Safe(REST)Script

Nuno Burnay, Antónia Lopes e Vasco T. Vasconcelos

LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal

Resumo O consumo de serviços REST é uma forma muito popular de invocar código fornecido por terceiros. Porque é particularmente predominante em aplicações web, muitos clientes destes serviços são escritos em JavaScript e, recentemente, também em TypeScript. Ao escolher o TypeScript, os programadores passam a usufruir de uma análise estática de tipos que consegue verificar a correção das chamadas às funções locais ou fornecidas por bibliotecas. No entanto, erros no consumo dos serviços REST são encontrados apenas em tempo de execução. Este artigo apresenta o SafeRESTScript, uma linguagem desenvolvida com o objetivo de mostrar que é possível estender o apoio da análise estática ao consumo de serviços web. O SafeRESTScript oferece uma coleção rica de tipos—com objetos, *arrays* e tipos refinados—que é partilhada com o HEADREST, a linguagem para descrever as APIs dos serviços REST consumidos.

1 Introdução

A arquitetura de um sistema de software tem um papel fundamental no alcance de propriedades como a fiabilidade, escalabilidade e facilidade de manutenção. Com a criação e crescimento da *World Wide Web*, surgiu a necessidade de criar estilos arquiteturais para aplicações que executam neste contexto. Durante as últimas décadas, entre os estilos que emergiram, o REST [5] destaca-se como o mais importante e mais popular [9].

Atualmente muitas aplicações são desenhadas de forma a oferecer interfaces aplicacionais que aderem a este estilo. Assim, o consumo de serviços REST tornou-se uma forma muito popular de invocar código fornecido por terceiros. No entanto, o apoio que os programadores têm disponível para escrever código que consome estes serviços é extremamente limitado quando comparado com o que em geral têm quando usam código de terceiros disponível em bibliotecas.

O consumo de APIs REST é particularmente popular em aplicações web e, portanto, muitos clientes de serviços web são programados em JavaScript e, mais recentemente, também em TypeScript. Apoiado num sistema de tipos rico, o compilador de TypeScript ajuda os programadores a encontrarem erros estaticamente. Em particular, os tipos ajudam a definir a interface de bibliotecas disponibilizadas a terceiros e o compilador ajuda a garantir que as chamadas às bibliotecas estão corretas. Em contraste, a linguagem não oferece nenhuma ajuda no que diz respeito ao consumo de serviços REST; erros nas chamadas a estes serviços só são encontrados em tempo de execução. O facto dos programadores de clientes de serviços web não terem forma de saber se as suas chamadas estão

corretas até à execução do código é identificado por Wittern et al. [13] como um dos quatro maiores desafios de investigação para o consumo de APIs web. O impacto prático disto é, por exemplo, atestado por um estudo recente numa companhia de pagamentos eletrónicos, que concluiu que erros no consumo de serviços REST causam a maioria das falhas nos clientes [2].

Este artigo apresenta a linguagem SafeRESTScript, uma linguagem desenvolvida com o objetivo de mostrar que é possível estender o apoio da análise estática ao consumo de serviços REST com a validação da boa formação dos pedidos REST, nomeadamente nos tipos dos parâmetros e do corpo da mensagem, bem como a validação da correta utilização das resposta (por exemplo, acesso aos membros dos objetos e vectores recebidos no corpo da resposta).

O SafeRESTScript capitaliza na linguagem HEADREST [12], uma linguagem de especificação de APIs REST que permite descrever individualmente cada uma das operações, num estilo reminescente dos triplos de Hoare e utilizando uma coleção rica de tipos, com objetos, vetores e tipos refinados. Apoiado num sistema de tipos, o compilador de SafeRESTScript ajuda os programadores a encontrarem erros estaticamente tanto nas chamadas locais como nas chamadas a serviços REST. O potencial desta ajuda é amplificado por uma tipificação que é sensível ao fluxo de controlo, realizada à custa de uma tradução para a linguagem intermédia de verificação Boogie [7] e da utilização do Z3 [4] para a verificação dos programas Boogie gerados.

A construção da linguagem SafeRESTScript é feita em dois passos: a construção de uma linguagem de *script* com tipos refinados e uma tipificação que é sensível ao fluxo de controlo, a que chamámos SafeScript, e a subsequente extensão desta linguagem com primitivas que permitem referir especificações de APIs REST e consumir estes serviços. O compilador de SafeRESTScript, tal como o de TypeScript, gera JavaScript. No entanto, ao contrário do TypeScript, SafeRESTScript não é uma extensão do JavaScript e não têm a ambição de ser uma linguagem de produção, mas antes contribuir para avançar o estado da arte da análise estática de programas e mostrar que é possível melhorar o apoio à escrita de código que consome serviços REST. A compilação de SafeRESTScript para JavaScript torna possível a utilização em conjunto das duas linguagens, ao mesmo tempo que permite automaticamente a execução de programas SafeRESTScript em muitos ambientes de execução diferentes.

