e., para atingir a um determinado nível, é necessário atender aos níveis mais baixos. Os níveis definidos foram:

- **TCL0:** nível básico, em que o conjunto de testes não atende a nenhum critério de cobertura estabelecido.
- **TCL1:** o conjunto de testes deve atingir 100% de cobertura de caminhos.
- **TCL2:** o conjunto de testes deve atingir 100% de cobertura de operações.
- **TCL3:** o conjunto de testes deve atingir 100% de cobertura do tipo de conteúdo de requisições e respostas.
- **TCL4:** o conjunto de testes deve atingir 100% de cobertura de parâmetros e classes de códigos de status.
- **TCL6:** o conjunto de testes deve atingir 100% de cobertura de propriedades do conteúdo das respostas.
- **TCL7:** o conjunto de testes deve atingir 100% de cobertura de fluxos de operações.

3 O método de testes proposto

A Figura 3 mostra os passos do método de teste utilizado, bem como as ferramentas utilizadas.

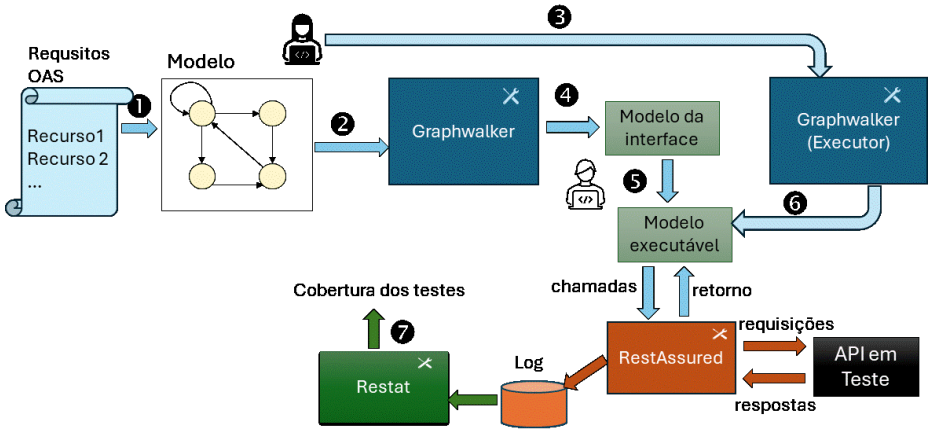


Figura 3: O método de testes utilizado.

Os passos realizados são brevemente descritos a seguir:

1. **Modelagem:** a partir da especificação, seja ela puramente textual ou no formato OAS, são obtidos os modelos de estado dos diferentes recursos. O método é incremental, o modelo representa cada recurso isoladamente.

2. **Validação do modelo:** é feita uma animação do modelo para que o testador possa determinar se é possível percorrê-lo para que se possa gerar os testes de forma automatizada.

3. **Definição dos critérios:** nesta etapa o testador define quais dos critérios de teste baseados em modelos de estado serão utilizados. Dentre os critérios apresentados em §2.1, a ferramenta utilizada, Graphwalker, permite a cobertura de vértices e cobertura de arestas.

4. **Geração das interfaces de testes:** nesta fase o testador define a interface de testes, constituída pelos estados e transições do modelo, que são chamados durante a execução na interação com a API em teste. A ferramenta utilizada dá apoio a este passo.

5. **Implementação do modelo:** nesta fase é criado o script de testes (casos de teste executáveis) para permitir a execução automática dos testes. Além da lógica de testes, o código de testes deve conter código de suporte. Por exemplo, para os testes de uma API, os seguintes passos são realizados: (i) carregar a URL da página com a operação a testar; (ii) preparar a requisição, identificando eventuais parâmetros e dados do corpo da requisição; (iii) enviar o comando HTTP que é parte da lógica do teste, utilizando bibliotecas existentes; (iv) capturar o código HTTP retorno; caso haja resposta, interpretar o corpo da resposta; (iv) comparar os resultados obtidos com os resultados esperados. Dado que o modelo de estados não deve mostrar todos estes detalhes, pois ele representa a lógica do negócio, é interessante garantir a separação de interesses, o que é conseguido com o uso da arquitetura em camadas para os casos de teste [7]; [17][cap 7.3]. Sendo assim, os casos de teste são divididos em três camadas, conforme recomendado por J. Smart (2015): camada de negócios, representada pelo modelo de estados executável, em que as transições e estados representam chamadas e análise de resultados, respectivamente, na linguagem da lógica de negócio. A camada de fluxo de negócios detalha os passos (i) a (iv) citados anteriormente e a camada técnica contém a interação direta com a API em teste. Estas duas últimas camadas são implementadas pela biblioteca Rest Assured [2].

6. **Geração e execução dos testes:** na execução online, a geração ocorre durante a execução dos testes, por isso é necessário um executor dos testes que produza os casos de teste em linha, conforme o critério (baseado em modelos de estado) escolhido pelo testador.

7. **Avaliação dos testes:** após a execução dos testes, avalia-se a cobertura da API alcançada pelos casos de teste aplicados. O uso da ferramenta Restats tem esse

[2]https://rest-assured.io

objetivo. As métricas devem ajudar ao testador onde focar maior esforço de testes, caso a API não tenha sido coberta de acordo com o nível de teste almejado pelos testadores (ver §2.4).

4 Ferramentas utilizadas

4.1 GraphWalker

A GraphWalker[3] é uma ferramenta para a realização de testes baseados em modelos [21]. Com ela, é possível executar modelagens em torno de estados e transições entre esses estados através de grafos direcionados. Existem alguns conceitos importantes para entender o funcionamento da modelagem com a GraphWalker e são eles:

- **Arestas:** representam uma ação(ou transição). Podem ser representados por exemplo como uma requisição à API ou um clique em um botão na tela.
- **Vértices:** representam uma verificação(ou asserção). Aqui é feita a validação para verificar se a resposta da API veio com o valor esperado ou não.
- **Modelo:** é o grafo representado pelo conjunto de vértices e arestas.
- **Algoritmo gerador:** é um algoritmo que decide como percorrer um modelo. Pode ser por exemplo um algoritmo que percorre de forma aleatória ou até mesmo um caminho já pré-definido pelo usuário.
- **Condições de parada:** define o critério de teste baseado em modelos de estado. Por exemplo, é possível definir se a cobertura deve incluir todos os vértices ou todas as arestas.

Com o modelo, a GraphWalker irá gerar um caminho passando pelos vértices e arestas respeitando o algoritmo gerador e a condição de parada. Para influenciar a forma como o modelo é percorrido e o seu comportamento, também é possível inserir **ações** e **guardas**. Ação é uma maneira de definir variáveis no modelo e guarda é uma forma de bloquear a aresta de ser percorrida caso uma condição não seja satisfeita.

Toda essa modelagem é feita utilizando a GraphWalker Studio que é uma ferramenta que facilita a criação e edição dos modelos de forma visual permitindo também, a execução do modelo para verificar o seu comportamento ao passar pelos vértices e arestas. A Figura 4 mostra o exemplo de uma modelagem dentro da GraphWalker Studio.

