Especificação de Interfaces Aplicacionais REST

Fábio Ferreira, Telmo Santos,
Francisco Martins, Antónia Lopes e Vasco T. Vasconcelos

Universidade de Lisboa, Faculdade de Ciências, LaSIGE

Resumo A programação de serviços *web* com interfaces aplicacionais REST é atualmente muito popular. O desenvolvimento de aplicações clientes destes serviços exige que as interfaces estejam bem documentadas. No entanto, e apesar de iniciativas importantes como a *Open API Specification*, o suporte à descrição destas interfaces é atualmente bastante limitado, focado essencialmente nos aspetos sintáticos. Neste artigo apresenta-se a linguagem HeadREST que permite ultrapassar estas limitações e descrever também os aspetos semânticos das interfaces num estilo reminescente dos triplos de Hoare e recorrendo a tipos refinados. A linguagem é apresentada através de pequenos exemplos extraídos de um dos estudos de caso que desenvolvemos com o intuito de a avaliar. Discute-se ainda a forma de validar a boa formação das especificações HeadREST e uma técnica para a geração de testes para verificar a conformidade de uma API REST relativamente à sua descrição.

1 Introdução

A programação de serviços *web* com interfaces aplicacionais que aderem ao estilo REST [5], designadas em inglês por *RESTful APIs*, e de aplicações consumidoras deste tipo de serviços é atualmente muito popular [8]. Por exemplo, aplicações como o Twitter, Instagram, Youtube e Uber, fornecem acesso programático às suas aplicações clientes através deste tipo de APIs. Isto acontece porque a utilização deste tipo de APIs, quando comparadas com as tradicionais interfaces de serviços *web* baseadas em SOAP, simplificam grandemente o desenvolvimento das aplicações clientes. Mais recentemente, com o advento da arquitetura baseada em micro-serviços, o desenho de aplicações como conjuntos de serviços tornou-se muito comum, alavancando ainda mais a utilização das APIs REST [3].

O desenvolvimento eficaz de aplicações clientes deste tipo de serviços exige que as suas interfaces estejam bem documentadas. Apesar de iniciativas importantes como a *Open API Specification*,[1] focada na criação e promoção de um formato aberto para a descrição de APIs REST, o suporte à descrição deste tipo de APIs é atualmente ainda bastante limitado e incide essencialmente na estrutura e representações dos dados trocados entre clientes e fornecedores.

De forma a ultrapassar as limitações existentes e suportar também a descrição dos aspetos semânticos subjacentes às APIs REST, desenvolveu-se a linguagem HeadREST que permite especificar cada um dos seus serviços individualmente, num estilo reminescente dos triplos de Hoare e utilizando tipos

[1] https://www.openapis.org

refinados. Trata-se de uma linguagem simples apenas com as características que foram identificadas como essenciais para ultrapassar as limitações das abordagens existentes. O objetivo da linguagem HeadREST não é assim estender a *Open API Specification* mas antes identificar primitivas que permitem alargar o seu poder expressivo e mostrar que é possível explorar estas descrições também para melhorar o estado da arte no que diz respeito à programação e teste de aplicações REST.

A linguagem é apresentada através de exemplos extraídos de um dos estudos de caso desenvolvidos para avaliar a linguagem: um sistema de gestão de labirintos, constituídos por quartos ligados entre si por portas. Este sistema, que possui uma interface aplicacional REST, encontra-se disponível em `https://mazes-demo.herokuapp.com/rest/v1/swagger.json`.

Discute-se ainda a verificação da boa formação das especificações HeadREST e uma técnica de geração de testes para aferir a conformidade de uma API REST relativamente à sua descrição. Os protótipos que implementam estas técnicas, ainda em desenvolvimento, tem a forma de um *plugin* para um *Interactive Development Environment* popular (o Eclipse) e utilizam uma ferramenta de verificação de teoremas (um SMT, o Z3 [2] em particular) para verificar as restrições geradas pelo sistema de tipos refinados e a relação de subtipos.