O editor e compilador de programas SafeRESTScript está disponível na forma de um *plugin* para o Eclipse IDE, de utilização livre. Encontra-se atualmente em curso o trabalho de avaliação da linguagem em diferentes frentes como seja a capacidade para lidar com uma coleção de referência de desafios de verificação e o impacto da utilização da linguagem na taxa de erros e tempo de desenvolvimento de consumidores de serviços REST.

2 Contexto e trabalho relacionado

Descrição de APIs REST As interfaces aplicacionais devem ser sempre acompanhadas de documentação que permita que os clientes possam utilizar corre-

tamente os seus serviços. Existem atualmente várias linguagens de especificação de APIs REST como a Open API Specification[1] (OAS, originalmente chamada *Swagger*) ou a API Blueprint[2]. Estas linguagens permitem descrever as várias operações fornecidas por um serviço e aspetos como o formato dos pedidos de cada operação e respetivas respostas. No entanto, são bastante limitadas no suporte à descrição de restrições nos parâmetros dos pedidos e, ainda mais, no impacto que estes têm no formato e conteúdo da resposta. Estas limitações levaram ao desenvolvimento do HeadREST [12].

A linguagem de especificação HeadREST baseia-se em duas ideias chave: (i) utilização de tipos para expressar propriedade do estado do sistema e dos dados no conteúdo das mensagens e (ii) pré e pós-condições para relacionar os dados enviados nos pedidos e os dados recebidos na resposta, bem como o resultado das mudanças no sistema. A maior expressividade do HeadREST face às alternativas, levou a que fosse a linguagem adotada pelo SafeRestScript para especificação das APIs REST dos serviços consumidos. Essa expressividade advém em parte de (i) tipos de refinamento, `(x:T where e)`, que consistem nos valores `x` do tipo `T` que satisfazem a propriedade `e` e (ii) um predicado, `e in T`, que avalia se `e` pertence ao tipo `T`.

Programação de clientes de serviços REST O JavaScript é uma das linguagens mais utilizadas para programar código que consome serviços REST, principalmente porque muito desse código é executado a partir de um browser e, assim, é uma escolha natural para explorar a análise estática de clientes REST. No entanto o JavaScript é dinamicamente tipificado, o que coloca enormes desafios.

A análise estática de JavaScript tem sido um tópico de investigação muito activo nos últimos anos [10] e foram desenvolvidas muitas soluções, sendo a maioria baseada na introdução de um sistema de tipos forte. Exemplos disso são as linguagens TypeScript[3], Dart[4] e Flow[5], que têm uma análise estática baseada no respetivo sistema de tipos e que compilam para JavaScript. Também existem ferramentas que atuam diretamente sobre código JavaScript, como o JSHint[6], que deteta estaticamente erros e potenciais problemas em JavaScript.

Análise de chamadas a serviços REST Em contraste, a literatura que endereça a análise estática de chamadas a serviços REST é bastante escassa e as soluções existentes são bastante limitadas. A maioria das abordagens que endereçam a análise estática do consumo de serviços web envolve a execução do código, nomeadamente no contexto de testes escritos à mão ou gerados automaticamente. Exemplos disto são, por exemplo, o PACT[7] (em que os testes seguem o paradigma do *contract by exame*) e RESTler [1], que gera sequências de testes.

Wittern et al. [14] propuseram uma abordagem para validar estaticamente pedidos REST em JavaScript baseada em análise estática interprocedimental de

[1] https://www.openapis.org
[2] https://apiblueprint.org/
[3] https://www.typescriptlang.org/
[4] https://dart.dev/
[5] https://flow.org/
[6] https://jshint.com
[7] https://docs.pact.io

strings. Esta foca-se exclusivamente em pedidos ajax feitos a partir da biblioteca JQuery, apesar do JavaScript suportar diferentes formas de realizar pedidos REST. A abordagem proposta passa por verificar se os URLs dos pedidos correspondem aos *UriTemplates* declarados na especificação do serviço (escrita em OAS) e se os dados enviados são os esperados. A abordagem utiliza um grafo de chamadas "field-based" para fazer as necessárias análises às *strings* passadas pelos argumentos de métodos, visto que o URL do pedido pode ser formado por variáveis fora do âmbito da respetiva função. Por outro lado, não é feita nenhuma verificação relativamente aos tipos dos dados esperados na resposta.