A GraphWalker também possui os conceitos de testes **offline** e **online**. Nos testes offline, a sequência de teste uma vez criada, pode ser executada depois quantas vezes forem necessárias utilizando o JUnit por exemplo. Essa sequência gerada pode ser muito útil para provar que o modelo, com o gerador de caminhos e a condição de parada, funciona. Já nos testes online, o modelo é percorrido para a geração dos testes juntamente com a execução da aplicação, sendo assim, os testes se tornam mais reais e dinâmicos. A GraphWalker por padrão gera 3 tipos diferentes de testes para o formato online:

[3] https://graphwalker.github.io

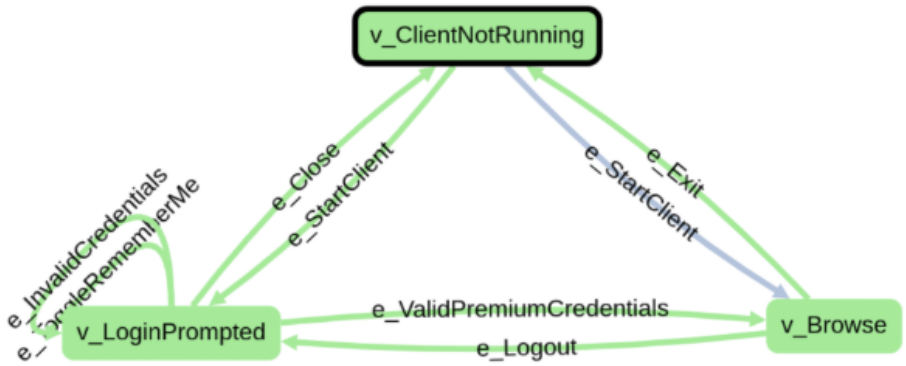


Figura 4: Exemplo de modelagem na GraphWalker Studio.

- **Teste de Fumaça (Smoke Test):** faz uma validação básica do modelo. É necessário informar um vértice inicial e um final para que o modelo seja percorrido através de um algoritmo de menor caminho.
- **Testes Funcionais (Functional Tests):** faz a cobertura total de arestas utilizando um algoritmo de passeio aleatório.
- **Testes de Estabilidade (Stability Tests):** permite que o modelo seja percorrido por um tempo determinado podendo ser de segundos até mesmo dias. O algoritmo utilizado é o de passeio aleatório e o teste ajuda a verificar a robustez do sistema em cenários de sobrecarga.

4.2 RestAssured

Rest Assured é uma biblioteca feita para facilitar o teste e validação de serviços REST utilizando Java [22]. Ela foi feita com o foco na usabilidade e eficiência, sendo assim, torna o trabalho do desenvolvedor uma tarefa mais intuitiva e produtiva.

A estrutura da Rest Assured segue a linguagem Gherkin, definida no paradigma do Desenvolvimento Guiado por Comportamento (ou BDD - Behavior Driven Development [17]), portanto, sua sintaxe é composta pelos comandos *given*, *when* e *then*.

- **Given:** aqui são indicadas as informações que a API precisa, como por exemplo, o token de autenticação ou os parâmetros da requisição.
- **When:** aqui é feita a requisição à API com o método a ser testado.
- **Then:** com o retorno da API, é extraída a resposta para verificar se o resultado veio como o esperado ou se aconteceu algum erro.

A Figura 5 mostra um exemplo de como fica a estrutura do código utilizando Rest Assured.

```java
import static io.restassured.RestAssured.*;
import static org.hamcrest.Matchers.*;

public class ApiTest {
    @Test
    public void testStatusCode() {
        given().
        when().
            get("/endpoint").
        then().
            statusCode(200);
    }
}
```

Figura 5: Exemplo de código Java da Rest Assured.

4.3 Restats

A Restats[4] é uma ferramenta feita em Python que permite a análise de cobertura de testes com API REST, conforme indicado na Seção 2.4. Ela calcula métricas de cobertura, estatísticas e o nível de cobertura (TCL) para suítes de teste. Essas métricas são calculadas da perspectiva da interface, considerando a completude da suíte de teste com relação aos elementos definidos na especificação OAS. Para o cálculo das métricas, são necessários alguns arquivos como entrada:

- A especificação OpenAPI da API REST a ser testada.
- Os arquivos com o conteúdo das requisições e respostas HTTP durante o teste.

A Figura 6 mostra o exemplo de como são os arquivos com as requisições e a Figura 7 o exemplo de respostas das requisições HTTP já formatadas no formato que a Restats espera.

1-request.txt

```
GET /petclinic/api/vets HTTP/1.1
Accept: */*
Content-Type: application/json
HOST: localhost:9966

```

Figura 6: Exemplo arquivo com a requisição HTTP.

[4]https://github.com/SeUniVr/restats/tree/main

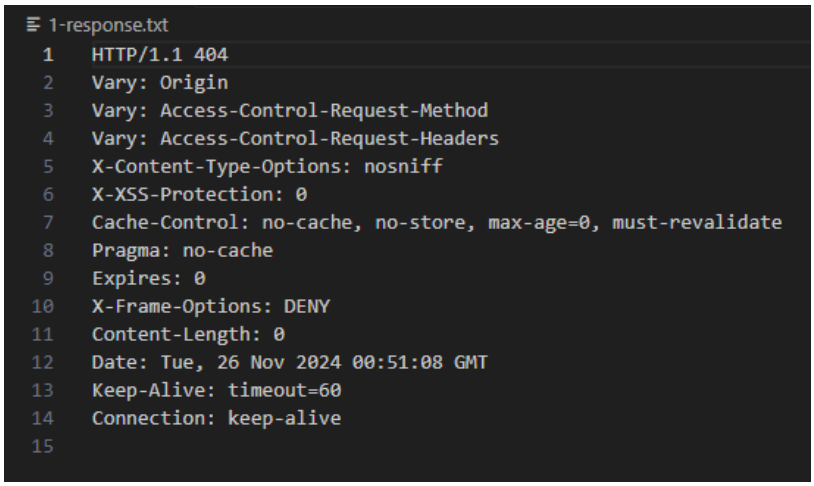


Figura 7: Exemplo arquivo com a resposta da requisição HTTP.

Dentro da pasta da Restats, é preciso configurar o arquivo *config.json* com a estrutura necessária para que os arquivos sejam lidos e gravados no local correto. Os elementos da configuração são:

- **modules:** indica que módulo a Restats deve executar. É possível executar o **dataCollection** onde a Restats vai ler os arquivos de log e gravar os dados no banco de dados ou **statistics**, onde ela vai ler as informações do banco de dados e gerar os relatórios. A opção **all** executa os dois módulos.
- **specification:** o caminho do arquivo OpenAPI com a especificação.
- **dumpsDir:** o caminho da pasta com os arquivos de log.
- **reportsDir:** o caminho da pasta com os relatórios gerados.
- **dpPath:** o caminho do arquivo de banco de dados SQLite[5] que a Restats gera com as informações.