2 Contexto e trabalho relacionado

APIs REST O REST (*Representational State Transfer*) é um estilo arquitetural desenvolvido como um modelo abstrato da arquitectura da *web*, alicerçado no conceito de *recurso* [4]. De acordo com Fielding e Taylor [5], um *recurso* é uma função $M_R(t)$ que associa a cada instante de tempo t um conjunto de valores, os quais podem ser *identificadores* ou *representações* de recursos. Os identificadores servem para identificar o recurso envolvido numa interação. Para executar ações sobre um recurso, os componentes REST utilizam representações que capturam o estado corrente ou o estado pretendido do recurso.

No sistema que vamos usar como exemplo ao longo do artigo—um sistema de gestão de labirintos constituídos por quartos ligados entre si por portas—os recursos são labirintos, quartos e portas. Na figura 1 apresenta-se um exemplo do conjunto de valores associados a um quarto de um labirinto num determinado instante. Podemos ver que o quarto tem nesse instante dois identificadores (por um lado é o quarto 1 do labirinto 1 e, por outro, é o quarto de entrada no labirinto 1) e tem duas representações, uma em JSON e outra em XML.

Neste artigo focamo-nos em sistemas REST que comunicam sobre HTTP e que interagem com sistemas externos através de recursos *web* identificados por URIs (*Unique Resource Identifiers*). Assim, as ações que podem ser executadas sobre um recurso correspondem a pedidos de execução dos métodos oferecidos pelo HTTP—GET, POST, PUT e DELETE—e os meta-dados e dados a transferir pelo cliente são enviados, respetivamente, nos cabeçalhos (*header*) e no corpo (*body*) do pedido. Em resultado é produzida uma resposta com os dados e

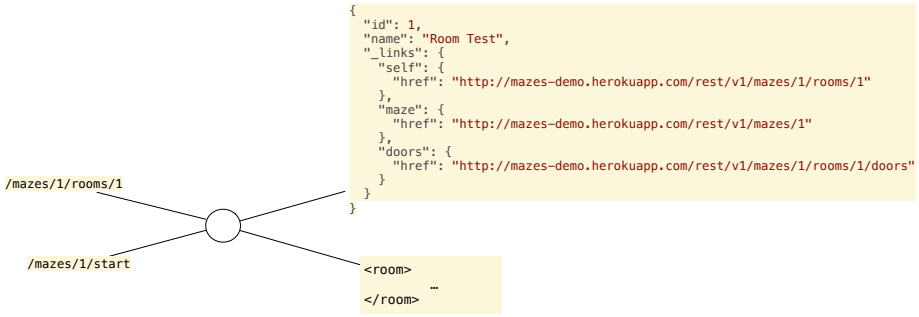

Figura 1. Um quarto de um labirinto num determinado instante

meta-dados a transferir para o cliente. A figura 2 mostra um exemplo de um pedido de execução de um PUT sobre o quarto referido anteriormente, identificado como o quarto 1 do labirinto 1. No pedido, para além de meta-dados, é comunicada ainda a representação (parcial) pretendida para esse quarto (ou seja, o novo nome). A resposta obtida, além de meta-dados, comunica a representação corrente do recurso sobre o qual foi realizada a ação.

As interfaces aplicacionais destes sistemas podem ser abstraídas como conjuntos de pares com um método e um *URI template*,[2] como ilustrado na primeira linha da figura 2.

```
PUT "http://mazes-demo.herokuapp.com/rest/v1/mazes/1/rooms/1"
-H "accept: application/hal+json"
-H "content-Type: application/json"
-d "{ \"name\": \"Room Test Updated\"}"
```

```
Code 200
Response body
{
  "id": 1,
  "name": "Room Test Updated",
  "_links": {
    "self": {
      "href": "http://mazes-demo.herokuapp.com/rest/v1/mazes/1/rooms/1"
    },
    "maze": {
      "href": "http://mazes-demo.herokuapp.com/rest/v1/mazes/1"
    },
    "doors": {
      "href": "http://mazes-demo.herokuapp.com/rest/v1/mazes/1/rooms/1/doors"
    }
  }
}
Response headers
content-type: application/hal+json
```