RESTyped Axios é uma ferramenta que verifica estaticamente pedidos REST feitos em TypeScript contra especificações de APIs em RESTyped[8] e que cobre exclusivamente pedidos efetuados através da framework Axios. A ferramenta verifica se os URLs das chamadas são válidos e se os tipos dos membros passados no pedido e acedidos a partir da resposta correspondem aos declarados na especificação. A técnica implementada tira partido de tipos *keyof* e *Lookup*, parâmetros *default*, e inferência genérica.

Nos dois casos apresentados, as soluções foram desenhadas para duas linguagens de programação já existentes. O facto de se pretender utilizar a linguagem de especificação HeadREST, cujos tipos refinados obrigam a uma avaliação semântica, torna uma solução que envolva adicionar estes tipos e sua validação a uma linguagem à séria, como o JavaScript ou o TypeScript, impraticável. Deste modo, decidiu-se desenvolver uma linguagem similar ao JavaScript fortemente tipificada e com tipos refinados, que se denominou SafeScript. O SafeRestScript é uma extensão desta linguagem que adicionalmente permite realizar pedidos REST e que verifica estaticamente a correção destes pedidos de acordo com as especificações dos serviços em HeadREST.

3 A linguagem SafeScript

Grosseiramente, o SafeScript pode ser visto como um subconjunto do JavaScript equipado com um sistema de tipos bastante expressivo.

O SafeScript herda o sistema de tipos da linguagem HeadREST e, desta forma, são suportados objetos (`{}`, `{l: T}`), *arrays* (`T[]`), tipos refinados (`x: T where e`), tipos escalares (`int`, `boolean` e `string`) e um tipo de topo (`any`). A relação de subtipos é semântica, sendo semelhante à definida por Bierman et al. [3] para uma linguagem funcional. Herdado do HeadREST é também o importante operador (`e in T`) que permite avaliar se uma determinada expressão `e` pertence a um tipo `T`.

A partir deste conjunto pequeno de tipos é possivel derivar um conjunto maior de tipos bastante úteis. Os tipos interseção `T & U` e união `T | U` são definidos respetivamente por `(x: any where x in T & x in U)` e `(x: any where x in T | x in U)`. Um objeto com vários membros, por exemplo `{l: T, m: U}`, é definido por `{l: T} & {m: U}`. Um objeto com um membro opcional `{?l: T}`, é derivado por `(x: {} where x in {l: any} => x in {l: T})`.

[8] https://github.com/rawrmaan/restyped

O SafeScript tem um conjunto de expressões muito semelhante ao JavaScript: operadores primitivos para aritmética de inteiros e booleana, concatenação de strings (`++`), acesso a *arrays* e objetos, entre outros. O operador ternário (`e ? e : e`) apresenta a propriedade de passar a informação da condição para ambos os ramos, de forma a ser considerada na avaliação dos tipos. Por exemplo, se `obj` é uma variável com tipo `{}`, a expressão `(obj in {l: boolean} ? obj.l : false)` é válida, visto que no acesso `obj.l` é garantido que a variável `obj` possui o campo `l`. Os operadores binários booleanos apresentam a mesma propriedade. Por exemplo, o operador conjunção (`x && y`) é derivado da expressão `(x ? y : false)`. Deste modo, a expressão anteriormente escrita com o operador ternário é equivalente a `(obj in {l: boolean} && obj.l)`.

Adicionalmente também é permitida a utilização de quantificadores universais (`forall x: T . e`) e existenciais (`exists x: T . e`). Porém, podem apenas ser utilizados dentro da declaração de tipos, isto é, em modo especificação. Assim, os tipos com quantificadores não podem ser utilizados numa expressão com o operator `in`, visto que não podem ser avaliados (num caso geral) em tempo de execução. A utilização de quantificadores aumenta bastante a capacidade expressiva dos tipos e permite, por exemplo, especificar os elementos dentro de um *array*. Por exemplo, o tipo que representa o conjunto dos números primos apenas pode ser representado usando quantificadores:

```
type Prime = (x: int where x > 1 &&
        forall i: int . 1 < i && i < x ==> x % i != 0)
```

Um programa SafeScript consiste num conjunto de declarações de variáveis globais (`var x: T`), abreviaturas de tipos (`type X = T`) e funções. A declaração de uma função consiste no tipo de retorno, o nome da função, uma lista de parâmetros com os respetivos tipos e o corpo do método. Este último é um subconjunto das instruções oferecidas pelo JavaScript: declaração e inicialização de variável (`T x = exp;`), atribuição (`x = exp;`), condicional (`if (condition)B1 else B2`), ciclo-*while* (`while (guard)inv invariant body`) e retorno (`return exp;`).