Após executar a ferramenta, os relatórios são gerados dentro da pasta *reportsDir* conforme indicado na Figura 8.

5 Aplicação

Para exemplificar tudo que foi descrito até agora, optamos por estudar a API da Pet Clinic[6]. Essa API é uma aplicação de referência usada frequentemente para demonstrar boas

[5]https://www.sqlite.org
[6]https://github.com/spring-petclinic/spring-petclinic-rest

Figura 8: Pasta com os relatórios gerados pela Restats.

práticas e conceitos no desenvolvimento de aplicações baseadas em arquitetura REST, principalmente no ecossistema Spring Framework. Para isso, vamos iniciar construindo o modelo dentro da GraphWalker Studio para conseguir depois fazer a geração dos testes.

Nosso objetivo aqui, é conseguir comparar a cobertura dos testes usando a Restats. Primeiro vamos criar um modelo onde não será incluída a cobertura de cenários com erros e depois vamos comparar os resultados com outro modelo onde os estados com erros serão incluídos.

5.1 Teste de cenário normal

5.1.1 Modelagem do recurso

Dentre os recursos disponíveis na documentação da Pet Clinic, optamos por modelar e testar o recurso de veterinários(Vet), pois ele apresenta todos os métodos HTTP que queremos testar. A Figura 9 mostra a especificação OpenAPI com todos os métodos que um veterinário tem disponível.

5.1.2 Validação do modelo na GraphWalker

Dentro da GraphWalker Studio, criamos um modelo conforme mostrado na Figura 10. Esse modelo cobre todos os estados(vértices) que um veterinário pode ter e suas arestas representam as chamadas à API com os métodos disponíveis. Vale ressaltar que, diferentemente do trabalho de Porres e Rauf, apresentado na Seção 2.3, não utilizamos os métodos HTTP diretamente no modelo de estados. Em conformidade com a arquitetura em camadas utilizado nos testes (Seção 3), o modelo representa a camada de negócios, e por esta razão, escolhemos para os eventos de entrada os nomes mais abstratos. Assim, em vez de usar como entrada POST(/api/vets), utilizamos e_create_vet, por exemplo.

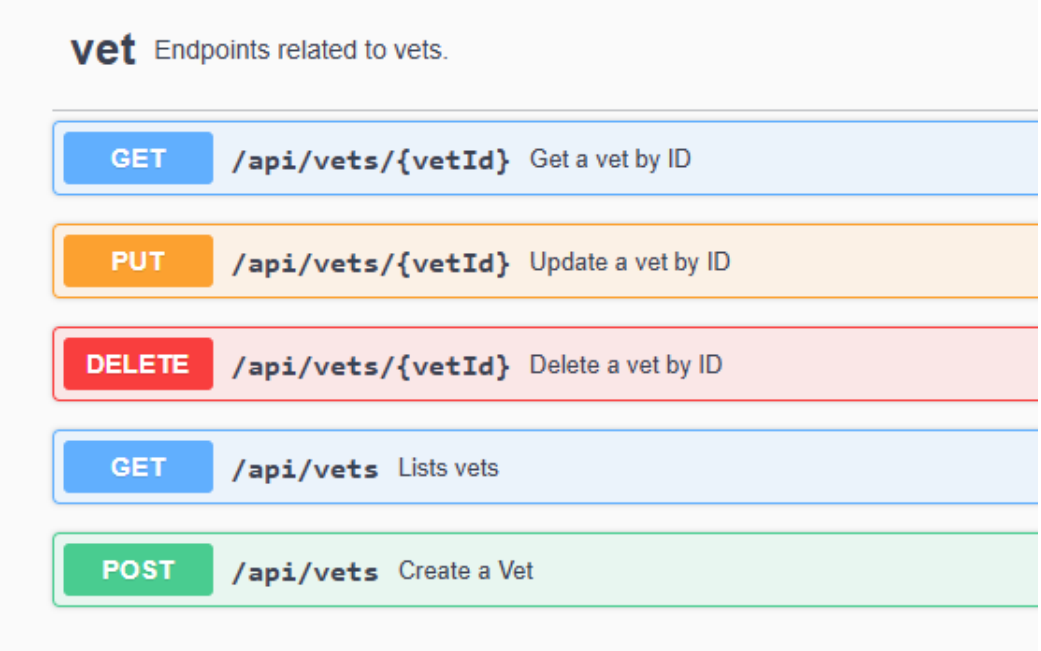


Figura 9: Descrição do recurso Vet na especificação OpenAPI.

5.1.3 Criação da interface de testes

Com o modelo criado dentro da GraphWalker Studio, é possível salvar um arquivo no formato JSON que será utilizado na geração da interface de testes. Para criar o projeto Java, a documentação da GraphWalker descreve como fazer o processo utilizando o gerenciador de dependências Maven. Além disso, também incluímos no projeto a Rest Assured e o JUnit para facilitar no processo de testes da API.

Com o projeto criado, foi necessário incluir o arquivo JSON com o modelo dentro da pasta *resources* para que a interface pudesse ser gerada de forma automática. A Figura 11 mostra como ficou a interface no final do processo.

5.1.4 Implementação da interface de testes

Para implementar a interface gerada, foi necessário criar uma classe auxiliar para concentrar os códigos dos vértices e arestas. A vantagem de fazer isso, foi que essa mesma classe pode ser utilizada tanto para os testes online como para os offline diminuindo assim, a duplicação de código. Nesta etapa, utilizamos a Rest Assured no código das arestas para fazer as requisições a API. Já nos vértices, utilizamos o JUnit para fazer as assertivas e verificar se o resultado vindo da API foi o esperado. A Figura 12 mostra o código do vértice e aresta para a criação de um veterinário. No código da aresta *e_create_vet* é feita a inserção do veterinário utilizando a Rest Assured e no código do vértice *v_vet_created* é feita a assertiva usando JUnit para verificar se o resultado está realmente correto. Já a Figura 13 mostra o código da Rest Assured para fazer a requisição à API e realizar a criação do veterinário no

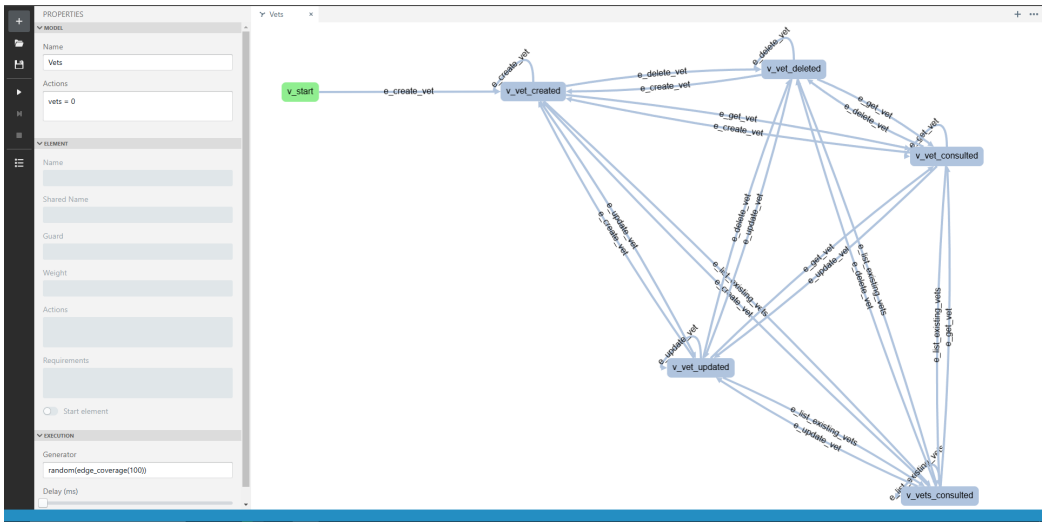


Figura 10: Modelo na GraphWalker Studio.

sistema.