Figura 2. Pedido de execução de PUT sobre um quarto através do identificador `/mazes/1/rooms/1` e respetiva resposta

[2] `https://tools.ietf.org/html/rfc6570`

Descrição de interfaces aplicacionais As interfaces aplicacionais REST foram originalmente propostas por Fielding no contexto da abordagem *Hypermedia as the Engine of Application State* [4], a qual assenta na ideia de que o código dos clientes não é escrito de encontro a uma descrição estática de um interface do serviço, mas antes deverá utilizar um conjunto de pontos bem conhecidos de entrada no serviço para ir descobrindo dinamicamente quais as interações que o serviço permite. Porém, a popularidade do REST foi conseguida à custa de serviços que não aderem a esta visão e que fornecem antes interfaces estáticas contra as quais esperam que as aplicações clientes sejam programadas. Neste contexto é importante ter descrições das interfaces REST que possam ser usadas como documentação e, portanto, lidas por humanos e também processadas automaticamente, nomeadamente para gerar artefactos de *software* que facilitem o desenvolvimento das aplicações clientes ou dos próprios serviços.

Existem atualmente várias linguagens de descrição de interfaces (IDLs) que foram desenhadas propositadamente para suportar a descrição formal de APIs REST.[3] Entre estas destacam-se a *Open API Initiative*[4] (originalmente chamada *Swagger*), a *RESTful API Modeling Language*[5] (RAML) e a *API Blueprint*[6] pelo impacto e a quantidade de apoios que reúnem. Estas IDL permitem descrever com detalhe e rigor os aspetos sintáticos dos dados transferidos nas interacções REST e tem associadas uma grande panóplia de ferramentas, nomeadamente para a geração de documentação, geração de código de clientes em diferentes linguagens de programação e geração de testes. A expressividade destas linguagens é porém bastante limitada no que diz respeito aos aspetos semânticos. Por exemplo, recorrendo às especificações *Open API Specification* não é possível descrever várias propriedades da API que podem ser importantes. Recorrendo ao caso de estudo do sistema de gestão de labirintos, não é possível, por exemplo, expressar a propriedade que os nomes fornecidos na criação de labirintos e de quartos têm de ter entre 3 e 50 caracteres, nem que os nomes dos labirintos são únicos (i.e., que um labirinto só será criado se não existir outro com o mesmo nome), que os nomes dos quartos de um labirinto também são únicos e que não é possível apagar o quarto designado como a entrada de um labirinto.

Descrições focadas nos recursos são suportadas pelo RDF (*Resource Description Framework*), um modelo padrão para intercâmbio de dados na *web*, desenvolvido pela W3C.[7] Este permite descrever recursos e as relações entre estes (na forma de um grafo etiquetado), mas não permite descrever o comportamento dos serviços e da ação deste sobre os recursos. Uma perspetiva semelhante é a dos SERIN (*Semantic RESTFul Interfaces*) [6], uma IDL que permite definir modelos sintáticos e semânticos das APIs REST baseada na junção de anota-

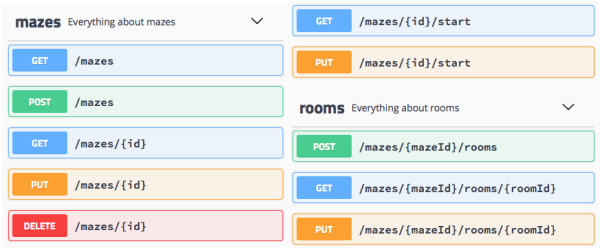

Figura 3. Excerto da API REST do sistema de gestão de labirintos

ções às classes das ontologias OWL *(Web Ontology Language)* para descrever que operações podem ser executadas e as restrições de integridade subjacentes.

O GraphQL é uma linguagem para consumir dados de APIs (não necessariamente REST), desenvolvida pela Facebook.[8] Permite aos clientes pedirem exatamente as propriedades de interesse, ao invés de se restringirem às representações disponibilizadas pelo servidor. Tal como no caso do RDF não permite descrever o comportamento dos vários serviços numa API.