Uma característica do SafeScript é o *flow typing*: uma variável, além de um tipo de declaração, tem um tipo que varia com o ponto no fluxo de controlo que se considera e que caracteriza o conjunto dos valores que a variável pode ter nesse ponto. Este tipo, que designamos por tipo efetivo, é necessariamente subtipo do tipo de declaração. Veja-se o seguinte programa SafeScript válido:

```
1 boolean f(int|boolean bit) {
2     if (bit in int) { return bit > 0; }
3     else { return bit; }
4 }
```

A variável `bit` é declarada com o tipo `int|boolean` e é utilizada em 3 sítios diferentes. É utilizada na avaliação do operator `in` na condição, que recebe uma expressão de tipo `any`, o tipo de topo. A partir da condição do condicional, o tipo da variável `bit` é mais restrito em cada um dos seus ramos: no ramo verdadeiro é do tipo `int` e por isso pode ser utilizada numa comparação, no falso é do tipo `boolean` e assim pode ser o valor de retorno deste método.

O tipo de uma variável pode também ser alterado por uma simples atribuição. Veja-se o seguinte exemplo:

```
1 boolean f(int|boolean bit) {
2     bit = true;
3     bit = 42;
4     return bit > 0;
5 }
```

Na primeira atribuição o tipo efetivo de `bit` passa a `(x: boolean where x == true)`, visto que, nos tipos de refinamento, o tipo de uma expressão corresponde ao conjunto de valores mais restrito que a respetiva expressão pode ter. Neste caso, depois da atribuição, o único valor que `bit` pode ter é **`true`**. De seguida é atribuído o valor `42` à variável, que apesar de não ser um valor do seu tipo no momento, é um valor do tipo com que foi declarada, logo a atribuição é válida. Por fim, a variável é utilizada numa comparação que espera um **`int`**, que é supertipo do tipo efetivo da variável, e assim a expressão é válida.

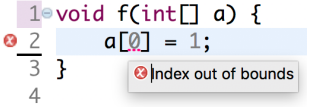
Os tipos refinados permitem detectar estaticamente erros comuns, como a divisão por zero:

```
1 type nat = (x: int where x >= 0);
2 int div(nat x) {
3     return 42 / x; // compilation error
4 }
```

Ao contrário de outros operadores sobre inteiros, a divisão restringe o segundo argumento a ser diferente de 0. Ou seja, para este sistema de tipos, o tipo do segundo argumento não é **`int`**, mas sim `(x: int where x != 0)`. Assim é assinalado um erro na divisão, visto que `x` pode ser `0` pelo seu tipo efetivo, que neste caso é igual ao declarado. Se, por exemplo, for adicionada a instrução `x = x + 1;` antes do retorno, o tipo efetivo de `x` passa a ser `(x: int where x >= 1)`, que é subtipo do tipo esperado pelo operador divisão e assim a expressão é válida.

Outro erro bastante comum é o acesso fora dos limites de *arrays*. Em JavaScript não existe distinção entre *arrays* e objetos, visto que os *arrays* são interpretados como objetos, i.e., um mapa de entidades para valores. Em SafeScript (como em muitas outras linguagens) os *arrays* são totalmente distintos dos objetos e no acesso aos seus elementos deve ser garantido um indíce dentro do tamanho do *array* (uma característica primitiva e imutável do mesmo).

A figura abaixo mostra este erro a ser detetado pela implementação do validador da linguagem descrito na Secção 5. Este validador está disponível num *plugin* para o Eclipse IDE.



No exemplo acima não é garantido que se possa aceder à posição `0` do array, isto é, que o *array* tenho um tamanho superior a `0`. Para tal basta, por exemplo, refinar o tipo do argumento a receber:

```
1 void f((x: int[] where length(x) > 0) a) {
2     a[0] = 1;
3 }
```

É bastante comum ter de iterar sobre todos os elementos de um *array* para realizar uma determinada operação. Neste caso, deve ser garantida que em qualquer momento da iteração o acesso é feito também dentro dos limites do *array*. O exemplo abaixo mostra um método que calcula a soma de todos os elementos de um *array* de inteiros. Apesar de fazer o pretendido corretamente, como se pode ver à esquerda na figura, o compilador da linguagem assinala um erro no acesso ao *array*. A guarda do ciclo garante que, na linha 5, `i` não excede o tamanho do array, porém não é garantido que a variável não é negativa. Para tal é necessário adicionar uma invariante ao ciclo ou então restringir o tipo declarado da variável `i`, como se pode ver à direita na figura.

```
1  int sum(int[] a) {
2      int i = 0;
3      int sum = 0;
4      while (i < length(a))
5      {
6          sum = sum + a[i];
7          i = i + 1;
8      }
9      return sum;
10 }
```

Index out of bounds

```
1  int sum(int[] a) {
2      (x: int where x >= 0) i = 0;
3      int sum = 0;
4      while (i < length(a))
5      {
6          sum = sum + a[i];
7          i = i + 1;
8      }
9      return sum;
10 }
```

As caracteristicas da linguagem SafeScript, nomeadamente o seu sistema de tipos e respetivos valores, foram escolhidas de forma a corresponderam à estrutura dos dados do formato JSON, o formato mais habitual do corpo das mensagens de pedidos REST. Deste modo, a linguagem SafeScript foi desenhada para ser estendida com a possibilidade de efetuar de pedidos REST.