5.1.5 Geração de testes em modo offline

Para a realização dos testes offline, optamos por utilizar um algoritmo que percorre o grafo de forma aleatória com cobertura de arestas de 100% *(random(edge_coverage(100)))*. A Figura 14 mostra um trecho de como ficou o código com a sequência de vértices e arestas do modelo. Como os testes são offline, eles foram gerados previamente e depois inseridos no código de forma manual.

5.1.6 Geração de testes em modo online

Nos testes online, optamos por executar 4 tipos diferentes de testes:

- **Teste de Fumaça (Smoke Test):** ao indicar um vértice inicial e um final, o modelo é percorrido através do caminho mínimo entre esses 2 vértices. Para o nosso exemplo, configuramos o vértices inicial como *v_start* e o final como *v_vet_deleted*.
- **Teste Funcional de Aresta (Functional Edge Test):** nesse teste, o modelo é percorrido de forma aleatória até conseguir cobertura de 100% das arestas.
- **Teste Funcional de Vértice (Functional Vertex Test):** nesse teste, o modelo é percorrido de forma aleatória até conseguir cobertura de 100% dos vértices.
- **Teste de Estabilidade (Stability Test):** no teste de estabilidade, é configurado um tempo determinado para que o modelo seja percorrido durante esse tempo e seja validado a robustez e a confiabilidade do sistema. Para o nosso exemplo, configuramos um tempo de **10 minutos**.

```
@Model(file = "com/petclinic/vets.json")  10 usages  5
public interface Vets {
    @Vertex()  34 usages  5 implementations
    void v_vet_created();

    @Edge()  21 usages  5 implementations
    void e_list_existing_vets();

    @Vertex()  39 usages  5 implementations
    void v_vet_updated();

    @Vertex()  27 usages  5 implementations
    void v_vet_deleted();

    @Edge()  39 usages  5 implementations
    void e_get_vet();

    @Vertex()  21 usages  5 implementations
    void v_vets_consulted();

    @Edge()  34 usages  5 implementations
    void e_create_vet();

    @Edge()  27 usages  5 implementations
    void e_delete_vet();

    @Vertex()  39 usages  5 implementations
    void v_vet_consulted();

    @Vertex()  1 usage  5 implementations
    void v_start();

    @Edge()  39 usages  5 implementations
    void e_update_vet();
}
```

Figura 11: Interface gerada a partir do modelo.

5.2 Testes de cenários de erro

Após testar os cenários normais, é necessário testar também os cenários de erro, para determinar se a API retorna os códigos de status HTTP especificados, como 4xx(erro do cliente) ou 5xx (erro no servidor). Também deve-se verificar se eventuais respostas com mensagens de erro são emitidas.

5.2.1 Validação do modelo na GraphWalker

Para representar o comportamento em situação de erro, o modelo normal foi aumentado, com mais 4 estados:

- **v_vet_delete_error:** estado para os erros envolvendo exclusão de um veterinário.
- **v_vet_update_error:** estado para os erros envolvendo atualização de um veterinário.
- **v_vet_consult_error:** estado para os erros envolvendo a consulta de um veterinário.
- **v_vet_create_error:** estado para os erros envolvendo criação de um veterinário.

```java
public void e_create_vet() {  4 usages  Joed Silva
    System.out.println("e_create_vet");

    Vet vet = this.vetFactory.nextElement();
    vet.setId(null);
    VetResponse createdVetResponse = vetClient.createVet(vet);
    this.vetsCounter += 1;
    this.lastVetId = Integer.parseInt(createdVetResponse.getVet().getId());

    this.vetFactory.setCurrentElement(createdVetResponse.getVet());
}

public void v_vet_created() {  4 usages  Joed Silva
    System.out.println("v_vet_created");

    Vet currentVet = this.vetFactory.getCurrentElement();
    VetResponse vetResponse = this.vetClient.getVet(currentVet.getId());

    assertEquals(vetResponse.getVet().getId(), currentVet.getId());
    assertEquals(vetResponse.getVet().getFirstName(), currentVet.getFirstName());
    assertEquals(vetResponse.getVet().getLastName(), currentVet.getLastName());
    assertEquals(vetResponse.getVet().getSpecialties(), currentVet.getSpecialties());
}
```

Figura 12: Exemplo de código de vértice e aresta.

Foram inseridas também 45 arestas (transições) que levam a estes estados de erro, gerando o modelo da Figura 15. Como se pode perceber, o modelo ficou mais complexo e sua visualização ficou mais difícil na Graphwalker, devido a limitações do editor de modelos.

5.2.2 Implementação da interface de testes

Uma vantagem dos testes baseados em modelos é que, no caso de atualização do modelo, o testador não precisa manter os casos de teste. Uma vez atualizado o modelo, basta regerar os casos de teste. Na Graphwalker, a atualização do modelo é simples. A interface é gerada automaticamente, e ao testador basta inserir o código dos novos vértices e arestas. A Figura 16 mostra os novos métodos que foram inseridos dentro da interface gerada.

5.2.3 Geração de testes em modo offline e online

Com a interface gerada automaticamente, foi necessário atualizar os códigos das classes dos testes offline e online para implementar os novos métodos. Um ponto importante a acrescentar é que, referente aos códigos das funções já existentes, não foi necessário fazer uma refatoração para adaptar o código para a nova versão do modelo. Isso ajudou muito, pois evitou um retrabalho que poderia levar muito tempo, e com isso, podemos focar apenas em desenvolver as novas funções. Os códigos implementados envolviam erros como por exemplo: veterinário não encontrado no sistema, informações inválidas enviadas ao servidor e erro interno no servidor. A Figura 17 mostra alguns exemplos de implementações de vértices e arestas relacionados a erros envolvendo a exclusão de um veterinário no sistema.

```java
public VetResponse createVet(Vet vet) {  4 usages  Joed Silva
    Gson gson = new Gson();
    String vetJson = gson.toJson(vet);

    Response response = given()
            .header( s: "Content-Type",  o: "application/json")
            .body(vetJson)
            .relaxedHTTPSValidation()
            .filter(new CustomRequestLoggingFilter())
            .when()
            .post( s: "/vets");

    int statusCode = response.statusCode();
    Vet createdVet = (statusCode == 201) ? response.as(Vet.class) : null;
    return new VetResponse(createdVet, statusCode);
}
```

Figura 13: Exemplo de código da Rest Assured.

```java
@Test  Joed Silva *
public void test1() {
    v_start();
    e_create_vet();
    v_vet_created();
    e_update_vet();
    v_vet_updated();
    e_update_vet();
    v_vet_updated();
    e_get_vet();
    v_vet_consulted();
    e_update_vet();
    v_vet_updated();
    e_list_existing_vets();
    v_vets_consulted();
    e_get_vet();
    v_vet_consulted();
    e_delete_vet();
```

Figura 14: Trecho do teste offline com a sequência de vértices e arestas.