3 HeadREST

A abordagem desenvolvida para especificar as interfaces REST assenta em duas ideias chave:

- a utilização de tipos para exprimir propriedades dos dados que são transmitidos nas diferentes interações e
- a utilização de pares de pré e pós-condições para exprimir (a) as relações que existem entre os dados enviados nos pedidos e obtidos nas respostas e (b) as mudanças de estado resultantes.

Estas ideias estão concretizadas na linguagem HeadREST, linguagem essa que é construída sobre dois conceitos fundamentais (cf. [1]):

- tipos de refinamento, `x:T where e`, que consistem em valores `x` do tipo `T` que satisfazem a propriedade `e` e
- um predicado, `e in T`, que devolve `true` ou `false` dependendo do fato do valor de `e` ser ou não do tipo `T`.

[3] Ver uma longa lista em `https://goo.gl/cY8knT`
[4] `https://www.openapis.org`
[5] `https://raml.org`
[6] `https://apiblueprint.org/`
[7] `https://www.w3.org/RDF/`
[8] `http://graphql.org`

Tipos de representação Para descrever a estrutura dos dados transferidos nas diferentes interações, consideram-se *tipos de representação*. Estes tipos, inspirados nos *modelos* usados nas especificações *Open API*, permitem ir além da estrutura dos dados e expressar propriedades sobre os seus valores.

Mais precisamente, os tipos de representação suportados são tipos de objetos (`{}` e `{l:T}`), tipos de vectores (`T[]`), tipos de refinamento (`x:T where e`), tipos escalares (incluindo `integer`, `boolean`, `string` e `URITemplate`) e o tipo de topo `any`. Estes tipos estão equipados com uma relação de subtipo definida semanticamente [1].

A linguagem de tipos base é extremamente rica, permitindo uma panóplia de tipos derivados. Por exemplo, o tipo singular, escrito `[e]`, é uma abreviatura do tipo refinado `(x:any where x==e)` quando `x` não ocorre livre em `e`. Uma utilização frequente deste padrão é `[null]`, o tipo que contém apenas o valor `null`. Tipos união `T||U`, intersecção `T&&U` e complemento `!T` são abreviaturas dos tipos `(x:any where x in T || x in U)`, `(x:any where x in T && x in U)` e `(x : any where !( x in U)`, respetivamente e onde `x` não ocorre livre nem em `T` nem em `U`. Utilizando os tipos intersecção podemos codificar tipos objecto multi-propriedade, como por exemplo o tipo `{l:T, m:U}` que aparece como abreviatura de `{l:T}&{m:U}`.

A linguagem de expressões conta com alguns operadores primitivos importantes. Entre eles encontra-se `matches` que verifica se uma *string* pertence à linguagem definida por uma dada expressão regular. Deste modo, o tipo `URI` dos identificadores de recursos é uma abreviatura de `(x:string where matches( e_uri, x))`, onde `e_uri` é uma *string* que denota a expressão regular descrita no RFC 6570.[9]

Por questões de conveniência de especificação, HeadREST permite associar nomes a tipos de representação e utilizar posteriormente esses nomes para referir os respetivos tipos.

```
type Link = {
    ?href: URI
}
type MRData = {
    name: (x: string where length(x) > 2 && length(x) <= 50)
}
type MazeGetData = {
    id: integer,
    name: string,
    _links: {
        self: Link,
        start: Link[] | [null]
    },
    _embedded: { orphanRooms: RoomGetData[] }
}
type RoomGetData = {
     id: integer,
     name: string,
```

[9] https://tools.ietf.org/html/rfc6570

```
    _links: {
        self: Link,
        maze: Link,
        doors: Link
    }
}
```

O exemplo introduz os nomes `Link`, `MRData` e `MazeGetData` e `RoomGetData`, os quais são associados a quatro tipos de representação: `Link` é um tipo de objeto apenas com uma propriedade opcional, com nome `href` e tipo `URI`; `MRData` é um tipo de objeto também apenas com uma propriedade cujo tipo é um refinamento de `string` que denota as sequências com um número de carateres entre 3 e 50; `MazeGetData` e `RoomGetData` são dois tipos de objeto com várias propriedades, todas obrigatórias, que utiliza o nome `Link` introduzido anteriormente (evitando assim ter de repetir a definição do tipo associado a esse nome).