4 A linguagem SafeRESTScript

A linguagem SafeRESTScript é uma extensão do SafeScript que permite realizar e validar estaticamente pedidos a APIs REST. Os pedidos são analisados contra uma especificação dos serviços em HEADREST.

Um programa SafeRESTScript importa no início a especificação HEADREST dos serviços consumidos (se for esse o caso), com `specification path of url`, onde `path` é o caminho para o ficheiro com a especificação, e o segundo corresponde ao URL base do serviço.

O programa que escolhemos para ilustrar a linguagem faz pedidos à Dummy API [9], um serviço feito para testes de aplicações clientes e que pretende simular uma base de dados de empregados. A especificação abaixo em HEADREST endereça apenas o *endpoint* do serviço que permite criar um novo empregado. De acordo com esta especificação, o *endpoint* `'/create'` exige um corpo de mensagem com um objeto que contem três membros (*name*, *salary* e *age*), todos do tipo

[9] http://dummy.restapiexample.com/

String. Se esta pré-condição for cumprida, então o serviço devolve uma resposta com o código 200 e com um corpo que consiste num objeto com o empregado criado, um objeto com os mesmos 3 membros que o pedido, e adicionalmente o identificador único do empregado. Para manter o exemplo simples, omitimos a especificação da unicidade do nome e não apresentamos a especificação dos casos onde os dados do pedido são incorreto ou o empregado não pode ser criado.

```
specification DummyAPI

type Employee = {name: String, salary: String, age: String}
type EmployeeWithId = Employee & {id: String}

{
    request in {body: Employee}
}
    post '/create'
{
    response.code == 200 &&
    response in {body: EmployeeWithId} &&
    response.body.name == request.body.name &&
    response.body.salary == request.body.salary &&
    response.body.age == request.body.age
}
```

Para fazer pedidos REST, a linguagem oferece a instrução `m s e`, onde `m` é o metodo HTTP (**get**, **post**, **put** ou **delete**), `s` é um literal **string** com o URL do recurso alvo relativo ao URL base declarado na importação da especificação, e `e` é uma expressão com o objeto que corresponde ao pedido.

Esta expressão tem de ter o tipo **Request**, enquanto que o valor retornado pelo pedido tem o tipo **Response**. Estes dois tipos são definidos, tendo em conta a estrutura das mensagens HTTP, da seguinte forma:

```
type Request = {
    location: string,
    headers: {},
    ?template: {},
    ?body: any
}
```

```
type Response = {
    code: int,
    headers: {},
    ?body: any
}
```

A linguagem JavaScript é *single thread* e o seu código não deve ser bloqueante, visto que um bloqueio na execução do código origina um bloqueio da página web onde o código está inserido. Deste modo, chamadas a métodos ou eventos que apresentem latência na resposta ou execução devem ser executados assincronamente, sendo colocados no fim da fila de execução. Os pedidos REST enquadram-se neste tipo de eventos e por isso devem idealmente ser assíncronos.

Recentemente o JavaScript introduziu as operações *async* e *await* para definir chamadas assíncronas utilizando promessas, em vez dos habituais *callbacks*. Esta nova sintaxe permite fazer chamadas assíncronas de forma mais simples e

próxima das chamadas síncronas. Utilizou-se a mesma ideia no SafeRESTScript. Para realizar um pedido REST assíncrono, este deve ser antecedido pela palavra reservada **await**. Os métodos que possuem chamadas assíncronas devem ser precedidos pela palavra reservada **async**, sendo que se um método faz uma chamada a outro que é assíncrono passa também a ser assíncrono e tem de ter a respetiva palavra reservada no cabeçalho.