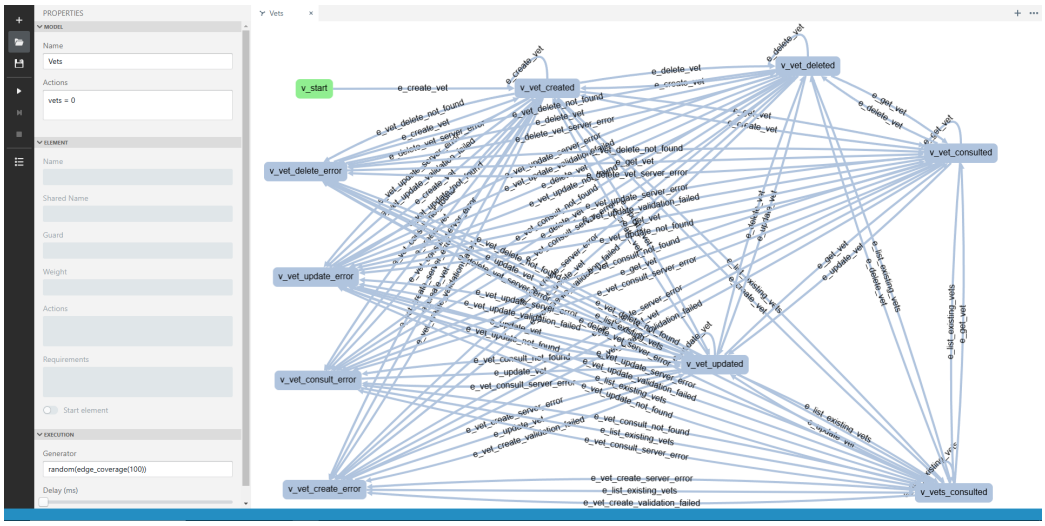


Figura 15: Modelo atualizado na GraphWalker Studio com os estados de erros.

```
@Edge()  13 usages  5 implementations
public void e_vet_create_validation_failed();

@Edge()  22 usages  5 implementations
public void e_vet_create_server_error();

@Vertex()  47 usages  5 implementations
public void v_vet_update_error();

@Vertex()  33 usages  5 implementations
public void v_vet_delete_error();

@Edge()  20 usages  5 implementations
public void e_vet_delete_not_found();

@Edge()  13 usages  5 implementations
public void e_delete_vet_server_error();

@Edge()  18 usages  5 implementations
public void e_vet_update_validation_failed();

@Edge()  14 usages  5 implementations
public void e_vet_update_server_error();

@Edge()  15 usages  5 implementations
public void e_vet_update_not_found();

@Vertex()  38 usages  5 implementations
public void v_vet_consult_error();

@Edge()  21 usages  5 implementations
public void e_vet_consult_server_error();

@Edge()  17 usages  5 implementations
public void e_vet_consult_not_found();
```

Figura 16: Interface atualizada com os novos vértices e arestas.

```java
public void v_vet_delete_error() {  4 usages  Joed Silva
    System.out.println("v_vet_delete_error");

    assertTrue(this.statusCodeErroList.contains(this.lastHttpStatusCode));
}

public void e_vet_delete_not_found() {  4 usages  Joed Silva
    System.out.println("e_vet_delete_not_found");

    int invalidVetId = this.lastVetId + 10;
    VetResponse vetResponse = this.vetClient.deleteVet(String.valueOf(invalidVetId));
    this.lastHttpStatusCode = vetResponse.getHttpStatusCode();
}

public void e_delete_vet_server_error() {  4 usages  Joed Silva
    System.out.println("e_delete_vet_server_error");

    VetResponse vetResponse = this.vetClient.deleteVet( id: null);
    this.lastHttpStatusCode = vetResponse.getHttpStatusCode();
}
```

Figura 17: Implementações dos vértices e arestas para os erros relacionados a remoção de um veterinário.

6 Resultados obtidos

Referente aos resultados, optamos por executar primeiramente todos os testes para o cenário normal (teste de fumaça, funcional de arestas, funcional de vértices e estabilidade) e depois executar os mesmo testes adicionando os cenários de erros.

6.1 Resultados por critério de cobertura

Para a análise da cobertura, a ferramenta utilizada, Restats, utiliza a especificação da API para determinar o que foi ou não coberto. Dado que só testamos o recurso veterinários (Vet), passamos para a ferramenta somente a especificação referente a este recurso. Com a Restats analisando apenas o veterinário, foi possível analisar a cobertura apenas dos elementos desta API. Ao analisar a especificação do recurso testado, a Restats obteve o número de itens documentados que devem ser exercitados, conforme descrito na Tabela 2:

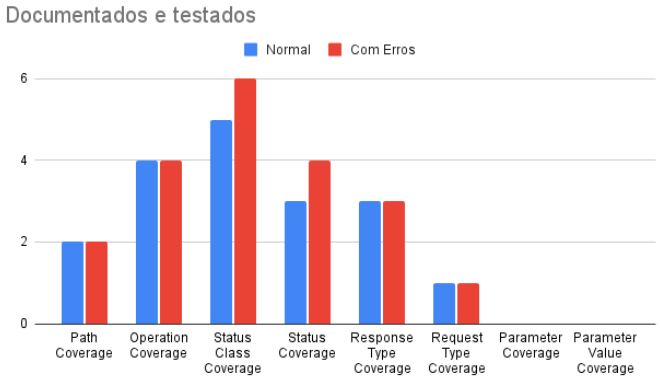
Métrica	**Total de itens**
Path Coverage	2
Operation Coverage	5
Status Class Coverage	14
Status Coverage	17
Response Type Coverage	6
Request Type Coverage	2
Parameter Coverage	0
Parameter Value Coverage	0

Tabela 2: Quantidade de itens para cada métrica analisada na Restats.