Os tipos objecto podem ser estendidos em largura, juntando mais propriedades, através da relação subtipos, e.g., `{l:T} & {m:U} <: {l:T}`. Mas isto não restringe a `U` o tipo da nova propriedade `m`, sendo que por vezes, pretendemos fixar este tipo. É o caso do tipo `Link` onde a propriedade `href`, a existir, tem ser do tipo `URI`. O tipo `{?href: URI}`, definido como uma abreviatura de `(x:any where x in {href: any} => x in {href: URI})`, captura essa restrição.

Os valores de um tipo de representação podem ser apresentados de diferentes formas. Por exemplo, um valor do tipo `RoomGetData` tanto pode ser apresentado em JSON como em XML. No corpo da resposta apresentada na Figura 2, há um exemplo de um valor deste tipo apresentado em JSON.

Pedidos e respostas Como discutido anteriormente, nas interações dos sistemas REST há dados transmitidos no pedido (do cliente para o fornecedor) e na resposta (do fornecedor para o cliente). Seguindo de perto os princípios do REST, ao mesmo tempo que se abstraem detalhes considerados pouco relevantes, em HeadREST é considerado que qualquer pedido de execução de um método sobre um recurso é do tipo `Request` e que a resposta obtida é sempre do tipo `Response`. Estes tipo estão predefinidos na linguagem HeadREST e têm, respetivamente, a seguinte definição:

```
type Request = {
  location: URI,
  ?template: {},
  ?header: {}
}
```

```
type Response = {
  code: integer
  ?header: {}
}
```

HeadREST dispõe de duas variáveis predefinidas, `request` e `response` pertencentes aos tipos supramencionados.

Os dados transmitidos num pedido podem estar sujeitos a condições adicionais por o pedido acontecer sobre um recurso que é identificado por um URI, obtido por expansão de um determinado *URI template*. Essas restrições podem ser capturadas por um subtipo do tipo `Request`. Por exemplo, o tipo dos pedidos de execução de `GET` sobre uma expansão de `mazes/{mazeId}/rooms/{roomId}` inclui

sempre as propriedades `mazeId` e `roomId`, não havendo restrições para os valores que lhes estão associados. Isto é capturado pelo seguinte subtipo de `Request`:

```
{
  location: URI,
  template: {mazeId: any, roomId: any},
  ?header: {}
}
```

Estados Através de uma interface aplicacional REST um sistema expõe interações que permitem observar e modificar os recursos do sistema, ou partes destes. Assim, no contexto das especificações das APIs REST, considera-se que o conjunto de valores associados a cada recurso do sistema num determinado instante de tempo é o que define o estado do sistema nesse instante.

Assume-se que cada recurso é classificado com sendo de um determinado tipo, de entre um determinado conjunto de *tipos de recursos*. Por exemplo, no caso do sistema de gestão de labirintos precisamos de três tipos de recursos. Estes são declarados na especificação HeadREST da API da seguinte forma:

```
resource Maze, Room, Door
```

A figura 4 mostra um exemplo de um estado do sistema de labirintos. Neste estado apenas dois recursos não estão associados ao conjunto vazio de valores, um deles é um labirinto e o outro é o quarto de início desse labirinto. Este exemplo mostra ainda que os estados representam explicitamente o papel que cada valor associado a um recurso tem através das relações binárias primitivas `resourceidof`, `representationof` e `in`.