Apresenta-se abaixo um exemplo simples de um programa SafeRESTScript com um método que faz um pedido à Dummy API para que seja adicionado um novo empregado.

```
1 specification "dummy.hrest" of "http://dummy.restapiexample.com/api/v1"
2
3 async string addEmployee(string name0, string salary0, string age0) {
4     Employee requestBody = {name = name0, salary = salary0, age = age0};
5     Response response = await post "/create" {body = requestBody};
6     return response.body.id;
7 }
```

Neste programa existem duas validações importantes relativas à chamada REST. Em primeiro lugar é verificado se o *endpoint* é válido, isto é, se na especificação do serviço está declarado que a operação **post** sobre o `/create` é suportada. Neste caso é simples pois o *endpoint* é definido por um URL. No caso geral, porque os *endpoints* são definidos por *URITemplates* (URIs com partes por preencher), isto não é trivial. Para a segunda validação é preciso começar por verificar se o objeto enviado no pedido verifica a respetiva pré-condição (o que é o caso). De novo, o caso geral é mais complicado porque o mesmo *endpoint* pode ter vários triplos e é preciso identificar quais os triplos cujas pré-condições são verificadas. De forma a suportar sub-especificação, o caso em que não existe nenhum triplo cuja pré-condição seja verdadeira não é considerado um erro. Por fim, é validado que a pós-condição do triplo para **post** `/create` garante que o objeto `response.body` existe, tem o campo `id` e tem um tipo que é subtipo de **string** (o tipo de retorno do método). No caso geral, usa-se a informação sobre a resposta derivada da disjunção das pós-condições dos triplos cujas prés foram identificadas no passo anterior.

5 Validação de programas SafeRESTScript

A validação de tipos é a parte principal e mais difícil da validação da linguagem. A Figura 1 apresenta uma visão geral de como é o processo de validação e compilação de programas SafeRESTScript.

A utilização de tipos refinados não permite uma simples análise sintática para verificar se um tipo é subtipo de outro ou se uma determinada expressão pertence a um tipo. Na linguagem HEADREST, a validação dos tipos é realizada com recurso a um SMT, de forma a resolver a relação se subtipos semanticamente, seguindo a abordagem proposta por Bierman et al. [3]. Esta abordagem não pode ser replicada diretamente no SafeRESTScript por se tratar de uma linguagem imperativa.

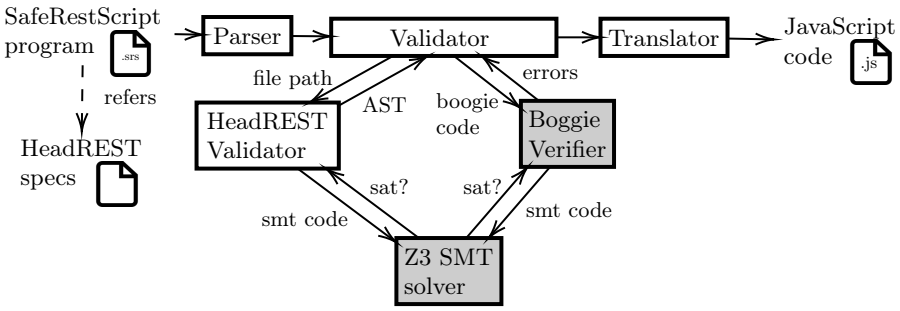


Figura 1. Esquema da compilação do SafeRestScript

Assim, para a validação de tipos nos programas SafeRestScript utilizou-se a linguagem intermédia de verificação Boogie [7]. A linguagem Boogie foi desenhada para acomodar verificações de condições para programas imperativos e orientados a objetos como por exemplo os que podem ser escritos em Dafny [6]. O verificador desta linguagem, com o mesmo nome, utiliza um SMT *solver* para validar as asserções do código Boogie. Para o SafeScript é utilizado o Z3, que também é o SMT *solver* por omissão do Boogie [4].

A axiomatização dos valores e dos tipos na linguagem Boogie segue de perto a usada na linguagem Whiley [8]. Whiley é uma linguagem de programação com uma forte análise estática, cuja validação é auxiliada por um provador de teoremas interno. Recentemente, Utting et al. proposeram uma validação da linguagem usando o Boogie, sendo este utilizado para provar as asserções necessárias, e não propriamente para a validação de tipos [11].

A modelação dos valores é feita numa perspetiva de conjuntos (como nos tipos refinados), onde todos os valores são do mesmo tipo no Boogie: `Value`. É através da definição de símbolos de função, e de axiomas que condicionam a sua interpretação, que são definidos os subconjuntos de valores para cada tipo. Por exemplo, para o tipo **`int`**, um dos casos mais simples, são definidas 3 funções:

```
1 function isInt(Value) returns (bool);
2 function toInt(Value) returns (int);
3 function fromInt(int) returns (Value);
4
5 axiom (forall i: int :: isInt(fromInt(i)));
6 axiom (forall i: int :: toInt(fromInt(i)) == i);
7 axiom (forall v: Value :: isInt(v) ==> fromInt(toInt(v)) == v);
```

Os axiomas definem as propriedades das funções e relacionam-nas, Por exemplo, o primeiro axioma afirma que todos os valores construídos a partir de inteiros primitivos do Boogie (com `fromInt`) são inteiros dos nosso sistema de tipos (`isInt`). Mais precisamente, a função `isInt` verifica se um valor é inteiro, a função `toInt` aplicado a um inteiro no nosso sistema de tipos dá o respetivo inteiro no Boogie e a a função `fromInt` efetua a operação inversa.