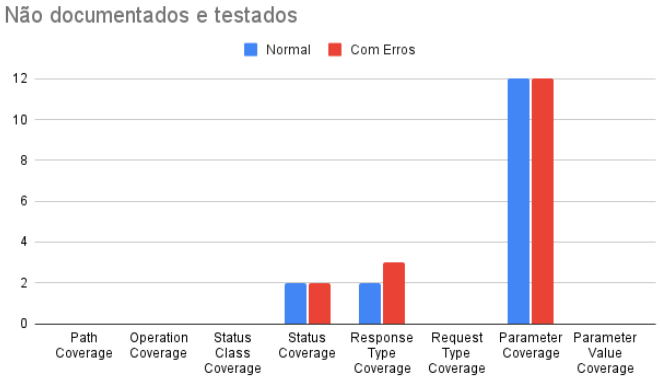
Nos gráficos da Figura 18 até a Figura 21, é possível visualizar o comparativo entre os dois cenários de testes. As barras em azul são referentes aos testes normais e as barras em vermelho são os testes com erros. Além da cobertura, a Restats ainda informa:

- **Itens documentados e testados:** são itens que estão na documentação da API e que foram testados.
- **Itens não documentados e testados:** itens que não estão na documentação da API mas que foram testados. Isso indica que provavelmente foram feitas implementações ou alterações no código que não foram adicionadas na documentação.

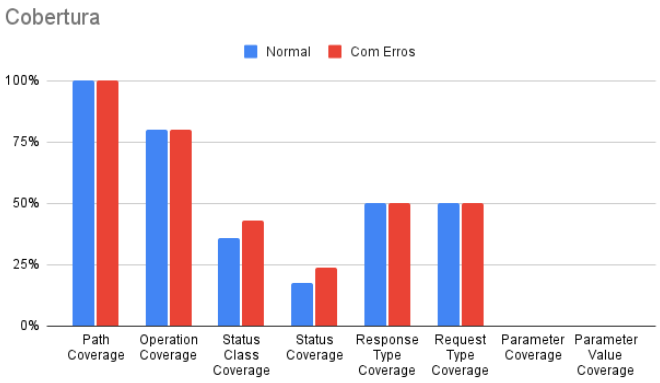
Teste de fumaça



((a)) Documentados e testados

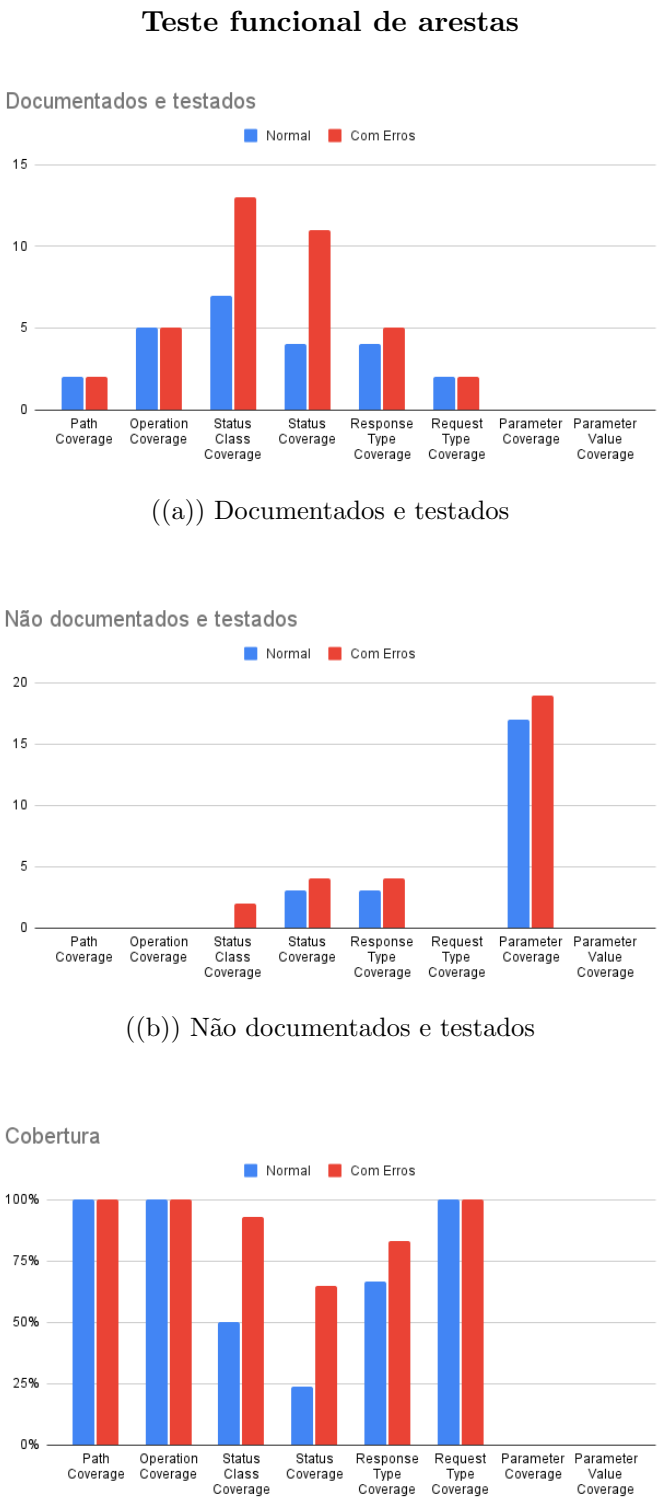


((b)) Não documentados e testados



((c)) Cobertura

Figura 18: Resultados do teste de fumaça para cenários normais e com erros.