Asserções HeadREST permite descrever formalmente as observações e as mudanças de estado resultantes das interações expostas numa API REST através de um conjunto de asserções com a estrutura dos triplos de Hoare. Concretamente, as asserções são triplos da forma

$$\{\phi\}\ (a\ \ t)\ \{\psi\}$$

onde a é uma ação (`GET`, `POST`, `PUT` ou `DELETE`), t é um *URI template* e ϕ, ψ são expressões do tipo booleano. A fórmula ϕ, chamada de pré-condição, endereça o estado em que a ação é executada e os dados transmitidos no pedido enquanto que ψ, chamada de pós-condição, endereça o estado resultante da execução da ação e os valores transmitidos na resposta. A asserção diz: se o pedido para a execução da ação a sobre uma expansão de t transporta dados que satisfazem ϕ e a ação é realizada num estado que satisfaz ϕ então os dados transmitidos na resposta satisfazem ψ, assim como o estado resultante da execução da ação.

Por questões de espaço, apresenta-se a linguagem usada para escrever pré e pós-condições apenas através de alguns exemplos. Os exemplos escolhidos fazem parte da especificação da API REST do sistema de gestão de labirintos em linguagem HeadREST que foi desenvolvido como caso de estudo.

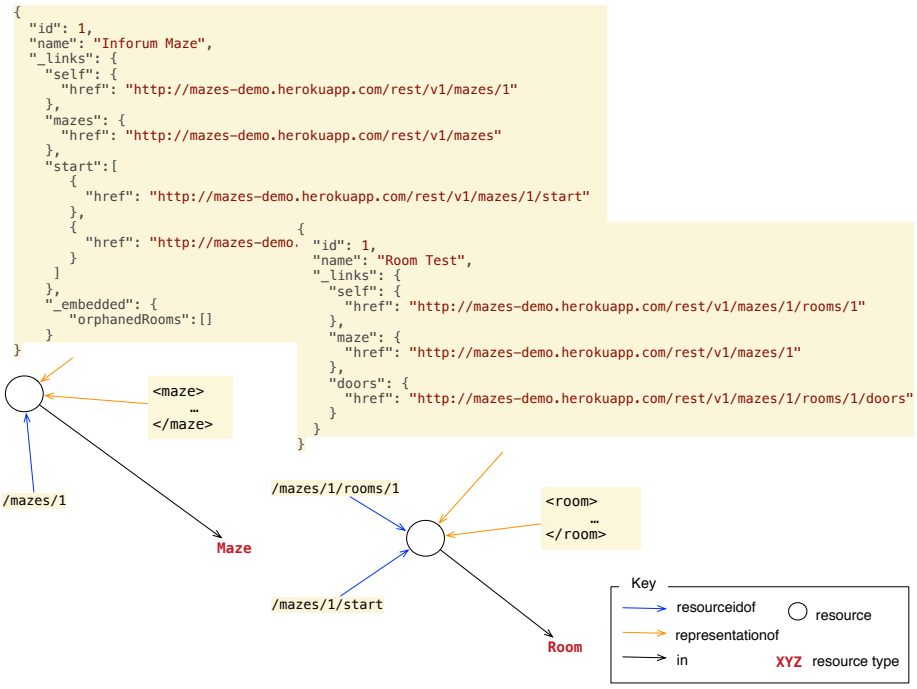

Figura 4. Um estado do sistema de labirintos

A primeira asserção endereça os pedidos de execução de `POST` sobre um recurso identificado por `/mazes` no caso em que os dados transmitidos no corpo do pedido são do tipo `MRData` (que como vimos anteriormente é um tipo objecto com uma propriedade com o nome `name`) e a ação é executada num estado em que ainda não existe qualquer outro recurso do tipo `Maze` com o nome indicado no pedido. A asserção declara que nesta situação o sistema efetivamente cria um recurso do tipo `Maze` com o nome indicado, que os dados transmitidos no corpo da resposta são uma representação desse recurso e o `URI` transmitido no cabeçalho um seu identificador, e que o código da resposta é o correspondente a `CREATED` (definido como 201). A asserção revela também que a representação do recurso acabado de criar que vem na resposta tem a propriedade `response.body._links.start` com o valor `null`.

```
1  // add maze, created
2  {
3    request in {body: MRData} &&&
4    (forall maze: Maze forall mazeRep:
5     (x: MazeGetData where x representationof maze) | mazeRep.
          name != request.body.name)
6  }
7    POST /mazes
8  {
```

```
9     response.code == CREATED &&
10    response in {body: MazeGetData, header: {Location: URI}}
          &&&
11    response.body.name == request.body.name &&
12    response.body._links.start == null &&
13    (exists maze: Maze | response.body representationof maze
14      && response.header.Location resourceidof maze)
15  }
```