Bierman et al. traduzem a expressão `(e in T)` numa fórmula da lógica de primeira ordem. No nosso caso é feito um processo semelhante, usando as funções definidas na axiomatização. Por exemplo, `(e in int)` é traduzido em `isInt(e')`, sendo `e'` a tradução de `e`. Outros tipos, como os objetos, os *arrays* ou os tipos refinados, apresentam traduções bem mais complexas.

As verificações de tipos são feitas à base da geração de código Boogie com asserções. Sempre que é feita uma operação ou uma chamada de métodos, é verificado se cada uma das expressões dadas como argumentos pertence ao tipo esperado. As verificações dos métodos são baseadas adicionalmente em hipóteses, traduzindo os tipos esperados dos argumentos. Asserções e hipóteses são ambas feitas à base da tradução de expressões `(e in T)`. Por exemplo, para validação do programa com o método `int div(nat x)` apresentado anteriormente é gerado um programa Boogie que inclui (entre outras coisas) algo equivalente ao seguinte:

```
1  assume isInt(x) && toInt(x) >= 0;       // div: assuming x's type
2  assert isInt(fromInt(42));              // / : checking first arg's type
3  assert isInt(x) && toInt(x) != 0;       // / : checking snd arg's type
4  return fromInt(toInt(42) / toInt(x));
```

Neste caso, a validação do programa Boogie dá erro na linha 3. Mais precisamente, o validador do Boogie não consegue garantir naquele ponto que `x` é um inteiro diferente de 0. Deste modo, é reportado que `x` não possui o tipo esperado, `(i: int where i != 0)`.

No SafeRestScript, os pedidos REST são abstraídos através de chamadas a funções não interpretadas de `Request` para `Response`, cuja interpretação é restringida através de axiomas que capturam os pares de pré e pós-condições incluídos na especificação HeadREST do serviço.

Após um programa ser considerado válido, o que acontece apenas se a ferramenta Boogie conseguir provar todas as asserções do programa gerado, o código SafeRestScript é traduzido de uma forma relativamente direta para JavaScript.

6 Conclusão

Neste artigo são apresentadas duas novas linguagens: SafeScript, um subconjunto do JavaScript equipado com tipos refinados e com um predicado que permite verificar se uma expressão é de um determinado tipo; e SafeRestScript, uma extensão da primeira que valida estaticamente chamadas a serviços REST contra especificações em HeadREST. Pretendemos com o SafeRestScript contribuir com uma melhor verificação e correção numa área onde existem várias lacunas na fiabilidade do software.

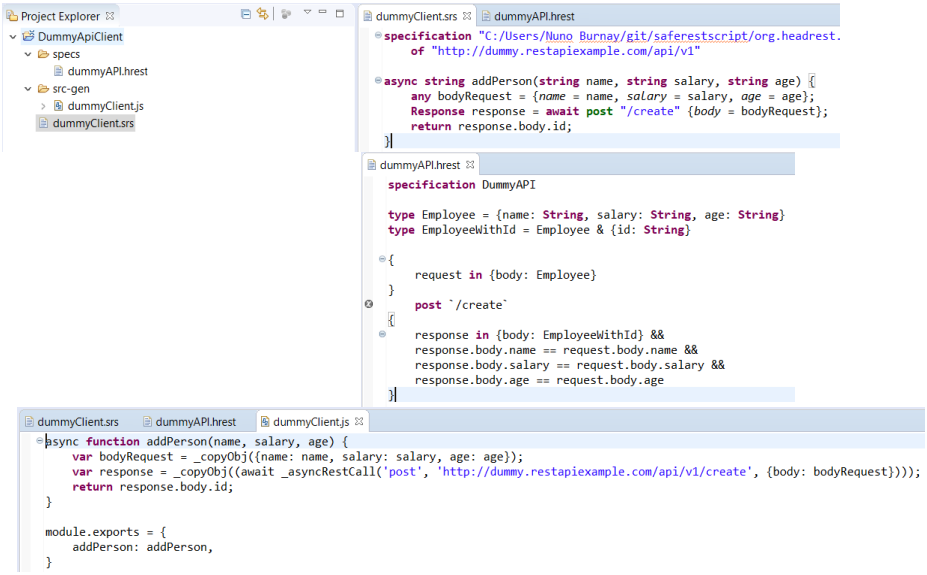
Agradecimentos. Este trabalho foi apoiado em parte pela FCT através do projeto *Communication Contracts for Distributed Systems Development*, PTDC/EEI-CTP/4503/2014 e da Unidade de Investigação LASIGE, UID/CEC/00408/2019.