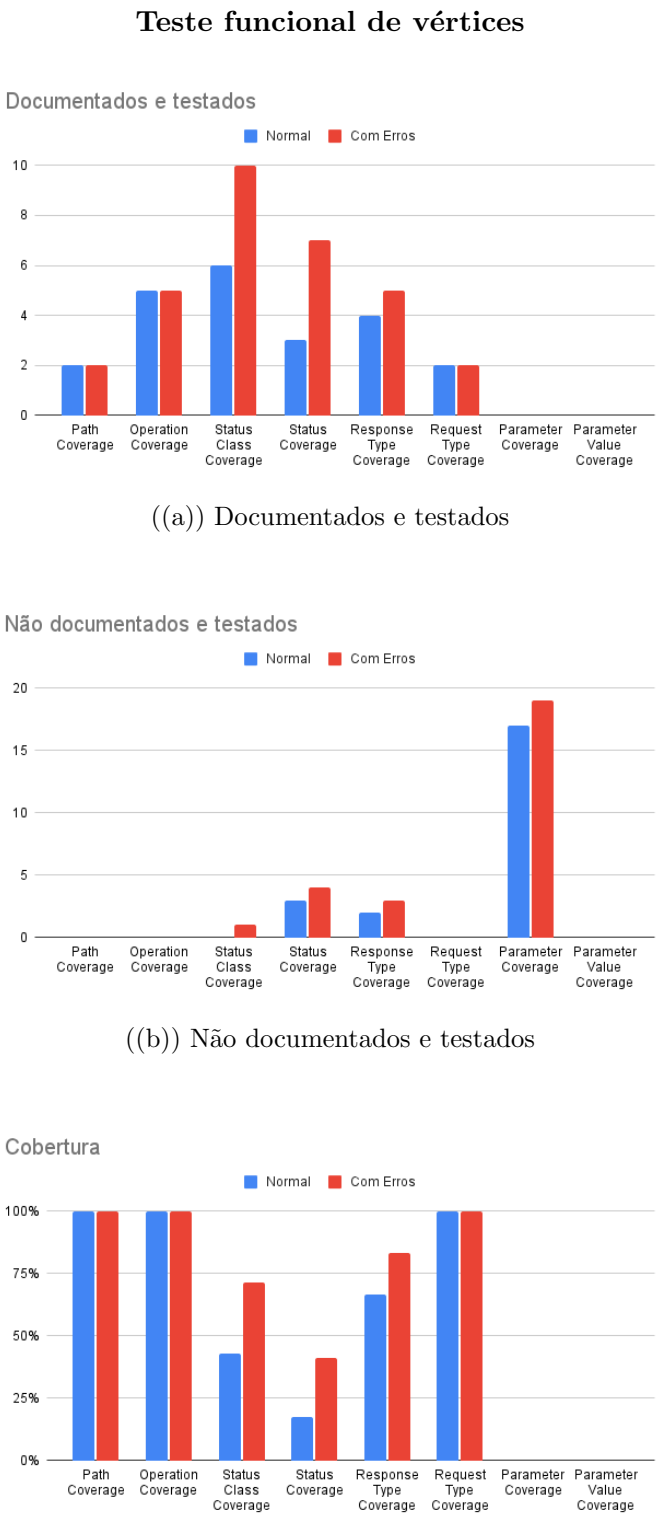


((a)) Documentados e testados

((b)) Não documentados e testados

((c)) Cobertura

Figura 19: Resultados do teste funcional de arestas para cenários normais e com erros.



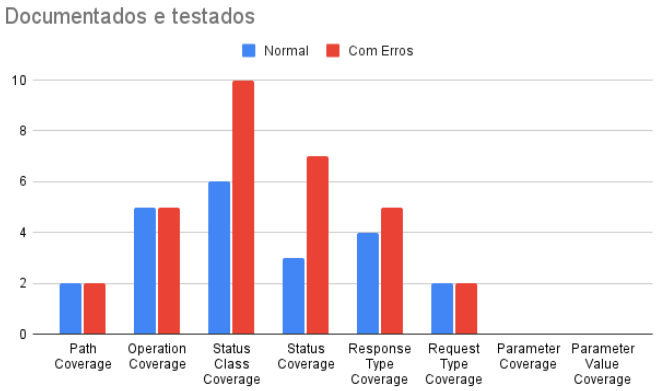
((a)) Documentados e testados

((b)) Não documentados e testados

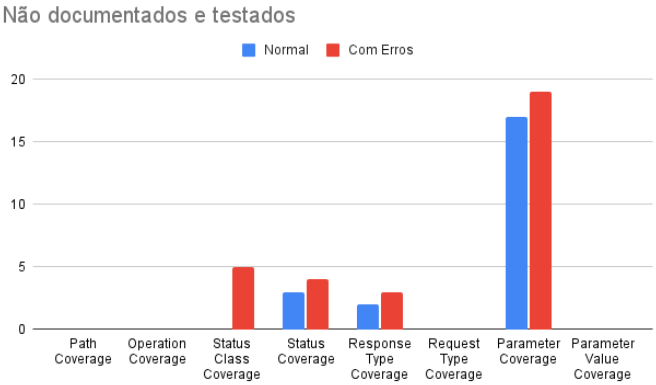
((c)) Cobertura

Figura 20: Resultados do teste funcional de vértices para cenários normais e com erros.

Teste de estabilidade



((a)) Documentados e testados



((b)) Não documentados e testados

((c)) Cobertura

Figura 21: Resultados do teste de estabilidade para cenários normais e com erros.

6.2 Análise de resultados

Referente aos resultados, logo de início percebemos que algumas métricas não tiveram variação de resultados ao comparar os diferentes modelos, são elas: *Path Coverage*, *Operation Coverage* e *Request Type Coverage*. Isto se deve ao fato de que a especificação deste recurso contém poucos itens a serem exercitados, como mostrado na Tabela 2. Com isso, até um teste mais simples como o teste de fumaça era capaz de exercitar todas as opções disponíveis.

Em contrapartida, as métricas que mais sofreram variação foram: *Status Class Coverage* e *Status Coverage*. Era esperado que isto acontecesse pois, ao adicionar os erros no modelo, foi possível testar erros das classes 4xx (Erro do cliente) e 5xx (Erro no servidor) incluindo os códigos 400, 404 e 500.

Uma métrica que teve um comportamento não esperado foi a *Parameter Coverage*. Isso porque em todos os testes ela apareceu dentro do grupo "Não documentados e testados". Aqui cabe uma explicação mais detalhada. Analisando os logs gerados, constatamos que se trata de parâmetros de cabeçalho, os quais não estão devidamente especificados para a aplicação testada. Esses parâmetros vêm logo após a requisição, como mostra a Figura 22, indicando, entre outras, o tipo de conteúdo aceito na resposta (json ou xml, por exemplo), o hospedeiro ou a linguagem (pt-br, en). Como esses parâmetros não foram definidos na especificação, foram definidos parâmetros default pela plataforma de execução dos testes, os quais aparecem no log. Por essa razão, a ferramenta de cobertura considera-os como testados, mas não documentados, dando indicação para os desenvolvedores de que estas informações deveriam constar da especificação.

```
"notDocumentedAndTested": {
    "/api/vets": {
        "get": [
            "Accept",
            "Content-Type",
            "HOST"
        ],
        "post": [
            "Accept",
            "Content-Type",
            "HOST",
            "\n",
            "{\"firstName\""
        ]
    },
    "/api/vets/{vetId}": {
        "get": [
            "Accept",
            "HOST"
        ],
        "delete": [
            "Accept",
            "HOST"
        ]
    }
}
```


Figura 22: Relatório gerado pela Restats.

Ao verificar os exemplos presentes no próprio repositório da Restats, percebemos que esse comportamento estranho também ocorre, dando a entender que provavelmente isso é algum problema dentro da própria ferramenta.

Outro ponto de atenção observado, foi em relação aos códigos HTTP retornados pela API. Em algumas operações, os códigos retornados não eram condizentes com o que estava documentado. Por exemplo, na operação para criar um veterinário usando um método POST, a API retornava 201, porém na documentação estava descrito que seria retornado o código 200. Outro exemplo era em relação a operação que removia um veterinário usando um método DELETE. Ao executar a operação, a API retornava 204 em caso de sucesso, porém na documentação estava descrito que seria retornado o código 200.

Situações como essa, onde a documentação não está 100% de acordo com o que foi implementado de fato, acaba dificultando a cobertura de testes da API, porém utilizando uma ferramenta como a Restats, é possível identificar pontos de melhoria da documentação.

Analisando mais especificamente cada teste, o teste de fumaça não apresentou mudanças significativas visto que é um teste relativamente simples e rápido comparado aos demais. Em relação a cobertura, em ambos os modelos, o TCL ficou com o valor de 1, atingindo 100% da cobertura dos caminhos.