A boa formação da asserção requer uma breve explicação. A boa formação de asserções é dada por um conjunto de regras que validam aspetos que não podem ser descritos sintaticamente. Em vez de apresentarmos as regras, apelamos à intuição do leitor. A linha 5 fala do objecto `request.body.name`. O tipo `Request` não nos permite concluir que a variável `request` tem uma propriedade `body` (e ainda menos que esta propriedade é um objecto com uma outra propriedade `name`). Mas o caso muda de figura se soubermos que `request` é do tipo `{body: MRData}`, porque `MRData` é um tipo objecto com uma propriedade `name`. A expressão `e &&& f` é uma abreviatura da expressão condicional `e ? f : false`. Deste modo a linha 5 está debaixo do ramo *then* do teste `request in {body: MRData}` (linha 3), o que nos permite assegurar que `request.body.name` está definido. Passando para a pós-condição, o operador `&&&` é usado crucialmente na linha 10 para estabelecer o tipo do objecto `response`, e permitir o acesso aos seus campos nas linhas seguintes. Além disso, o objecto `request` tem a propriedade `body` e esta a propriedade `name` (linha 11) porque o contexto da pré-condição estende-se à pós-condição.

A especificação inclui ainda duas outras asserções para o par `POST /mazes`: uma que endereça a situação em que já há um labirinto com o nome fornecido no pedido e uma outra para a situação em que `!(request in {body: MRData})`, ou seja o pedido não tem corpo ou tem mas os dados que vêm no corpo não têm o tipo certo. Em ambos os casos não é criado nenhum labirinto e os dados da resposta identificam o tipo de situação de erro ocorrida.

A asserção mostrada abaixo endereça por seu lado os pedidos de execução de `GET` sobre expansões de `/mazes{?page,limit}`. A asserção descreve que no caso de os valores dos parâmetros estarem dentro de determinados intervalos, a resposta ao pedido traz uma lista com as representações dos recursos existentes.

```
type RequestTemplate = {
  template: {
    page: (i: integer where 1<=i && i<=100000),
    limit: (i: integer where 1<=i && i<=50)
  }
}
// get mazes
{
    request in RequestTemplate
}
    GET /mazes{?page,limit}
{
    response.code == SUCCESS &&
    response in {body: MazeList} &&&
```

```
    response.body.meta.totalResults >= 0
}
```

4 Validação de especificações e geração de testes

A par da definição da linguagem HeadREST temos vindo a desenvolver uma ferramenta para verificar a boa formação das asserções e outra para testar APIs.

Validação de especificações A validação da boa formação das asserções baseia-se no trabalho de Bierman et al. [1]. A axiomatização original foi estendida com novas regras para definir a semântica subjacente às novas caraterísticas da linguagem HeadREST, nomeadamente recursos e vetores.

O verificador de tipos foi implementado como um sistema bidirecional que recorre a uma função para verificação e a outra para síntese [7]. No ponto de encontro entre a função de verificação e a de síntese encontra-se a relação de subtipos semântica, relação essa que é resolvida através de um SMT. A resposta do SMT é analisada, sendo emitida uma mensagem de erro ou de aviso. A geração de código de verificação a enviar ao SMT apresenta vários desafios, em particular, no que diz respeito à representação de *strings*, que endereçamos optando pela utilização da teoria implementada no Z3str2 [9]. Um outro desafio, sobre o qual ainda estamos a trabalhar, reside em conseguir uma axiomatização para o SMT que apresente uma eficiência razoável sem afetar a correção da mesma.

Geração de testes A técnica habitual de testes deste tipo de APIs envolve a criação manual de pedidos para cada operação, o que pode revelar-se um processo trabalhoso e conducente a erros, especialmente em APIs mais complexas. Ora, a partir da especificação da API é possível gerar casos de teste de forma automática que simplificam grandemente o processo de teste.