Referências

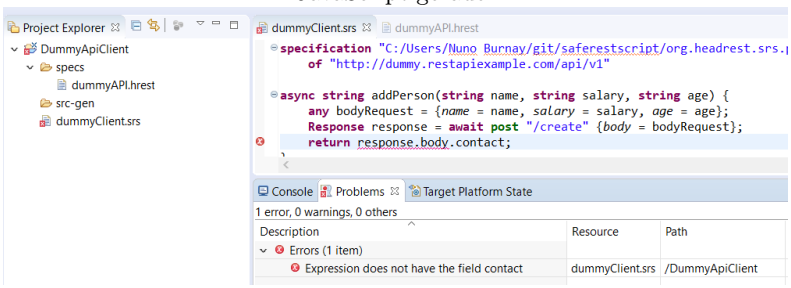
1. Atlidakis, V., Godefroid, P., Polishchuk, M.: Restler: Stateful rest api fuzzing. In: Proceedings of the 41st International Conference on Software Engineering. pp. 748–758. ICSE '19, IEEE Press, Piscataway, NJ, USA (2019), https://doi.org/10.1109/ICSE.2019.00083
2. Aué, J., Aniche, M., Lobbezoo, M., van Deursen, A.: An exploratory study on faults in web API integration in a large-scale payment company. In: Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice. pp. 13–22. ICSE-SEIP '18, ACM, New York, NY, USA (2018), http://doi.acm.org/10.1145/3183519.3183537
3. Bierman, G.M., Gordon, A.D., Hriţcu, C., Langworthy, D.: Semantic subtyping with an SMT solver. SIGPLAN Not. **45**(9), 105–116 (Sep 2010), http://doi.acm.org/10.1145/1932681.1863560
4. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg (2008), http://dl.acm.org/citation.cfm?id=1792734.1792766
5. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis (2000), aAI9980887
6. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning. pp. 348–370. LPAR'10, Springer-Verlag, Berlin, Heidelberg (2010), http://dl.acm.org/citation.cfm?id=1939141.1939161
7. Leino, R.: This is Boogie 2. Microsoft Research (June 2008), https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/
8. Pearce, D.J., Groves, L.: Whiley: A platform for research in software verification. In: Erwig, M., Paige, R.F., Van Wyk, E. (eds.) Software Language Engineering. pp. 238–248. Springer International Publishing, Cham (2013)
9. Richardson, L., Ruby, S.: Restful Web Services. O'Reilly, first edn. (2007)
10. Sun, K., Ryu, S.: Analysis of JavaScript programs: Challenges and research trends. ACM Comput. Surv. **50**(4), 59:1–59:34 (Aug 2017), http://doi.acm.org/10.1145/3106741
11. Utting, M., Pearce, D.J., Groves, L.: Making whiley boogie! In: Polikarpova, N., Schneider, S. (eds.) Integrated Formal Methods. pp. 69–84. Springer International Publishing, Cham (2017)
12. Vasconcelos, V.T., Martins, F., Lopes, A., Burnay, N.: HeadREST: A Specification Language for RESTful APIs, pp. 428–434. Springer International Publishing, Cham (2019), https://doi.org/10.1007/978-3-030-21485-2_23
13. Wittern, E., Ying, A., Zheng, Y., Laredo, J.A., Dolby, J., Young, C.C., Slominski, A.A.: Opportunities in software engineering research for web API consumption. In: Proceedings of the 1st International Workshop on API Usage and Evolution. pp. 7–10. WAPI '17, IEEE Press, Piscataway, NJ, USA (2017), https://doi.org/10.1109/WAPI.2017..1
14. Wittern, E., Ying, A.T.T., Zheng, Y., Dolby, J., Laredo, J.A.: Statically checking web api requests in JavaScript. In: Proceedings of the 39th International Conference on Software Engineering. pp. 244–254. ICSE '17, IEEE Press, Piscataway, NJ, USA (2017), https://doi.org/10.1109/ICSE.2017.30

Apêndice

Este apêndice ilustra as ferramentas que suportam o desenvolvimento de programas em SafeRESTScript. O exemplo mostra o programa SafeRESTScript apresentado no texto do artigo, e a especificação HEADREST de que este depende, no contexto do Eclipse IDE com um plugin implementado para dar suporte a estas linguagens.



(1) Especificação HEADREST. (2) Programa SafeRESTScript. (3) Programa JavaScript gerado.



Programa SafeRESTScript com um erro: o programa usa um campo do corpo da resposta que, de acordo com a especificação, não há qualquer garantia que exista.