No teste funcional de arestas, houve uma melhora da cobertura referente às métricas *Status Class Coverage* e *Status Coverage*, isso porque os códigos de erros foram testados. Segundo a Restats, o TCL atingiu o nível 2 com 100% de cobertura de operações. Porém, analisando melhor os relatórios, percebemos que o TCL poderia ter sido maior e chegado ao nível 3 (100% de cobertura do tipo de conteúdo de requisições e respostas), caso não houvesse diferenças entre as informações que a API retorna e o que está documentado, conforme discutido anteriormente.

A mesma coisa ocorreu para o teste funcional de vértices e para o teste de estabilidade. A cobertura teve uma melhora em relação às métricas *Status Class Coverage* e *Status Coverage* e o TCL também ficou no nível 2 devido a diferenças entre a implementação da API e sua documentação. Caso contrário poderia ter chegado no nível 3.

6.3 Resumo

De fato, o modelo com os cenários de erros ajudaram a melhorar a cobertura. Os resultados mostraram que algumas métricas, como *Path Coverage* e *Operation Coverage*, não variaram devido à simplicidade dos cenários testados, enquanto outras, como *Status Class Coverage* e *Status Coverage*, se beneficiaram da inclusão de erros nos modelos, testando códigos 4xx e 5xx. Além disso, trabalhar com uma especificação OAS contendo apenas o que é necessário para o teste, ajudou na análise dos resultados. Por outro lado, a existência de itens não documentados, como o campos de cabeçalho, e de itens desatualizados na especificação, o nível de testes atingido foi mais baixo do que deveria. O resultado mostra a importância da análise da cobertura, pois ajuda não só a melhorar o conjunto de testes, mas indica também melhorias para a documentação da API.

7 Conclusões

Nesse projeto, trabalhamos e discutimos a avaliação e melhoria de testes baseados em modelos de estados para APIs REST. Vimos que os testes baseados em modelos de estados são utilizados para representar o comportamento esperado de sistemas, permitindo automatizar a geração de casos de teste e identificar discrepâncias entre o comportamento real e o especificado. Para aplicar na prática, escolhemos um recurso da Pet Clinic e modelamos dentro da GraphWalker primeiramente o modelo com a execução normal e depois a execução com erros. Após a criação do modelo, geramos a interface com os métodos dos vértices e arestas e implementamos esses métodos dentro de um projeto Java junto com o JUnit e a Rest Assured. Com o código implementado, executamos os testes e utilizamos a Restats para verificar as métricas de cobertura.

Feito isso para o cenário normal, repetimos os mesmos passos para os cenários com erros. Um ponto positivo é que, como a interface é gerada automaticamente, foi possível saber quais funções deveriam ser implementadas e todo o código criado anteriormente foi totalmente reaproveitado sem necessidade de alterações.

Com todos os relatórios da Restats disponíveis, foi possível identificar que houve uma melhora na cobertura da métricas envolvendo os status das requisições no modelo com cenários de erros. Entretanto, também notamos alguns problemas envolvendo o gerador de relatórios da Restats e incongruências entre a implementação da API e a especificação OAS. Para testes futuros, o ideal seria garantir que a documentação da API esteja com os dados corretos para que seja possível obter resultados melhores relacionados à cobertura.

Uma sugestão de trabalho futuro seria testar recursos com parâmetros especificados, que levariam à necessidade de complementar os testes baseados em modelos de estado com testes de dados para cobrir o conteúdo de mensagens e de parâmetros.

Referências

[1] P. Ammann, J. Offutt. Introduction to Software Testing. Editora Cambridge University Press, 2017.

[2] Bonifacio, A. L. and Moura, A. V. (2023). Conformance checking and pushdown reactive systems. CLEI Electronic Journal, 25(2).

[3] Corradini, D., Zampieri, A, Pasqua M. and Ceccato, M. (2021). restats: A Test Coverage Tool for RESTful APIS. 37th International Conference on Software Maintenace and Evolution (ICSME 2021).

[4] Freitas, F.G.; Maia, C.L.B.; Campos, G.A.L.; Souza, J.T. Otimização em Testes de Software com Aplicação de Metaheurísticas. Revista de Sistemas de Informação da FSMA n. 5 (2010) pp. 3-13.

[5] Fielding, R.T., Architectural styles and the design of network-based software architectures. University of California, Irvine Tese de Doutorado, 2000. Disponível em: https://ics.uci.edu/ fielding/pubs/dissertation/rest_arch_style.htm. Acesso em: jul/2024.

[6] Jorgensen, P.C. The Craft of Model-Based Testing. CRC Press, 2017.

[7] Li, B. Layered Architecture for Test Automation. INFOQ. Acesso em novembro/2024 em https://www.infoq.com/articles/layered-test-automatation/.

[8] Meyer, B., (1992). 'Applying design by contract'. IEEE Computer, 25(10), pp.40-51.

[9] Open API Specification v. 3.1.0. (fev/2021). Acesso em julho/2024 em https://spec.openapis.org/oas/latest.html.

[10] (2011). OMG unified modeling language, superstructure version 2.4.1. http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF.

[11] Pinheiro, P. V. P., Endo, A. T., & Simao, A. (2013). Model-based testing of RESTful web services using UML protocol state machines. In Brazilian workshop on systematic and automated software testing (pp. 1-10).

[12] M. Pezzè, M. Young. Teste e Análise de Software. ARTMED Editora, 2008.

[13] Porres, I., & Rauf, I. (2011). Modeling behavioral RESTful web service interfaces in UML. In Proceedings of the 2011 ACM Symposium on Applied Computing (pp. 1598-1605).

[14] Rauf, I. (2014) Design and Validation of Stateful Composite RESTful Web Services. PhD Thesis, Turku Center for Computer Science.

[15] Richardson, L.; Ruby, S. RESTful Web Services. O'Reilly, 2007. Visto em novembro/2024 em https://www.oreilly.com/library/view/restful-web-services/9780596529260/ch04.html.

[16] Sanoja, D. (2023). Types of API Architecture Diagrams. Acesso em julho/2024 em https://bump.sh/blog/api-architecture-diagrams.

[17] J. Smart. BDD In Action. Manning Publications Co., 2015, cap 7.3.

[18] J. L. Szwarcfiter. Algoritmos e Grafos: Uma Introdução. #ª Escola de Computação. Departamento de Informática PUC/RJ. Rio de Janeiro, 1982

[19] Utting, M.; Legeard, B. Practical Model-Based Testing. Editora Morgan-Kaufmann, 2007.

[20] E. J. Weyuker. Axiomatizing Software Test Data Adequacy. IEEE Trans. on Software Engineering, vol.SE-12, nº 12, dez/1986, pp. 1128-1138.

[21] Model-Based Testing using GraphWalker and Java. Acesso em novembro/2024 em https://docs.getxray.app/display/XRAY/Model-Based+Testing+using+GraphWalker+and+Java.

[22] SoftDesign. "RestAssured: O que é e como usar?"SoftDesign Blog. Acesso em novembro/2024 em https://softdesign.com.br/blog/restassured-o-que-e-e-como-usar/#h-configuracao-do-ambiente-com-rest-assured