A nossa abordagem parte do princípio que o sistema a ser testado tem um estado inicial fixo e bem conhecido. Cabe ao testador construir um estado do sistema que satisfaça a pré-condição da asserção a testar. A partir daí, a nossa ferramenta (1) verifica se o estado a construir satisfaz de facto a pré-condição da asserção a testar, (2) gera testes para exercitar a API descrita por este axioma, e (3) executa os testes, construindo pedidos e verificando a pós-condição da asserção em teste.

Por exemplo, para a asserção que descreve a criação de labirintos apresentada anteriormente encontramos uma pré-condição que exige que não existam labirintos com o mesmo nome daquele que se pretende criar. A partir desta informação o testador tem de fornecer uma lista de operações que constrói este estado. Pode, por exemplo, utilizar a operação `POST /mazes` três vezes para construir um estado com três labirintos. A ferramenta coleciona as pré e pós-condições referente à criação dos três labirintos e com isso verifica a validade da pré-condição da asserção a testar. A geração dos dados para os pedidos é feita pedindo ao SMT valores dos tipos respetivos. Por último, a resposta da chamada ao serviço é verificada de encontro à pós-condição da asserção.

5 Conclusões

Neste artigo apresentamos a HeadREST, uma linguagem para descansarmos a cabeça no que toca a APIs REST. A linguagem, introduzida de um modo informal, destina-se a suportar todo do ciclo de vida de aplicações REST. Falamos brevemente do processo de geração de testes baseados em descrições HeadREST. Para além da geração automática de testes, tencionamos usar a linguagem também para a) verificar dinamicamente a aderência dos serviços face à sua especificação, b) gerar *stubs* servidor e SDKs (*Software Development Kit*) cliente a partir de especificações descritas usando HeadREST e c) verificar estaticamente código cliente e/ou servidor de encontro a especificações HeadREST. Pretendemos ainda explorar a especificação de questões de segurança em contexto REST, em particular como utilizar a linguagem HeadREST para assegurar a conformidade com requisitos de autenticação e de confidencialidade.

Agradecimentos Este trabalho foi suportado pela Fundação para a Ciência e Tecnologia (FCT) através do projeto CONFIDENT (PTDC/EEI-CTP/4503/2014) e a unidade de investigação LASIGE (UID/CEC/00408/2013).

Referências

1. Bierman, G.M., Gordon, A.D., Hriţcu, C., Langworthy, D.: Semantic subtyping with an SMT solver. J. Funct. Program. 22(1), 31–105 (2012), http://dx.doi.org/10.1017/S0956796812000032
2. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: TACAS. pp. 337–340. Springer-Verlag, Berlin, Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-78800-3_24
3. Erl, T., Carlyle, B., Pautasso, C., Balasubramanian, R.: SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutions with REST. Prentice-Hall (2013), http://dx.doi.org/10.1145/2464526.2464551
4. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis, University of California, Irvine (2000)
5. Fielding, R.T., Taylor, R.N.: Principled design of the modern web architecture. ACM Trans. Internet Techn. 2(2), 115–150 (2002), http://dx.doi.org/10.1145/514183.514185
6. Lira, H., Dantas, J., Muniz, B., Nunes, T., Farias, P.: An approach to support data integrity for web services using semantic RESTful interfaces. In: WWW. pp. 1485–1490. ACM (2015), http://dx.doi.org/10.1145/2740908.2743042
7. Pierce, B.C., Turner, D.N.: Local type inference. ACM Trans. Program. Lang. Syst. 22(1), 1–44 (2000), http://dx.doi.org/10.1145/345099.345100
8. Richardson, L., Ruby, S.: Restful Web Services. O'Reilly, first edn. (2007)
9. Zheng, Y., Ganesh, V., Subramanian, S., Tripp, O., Berzish, M., Dolby, J., Zhang, X.: Z3str2: an efficient solver for strings, regular expressions, and length constraints. Formal Methods in System Design 50(2-3), 249–288 (2017), http://dx.doi.org/10.1007/s10703-016-0263-6